Automatic extraction of x86 formal semantics from its natural language description

NGUYEN, Lam Hoang Yen

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

March, 2018

Master's Thesis

Automatic extraction of x86 formal semantics from its natural language description

1610062 NGUYEN, Lam Hoang Yen

Supervisor : Professor Mizuhito Ogawa
Main Examiner : Professor Mizuhito Ogawa
Examiners : Associate Professor Nguyen Minh Le
Professor Kazuhiro Ogata
Associate Professor Nao Hirokawa

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

Abstract

Nowadays, computers have become an essential device of almost every activity for everybody at any age. From personal demands to industrial and business sectors, they are used to improve human life as well as the efficiency and the productivity. The more important they are, the more attractive target they become for being attacked and serving malicious purposes. There are various threats to a computer system. One of the most common manners that penetrates or damages a system with bad impacts to most of the computer users is malware. Malware detection and malware classification are two of the most attractive problems in not only industry area but also academic research. The bits-based fingerprint also known as the signature-based pattern recognition is applied popularly in commercial anti-virus software due to its light-weight and fast features. However, it is easily cheated by advanced polymorphic techniques in malware. Therefore, malware analyses based on control flow graph (CFG) have been attracting a lot of attention, e.g., VxClass at Google. Semantic fingerprints are used to defeat advanced techniques of malware, which cause the restriction of bits-based fingerprints. They consist of code and CFG fragments that are acquired by disassembly. But malware classification requires further investigation of CFGs more deeply to clarify what types of techniques are applied. Our long-term target is to classify malware based on the observation of acquired obfuscation techniques. Therefore, we need to precisely generate highly accurate CFG of x86 binary that involves typical obfuscation techniques.

Hence, in order to construct the CFG, our collaboration at Ho Chi Minh City University of Technology have been developing BE-PUM (Binary Emulation for PUshdown Model generation), which is a binary analyzer. BE-PUM concentrates on malware, which is often small and obfuscated. BE-PUM applies symbolic execution to analyze an input file and constructs CFG for the given file. The binary emulator of BE-PUM takes responsibility for handling assembly instructions. Each instruction can modify the environment and the path condition in BE-PUM according to the Architectures Software Developer's Manual in English provided by Intel. There are several remaining problems in BE-PUM need to be addressed, such as the loop handling for the efficiency and huge manual effort for the implementation of symbolic execution.

This research is dealing with the second problem, especially the lack of x86 binary emulation support, which restricts the ability of analysis due to the unexpected termination when an input file involves an unsupported x86 binary instruction. Hence, an output CFG may miss disassembling significant characteristics for malware detection. Among more than 500 common x86 instructions, BE-PUM supports about 250 instructions after

3-years human effort. The emulator implementation and the path condition generation starts manually from reading its x86 assembly instruction specification in natural language (NL) and its pseudo-code description. However, the idea for using a commercial emulator is not enough since path conditions must be maintained for symbolic execution. Our aim is to automate the implementation of x86 binary emulation, based on know-how obtained during manual implementation.

However, writing computer programs from using natural language description is a challenging problem. Specifications in NL can be long, complicated or ambiguous, whereas programs have to be explicit, restricted and formal. Not only reducing manual implementation tasks, it may allow a product line-like approach to different platforms other than x86. In natural language processing, the semantic analysis that maps natural language sentences into a formal representation is semantic parsing. Recently, there are several works in semantic parsing. They have exploited natural language to address problems in domain specific applications, such as recipe from the description, robot commands, operating systems, scripts in smart-phone and spreadsheet data analysis. In order to develop such language-to-code systems, efforts for manually constructing parsers or large corpus of appropriate training samples are generally required.

Based on our observation, we found that x86 instruction specifications have particular features, such as type convention specified by Intel, the pseudo-code and the flag-change description, which may often contain informal keywords. These keywords require Natural Language Processing (NLP) to extract their semantics. From the obtained semantic, the emulator can be automatically generated. In order to claim the correctness of the generation, the information about the operand types and the pseudo-code in the document can also be used for the automatic test-case generation. In NLP, the ambiguity of the semantic parsing and interpretation is one of difficulties. However, we can apply testing to explicitly choose and clarify the accurate interpretation.

The research will contribute not only for automatic enhancement of BE-PUM, which is regarded as a case study of automatic code generation under rigid natural language specification and executable environments, but also automatic test data generation for validating whether implementation matches to the specification documents. We address the problem of automatic generation by building a parser to take advantage of pseudo-code and applying simple techniques in NLP to extract appropriate actions for text sentences in the flag-change descriptions. The flag-change description shows how an instruction execution affects to system flags. It consists of one or more sentences, in which each one can be classified into two types including an interpretation and an essential supplement for the pseudo-code accompanying within the same document. By measuring the similarity between a sentence in the flag-change description and prepared template sentences, the system can extract the way how system flags mentioned are affected. As a result of observation, the number of the template sentences is set to be five. To establish the final sensible decisions, the research needs to accomplish a statistic on the total collected sentences. However, there are still several ambiguous parts in pseudo-code which cannot be overcome. By manually preparing beforehand prerequisites for such unclear pieces, the approach can achieve the good efficiency with the minimum human effort. After

that, to verify the generated code for the binary emulation, we have to carry out a testing by comparing a pair of environments before and after executing an instruction in a commercial emulator or debugger (e.g., Intel/PIN, OllyDbg, x64dbg) in comparison with these by BE-PUM. To ensure the correctness, the research has to conduct a testing with high test coverage that is able to cover all of the statements in the generated code. This task firstly requires test-cases that are binary programs containing target x86 instructions. We aim to automatically generate them from the document. In order to generate test programs, we have to obtain the following essential specifications. The first is conditions for the validity of arguments (operands of instructions) to automatically generate all valid and possible forms of a given instruction. The second is conditions for covering branches of an execution for the test coverage.

The system requires some manual beforehand preparation as the prerequisites supporting the automatic code generation for x86 binary emulation in BE-PUM. Therefore, we manually implemented default rule-based flag-change modifications and 30 undefined functions frequently used in pseudo-code. Finally, the system can generate successfully 299 instructions among 530 collected specifications. In addition to the current result, the system is highly expected to be adapted to work on other specifications of different architectures/platforms with small modifications.

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Professor Mizuhito Ogawa for his continuous support and insightful encouragement in not only my research but also my student life at JAIST. His vast knowledge, inspiring ideas and critical thinkings are key factors that affect directly to the completion of my master's degree. I always admire him for his hard-working style and his care for all students.

Then, I would like to thank Associate Professor Nguyen Le Minh and Associate Professor Nao Hirokawa for their precious suggestions and useful advice in the progress of my research. I appreciate their feedback and comments, which have been a great help in improving my work.

Besides, I would like to show my appreciation to several friends, staffs and lecturers in Japan Advanced Institute of Science and Technology, who support me during the time I am staying in Japan. I learned a lot from them through their hard work and dedication. I got chances to achieve necessary knowledge as well as to immerse in a high-level educational environment.

Last but not least, my deepest appreciation from the bottom of my heart goes to my family and my friends in Viet Nam for always standing by and encourage me.

Contents

Abstract **i**

Acknowledgements **i**

Table of Contents **vi**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 x86 Architecture and Malicious Threats **4**
2.1 x86 Instruction 4
2.2 Malware Techniques on x86 Instructions 7

3 Dynamic Symbolic Execution of Binary and BE-PUM System **8**
3.1 Binary Emulation and Path Condition for Symbolic Execution 8
3.2 What is BE-PUM 9

4 Natural Language Processing in Automatic Generation of Dynamic Symbolic Execution **13**
4.1 x86 Specification In Intel Software Developer's Manual 13
4.2 WordNet and Sentence Similarity 14

5 Observation on Automatic Generation with Specification **17**
5.1 Required x86 Specification 17
5.1.1 Specification for Generation 18
5.1.2 Specification for Test-case Generation 19
5.2 Prerequisite for Automatic Generation 20

6 Specification Extraction **22**
6.1 Operation from Pseudo-code 22
6.2 Flag Modification from Natural Language Specification 25
6.3 Type Information 26

7 Automatic Generation **28**
7.1 Test-case Generation . . . 28
7.2 Conformance Testing . . . 30

8 Implementation **32**
8.1 Module Extraction . . . 32
8.2 Module Generation . . . 33
8.3 Module Testing . . . 37

9 Experiment **39**
9.1 Successfully Generated x86 Instruction . . . 39
9.2 Comparison with Manual Implementation . . . 40
9.3 Discussion of Failed Cases in Automatic Generation . . . 40

10 Related Work and Conclusion **42**
10.1 Related Work . . . 42
10.2 Conclusion and Future Work . . . 43

List of Figures

2.1 Registers in an x86 processor 5
2.2 Registers in an x86 processor 5

3.1 An illustration of the generation of multiple destinations 9
3.2 BE-PUM architecture 11
3.3 The binary emulation in BE-PUM 12

4.1 An sample of the specification of CWD/CDQ instructions 14
4.2 A fragment example of hierarchy in WordNet 15

5.1 Examples of an interpretation and an essential supplement of "flags-affected" descriptions 18
5.2 A part of the table in the document *Jcc* describes conditions for the conditional jump instructions 19

6.1 An example of the pseudo-code containing dead code in the description of *PUSHF/PUSHFD* 24
6.2 An example of the determination of positions of operands in a set U 25
6.3 An example of the determination of the interpretation in flag-change description 27

7.1 An example of the pseudo-code section that describes operations for multiple instructions 28
7.2 An example of the pseudo-code section that describes operations for multiple types of operands 29
7.3 An example of the pseudo-code section that consists of a conditional expression without the modification of operands in the expression 30
7.4 An example of the pseudo-code section that consists of a conditional expression with the modification of operands in the expression 30
7.5 An illustration of clarification by comparing environments in BE-PUM and debugger 31

8.1 Information extraction from specification 32
8.2 An example of concolic testing on conditional expressions in pseudo-code (see 7.3) 38

9.1 An example of unsupported architecture in the specification for FLDENV . 41
9.2 An example of unsupported function call in the specification for FBSTP . 41

10.1 The number of IoT malware samples detected each year (2013 - 2017) . . . 44

List of Tables

5.1 The convention for symbolizing operand types from Intel 20

6.1 The selected templates and their implications 26

Chapter 1

Introduction

Motivation and Problems

Malware is an general term used for referring to several forms of malicious software, including computer viruses, trojan horses, ransomware, spyware, adware and other unwanted programs. They are defined by their malicious aim which counteracts the expectations of users and damages computer systems. Their releases accomplish infection and harm via binary executable files, without source code. In order to hide and make them more difficult to be detected by typical binary pattern recognitions and dynamic analyses the obfuscation techniques are utilized in malware. They are the obfuscation (e.g., dead code insertion, instruction replacement and indirect jump), polymorphic algorithms (e.g., self modification and self encryption) and the sandbox detection (s.t. they do not reveal malicious behaviors in an emulator or a virtual environment). So that Symantec admits that commercial anti-virus software can detect less than 45% of the threats in 2015.

Model checking based approaches are attractive and draw many consideration for malware detection. First of all, a binary executable file is analyzed in order to obtain its abstract model, e.g., Control Flow Graph (CFG). Then, we can carry out analysis techniques based on model checking [19] [4] [8] [7] to complete the detection task. However, the challenge of this approach is on the model generation part because of the obfuscation techniques. The model checking part is more likely.

There are various analysis and model generation tools for a binary executable program, such as BIRD, CodeSurfer/x86, BINCOA/OSMODE, Renovo and Syman. However, Syman is the only tool having the support of system calls with a Window API emulator. **BE-PUM** (Binary Emulation for PUshdown Model Generation) [16] is a binary analyzer and focuses on malware program. BE-PUM applies dynamic symbolic execution to analyze an input file and constructs its Control Flow Graph (CFG). Binary emulator in BE-PUM takes responsibility for handling assembly instructions. Each instruction can modify the environment and path condition in BE-PUM according to the Architectures Software Developer's Manual from Intel written in English.

There are several remaining problems in BE-PUM need to be addressed, such as efficient loop handling and huge manual effort for implementation in symbolic execution. This

research is dealing with the second problem. The lack of x86 binary emulation support causes the unexpected termination when an input file involves an unsupported x86 binary instruction. Hence, an output CFG may miss to disassemble significant characteristics of malware. Among more than 500 common x86 instructions, BE-PUM supports about 250 instructions after 3-years human effort. Typically, an emulator implementation starts from reading its x86 assembly instruction specification and pseudo-code then writing code manually by human. This process is repeated one by one. By observation, we found that the Intel Software Developer's Manual contains the pseudo-code and the "flag-affected" descriptions of x86 semantics, which may contain informal keywords. These keywords require Natural Language Processing (NLP) to extract its semantics. From the obtained semantic, the dynamic symbolic execution can be automatically generated. In order to claim the correctness of the generation, the information about the operand types and the pseudo-code in the document can also be used for the automatic test-case generation which consists of the ambiguity removal in NLP and the conformance testing.

Our aim is to automate the implementation of x86 binary emulation, based on know-hows obtained during manual implementation. The research will contribute not only for automatic enhancement of BE-PUM, which is regarded as a case study of automatic code generation under semi-formal natural language specification and executable environments, but also automatic test data generation for validating whether implementation matches to the specification documents.

Contribution

We observe that the generation of x86 instruction emulator from its natural language specification requires restricted information. Besides, testing with executable test-cases and comparing the before and after environments in BE-PUM with the environments in a debugger can be applied to avoid ambiguity in information extracted from the document. At the moment, we have constructed a system for x86 instruction emulator generation.

- The system collected about 530 x86 instruction specifications from http://www.felixcloutier.com/x86/.
- The system can extract x86 semantics and generate instruction emulator. There are about 300 instruction supports having been successfully generated and tested.
- The generated instruction supports allow BE-PUM to handle more x86 instructions and point out 4 errors in manual implementations.

The thesis is organized in 10 chapters. Chapter 2 brings the brief background knowledge. Chapter 3 and 4 introduces BE-PUM system and our observation of the specification from Intel with NLP techniques used for the research. Chapter 5 shows our significant observations for deciding the method and implementation. Chapter 6 and 7 describe in detail our methodology to extract x86 semantics in which the binary emulation, the path condition and the test-cases to cover branches are generated. Chapter 8 and 9 expresses

our implementation and experiments. And lastly, the chapter 10 discusses our conclusion and future works.

Chapter 2

x86 Architecture and Malicious Threats

2.1 x86 Instruction

x86 is a family of backward-compatible instruction set architectures. The term "x86" derives from the names of several processor ending with "86"[1]. Nowadays, x86 usually implies a binary compatibility with the x86 32-bit instruction set. It obeys the design of complex instruction set computing (CISC), where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store).[2] [3] The x86 instruction set is a set of assembly instructions supported by x86-compatible processors. These instructions have variable binary lengths and variable operands (i.e. accompany with no operand or up to 3 operands).

A typical x86 32-bit processor consists of the following components:

- 8 general-purpose registers (GPR)
  The following list shows the order that is used not only in a push-to-stack operation, but also for covering back to the register later.

  1. Accumulator register (AX): arithmetic operations
  2. Counter register (CX): shift/rotate instructions and loops
  3. Data register (DX): arithmetic operations and I/O operations
  4. Base register (BX): pointer to data (located in segment register DS)
  5. Stack Pointer register (SP): pointer to the top of the stack
  6. Stack Base Pointer register (BP): pointer to the base of the stack
  7. Source Index register (SI): pointer to a source in stream operations

[1] https://en.wikipedia.org/wiki/X86

[2] https://en.wikipedia.org/wiki/Complex_instruction_set_computer

[3] Tanenbaum, Andrew S. (2006) Structured Computer Organization, Fifth Edition, Pearson Education, Inc. Upper Saddle River, NJ.

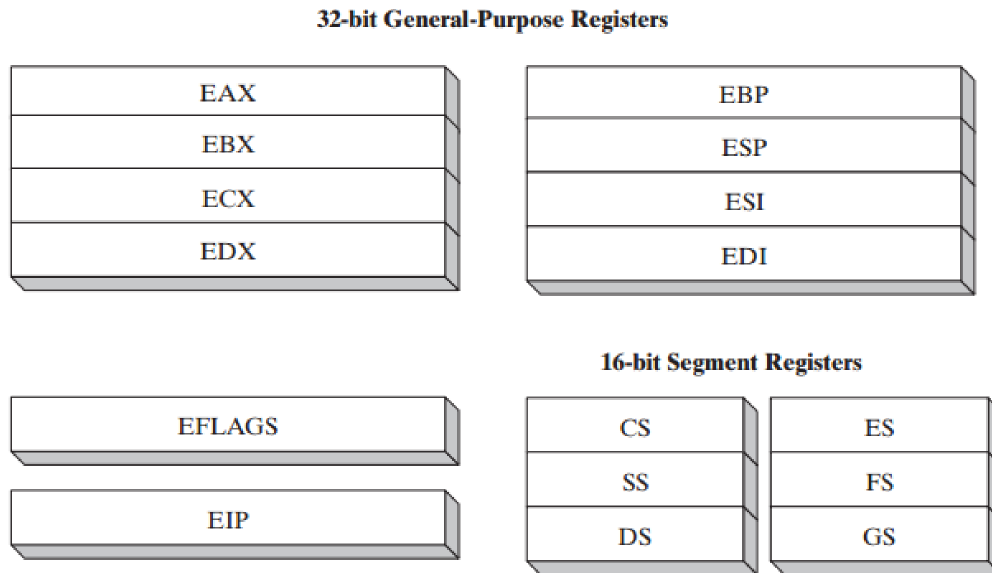

Figure 2.1: Registers in an x86 processor

8. Destination Index register (DI): pointer to a destination in stream operations

General-purpose registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, a register is recognized by its two-letter according to the list above. In 32-bit mode, these abbreviations are prefixed with a letter 'E' (extended). For instances, 'EAX' is the 32-bit accumulator register. The first four registers (AX, CX, DX and BX) can be accessed in their size of two 8-bit, by replacing the letter 'X' with the letter 'H' (higher part) or 'L' (lower part).

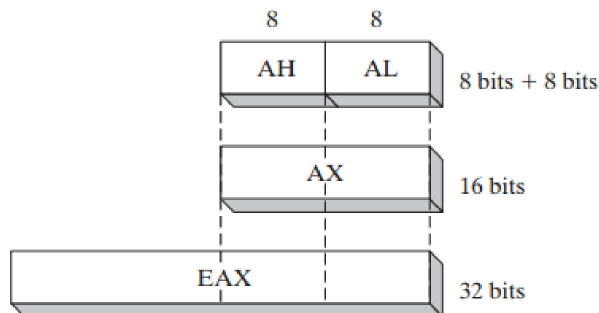

Figure 2.2: Registers in an x86 processor

- 6 segment registers:
  - Stack Segment (SS): pointer to the stack

  - Code Segment (CS): pointer to the code
  - Data Segment (DS): pointer to the data
  - Extra Segment (ES): pointer to extra data
  - F Segment (FS): pointer to more extra data (comes after 'E')
  - G Segment (GS): pointer to still more extra data (comes after 'F')

- Flags register (EFLAGS) is a 32-bit register used as the list of bits that stores Boolean values, representing characteristics of results of operations, and the processor state as follows:

  - Carry Flag (CF): it is set when the last arithmetic operation carries (addition) or borrows (subtraction) a high-oder bit (leftmost) outside the size of of registers.
  - Parity Flag (PF): it is cleared if the number of 1-b.its in the last arithmetic operation is a multiple of 2, otherwise the set value is stored
  - Adjust Flag (AF): the carry of the last arithmetic operation on Binary Code Decimal (BCD) numbers
  - Zero Flag (ZF): the result of an operation is zero (0) or not.
  - Sign Flag (SF): it indicates the result of an operation. The negative result sets it to 1.
  - Trap Flag (TF): single-step mode indication for debugging.
  - Interruption Flag (TF): if interrupts are enabled, it contains set value.
  - Direction Flag (DF): stream direction indicates the direction for moving or comparing is left or right.
  - Overflow Flag (OF): if a register can not contain the signed arithmetic operations result due to a too large value, the flag is set.

- The instruction pointer, which is represented by EIP register, contains the address of the next instruction to be executed.

- The memory uses the little-endian architecture. It means that multi-byte values are written least significant byte first.

- The stack (Last In First Out data structure) stores values which are popped from it in the reverse order that they are pushed onto. It is the temporary storage that allows a processor to rapidly save and retrieve data.

2.2 Malware Techniques on x86 Instructions

Malware detection and malware classification are two of the most attractive problems in not only industry area but also academic research. Bits-based fingerprint also known as signature-based for pattern recognition is applied popularly in commercial anti-virus software due to its light-weight and fast features. However, it is easily cheated by advanced polymorphic techniques in malware. Therefore, malware detection based on control flow graph (CFG) has been appealing to a lot of attention recently. Semantic fingerprints [11] [2] are used to defeat advanced techniques of malware, which cause the restriction of bits-based fingerprints. They consist of code and CFG fragments that are acquired by disassembly. But malware classification requires further precise CFG to investigate more deeply what types of techniques are applied.

Nevertheless, the task of obtaining precise CFG is difficult. In general, there are three steps applied in malware techniques:

1. *Obfuscation* (i.e., complicating work flows in order to discard the signature-based detection) and anti-debug to curtail malicious characteristics during virtual environment emulation.

2. *Infection*, i.e., a malware embeds itself in other executable programs.

3. *Malicious actions* (e.g., taking control illegally, destroying data, information theft).

Our long-term target is to classify malware based on the observation of acquired obfuscation techniques. Therefore, we need to precisely generate highly accurate CFG of x86 binary that involves typical obfuscation techniques. The typical obfuscation techniques such as *indirect jump, dead code insertion, instruction replacement and self modification* can overcome commercial dissemblers (e.g., IDA Pro, Capstone). Besides, if a malware is executed in a sandbox environment in order to observe behaviors and its CFG, *sandbox detection* may help it to curtail malicious characteristics.

For example, we present the technique indirect jump, which hides the target location in a register or memory, used in a code fragment from the real malware `Virus.Adson.1559` in the below code section. The code shows that it eventually wants to call the Windows API `FindFirstFileA` by dynamically loading the library `Kernel32` beforehand and calling the Windows API `GetProcAddress`. As for the two first instructions, the value of the pointer to `Kernel32` handler and the string “FindFirstFileA” are pushed in the stack. Next, `GetProcAddress` is invoked and the address of `FindFirstFileA` is returned and stored in the register EAX. Finally, the instruction at `004024B3` call `FindFirstFileA` through the technique indirect jump.

```
004024A6        PUSH EAX                          ; Kernel32 Handle
004024A7        PUSH DWORD PTR SS:[EBP + 403236]  ; 'FindFirstFileA'
004024AD        CALL DWORD PTR SS:[EBP + 40323A]  ; Call GetProcAddress
004024B3        JMP EAX                           ; Call FindFirstFileA
```

Chapter 3

Dynamic Symbolic Execution of Binary and BE-PUM System

3.1 Binary Emulation and Path Condition for Symbolic Execution

Symbolic execution, a natural extension of normal execution, is a traditional technique to execute a program symbolically. It is extended to accept symbolic inputs and construct symbolic formulas as output. It maintains the state of a program execution (l, pc) that includes a instruction location l and a path condition pc. The path condition pc is a boolean expression over symbolic values interpreting the precondition from the program entry point to the current instruction l. If it satisfiable, the execution path is feasible.

As for **symbolic execution**, a variable's value can be represented by a symbolic value or a symbolic expression [9]. For instance, the value of the register *eax* in computer can be presented by a symbolic expression "$\alpha_1 + \alpha_2 + 5$", where $\{\alpha_i\}$ is the set of symbolic constants. Then, an environment is obtained by collecting all variables and their corresponding values. A path condition is a logical constraint over symbolic for the path from the entry point to a particular location in a program. The program updates both of its environment and path conditions during its execution flows. Thus, according to the theory, a state of these two components need to be maintained. However, in practice, the environment and the path conditions are usually implemented separately in order to decrease memory space and update the environment easily.

In each step of on-the-fly manner, we can obtain an x86 assembly instruction *asm* (together with its parameters) by disassembling a binary sequence of an input program at a specific location k. If the instruction *asm* is not an external function call (i.e. invoking a system call of an operating system), it will be handled the binary emulation in BE-PUM. According to the semantic of *asm* extracted from the Intel manual, the binary emulation will modify the state of the environment and the path conditions.

Based on our observation, most of data instructions (e.g., *inc*, *mov*, *add*) do not cause branching on a CFG or affect to the path conditions. Except that parameters in conjunction with values in the environment lead to exceptions and make the path condition

unfeasible. For example, the state of the path condition P is $(\{eax = \alpha_1, ebx = \alpha_2\}, true)$ (for simplicity, many variables of the environment are omitted) and the next x86 instruction *asm* is "*inc eax*". The x86 instruction "*inc*" adds one to the operand "*eax*" according to the Intel manual. Thus, the path condition P' is $(\{eax = \alpha_1 + 1, ebx = \alpha_2\}, true)$ without the update of the path constraint. As for data instructions, the next location depends on the length of the instruction *asm* statically.

However, control instructions (e.g., *jz*, *jc*, *jmp*) may cause branching because of multiple possible destinations. The next location can not be statically decided if a condition or a target address of the instruction consists of symbolic values. Symbolic execution can deal with them to explore destinations in two ways:

- **Static symbolic execution** finds candidates statically. It checks whether each destination P' where $PathConst'$ is feasible or unfeasible by the satisfaction of $PathConst \wedge next = PathConst'$.
- **Dynamic symbolic execution** explores next destinations $P', P'', \ldots$ by testing satisfiable instances of $PathConst$ (*concolic testing*) [17].

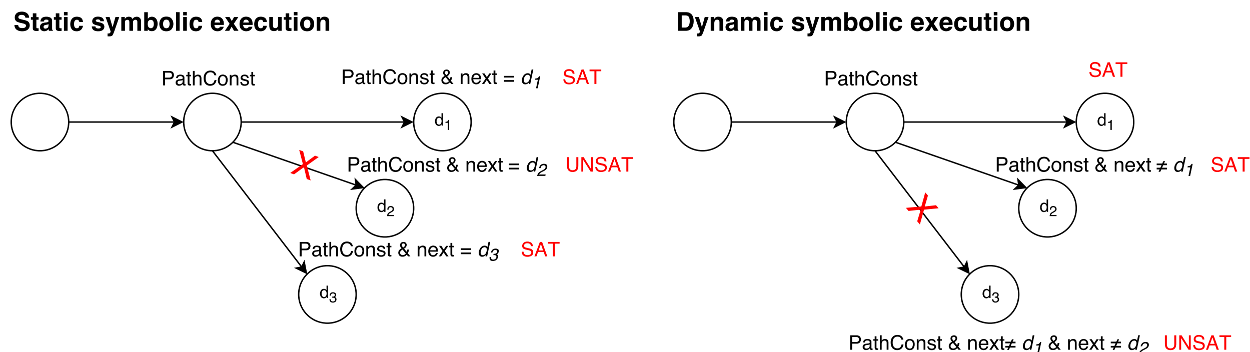

Figure 3.1: An illustration of the generation of multiple destinations

3.2 What is BE-PUM

BE-PUM (Binary Emulation for PUshdown Model generation) is a binary analyzer. Currently, BE-PUM concentrates on malware, which are often small and obfuscated. BE-PUM inputs an x86/Win32 executable binary file and then outputs its CFG (control flow graph). In order to construct the CFG, BE-PUM chooses on-the-fly manner to symbolically execute the input program. There are two reasons for BE-PUM to prefer on-the-fly-construction of CFG:

- From the perspective of binary level, data and instructions are treated in the same way. Hence, the execution of binary can modify not only data, but also instructions of the given binary executable.

- The current instruction together with environment determine the location of the next instruction while executing the program. For instance, if the current instruction is an indirect jump operation *"jump eax"*, then the next instruction is relied on the value of the register eax at that time.

Symbolic execution is adapted to execute the input program in BE-PUM. By concolic testing with a instance of *pc*, BE-PUM is able to decide the next instruction. An implementation of virtual simulation is compulsory for handling stepwise executions.

The implementation of BE-PUM is written in Java. We use the figure 3.2 to represent and briefly describe the architecture of BE-PUM. BE-PUM reuses and takes advantage of JakStab 0.8.3 to disassemble binary code of an input executable file to assembly instructions. Besides, SMT Z3.4.3 is also applied as a backend engine to produce a test instance for concolic testing. The architecture of BE-PUM consists of three elements: *symbolic execution*, *binary emulation*, and *CFG storage*. One symbolic state is taken out of the frontiers in turn at the ends of explored execution paths. Then, the symbolic execution attempts to extend one more step from it. If the instruction is a arithmetic instruction (i.e., only update the environment and the next location is statically determined), BE-PUM will simply disassemble the next instruction. If a control instruction (e.g., conditional jump instruction) is encountered, we will apply the concolic testing to figure out the next location. Thereafter it find out a new CFG node or CFG edge, this new exploded information is stored in the CFG storage and the frontiers obtain a corresponding configuration. This repetitive process keep continuing until either we explore all branches, or encounter an unsupported instruction, a system call or an unknown address.

BE-PUM executes an input program symbolically on the path condition *pc* (on the symbolic value) and the environment *Env* (the mapping table storing variables and their corresponding values), which are independent, in the implementation. The environment *Env* is represented by the tuple (Env_R, Env_S, Env_M), where Env_R, Env_S and Env_M contains values of the register, the stack and the memory (excludes the stack values), respectively. For a memory location $k = Env_R(eip)$, let $asm = instr(Env_M, k)$ be an x86 instruction (with its arguments) obtained by disassembling a sequence binary code starting from k. A binary emulation is used for handling x86 assembly instructions of the given executable file. According to the technical description of Intel software developer's manual, the binary emulation is implemented to deal with the modification on the environment and the path condition of the current x86 instruction *asm*.

The binary emulation in BE-PUM (described in the figure 3.3) consists of three components: *a pre-condition P*, *the binary emulation*, and *a post-condition P*. The path condition in BE-PUM is presented by a tuple $(Env, PathConst)$. The $PathConst$ is a boolean expression over symbolic values, represents the condition for the path from the initial entry point to the current location k and never contains any variables of the environment *Env*. The path condition also can be branched due to several causes. One of these reasons is multiple possible destinations, which is produced by a conditional jump instruction whose condition contains symbolic values. Currently, BE-PUM can handles about 250 x86 instructions and they were manually implemented. However, these implementations may consists of errors and there are more or less 500 x86 instructions in

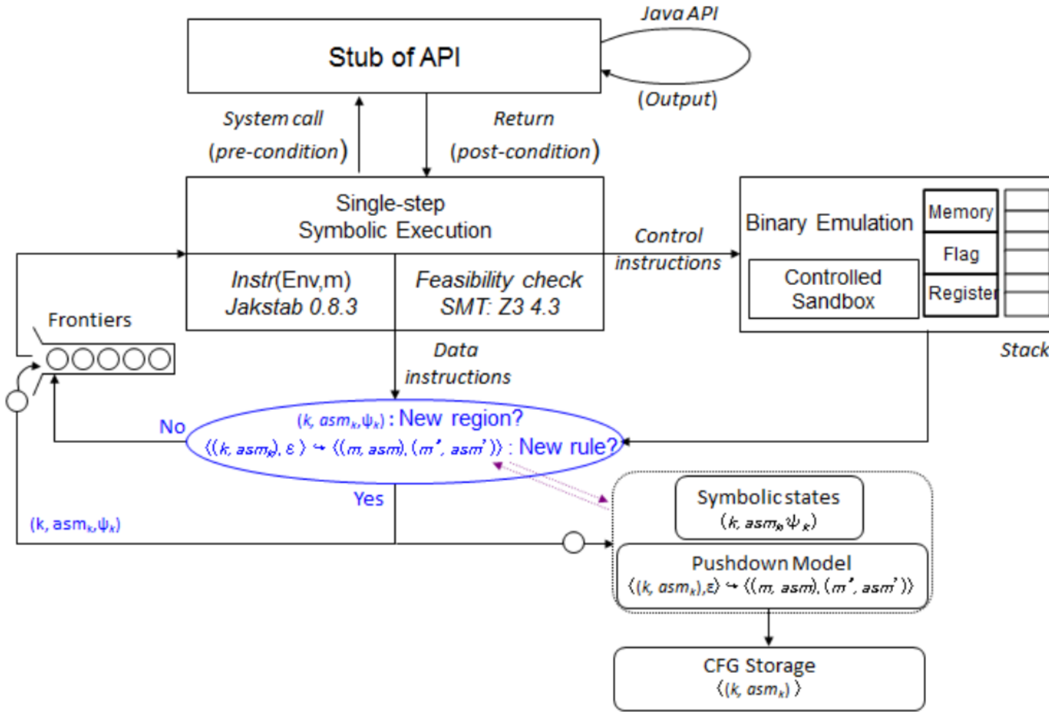

Figure 3.2: BE-PUM architecture

total. The lack of supporting in binary emulation may cause an unexpected termination in BE-PUM. For that reason, in order to enhance the ability of BE-PUM and reduce human intervention, we aim to conduct the research in the hope of applying its method to expand BE-PUM into other platforms (e.g., ARM).

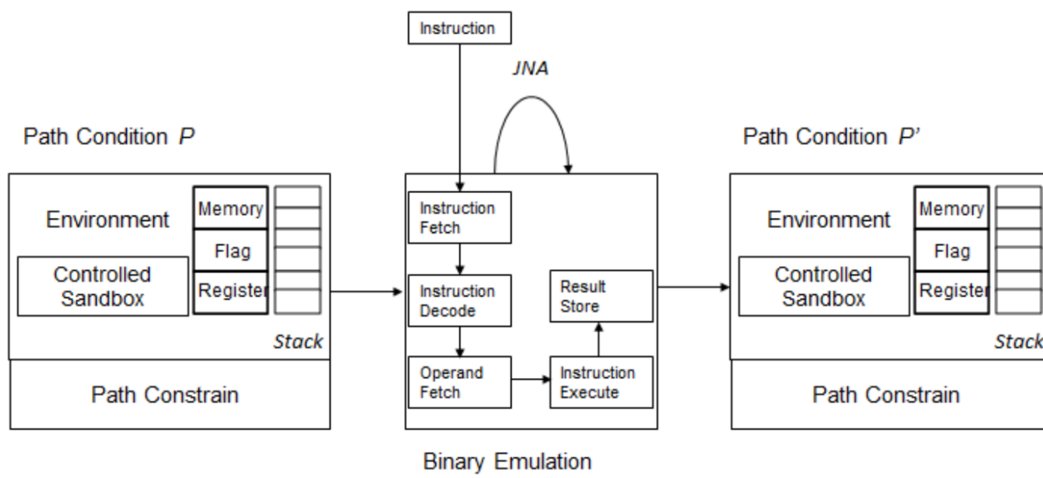

Figure 3.3: The binary emulation in BE-PUM

Chapter 4

Natural Language Processing in Automatic Generation of Dynamic Symbolic Execution

4.1 x86 Specification In Intel Software Developer's Manual

The x86 instruction set document are officially published in the Intel Software Developer's Manual under the PDF format. This such format hinders the automatic parsing on the content of document. Therefore, it turns out that the extraction module need an alternative format of such manual that is easier to be parsed. The alternative document consists of numerous HTML web pages (semi-structured data) and each page (e.g., the figure 4.1) is a specification of an x86 instruction. Each specification of an x86 instruction contains the following descriptions in order:

- **Instruction name**: It indicates the instruction(s) is(are) specified in the current specification.
- **Type information table**: It consists of several rows, in which each row mentions one specific opcode (operation code), which is represented in hexadecimal number and is the piece of a machine language instruction that implies the instruction to be executed, a corresponding assembly statement with valid operand type(s) and its description.
- **Instruction description**: It describes the function and operation of the current instruction(s) in natural language.
- **Pseudo-code (operation)**: By using pseudo-code, it specifies the operation which describes how the environment is affected after an execution of the mentioned instruction(s).

- **Flag-change description**: It is described in natural language by one or many sentences. For each sentence, it mentions flags and an effect that will be applied on these such flags. Due to the small amount of all sentences appearing in the total specifications, we give a try to the most naive manner to extract the effect to mentioned flags by measuring the similarity of a given sentence with sentence templates. Then, the system is hoped to acquire a good result with the minimal effort of this approach.

CWD/CDQ

Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Mnemonic	Description
99	CWD	DX:AX = sign-extend of AX
99	CDQ	EDX:EAX = sign-extend of EAX

Description
Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register (see Figure 7-6 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1). The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

Operation

```
if(OperandSize == 16) DX = SignExtend(AX); //CWD instruction
else EDX = SignExtend(EAX); //OperandSize = 32, CDQ instruction
```

Flags affected
None.

Figure 4.1: An sample of the specification of CWD/CDQ instructions

4.2 WordNet and Sentence Similarity

WordNet[15] is a large semantic lexicon database for English. It contains nouns, verbs, adjectives and adverbs and groups them into sets of synonyms (synset) which expressing a distinct concept i.e. one sense. If a word has more than one sense, it will appear in several synsets. Synsets are organized into hierarchies based on the hypernym/hyponym relation. As for two given concepts X and Y , if X is a kind of Y , then Y is hypernym of X and X is hyponym of Y. WordNet 3.0 provides 117,798 nouns in 82,115 synsets, 11,529 verbs in 13,767 synsets, 21,479 adjectives in 18,156 synsets, and 4,481 adverbs in 3,621 synsets, in a total of 155,287 words in 117,659 synsets.[1] The figure 4.2 shows an example of the hypernym/hyponym taxonomy in WordNet which is used for several word-to-word similarity measurements. A common parent of two synsets is known as a subsumer. The

[1] http://wordnet.princeton.edu/wordnet/man/wnstats.7WN.html

least common subsumer (LCS) is the subsumer that does not have any hyponym which is also a subsumer of two given synsets. In other words, the LCS is the closest subsumer to the concerning synsets. For example, the synset "wheeled vehicle" is the LCS of two synsets "car" and "truck".

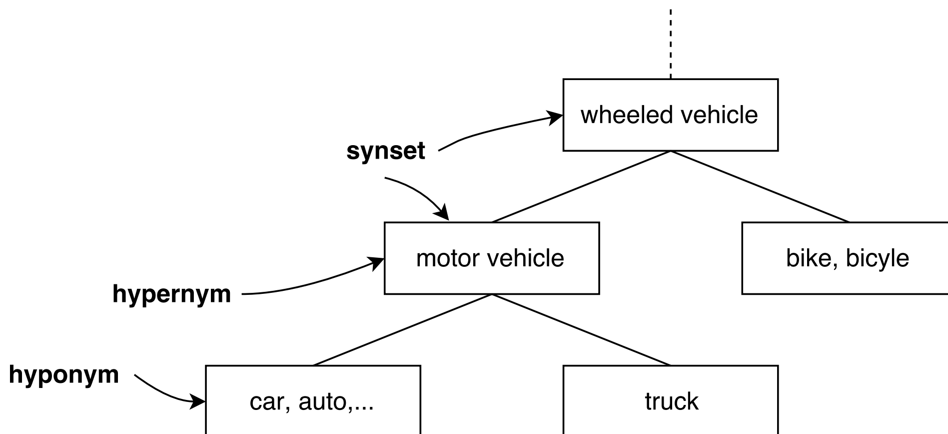

Figure 4.2: A fragment example of hierarchy in WordNet

Given two text inputs T_1 and T_2, we would like to automatically measure their **sentence similarity** in meaning by deriving a score. Thus, it goes beyond the plain lexical matching method. The knowledge-based measure of text semantic similarity (Mihalcea, R, et al., 2006) [14] is a simple approach but achieve an acceptable result for this task. A formula of this method is used for computing a similarity score and accomplished by combining measurements of word-to-word similarity and word specificity. The consideration of word specificity supports the formula to give a higher weight when matching a pair of specific words and give less importance to a measurement between generic concepts. In order to determine the specificity of words, it uses the inverse document frequency (*idf*) introduced by Sparck-Jones (1972). The idf is a statistic that reflects the importance of a word belonging to a document in a collection. It is defined as the total number of documents in the collection divided by the total number of documents consisting that word.

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

where N is the total number of documents in the collection and $|\{d \in D : t \in d\}|$ is number of documents where the word t appears.

There are a number of measures that were well-developed and work well on the hierarchy of WordNet for the task of measuring word-to-word similarity. All these measurements accept to input as a pair of concepts and give out a value implying their similarity. One of the popular and simple formula for this task is proposed by Wu and Palmer (1994) [20]. Its measure is based on the depth of two given concepts and the LCS in the WordNet taxonomy. Then, these values are combined into a similarity score by the following formula:

$$sim_{wup}(concept_1, concept_2) = \frac{2 * depth(LCS)}{depth(concept_1) + depth(concept_2)}$$

The formula of sentence similarity is defined by firstly picking each word w in the sentence S_1 and attempting to identify the word in the sentence S_2 that forms the highest word-to-word similarity ($maxSim(w, S_2)$). The same process is carried out for each word in the sentence S_2 afterward. Next, the obtained word similarities are multiplied with the corresponding word specificity. Finally, these values are summed up, normalized with the length of each sentence and combined with a simple average.

$$sim(S_1, S_2) = \frac{1}{2}\left(\frac{\sum_{w \in \{S_1\}}(maxSim(w, S_2) * idf(w)))}{\sum_{w \in \{S_1\}} idf(w))} + \frac{\sum_{w \in \{S_2\}}(maxSim(w, S_1) * idf(w)))}{\sum_{w \in \{S_2\}} idf(w))}\right)$$

$$maxSim(w, S) = max\left[sim_{wup}(w, s) | s \in S\right]$$

Chapter 5

Observation on Automatic Generation with Specification

5.1 Required x86 Specification

The manual of x86 instructions consists of various specifications, in the context of this research we only need a few parts of them. Concerning the entire system, we have to extract specific information for corresponding tasks as follows:

- To implement the dynamic symbolic execution, we have to extract operations of x86 instructions to update the environment and generate next path conditions when a current execution causes branching.
- To verify the generated code for the binary emulation, we have to carry out a testing by comparing a pair of environments before and after executing an instruction in a real emulator (e.g., Intel/PIN, OllyDbg, x64dbg) with another pair in BE-PUM. This task requires test-cases that are binary programs containing corresponding target x86 instructions. We aim to automatically generate them from the document. In order to generate test programs, we have to obtains the following specifications:
  - *Conditions for the validity of arguments (operands of instructions)*: To automatically generate all possible forms of a given instruction, the number of operands (in the range of zero to three) and data types of them must be valid and initialized before invoking the instruction in a test program.
  - *Conditions for covering branches of an execution*: In an operation of each instruction, there may be more than one execution flow. For entering a branch, we can find at least one set of valid initialized operands. To satisfy the test coverage, several test programs in which each branch have to be executed need to be generated automatically.

5.1.1 Specification for Generation

First and foremost, the most important requirement for the generation is the operation of x86 instructions. Each document of an x86 instruction contains **the pseudo-code section and the "flags-affected" description**. They describe how an instruction execution affects to the environment. The "flags-affected" section consists of one or several sentences. From our observation, each sentence can be categorized into the following two cases and automatically classified by the manner described in the section 6.2

- **An interpretation** describes the way the pseudo-code section updating system flags in English. We ignore this kind of sentence in the specification. We observe that a sentence is an interpretation when the mentioned system flags are also appear in the pseudo-code section.
- **An essential supplement** supplies the modifications of system flags that the pseudo-code section does not represent. The mentioned system flags in this case are not included in the pseudo-code section based on our observation. For solving this problem, we extract system flag names from the sentence and predict its action by measuring the similarity between it and prepared template sentences.

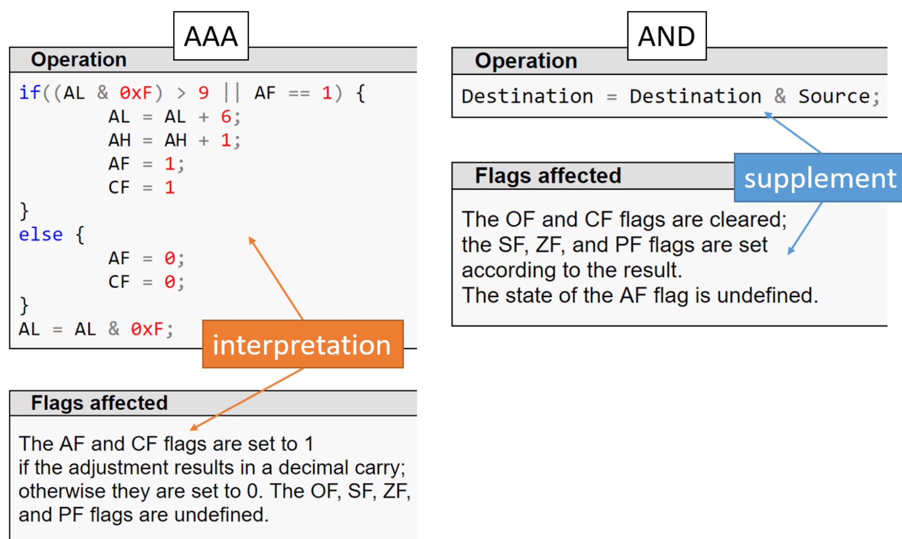

Figure 5.1: Examples of an interpretation and an essential supplement of "flags-affected" descriptions

For example, the figure 5.1 shows the pseudo-code parts and the "flags-affected" descriptions of two instructions *AAA* and *AND*. In regard to the specification of *AAA*, the first sentence of the "flags-affected" description explains the effects of the pseudo-code to the two system flags *AF* and *CF*. As for the instruction *AND*, there is no statement about the changes of the system flags *OF*, *CF*, *SF*, *ZF* and *PF* in the pseudo-code. However, they are described in natural language by the "flags-affected" description.

From our observation, **a document (i.e., a web page) can specify descriptions for more than one instruction**. In this case, the pseudo-code contains the operations that can be applied for many instructions. These documents are recognized by their titles. For instance, the instructions *RCL*, *RCR*, *ROL*, *ROR* are defined in the same document labeled "*RCL/RCR/ROL/ROR*". By extracting semantics from one document, we can generate code for supporting several instructions. Besides the title which contains several instruction names, the next indication is ending with letters "*cc*". This indication implies these documents describe conditional instructions (i.e., executed if the condition is met). And the operation is described not only in pseudo-code but also in the type information table. For these documents, the type information table consists of several rows, in which each row mentions one specific instruction and its condition. For example, the figure 5.2 shows a part of the type information table in the document "*Jcc*". The first row in the table gives the condition for executing the instruction *JA* which is the values of *CF* and *ZF* equal to zero.

Opcode	Mnemonic	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0).
73 cb	JAE rel8	Jump short if above or equal (CF=0).
72 cb	JB rel8	Jump short if below (CF=1).
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1).
72 cb	JC rel8	Jump short if carry (CF=1).
E3 cb	JCXZ rel8	Jump short if CX register is 0.
E3 cb	JECXZ rel8	Jump short if ECX register is 0.
74 cb	JE rel8	Jump short if equal (ZF=1).
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF).

Figure 5.2: A part of the table in the document *Jcc* describes conditions for the conditional jump instructions

5.1.2 Specification for Test-case Generation

As mentioned in the section 5.1, the test-case generation requires two compulsory specifications including conditions for the validity of operands and covering branches. **The descriptions for valid operand types** are described in the column *Mnemonic* of the type information table (e.g., figure 5.2). After the instruction name, a sequence of valid operand types are listed. These types are named by the conventions from Intel as symbols. A symbol is a concatenation of letters and a number, which represent a data type and a data size, respectively. The table 5.1 shows the convention for symbolizing operand types in detail. From a type description, by replacing symbols with valid operand values, we eventually acquire a satisfiable x86 instruction statement used in a test program. For instance, as for the type information "*ADD r/m8, imm8*", a candidate can be used for testing is "*ADD al, 10*".

The conditions for covering branches to satisfy test coverage are obtained from the pseudo-code section of each document. The system extracts conditions from the operation. There are several variables can appear in a condition. Some of variables are determined by instruction statement disregarding the values of operands. For examples, the variables *NumberOfOperands* and *OperandSize* are statically decided by the number

Starting letters	Data sizes (in bits)	Description	Examples
rel	8, 16, 32	A relative address	rel8, rel16
r	8, 16, 32	A general-purpose register	AL, AX, EAX
imm	8, 16, 32, 64	An immediate value (signed number)	imm8 is a signed number between −128 and +127
r/m	8, 16, 32, 64	Either the value of a general-purpose register or a memory operand	AL, AX, 0x4001001
m	8, 16, 32, 64, 128	An operand in memory	0x4001001

Table 5.1: The convention for symbolizing operand types from Intel

of operands of an assembly statement and the data size that the instruction operating on (e.g., "*ADDB*" is the byte addition operation, thus, *OperandSize* is 8). Based on our observation, the generation of test programs from the type information table can cover branches caused by conditions that can not be affected by operand values.

5.2 Prerequisite for Automatic Generation

Because the operation of instructions are represented by pseudo-code, an informal programming language, and there are many pieces of existing basic knowledge that are not explicitly defined in the document, the system requires some prerequisites that need to be prepared manually in order to automatically generate code for x86 binary emulation in BE-PUM. These necessary preparations are reused frequently in several instructions. Therefore, with a little manual effort, the system can deal with a large number of instructions and enhance the number of successful generated code.

The default rules of modifying system flags are compulsory prerequisites. In the Intel software developer's manual, there is no specification for the changes of system flags after executing an instruction. These rules are generic and always applied to all instructions that make use of arithmetic operations in the operation. Based on the last arithmetic operations l and r and its result t, the final states of flags are decided. There are 6 system flags are affected by the following rules:

1. **OF**: It is set to true if the last operation on signed numbers causes overflow of data. Otherwise, its value is false.
2. **SF**: A positive value of t clears its value and a negative result set it to true.
3. **ZF**: A non-zero value of t clears it to false and a zero value of t turns it to true.
4. **AF**: The carry of the last arithmetic operation on Binary Code Decimal (BCD) numbers.

5. **PF**: It contains the parity bit (or check bit) of the value of t.

6. **CF**: It indicates the carry from left-most bit after the last arithmetic operation. For instance, when the last operation is addition, its value is true only if both l and r are positive and t is negative, or both l and r are negative and t is positive.

The prerequisites for undefined functions in pseudo-code helps the system out to successfully automatically generate code. The pseudo-code sections consists of several ambiguous function calls. Extracting semantics and specifications for these functions is difficult. Besides, they are reused many times in individual instructions, by the manual implementation for ambiguous functions, we can still enhance the ability of BE-PUM with the minimum human effort.

Chapter 6

Specification Extraction

6.1 Operation from Pseudo-code

The description of each x86 instruction always consists of a pseudo-code section, which describes operation in an informal programing language. It is the most important part in the description that is used for automatically extracting semantic and generating code for the binary emulation in BE-PUM. Unfortunately, there is not the certain definition for the syntax of the pseudo-code presented in the document.

From our observation, **the pseudo-code can be parsed by a simple parser** because the syntax is not complicated. Therefore, we observed and deduced manually **the context free grammar** (including 20 rules) for parsing the pseudo-code as follows:

```
code                 → statement+
statement            → block
                     | structuredStatement
                     | exceptionStatement `;'
                     | simpleStatement `;'
block                → `{' statement* `}'
simpleStatement      → assignmentStatement
                     | expression
                     | breakStatement
structuredStatement  → conditionalStatement
                     | repetitiveStatement
                     | forLoopStatement
                     | switchStatement
exceptionStatement   → `#' IDENT
parameterList        → expression (`,' expression)*
assignmentStatement  → <assoc=right> expression `=' expression
                     | `(' parameterList `)' `=' expression
expression           → (`+' | `−')? factor
                     | `~' expression
                     | factor `.' identifier
```

```
              | expression (`*' | `/' | `div' | '%') expression
              | expression (`|' | `&') expression
              | expression (`+' | `-') expression
              | expression (`<<' | `>>') expression
              | expression (`==' | `!=' | `>' | `<' | `<=' | `>=')
                expression
              | expression (`||' | `^' | `&&') expression
              | <assoc=right> expression `=' expression
indexing         → identifier `[' expression `..' expression `]'
                 | identifier `[' expression `:' expression `]'
                 | identifier `[' expression `]'
concatenate            → identifier `:' identifier
factor                 → identifier
                       | funcCall
                       | `(' expression `)'
                       | unsignedConstant
                       | indexing
                       | accessAddress
                       | concatenate
accessAddress          → (`CS:' | `SS:' | `DS:') expression
funcCall               → IDENT `(' parameterList? `)'
unsignedConstant       → NUM_INT | NUM_HEX | STRING_LITERAL
conditionalStatement   → `if' `(' expression `)'
                         statement (`else' statement)?
repetitiveStatement    → `while' `(' expression `)' statement
forLoopStatement       → `for' `(' assignmentStatement+ `;'
                         expression `;' expression `)' statement
switchStatement        → `switch' `(' expression `)' `{'
                         caseStatement+ `}'
caseStatement          → `case' expression `:' statement*
```

An abstract syntax tree (AST) can be obtained from a pseudo-code section by parsing. It is a representation as a tree of a programing language source code. The AST represents the abstract syntactic structure and usually forms an input of code analysis and code generation. However, the code generation can not directly uses the AST obtained from the pseudo-code since it consists of redundant information. A computer processor has several operating modes such as real mode, protected mode and virtual mode to be selected and executed on. Thus, the Intel software developer's manual also describes operations for all supported modes. Nevertheless, BE-PUM concentrates on malware that shows its behaviors in the real environment status.

Hence, **the system need to removes all dead branches** (i.e., the branches describing the operations in non-real modes) before moving to the next step. The specification from Intel has the convention for using some variables in the following list to imply operating modes of computer.

- **PE**: boolean variable representing protected mode
- **VM**: boolean variable representing virtual mode
- **IOPL**: integer variable representing I/O privilege level

Then, *in the real mode, the values of both PE and VM have to be zero (false)*. The system can identify dead branches by statically evaluating conditional expressions, and consequently point out the expressions whose values are always concrete. The figure 6.1 shows an example of removing dead code from the pseudo-code of the instruction *PUSHF/PUSHFD*. As for the first conditional statement, by calculating the condition with the fixed zero values of PE and VM, it turns out that the second branch will never be executed. Therefore, in this case, the system only keeps the first branch of the first conditional statement in the pseudo-code section and dismisses everything else.

always true *(PE = 0 and VM = 0)*

```
if(PE == 0 || (PE == 1 && (VM == 0 || (VM == 1 && IOPL == 3)))) {
     if(OperandSize == 32) Push(EFLAGS & 0xFCFFFF);
 //VM and RF EFLAG bits are cleared in image stored on the stack
     else Push(EFLAGS[0..15]);
}
//In Virtual-8086 Mode with IOPL less than 3
else Exception(GP(0)); //Trap to virtual-8086 monitor
```

need to be removed/ignored

Figure 6.1: An example of the pseudo-code containing dead code in the description of *PUSHF/PUSHFD*

Code analysis is applied on the AST that has been pruned dead branches in order to extracts essential information for code generation. It extracts variables (including registers, flags, defined variables, global variables, etc.) and function calls. We define the following notations for this extraction task:

- The set of registers: R = {'EAX', 'EBX', 'ECX', 'EDX', 'AX', 'AH', 'AL', 'BX', 'BH', 'BL', 'CX', 'CH', 'CL', 'ESI', 'EDI', 'ESP', 'EBP', 'EIP', 'DX', 'DH', 'DL', 'SI', 'DI', 'SP', 'BP', 'DS', 'ES', 'FS', 'CS', 'SS', 'GS'}
- The set of registers system flags: F = {'AF', 'CF', 'DF', 'IF', 'OF', 'PF', 'SF', 'TF', 'ZF'}
- The set of variables: V = the set of identifies appearing in a given AST
- The set of indicated flags: $IF = V \cap F$ (i.e., the flags appearing in a given AST)

- The set of indicated registers: $IR = V \cap R$ (i.e., the registers appearing in a given AST)
- The set of updated variables: $UV = \{x | x \in V \wedge x \text{ appears in the left hand side of assignment statements}\}$
- The set of flags updated in a given AST: $UF = IF \cap UV$
- The set of flags specified in flag-change specification: FF
- The set of global variables: $G =$ specific variables and used in all descriptions (e.g., Destination, Source, OperandSize, etc.)
- The set of unknown variables: $U = V - G - F - R - UV$

By observation, we consider variables in U are operands of a given instruction. The corresponding positions of operands are unclear at the time of code generation. Therefore, the system need to produce a combination of operands and feasible positions. Then it generates code for each case and carries out a test to clarify which one is correct. For example, the pseudo-code in the description for the instruction *BTS* contains two unknown variables ($U = \{$'BistBase', 'BitOffSet'$\}$). Hence, the system attempts to produce the possible forms of the current instruction including '*BTS BitBase BitOffSet*' and '*BTS BitOffSet BitBase*'. In other words, there are possibilities for the positions of the operands. The first possibility is 'BistBase' at the first operand and 'BitOffSet' at the second operand. The second one is 'BitOffSet' at the first operand and 'BistBase' at the second operand. For each case, the system can generate the corresponding code for the binary emulation. The testing step finally is performed to clarify the correct one, which is '*BTS BitBase BitOffSet*'.

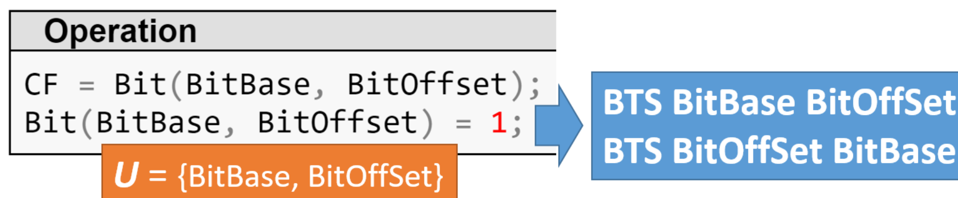

Figure 6.2: An example of the determination of positions of operands in a set U

6.2 Flag Modification from Natural Language Specification

As mentioned in the section 5.1.1, a flag-change description describes how an instruction execution affects to system flags in natural language. It consists of one or more sentences, in which each one can be classified into two types including an interpretation and an essential supplement.

By measuring the similarity between a sentence in flag-change description and prepared template sentences, the system can extract the way how system flags mentioned are affected. Intuitively, we notice that there are at least three operations that can be applied on system flags. They are “do nothing”, “set to 1” and “clear to 0”. From our observation, there are two more operations including “set according to the result” and “set according to the value of a register”. As a result, the number of template sentences is supposed to be five. To establish the final sensible decisions, the research need to accomplish a statistic on the total 289 sentences. Due to the simplify of such sentences and based on our observation, the statistic is performed by grouping sentences having the same last word. Eventually, it leads to the conclusion for the templates, which occupy the majority of cases, as showed in the table 6.1.

<table>
<tr><th>Template</th><th>Proportion (%)</th><th>Implication</th></tr>
<tr><td>“None”</td><td>32.2%</td><td>There is no system flags which are affected</td></tr>
<tr><td>“The flags are undefined”</td><td>17%</td><td rowspan="2">The system flags mentioned in the sentence will be modified by the default rules (see 5.2)</td></tr>
<tr><td>“The flags are set according to the result”</td><td>8.3%</td></tr>
<tr><td>“The flags are cleared”</td><td>6.2%</td><td>The value of the system flags mentioned in the sentence will be cleared to zero</td></tr>
<tr><td>“The flags are unaffected”</td><td>3.8%</td><td>There is not effect to the system flags mentioned in the sentence</td></tr>
</table>

Table 6.1: The selected templates and their implications

The interpretation is ignored due to the ambiguity of natural language and the clarity of pseudo-code. There are two steps to deal with this task:

- From a flag-change description, the system extracts system flags mentioned and represents them in the set FF.
- The system only makes use of the set of the flags that do not appear in pseudo-code: $FF' = FF - UF$

For example, the flag-change description of the instruction AAA consists of the interpretation at the first sentence (figure 6.3). This such sentence mentions the modification of two flags including ‘AF’ and ‘CF’, which are elements of the set FF. From the pseudo-code section, the system extracts the set UF, which contains the same elements as the set FF. Therefore, the specification of modification on these two system flags in the flag-change description are disregarded.

6.3 Type Information

A type information table (e.g., the figure 5.2) specifies not only possible prototypes but also conditions for execution of a document. It consists of several rows and three columns.

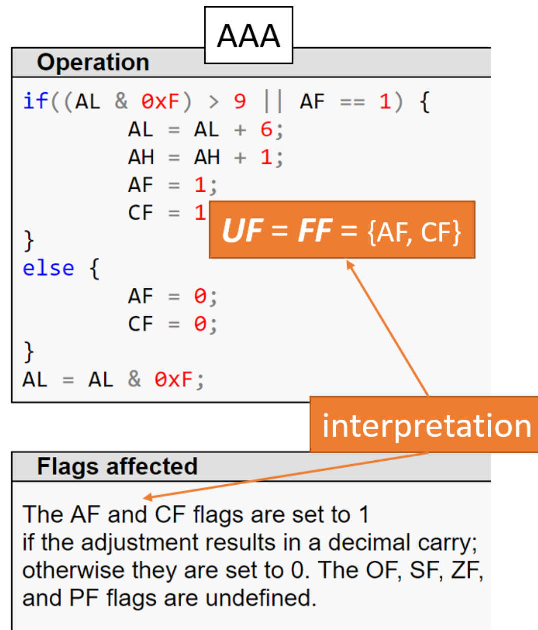

Figure 6.3: An example of the determination of the interpretation in flag-change description

Among these rows, each one describes a specific prototype and its condition. The system extracts information on the second column ("Mnemonic") and the third column ("Description").

- In the case that the table that does not contain conditions for execution, only the second column is needed in order to extract possible prototypes for generating test programs.
- If the table contains conditions for execution, in addition to the necessary information as below, each row of the third column need to be parsed to obtain the condition being dedicated to the instruction name in the second column.

Chapter 7

Automatic Generation

7.1 Test-case Generation

The operation of an x86 instruction consists of several branches. To ensure the correctness, the research has to conduct a testing with high test coverage that is able to cover all of the statements in the generated code. The generated code can have the following cases.

- An operation consists of procedures for dealing with several instructions.

  $\langle Problem \rangle$ A grouped specification, whose the indication is its title, can specify descriptions for more than one instruction. The same pseudo-code section in this kind of specification can be applied for many instructions. For instance, the instructions including *FADD*, *FADDP* and *FIADD* are defined in the same specification labeled "*FADD/FADDP/FIADD*" (the figure 7.1 shows the pseudo-code section of the specification).

  $\langle Solution \rangle$ A type information table of such the specification consists of all instruction names together with valid operands that can invoke all instructions described in the pseudo-code section. Therefore, based on the type information table, the system generates test programs in the case of grouped specification.

```
Operation
if(Instruction == FIADD) Destination = Destination + ConvertToExtendedDouble(Source);
else Destination = Destination + Source; //source operand is floating-point value
if(Instruction == FADDP) PopRegisterStack();
```

Figure 7.1: An example of the pseudo-code section that describes operations for multiple instructions

- An operation consists of procedures for dealing with several ways of inputing operands.

  $\langle Problem \rangle$ Concerning the properties of operand, such as operand size and number of operands, an instruction may have several way to behave and affect to the environment. For instance, the operation specification for "*CBW/CWDE*", which is

showed in the figure 7.2, has two ways to affect to the environment. If the operand size is 16, the value of the register AX will be modified. Otherwise, the register EAX will be updated.

⟨*Solution*⟩ A type information table of the specification in this case consists of all possible types of operands. The testing can cover all treatments for individual operand properties by generating test programs from this such table.

Operation
```
if(OperandSize == 16) AX = SignExtend(AL);
else EAX = SignExtend(AX);
```

Figure 7.2: An example of the pseudo-code section that describes operations for multiple types of operands

- An operation consists of various conditional statements with conditions formed from elements of environment.

  ⟨*Problem*⟩ Depending on the state of an initialized environment, an instruction may has many ways to behave. The figure 7.3 shows an example of branching in operation (of the instruction AAA) based on the values of the register AL and the flag AF.

  ⟨*Solution*⟩ In principle, the system need to explore all feasible branches. And for each branch, a path condition can be acquired by the similar methodology of BE-PUM. Thereafter, by conducting concolic-testing in turn for each path condition, the system can eventually obtains concrete instances as required initialized arguments for test programs that can cover all branches. Based on our observation, *there are only 6 specifications that modifies values of operands in conditional expressions before evaluating them.* For example, the figure 7.4 shows the pseudo-code section of the instruction TEST. For choosing a branch, the operation invokes an evaluation of the expression "Temporary = 0", where the value of "Temporary" is decided by a fore-assignment.

  The statistic turns out that most of the specifications do not update the environment before executing a conditional statement involving the environment state. It leads to the fact that the implementation for an automatic generic test-case generation system can take more time than a manual test-case generation. In addition to, branches and conditions in specifications are not complicated. Hence, in the case of conditional expressions with modification, we manually add more test-cases to satisfy test coverage. As for conditional expressions without modification (150 cases), the system extracts conditional expressions and generates combinations of them in both cases being true and false. Then, the concolic-testing are directly performed to obtain necessary inputs.

```
Operation
if((AL & 0xF) > 9 || AF == 1) {
        AL = AL + 6;
        AH = AH + 1;
        AF = 1;
        CF = 1;
}
else {
        AF = 0;
        CF = 0;
}
AL = AL & 0xF;
```

Figure 7.3: An example of the pseudo-code section that consists of a conditional expression without the modification of operands in the expression

```
Operation
Temporary = Source1 & Source2;
SF = MSB(Temporary);
if(Temporary = 0) ZF = 1;
else ZF = 0;
PF = BitwiseXNOR(Temporary[0:7]);

CF = 0;
OF = 0;
AF = Undefined;
```

Figure 7.4: An example of the pseudo-code section that consists of a conditional expression with the modification of operands in the expression

7.2 Conformance Testing

After the code generation and test-case generation, the system need to conduct a testing to clarify the correctness of generated code in the binary emulation of BE-PUM. The idea is to compare a pair of environments before and after executing an instruction in a commercial emulator/debugger (e.g., Intel/PIN, OllyDbg, x64dbg) with another pair in BE-PUM (illustrated by the figure 7.5). By executing the testing task for each test program in the set of generated test-cases, the system not only verifies whether generated code of an instruction is correct or not, but also clarifies the accurate choice for positions of operands in the set U (see 6.1).

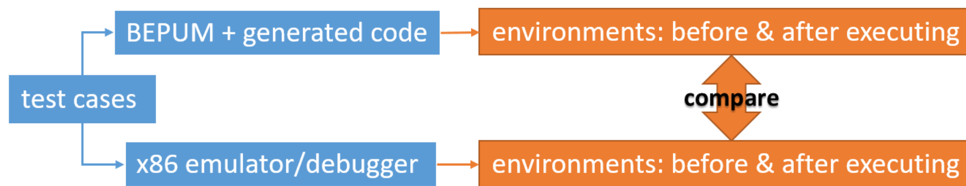

Figure 7.5: An illustration of clarification by comparing environments in BE-PUM and debugger

Chapter 8

Implementation

The implementation of automatic generation of x86 instructions for the binary emulation in BE-PUM is written in Java, Python, Velocity script language and context free grammar for ANTLRv4. The entire system is divided into 3 modules including extraction, generation and testing.

8.1 Module Extraction

As mentioned in the section 6, this module firstly performs the extraction task and obtains raw information as the figure 8.1.

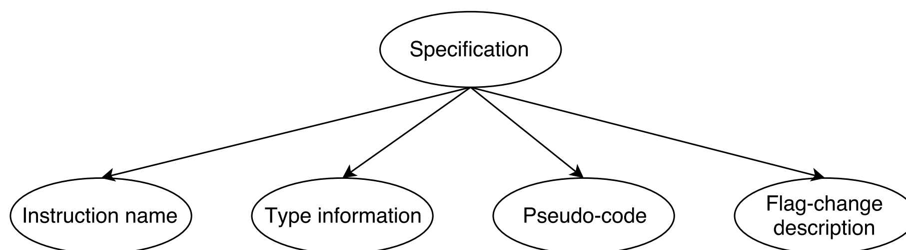

Figure 8.1: Information extraction from specification

In the implementation, **Jsoup**[1], which is a Java HTML Parser library, is used for parsing specification in HTML data structure. The necessary information is located in fixed and orderly tags in each specification. The module can easily extracts instruction name from the second *h1* tag. Type information table, pseudo-code and "flags-affected" description are acquired by parsing the first, third and forth *table* tags, respectively.

ANTLR (ANother Tool for Language Recognition) v4[2] accepts the context free grammar mentioned in the section 6.1, which is rewritten according to the syntax of the

[1]jsoup: Java HTML Parser - `https://jsoup.org/`

[2]ANTLR (ANother Tool for Language Recognition) - `http://www.antlr.org/`

tool, as the input and generates a parser for pseudo-code. Then, an abstract syntax tree (AST) is obtained by applying the parser on the acquired pseudo-code section.

An implementation of the simple boolean expression evaluator is required for pruning dead branches from the AST afterwards. The evaluator obeys short-circuit evaluation strategy, where if the expression contains more than one sub-expressions, the second sub-expression is evaluated only if the first one can not be sufficient for determining the returned value of the expression. As for the `AND` operation, if there is at least one sub-expression evaluates to `false`, the final result must be `false`. And in the case of the `OR` operation, if one or more sub-expressions evaluate to true, the `true` value is returned. The evaluator turns the variables including `PE` and `VM` into zero before evaluating. If all factor in the expression is evaluable, the result is concrete boolean value; otherwise, it is an unknown value. The module only removes the branch that gets the false conditional expression. After all, the pruned AST is traversed to apply code analysis and extract sets of variables and function calls as described in the section 6.1.

NLTK (Natural Language Toolkit)[3] in Python is used to take advantage of easy-to-use interfaces such as WordNet, lemmatization and tagging. As mentioned in the section 6.2, the module prepares the sentence templates showed in the table 6.1 and measures the similarity between each sentence in the "flags-affected" description with sentence templates (by the method described in the section 4.2) to extract affected flags and the corresponding code template that need to be applied for such flags. After processing all sentences in the "flags-affected" description, several lists including affected flags with default rules, affected flags with specified value, cleared flags, and unaffected flags are obtained and stored into an instance of *"flags-affected" data structure*.

8.2 Module Generation

This module accepts extracted information from the module extraction to automatically generate code for the binary emulation in BE-PUM. Before generating, we have to prepare prerequisite code as mentioned in the section 5.2.

- **Instruction name** (title of specification): It describes the name of the instruction that is specified in the current specification, excepts the special cases as follows:
  - *The name contains the character "/"*: It implies the specification describes semantic for several instructions in the same document, where each instruction is split by such character. In this case, the same generated code will be applied for several instructions mentioned in the title.
  - *The name ends with letters "cc"*: It implies these documents describe conditional instructions (i.e., executed if the condition is met). And the operation is described not only in pseudo-code but also in the type information table. For these documents, the **type information table** consists of several rows, in which each row mentions one specific instruction and its condition. For

[3] Natural Language Toolkit - `http://www.nltk.org/`

example, the figure 5.2 shows a part of the type information table in the document "*Jcc*". The first row in the table gives the condition for executing the instruction *JA* which is the values of *CF* and *ZF* equal to zero.

- **Abstract syntax tree**: Each node in the AST is traversed to transform pseudo-code to Java code for the binary emulation. After having traversed, the module acquires attributes (i.e., variables used in the operation), a set of conditional expressions, an initialization code for attributes (i.e., variables need to be loaded values before executing, for example, the value of the general-purpose register AL need to loaded to the attribute `AL`) and an operation code. These such information are stored into attributes of the object `data` and this module uses the object to fill code into a dynamic template. This process is implemented in Java and Velocity[4] library, which is a Java-based template engine and allows to use a simple template language to refer objects in Java. The dynamic template is defined as follows:

```
public class $data.getName() extends X86GeneratedStub {

    $data.attribute

    @Override
    protected BPState preExecute() {
        if (mConditionValueMap == null) {
            mConditionValueMap = new HashMap<>();
            #foreach( $entry in $data.formulaMap )
            mConditionFormulaMap.put($entry.getKey(),
                                        $entry.getValue());
            #end
        }
        return super.preExecute();
    }

    @Override
    protected boolean getConditionValue(int pIndex) {
        switch (pIndex) {
            #foreach( $entry in $data.conditionMap )
            case $entry.getKey():
                $entry.getValue()
            #end
        }
        if (mConditionValueMap.get(pIndex) != null)
            return mConditionValueMap.get(pIndex);
        return generatePathCond(pIndex);
```

[4]The Apache Velocity Project - `http://velocity.apache.org/`

```
    }

    @Override
    protected void initAttributes() {
        super.initAttributes();
        $data.init
    }

    @Override
    public BPState execute() {
        $data.code
        return null;
    }
}
```

By using the above template, Velocity attempts to replace all strings that has the convention `$object.attributes` with appropriate strings stored in `object`. The library also provides the syntax for looping (by the block `#foreach...#end`).

- **"flags-affected" data structure**: The instance of "flags-affected" data structure contains several lists including affected flags with default rules, affected flags with specified value, cleared flags, and unaffected flags. For each list, the module applies a relevant code template on mentioned flags.

For instance, by using the above template with extracted information from the module extraction, the generated code for the binary emulation in the case of instruction JZ is shown below:

```
public class jz extends X86GeneratedStub {
    Value Destination = new LongValue(0);
    Value CS = new LongValue(0);
    Value EIP = new LongValue(0);
    Value ZF = new LongValue(0);

    @Override
    protected BPState preExecute() {
        if (mConditionValueMap == null) {
            mConditionValueMap = new HashMap<>();

            mConditionFormulaMap.put(0,
             new HybridBooleanValue(ZF, "==", new LongValue(1L)));
        }
```

```
        return super.preExecute();
    }

    @Override
    protected boolean getConditionValue(int pIndex) {
        if (mConditionValueMap.get(pIndex) == null)
            switch (pIndex) {
                case 0:
                    if (checkIsConcreteValues(ZF))
                        mConditionValueMap.put(0,
                            BPOperation.toLong(ZF) == 1L);
            }
        if (mConditionValueMap.get(pIndex) != null)
            return mConditionValueMap.get(pIndex);
        return generatePathCond(pIndex);
    }

    @Override
    protected void initAttributes() {
        super.initAttributes();
        Destination = getValue(dest);
        CS = env.getRegister().getRegisterValue("cs");
        EIP = env.getRegister().getRegisterValue("eip");
        ZF = env.getFlag().getZFlag();
    }

    @Override
    public BPState execute() {
        mIsSF = false; mIsCF = false; mIsZF = false;
        mIsAF = false; mIsPF = false; mIsOF = false;

        if (getConditionValue(0)) {
            EIP = mBPOperation.add(EIP,
                    mFunctionCall.SignExtend(Destination));
            env.getRegister().setRegisterValue("eip", EIP);
            if (mOpSize == 16L) {
                EIP = mBPOperation.and(EIP, new LongValue(0xFFFF));
                env.getRegister().setRegisterValue("eip", EIP);
            } else {
                if ((BPOperation.toLong(EIP) < CodeSection.Base) ||
                    (BPOperation.toLong(EIP) > CodeSection.Limit)) {
                    return exceptionSEH();
                }
```

```
            }
        }

        return null;
    }
}
```

8.3 Module Testing

In order to verify automatically generated code, the module testing is conducted by two steps:

- **Test program generation**: As mentioned in the section , in order to reach the test coverage on operation of x86 instructions, which consists of several branches, test programs are automatically generated from the type information table and the pruned AST. The descriptions for valid operands are described in the column *Mnemonic* of the table. Based on the convention shown in the table 5.1 of the section 5.1.2, this module replaces symbols with valid operand values and acquires an assembly statement, which is written to an assembly code file. Netwide Assembler (NASM) [5], which is an assembler for the Intel x86 architecture, is used to compile the assembly code file to an executable file as a test program.

  In the case of operation contains various conditional statements with conditions formed from elements of environment, the statistic shows that most of the specifications do not update the environment before executing conditional statements. For conditional expressions with modification, we manually add more test-cases to satisfy test coverage. Concerning conditional expressions without modification, the system extracts conditional expressions and generates combinations of them in both cases being true and false. Then, the concolic-testing are carried out by Z3 [6], which is a theorem prover from Microsoft Research, to acquire concrete values. The figure 8.2 demonstrates this process for the pseudo-code of the instruction AAA (see 7.3).

- **Comparison of environments in BE-PUM and Debugger**: This step compares a pair of environments before and after executing an instruction in a debugger with another pair in BE-PUM (illustrated by the figure 7.5). In the implementation, we pick the x64dbg debugger [7], which is an open-source x64/x32 debugger for Windows OS, due to the scriptable feature that provides an integrated scripting language in Python. A script in Python is written to extract automatically environments on the real machine for all generated test programs. Thereafter, these such

[5] Netwide Assembler - `http://www.nasm.us/`

[6] Z3 - `https://github.com/Z3Prover/z3/`

[7] x64dbg - `https://x64dbg.com`

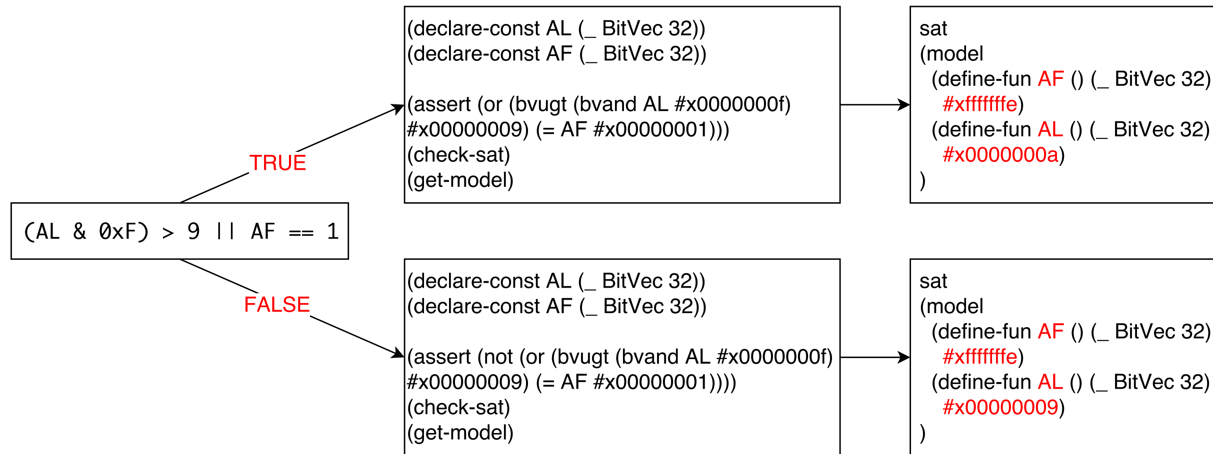

Figure 8.2: An example of concolic testing on conditional expressions in pseudo-code (see 7.3)

information are used for comparison with the implementation of generated code of the binary emulation in BE-PUM.

Chapter 9

Experiment

9.1 Successfully Generated x86 Instruction

As mentioned in the section 5.2, the system requires some manual beforehand preparation as the prerequisites supporting the automatic code generation for x86 binary emulation in BE-PUM. Therefore, we manually implemented default rule-based flag-change modifications and 30 undefined functions frequently used in pseudo-code. Finally, among the specifications for 530 x86 instruction that have been collected, **the system successfully generated code for binary emulation in 299 instructions** including the groups of the following instructions:

- Arithmetic instructions, e.g., ADD, SUB, MUL, INC
- Data instructions, e.g., MOV, BSWAP, CMOVC
- Logical instructions, e.g., AND, OR, XOR, NOT
- Control instructions, e.g., JMP, JA, JZ
- Flag-control instructions, e.g., STC, CMC, CLD
- Partly FPU instructions (which work with floating-point unit registers and perform floating-point arithmetic), e.g., FADD, FSUB, FMUL
- Partly MMX instructions (which work with MMX registers), e.g. MOVD, PADDQ, PSUBB

The failed instructions belongs to the following groups and are explained in the section 9.3.

- Architecture dependence, e.g.,FSUBR, FSTCW
- Unsupported functions in BE-PUM, e.g., FLDENV, CVTDQ2PD

9.2 Comparison with Manual Implementation

Beforetime the research is conducted, BE-PUM officially supported about 250 x86 instructions after 3-years human effort. These supports were implemented manually and have several problems due to the limitation of writing test-case and test-coverage. In comparison with the manual implementation, our system excels in the following points:

- The larger number of supported instructions (better than the manual implementation 49 instructions) in the shorter time period of implementation.
- The system points out errors in the manual implementation:
  - The preprocessor JakStab of BE-PUM incorrectly disassembles the instruction IMUL (in the case of one operand) and the instructions CWD/CDQ, where both of them are disassembled into the same instruction.
  - The implementation of the function `MSB` (i.e., calculating the most significant bit) might return unexpected value in the case of inputting 32-bit argument and affect to 8 instructions including ROR, ROL, RCL, RCR, SAL, SAR, SHL, SHR.
  - The implementation of the method `evaluateAddress` wrongly evaluates the memory address which consists of segment register and might lead to the incorrectness for instructions which reference memory.
- The system can be modified to adapt to the specification for instructions in other architecture and help BE-PUM to expand abilities in analyzing several platforms beyond x86.

9.3 Discussion of Failed Cases in Automatic Generation

Our system fails in automatically generating code for the remaining 131 instructions. There are two reasons causing this problem:

- **Due to the lack of support in BE-PUM** for the architectures that are rarely used such as floating-point arithmetic (with FPU register stack, XMM register, etc.), the system can not be configured to generate proper code statements being compatible existing implementation in BE-PUM. Moreover, it does not have appropriate environment to be executed and perform testing. The tasks of supporting these such architectures are too difficult for automation because they require certain technical know-how in computer architecture and are described in several kinds of documents. For instance, the figure 9.1 shows the pseudo-code section of the specification for the instruction FLDENV, which consists of variables (e.g., `FPUConstrolWord`, `FPUStatusWord`, `FPUTagWord`) that are available only in the partially unsupported architecture FPU register stack.

Operation

```
FPUControlWord = Source.FPUControlWord;
FPUStatusWord = Source.FPUStatusWord;
FPUTagWord = Source.FPUTagWord;
FPUDataPointer = Source.FPUDataPointer;
FPUInstructionPointer = Source.FPUInstructionPointer;
FPULastInstructionOpcode = Source.FPULastInstructionOpcode;
```

Figure 9.1: An example of unsupported architecture in the specification for FLDENV

- **Pseudo-code invokes an unsupported function** that need to be manually prepared beforehand (see 5.2). At the present, the system tries to minimize the hand-operated implementation by only manually preparing functions commonly used in the entire manual. It turns out that the functions seldom appearing in the manual are missing and the generated code can not invoke them. For instance, the pseudo-code, which describes the operation for the instruction FBSTP, in the figure 9.2 calls the unprepared function `BCD`.

Operation

```
Destination = BCD(ST(0));
PopRegisterStack();
```

Figure 9.2: An example of unsupported function call in the specification for FBSTP

Due to the causes of fail cases in automatic generation in the above section, the system will be improved if we eliminate the mentioned limitations. The task of implementing lacking architectures requires human efforts because of its difficulty. And besides, to ensure the precision and compatibility of new architectures, the implementation ought to be revised carefully and it is not one of repetitive works that can be handled by automation. The simple idea for supporting new architectures without manual implementation is to make use of an existing emulator to input arguments and receive returned values. From our observation, instructions working on such architectures can affect to path condition. Thus, this idea is ineffective and may cause shortcomings during disassembling.

As for unsupported functions, there are about 40 functions need to be added to the implementation of our system. Among them, there are about 10 functions that are designed to convert data type (e.g., `ConvertDoubleToFloat`, `ConvertDoubleToInteger`). These such functions accept the same parameter type and convention. Hence, their implementations may be automatically generated. The remaining functions require several practical knowledges and are specified in separated documents. Therefore, the required tasks for improvement need to be investigated further and are left for future work.

Chapter 10

Related Work and Conclusion

10.1 Related Work

Writing computer programs from using natural language description is a challenging problem. The advantage of using natural language processing to automatically generate programs is to ease human effort in manual works having the characteristic of repetitiveness. Besides, it may allow newcomer to utilize existing systems to apply them for similar tasks. In natural language processing, the process for semantic analysis mapping natural language sentences into a formal representation is semantic parsing. Recently, there are several works in semantic parsing, which have exploited natural language sentences to address problems in domain specific applications, such as recipe from description [18], robot commands [6], operating system [1], scripts in smart-phone [13], spreadsheet data analysis and manipulation [3]. In order to develop such language-to-code systems, efforts for manually constructing parsers or large corpus of appropriate training samples are generally required.

The approach proposed by Quirl et al [18] from Microsoft Research learns to map simple natural language descriptions (i.e., written in the form of "if-then") to to naturally-happening code statements (called "recipes"). Recipes are simple rules that allow users to control smart electric devices. There are several services and applications, such as Tasker and IFTTT [1], provides graphic user interface to support users to create uncomplicated programs with trigger and actions. For instance, an air-conditioner can be programmed to be operated only in a certain period of time from 8 AM to 6 PM. This approach is performed and evaluated on the corpus of pairs of recipe (which is thereafter extracted to an abstract syntax tree representation) and description collected from the website of IFTTT. It utilizes a probabilistic log-linear classifier with character and word n-gram features in machine learning to build a correlation between queries and recipes.

NLyze [3] is a implementation of an Excel add-in using the method proposed by Gulwani et al. Spreadsheet systems (e.g., Microsoft Excel) allows users to program scripts/macros using a built-in library that works on string and numeral data. However, inexperienced end users may find it too complicated to write applications as their wishes. The add-in

[1] If This Then That - `https://ifttt.com/`

supports users to automate popular tasks by stating natural language requirements. It makes use of a rule-based translation algorithm for converting a description in natural language with the context of a given spreadsheet into a set of type domain-specific language (DSL) programs. The method is dedicated to perform on spreadsheet with tasks being the combination of actions predefined in the DSL.

Nowadays, people tend to use their electric devices in most of daily-life activities. In order to automate and program such activities, general-purpose programming languages such as Java, C#, Objective C are not suitable for most of end users. Instead, an end user can interact with an assistant system (e.g., Google Assistant, Siri, Cotana) through natural language to command mobile devices. Beyond limited common instructions, SmartSynth [13] can synthesize smartphone automation scripts by specifying description in natural language. Firstly, it extracts the set of components occuring in the description and their incomplete dataflow relations by NLP techniques including regular expression, bags-of-words [5], parse tree [10], etc. Then, rule-based relation detection algorithms and type-based synthesis techniques are applied to discover missing dataflow relations.

As for the most related work, Le Vinh [12], our former colleague, has conducted his Master's thesis of automatic Windows API stub generation from natural language description. His thesis contributed to support a large number of API stubs in BE-PUM, which overcomes the major cause of the unexpected termination. His research aims to avoid ambiguity in Windows API specification. Therefore, semantic parsing has been applied to deal with this situation and find out the correct representation of input data. Bayesian learning and sentence similarity are applied to extract and disambiguate specifications about data types of parameters.

Inspired by these above works and based on our observation, x86 instruction specifications have particular features, such as the type convention specified by Intel, the pseudo-code and the "flag-affected" description. Hence, we address the problem by building a parser to take advantage of pseudo-code and applying simple techniques in NLP to extract appropriate actions for text sentences in "flag-affected" descriptions. However, there are still several ambiguous parts in the pseudo-code which cannot be overcome. By manually preparing beforehand prerequisites, the approach can achieve a good efficiency with the minimum of human effort.

10.2 Conclusion and Future Work

This thesis presents our study on automatic generation of dynamic symbolic execution of x86 instructions from their natural language specification. The implementation target is BE-PUM. By using the combination of the parser and simple techniques in natural language processing, we successfully generate dynamic symbolic execution covering 299 instructions among 530 specifications collected from Intel Software Developer's Manual. In addition to the current result, the system is highly expected to be adapted to work on other specifications of different architectures/platforms by small modifications.

We aim to reduce human efforts by automation on an easier part. Then human can concentrate on real difficulties. From this view, we hope to extend the methodology to

platforms other than x86, e.g., ARM. Beyond x86 architecture, we would like to expand the restriction of BE-PUM in supported platforms. According to Gartner, nowadays there are over 6 billion IoT (Internet of Things) devices in the world. Up to May 2017, Kaspersky Lab collected several thousand individual IoT malware samples, in which a half of them were found in 2017 (figure 10.1) [2]. Besides, a case study [3] conducted by HP in 2015 claimed that 100 percents of IoT home security devices have critical vulnerabilities. Therefore, IoT device firmwares currently become one of the significant and attractive targets. We expect the current system can be modified to generate code for the binary emulation that can deal with other platforms in BE-PUM.

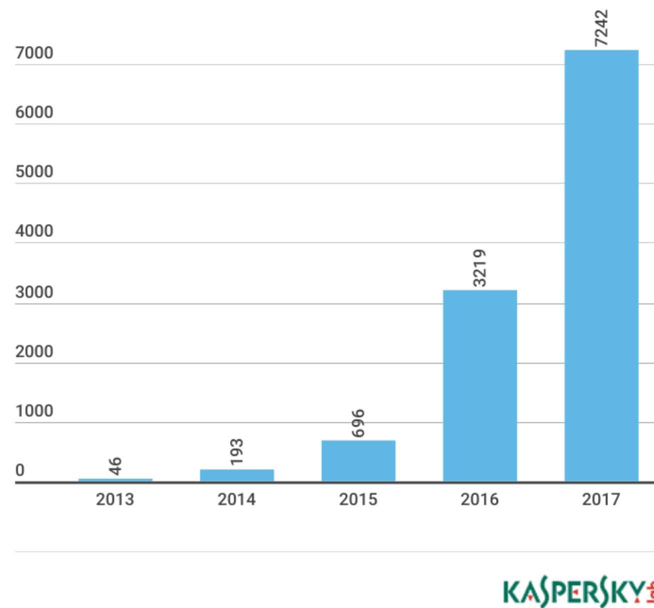

Figure 10.1: The number of IoT malware samples detected each year (2013 - 2017)

[2] https://securelist.com/honeypots-and-the-internet-of-things/78751/
[3] http://files.asset.microfocus.com/4aa5-4759/en/4aa5-4759.pdf

Bibliography

[1] S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 82–90, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[2] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[3] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 803–814, New York, NY, USA, 2014. ACM.

[4] Andreas Holzer, Johannes Kinder, and Helmut Veith. *Using Verification Technology to Specify and Detect Malware*, pages 497–504. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[6] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. Learning to transform natural to formal languages. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pages 1062–1068. AAAI Press, 2005.

[7] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4):424–438, Oct 2010.

[8] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. *Detecting Malicious Code by Model Checking*, pages 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[9] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[10] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[11] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[12] Vinh Le. Automatic stub generation from natural language description. September 2016.

[13] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 193–206, New York, NY, USA, 2013. ACM.

[14] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 775–780. AAAI Press, 2006.

[15] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[16] Nguyen Minh Hai, Mizuhito Ogawa, and Tho Quan. Obfuscation code localization based on cfg generation of malware. volume 9482, pages 229–247, 02 2016.

[17] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 878–888, Beijing, China, July 2015.

[19] Fu Song and Tayssir Touili. Pommade: Pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 607–610, New York, NY, USA, 2013. ACM.

[20] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32Nd Annual Meeting on Association for Computational Linguistics*, ACL '94, pages 133–138, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.