Master's Thesis

Generating a Dynamic Symbolic Execution Tool from MIPS Specifications

1710459 Quang Thinh Trac

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

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

August, 2019

Abstract

Nowadays, malware has evolved and caused serious consequences to society and business. Hence, malware analysis has been receiving a great deal of attention in both academic community and industries to detect and prevent malware dangerous impacts. However, it is a challenge since modern malware uses many obfuscations techniques (e.g. indirect jumps or self-modifying loops) to hide its behaviors and guard against the detection of antivirus software. As a result, various methods have been proposed for analyzing the real behaviors of malware. Unlike other methods are easily fooled by obfuscation techniques, Control Flow Graph (CFG) based approaches (e.g. VxClass) can overcome this problem. A well-known extension of Symbolic Execution as Dynamic Symbolic Execution (or concolic testing) has been widely used to reconstruct CFGs of malware. Dynamic Symbolic Execution consists of both concrete and symbolic execution. It is capable of exploring all feasible execution paths and dealing with obfuscation techniques like indirect jump. In recent years, we have introduced BEPUM (Binary Emulation for PUshdown Model) and Corana, two Dynamic Symbolic Execution tools for x86 and ARM Cortex M, respectively. Following these studies, our goal is expanding this idea for a new architecture - MIPS.

Similar to ARM architecture, MIPS is a RISC (Reduced Instruction Set Computer) instruction set architecture, which is one of the popular architectures of IoT devices. Since the number of IoT devices has been growing rapidly, it leads to the dramatic increase of IoT malware infection rates. Although IoT malware does not contain sophisticated obfuscation techniques, dealing with indirect jumps is necessary to reconstruct precise CFGs of malware. Afterward, generated CFGs can be used for further detection and classification analysis. Although the number instructions of MIPS is quite smaller than others such as x86 or ARM, it is beneficial to propose a systematical method to implement Dynamic Symbolic Execution for further similar studies to reduce human efforts on implementations.

This thesis proposes a semi-automatic extraction of the formal semantics of MIPS architecture from the pseudocode description in MIPS instruction manual. Among 127 collected instructions, we focus on the 63 instructions of the CPU category. After manually preparing 21 primitive functions in the pseudocode description, their semantics are successfully generated as Java methods, which are unified as a dynamic symbolic execution tool SyMIPS. We perform an empirical experiment on 3219 MIPS32 IoT malware collected from ViruSign and observe that SyMIPS successfully traces 2725 samples. The rest is interrupted by either system calls or out of memory error. Although the current implementation is preliminary, SyMIPS finds the destinations of indirect jumps by concolic testing and

discovers dead conditional branches in some samples.

Keywords: Dynamic Symbolic Execution - IoT Malware - MIPS32.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Malware is the shorthand term of malicious software. It is developed with the intention of causing damages or accessing a system or a network without the approval of the owners. Like lawful software, malware has evolved over the decades and come armed with various functions depending on the goals of the attackers. There are a large number of different forms of malicious software such as computer viruses, trojan horses, ransomware, spyware, adware. Because of their harmful impacts and the rapid infection rate, it is essential to do research on detecting, classifying and then preventing the affection of malicious code into our systems. However, it is a challenge for the reason that almost malware are created by using many obfuscation techniques to mask its actual behaviors and also guard against the detection of antivirus software. In order to unmask its harmful behaviors, several methods were proposed for analyzing binary malware including static analysis, dynamic analysis and also model checking based approaches. Unlike other methods that are becoming misleading without difficulty by obfuscation techniques, model checking based approaches reveal malware's original behaviors by analyzing binary files to obtain its abstract models. From these models, we conduct further experiments to investigate malware and find out effective ways of preventing malicious codes. In recent years, we have proposed BEPUM (Binary Emulation for PUshdownModel)[12] with the aim of building up the CFG of malware under the presence of obfuscation techniques for x86 architecture. Furthermore, a Dynamic Symbolic Execution - Corana[17] was proposed for ARM Cortex M architecture. This study applies natural language processing techniques to automatically extract ARM formal semantics from its natural language specifications and utilizes generated methods to develop Corana. Following these studies, our goal is expanding this idea for a new architecture - MIPS.

MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture developed by MIPS Technologies. Originally, MIPS has been used for general-purpose computer but recently, it has been used for embedded systems such as routers and residential gateways. Because of its applications, almost current MIPS malware is IoT malware. According to McAfee[1], from June to August-2018, McAfee detected more than 450 IoT malware threats every minute. Furthermore, the report informed that new malware samples grew by 53 percents. The total number of IoT malware had been grown up dramatically about 200 percent over the previous four quarters in August 2018, which provides the evidence that IoT devices are new targets for attackers. Although IoT malware does not contain sophisticated obfuscation techniques such as self-decryption loops, studying indirect jumps plays an important role with the aim of understanding malware behaviors by using Dynamic Symbolic Execution to reconstruct its CFG. Developing a Dynamic Symbolic Execution tool requires a binary emulator and path constraints generation. Although the number instructions of MIPS is quite smaller than others such as x86 or ARM, it is beneficial to propose a systematical method for further similar studies to reduce human efforts on implementations.

Hence, the aim of this study is proposing a semi-automatic extraction of the formal semantics of MIPS architecture from the pseudocode description in MIPS instruction manual and then utilizing generated methods to developing a Dynamic Symbolic Execution tool - SyMIPS. Furthermore, we perform an empirical experiment on IoT malware to trace its behaviors.

1.2 Contribution

This study applies a semi-automatic extraction of the formal semantics on the MIPS architecture, similarly to x86 [11] and ARM [17]. The extracted semantics is unified as a dynamic symbolic execution tool SyMIPS on MIPS. Among variations of the MIPS architecture such as MIPS I-V and MIPS32/64, we focus on MIPS32 (release 5) of which the specifications are available in MIPS32 instruction set manual [2]. Among 127 collected MIPS32 instructions, we focus on the 63 instructions of the CPU category. After manually preparing 21 primitive functions in the pseudocode description, their semantics are successfully generated as Java methods. We perform an empirical experiment on 3129 MIPS32 IoT malware collected from ViruSign and observe that SyMIPS successfully traces

[1]https://www.iottechnews.com/news/2018/dec/20/mcafee-new-iot-malware-variants-minute/

[2]https://www.mips.com/products/architectures/mips32-2

2725 samples. The rest is interrupted by either system calls or the stack overflow. Although the current implementation is preliminary, SyMIPS finds the destinations of indirect jumps by concolic testing and discovers dead conditional branches in some samples. The current version of SyMIPS is downloadable at `https://github.com/tracquangthinh/SyMIPS`.

1.3 Thesis Structure

This thesis is organized as follows:

- Chapter 2 explains the overview of MIPS architecture and IoT malware analysis techniques.
- Chapter 3 introduces the formal semantics of ARM, which including both operational semantics and Java semantics.
- Chapter 4 briefly introduces the process of MIPS specification extraction.
- Chapter 5 introduces a conformance testing method to validate the correctness of our implementation.
- Chapter 6 presents SyMIPS - a Dynamic Symbolic Execution tool by utilizing Java generated methods.
- Chapter 7 shows the results of the practical experiment including semantics extraction and SyMIPS executions on IoT malware.
- Finally, Chapter 8 summaries the contribution of this study and proposes some directions to improve and extend our current method.

1.4 Extraction Overview

Fig. 1.1 describes the overview of our proposed method. The specification of instructions is parsed to abstract syntax trees by defining the context-free grammar of pseudocode and then, Java methods are generated by applying depth-first search algorithm. When an instruction is executed, the environment of the binary emulator is updated and the path conditions are generated. Finally, the correctness of generated methods is validated by conformance testing.

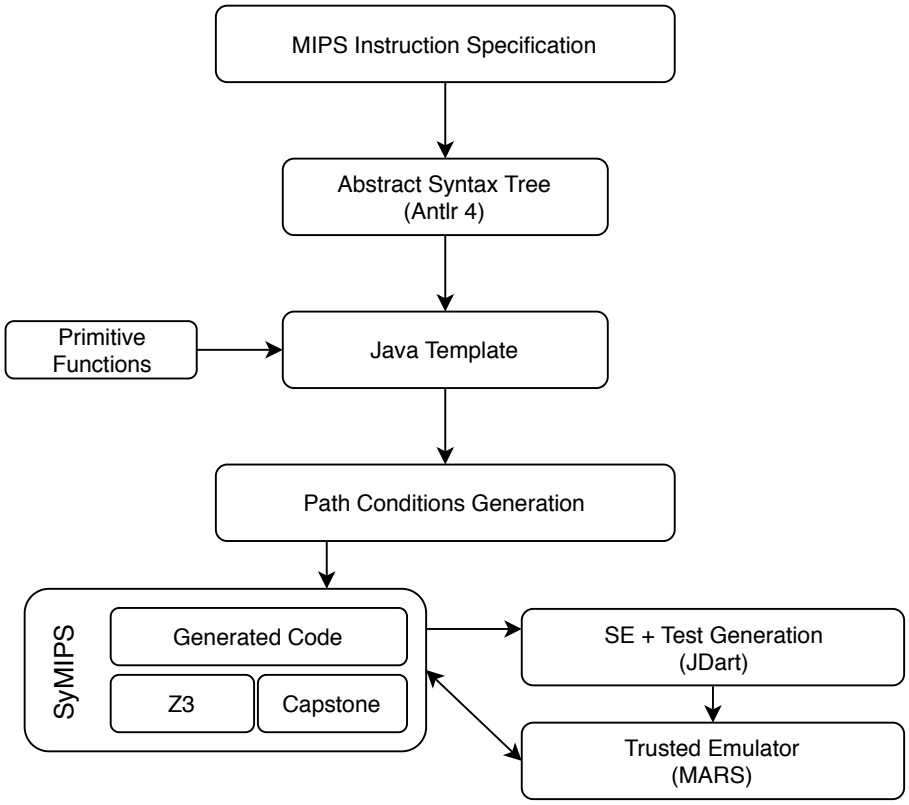

Figure 1.1: The overview of the proposed method

Chapter 2

Related Work

2.1 Binary Symbolic Execution Tools

Although developed software have been tested carefully by humans before releasing them to their customer, they still have several potential bugs and security vulnerabilities. Recently, various methods have been proposed by applying symbolic execution to analyze programs, especially for binary code such as McVeto[16], MiAsm[6], CoDisasm[2], BE-PUM[12], Mayhem[5], KLEE-MC[4], Angr[15], Corana[17]. Most of them are developed for x86 architecture except Corana is for ARM. The first essential tasks for developing binary analyzers is extracting formalizing the semantics of the instructions, which is given often in the manual for human or appear implicitly among the tools like debuggers and disassemblers.

The implementation of Binary Symbolic Execution tools is classified by two approaches.

- By using the existing disassemblers, e.g., CAPSTONE[1], binary code is translated to an intermediate machine language, e.g., LLVM in KLEE-MC, VEX in Angr.
- Directly interpreting the binary code, e.g., McVeto and BE-PUM for x86, and Corana for ARM.

BE-PUM (Binary Emulation for PUshdown Model) is a binary code analyzer for x86 architecture. By applying dynamic symbolic execution, BE-PUM inputs binary files to generate control flow graphs under the presence of obfuscation techniques like indirect jump or overlapping instructions. It consists of three main

[1] http://www.capstone-engine.org

components including a binary emulator, a CFG storage and a symbolic executor. Fig. 2.1 shows the architecture of BE-PUM.

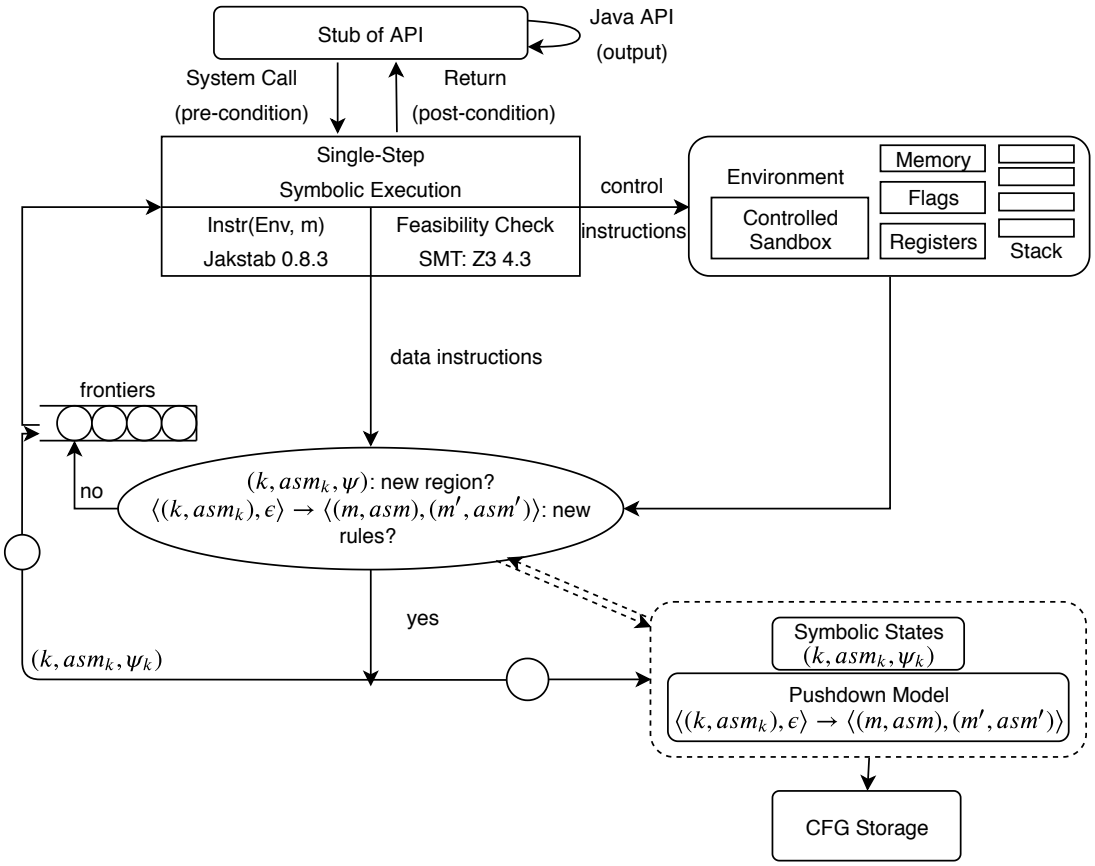

Figure 2.1: BE-PUM architecture

Furthermore, we have proposed a method to systematically extract the formal semantics of ARM instructions from their natural language specifications. It consists of the semantics interpretation by applying translation rules, augmented by the sentences similarity analysis to recognise the modification of flags. Afterward, Corana - a Dynamic Symbolic Execution tool for Cortex-M was built by utilising extracted instructions. Fig. 2.2 shows the architecture of Corana.

2.2 Formal Binary Semantics Extraction

Although the specifications are divergent, they are classified to

- Either natural language description only (e.g., ARM), or with pseudocode

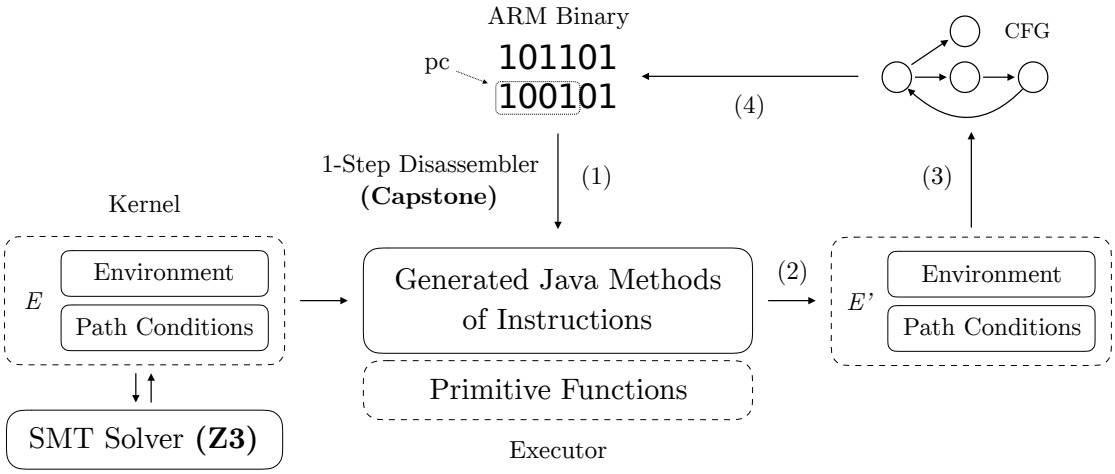

Figure 2.2: Corana architecture

description (e.g., x86, MIPS). The latter obeys a quite unified grammar and much easier to handle by manually defining primitive functions.

- Either in html (e.g., x86, ARM Cortex-M). or in pdf (e.g., MIPS, ARM Cortex-A/R), the latter requires the pdf to text translation, which may lose the information figured in tables.

Their semantics framework basically consists of the transition systems over the quadruplets of registers, flags, memory and the stack. Note that since most of malware is a sequential user-mode process and we avoid the concurrency, the weak memory model, and co-processor operations. Still, there are some variations. For instance, MIPS does not have flags (instead the condition is stored in a register and cache operations explicitly. Due to these differences, it is required to produce an unprecedented approach to extract the formal semantics of MIPS architecture.

The first trial to automatically extract the formal semantics appears in x86 [11] for extending BE-PUM, in which the flag updates are recognized by the similar analysis in natural language processing and the operation of instruction is directly obtained from the pseudocode description. The experiment shows that among 530 collected specifications from Intel Developer's Manual [2], Java method descriptions of 299 x86 instructions are successfully generated by manually preparing 30 primitive functions in the pseudocode.

The extraction of the semantic of ARM instructions [17] is more challenging, since the ARM specification is described only in English. By manually prepar-

[2]https://www.felixcloutier.com/x86

ing 228 semantics interpretation rules that rewrite a noun phrase to a Java code fragment, the experiment shows that among 1039 collected ARM Cortex-M specifications from ARM manual[3], the semantics of the 662 instructions are successfully extracted. Note that they apply the conformance testing by comparing the execution results of instructions between generated Java methods and the existing emulators, e.g., Ollydbg [4] for x86 and μVision [5] for ARM. Fig. 2.3 illustrates the extraction approach of Corana.

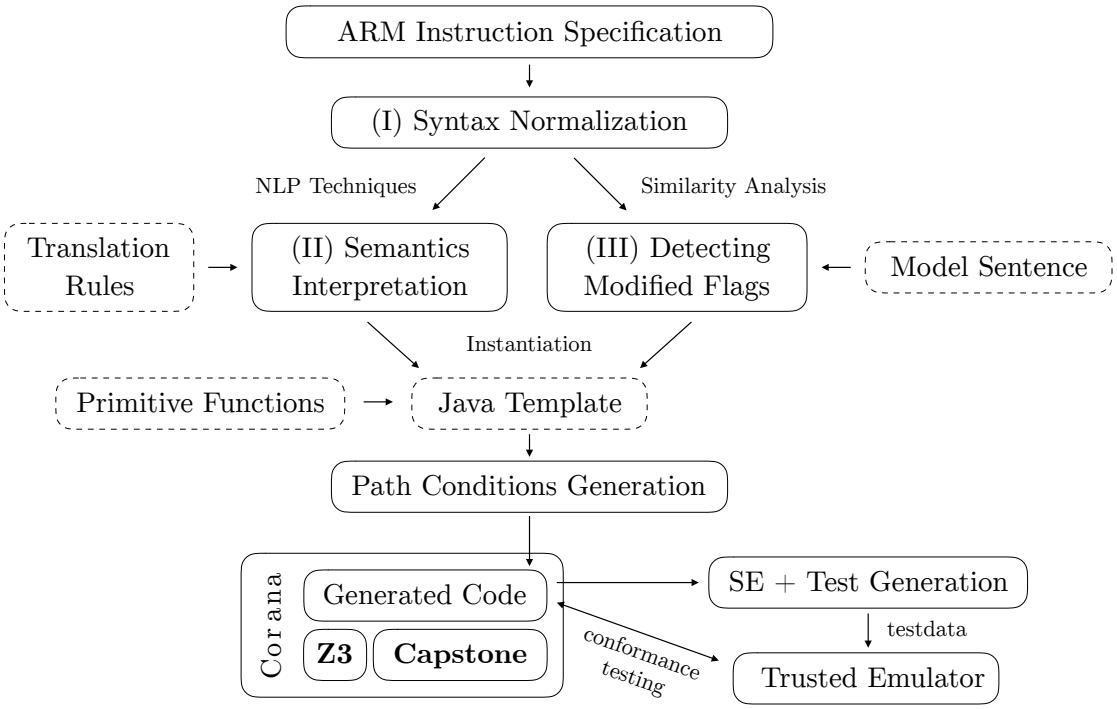

Figure 2.3: The overview of the extraction approach of Corana

[3]https://developer.arm.com
[4]http://www.ollydbg.de
[5]http://keil.com/mdk5/uvision

Chapter 3

Symbolic Execution Techniques

The most disadvantages of common malware analysis techniques are that it can not be executed successfully in the presence of obfuscation techniques. To overcome this problem, Dynamic Symbolic Execution technique was proposed to explore the Control Flow Graph of malware binaries. As the above-mentioned definition, Control Flow Graph is a directed graph in which its nodes represent basic blocks and its vertices represent control flow paths. A basic block is a straight-line instructions sequence with no branches in except to its entry and no branches out except at its exit. Control Flow Graph is used to capture the behavior and the program structure of malware. In the example 3.1, a Control Flow Graph contains four basic blocks as four nodes. The path between node a_1 and node a_4 may represent a condition statement while another path between block a_2 and a_3 may represent a loop statement.

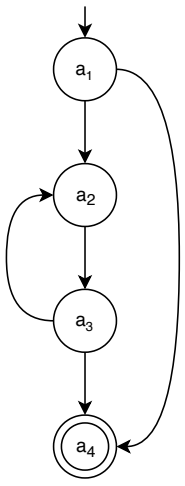

Figure 3.1: An example of Control Flow Graph

3.1 Symbolic Execution

Symbolic Execution[9] is a well-known program analysis technique to test whether violated properties appear in some locations of software. These properties may be division by zero, NULL pointer or backdoor exists. In a concrete execution, a specific input is used to execute the program and only one control flow path is obtained. In contrast, symbolic execution can simultaneously explore multiple paths of a program under different inputs. Instead of executing concrete values, input variables are considered as symbolic variables like φ or ψ. The path condition will be updated by adding constraints at each location of conditional branches.

Fig. 3.2 shows an example of generating test cases by applying symbolic execution method. Assuming that `bar`, which has two parameters including `x` and `y`, is the function needed to test. Furthermore, this function has a potential error `ERROR`. During generating test cases of this function, the error can be detected by symbolic execution.

```
public int bar(int a, int b){
    if(a == b){
        return 0;
    } else {
        if(2*a+b > 2019){
            if(3*b + 10 < 100){
                ERROR();
            } else {
                b = b + 2018;
            }
        } else {
            a = a + 2020;
        }
    }
    return a;
}
```

Figure 3.2: An example of a testing function

Let assuming that α, β are symbolic values representing for two parameters `x` and `y`. The execution tree of this function is described in Fig. 3.3. As a result, from four path conditions in Fig. 3.3, the input and output of four respective test cases are obtained by using Z3 to check the satisfiability of these conditions.

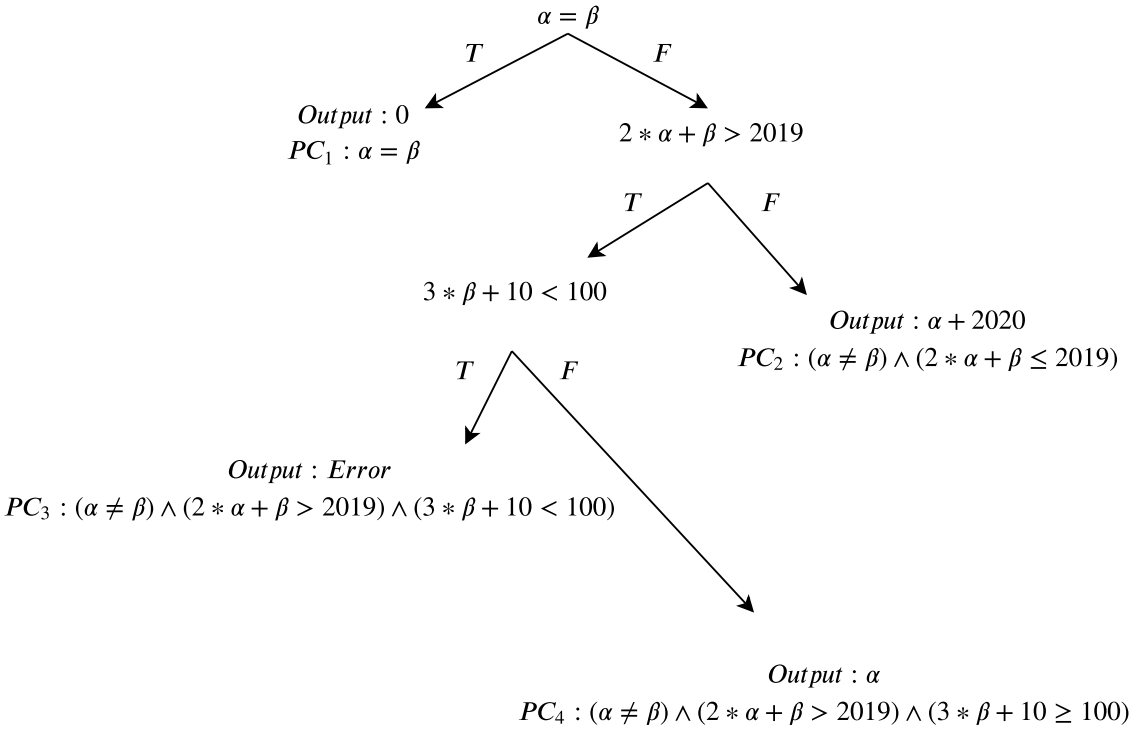

Figure 3.3: The execution tree of the function `bar`

- From PC_1, we have $\langle \texttt{Input} = (\texttt{x} = 0, \texttt{y} = 0), \texttt{Output} = 0 \rangle$
- From PC_2, we have $\langle \texttt{Input} = (\texttt{x} = 0, \texttt{y} = 1), \texttt{Output} = 2020 \rangle$
- From PC_3, we have $\langle \texttt{Input} = (\texttt{x} = 1010, \texttt{y} = 0), \texttt{Output} = \texttt{Error} \rangle$
- From PC_4, we have $\langle \texttt{Input} = (\texttt{x} = 673, \texttt{y} = 674), \texttt{Output} = 673 \rangle$

3.2 Dynamic Symbolic Execution

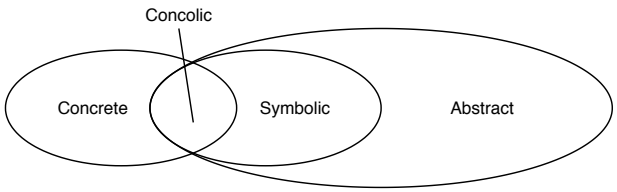

Figure 3.4: Concrete and Symbolic Execution

Although Symbolic Execution has been using in many areas to verify the correctness of programs for a long time, it still has limitations. As can be observed, when the path conditions become sophisticated e.g. non-linear equations or black-box functions, the time explosion of the theorem solver maybe happened and the feasibility can not be decided. To overcome this limitation, Dynamic Symbolic Execution or concolic testing [7] had been proposed by Patrice Godefroid and his colleagues. It is an extension of symbolic execution by using concrete execution to drive symbolic execution. In similar to static symbolic execution, concolic testing stores symbolic values and path conditions but it also keeps concrete values. After beginning with an arbitrary input, it executes the program both concretely and symbolically by simultaneously updating the concrete - symbolic values and the path constraints. This approach can overcome the mentioned above problem. Dynamic Symbolic Execution is also necessary to deal with indirect jump while the jump destination is an expression of symbolic values that need to be calculated concretely. Fig. 3.5 illustrates an example of a testing function, which is difficult to deal with Symbolic Execution. In this example, `complex` is a black-box function such as cryptographic function, non-linear integer, floating-point arithmetic, or calls to kernel-mode function. Hence, it is hard to generate values for `a` and `b` that satisfies `a = complex(b)`.

```
public int foo(int a, int b){
    if(a == complex(b)){
        ERROR();
    }
    return 0;
}
```

Figure 3.5: An example of a difficult analysis function

However, Dynamic Symbolic Execution can solve this problem as follows:

1. Start with random values of a and b: a = 10, b = 43.
2. Execute both concretely and symbolically.
   - Concrete value: `if(10 == 257)`.
   - Symbolic value: `if(a == complex(b))`.

   Because the symbolic constraint is too complex, we simplify it by using the concrete value of `b`: `if(a == 257)`. After solving this constraint, we have a new test case `a = 257, b = 43`.
3. Go back to step 1 with the new test case until all branches are exectuted.

3.3 Existing Symbolic Execution Tools

Recently, Symbolic Execution tools for binary have been receiving a great deal of attention. Comparing to binary Symbolic Execution, Symbolic Execution tools for high-level programming language have a longer history when several tools have been proposed since 2000s, e.g. jCute[14], JPF-Symbc[13], JDart[10], Acteve[1] for Java, CREST[3] for C.

For Java Symbolic Execution tools, although it has been introduced by using various techniques, there are two key mechanisms including:

- Bytecode instruction factory.
- Attributes associated with the program state.

These tools analyse Java source code by interpreting its bytecodes in a virtual machine. Afterward, the instruction factory allows extending standard concrete execution with symbolic execution. For instance, JPF-Symbc and JDart are developed based on JavaPathFinder(JPF)[8] as bytecode instruction factory while Soot[1] are the instrumentation of jCute and Acteve. Furthermore, the symbolic information is stored in attributes associated with program data (fields, stack operands and local variables). The approach for C is expressed in a different way since Symbolic Execution is executed from source code rather than bytecodes like Java. For example, CREST uses concolic testing in source code to generate test cases.

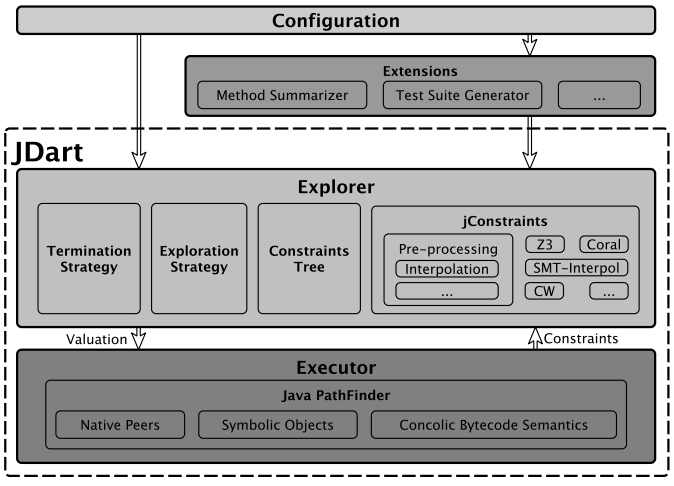

Figure 3.6: The architecture of JDart

[1]https://github.com/Sable/soot

In this study, we use JDart to generate test cases for conformance testing. The key distinguishing feature of JDart is its modular architecture. Hence, it is easily extensible and configurable and also can be used as a component within other tools. Fig. 3.6 illustrates the modular architecture of JDart. It has two main modulars including Executor for executing the analyzed program and recording symbolic constraints, Explorer for organizing recorded path constraints to a constraint tree and deciding the next execution.

Chapter 4

MIPS Formal Semantics

Formal Semantics is a fundamental aspect of binary analysis methods. In recent years, several studies were proposed with the implementation of formal semantics such as CoDisasm, BE-PUM, KLEE-MC, MiAsm, McVeto. Unlike other architecture, MIPS does not use flags. Hence, the implementation of MIPS semantics consists of three components including registers, memory, and stack instead of four as usual. The execution of an instruction is considered as the transition of the triplets of registers, memory, and stack. Furthermore, because almost IoT malware is a sequential user-mode process, we currently bypass the existence of float points, co-processor, and privileged instructions to simplify the implementation.

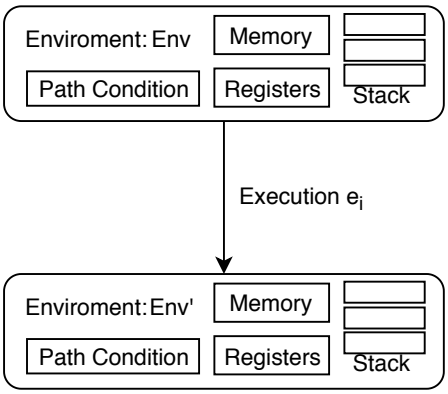

Figure 4.1: MIPS semantics transition

4.1 MIPS Architecture

4.1.1 Overview

MIPS is a RISC instruction set architecture, which was first introduced in 1985. Although there are exists multiple versions and extension, all of them are being followed strictly by MIPS architecture requirements. In a similar way to ARM, MIPS is a load/store architecture (or known as register-register architecture). It means that all instructions operate only on registers, except for the load and store instructions are used for accessing the memory. A conventional MIPS processor contains the following main components:

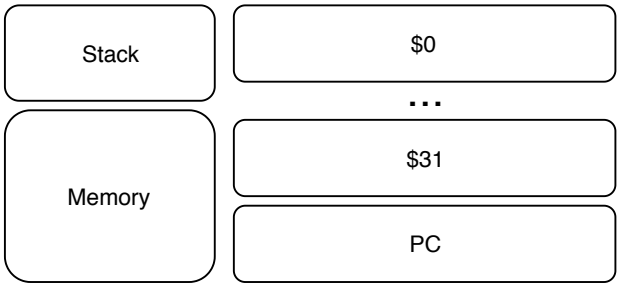

Figure 4.2: The main components of MIPS processors

1. **Registers**: is a small set of high-speed storage cells inside the CPU. MIPS provides 32 general-purpose registers which named from $0 to $31 and PC register:
    - Register $0 is hardwired to zero and instructions could not be able to write to it.
    - Register $29 is the stack pointer register which contains the address of the top value of the stack.
    - Register $31 is the link register where the functions return its values.
    - Register PC is the program counter register that holds the next executed instruction.
2. **Memory**: is a physical device capable of storing information temporarily.
3. **Stack**: is a special religion of the memory that stores temporary information created by functions.

In contrast to the x86 and ARM architectures, MIPS does not have flags and flags-sensitive instructions. Instead, it uses general registers for storing the Boolean values of conditions. Furthermore, the MIPS instructions except for the load/store instructions, `lb`, `sb`, `lw`, `sw`, cannot access memory directly

4.1.2 Instruction Format

MIPS instructions are divided into three types: R, I and J. The following formats are used for the core instruction set:

- **R-type**: Starts with a 6-bit opcode and then three specifications for three registers. It also consists of a shift amount fied and a function field.
- **I-type**: Also starts with a 6-bit opcode but it has only two registers and a 16-bit immediate value.
- **J-type**: follows a 6-bit opcode with a 26-bit jump target.

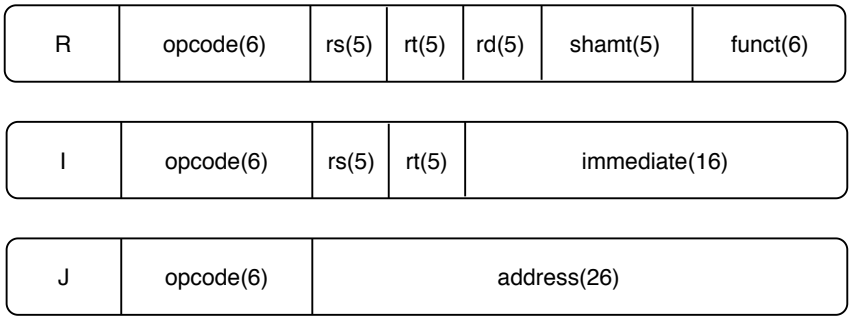

Figure 4.3: MIPS instruction formats

4.1.3 MIPS32

After spinning-out of Silicon Graphics in 1998, MIPS Technologies refocused on the embedded market. MIPS32 was based on MIPS II with some additional features from other previous versions. MIPS32 instruction set consists of 32-bit instructions including loads and stores, ALU, shift, multiplication and division, jump and brach, exception.

4.2 Environment Model

An environment model of a MIPS binary program is defined as a tuple $\langle \mathrm{R}, \mathrm{M}, \mathrm{S} \rangle$ where

- R is the set of 33 registers including 32 general purpose registers and register PC.
$$\mathrm{R} = \{\mathrm{r}_0, \mathrm{r}_1 ... \mathrm{r}_{30}, \mathrm{r}_{31}, \mathrm{pc}\}$$
- M is the set of stored memory locations.
$$\mathrm{M} = \{\mathrm{m}_0, \mathrm{m}_1 ... \mathrm{m}_\mathrm{n}\}$$
- S($\subseteq$ M) is the set of contiguous memory locations for a stack.
$$\mathrm{S} = \{\mathrm{s}_0, \mathrm{s}_1 ... \mathrm{s}_\mathrm{k} \mid \mathrm{k} < \mathrm{n}\}$$

We consider that each register $\mathrm{r_i}$ is represented by a 32-bit vector while each memory location $\mathrm{m_i}$ and $\mathrm{s_i}$ are represented by 8-bit vectors. In the beginning of the execution, the register pc stores the address of next instruction while sp stores the address of the top location of the stack. All objects in the environment model keep their initial symbolic values until the execution of instructions.

4.3 Operational Transitions

Fig. 4.4 describes some examples of MIPS operational transitions which follow MIPS technical documentation including `addi, b, lb, or`.

$$\frac{\mathrm{R_{pc}} = \mathrm{n}; \mathrm{instr(n)} = \mathrm{addi\,i\,j\,im}; \mathrm{R_j} = \mathrm{x}; |\mathrm{im}| \leq 2^{16} - 1; \mathrm{z} = \mathrm{x} + \mathrm{im}; \mathrm{z} \leq 2^{32} - 1}{\langle \mathrm{R}, \mathrm{M}, \mathrm{S} \rangle \rightarrow \langle \mathrm{R}[\mathrm{pc} \leftarrow \mathrm{n} + |\mathrm{instr(n)}|; \mathrm{R_i} = \mathrm{z}], \mathrm{M}, \mathrm{S} \rangle}[\mathrm{ADDI}]$$

$$\frac{\mathrm{R_{pc}} = \mathrm{n}; \mathrm{instr(n)} = \mathrm{b\,i}; \mathrm{R_i} = \mathrm{x}}{\langle \mathrm{R}, \mathrm{M}, \mathrm{S} \rangle \rightarrow \langle \mathrm{R}[\mathrm{pc} \leftarrow \mathrm{pc} + \mathrm{x}], \mathrm{M}, \mathrm{S} \rangle}[\mathrm{B}]$$

$$\frac{\mathrm{R_{pc}} = \mathrm{n}; \mathrm{instr(n)} = \mathrm{lb\,i\,j\,k}; \mathrm{R_j} = \mathrm{x}; \mathrm{y} = \mathrm{x} + \mathrm{k}; \mathrm{M_y} = \mathrm{z}}{\langle \mathrm{R}, \mathrm{M}, \mathrm{S} \rangle \rightarrow \langle \mathrm{R}[\mathrm{pc} \leftarrow \mathrm{n} + |\mathrm{instr(n)}|; \mathrm{R_i} = \mathrm{z}], \mathrm{M}, \mathrm{S} \rangle}[\mathrm{LB}]$$

$$\frac{\mathrm{R_{pc}} = \mathrm{n}; \mathrm{instr(n)} = \mathrm{or\,i\,j\,k}; \mathrm{R_j} = \mathrm{x}; \mathrm{R_k} = \mathrm{y}; \mathrm{z} = \mathrm{x\,or\,y}}{\langle \mathrm{R}, \mathrm{M}, \mathrm{S} \rangle \rightarrow \langle \mathrm{R}[\mathrm{pc} \leftarrow \mathrm{n} + |\mathrm{instr(n)}|; \mathrm{R_i} = \mathrm{z}], \mathrm{M}, \mathrm{S} \rangle}[\mathrm{OR}]$$

Figure 4.4: Some examples of MIPS operational transitions

4.4 Java Specification

We describe the formal semantics of MIPS instructions by Java methods, which are interpreted on a Java class `BitVec` prepared for Corana [17]. The value of the

`BitVec` class is a pair $\langle \texttt{bs}, \texttt{s} \rangle$, where `bs` is a variable in the `BitSet` class supporting 32-bit vector representation and `s` is a string variable that stores a symbolic value in the BitVector theory of SMT solvers. We manually prepare 21 primitive functions (listed below) appearing in the pseudocode description.

1. Jump Operators
    - `j`: Jump to an address.
    - `jr`: Jump to a stored value of a register (indirect jump)
2. Bitwise Operators
    - `and`: Bitwise `AND`
    - `or`: Bitwise `OR`
    - `xor`: Bitwise `XOR`
    - `nor`: Bitwise `NOR`
3. Arithmetic Operators
    - `add`: Addition of two `BitVec` values.
    - `sub`: Subtraction of two `BitVec` values.
    - `mul`: Multiplication of two `BitVec` values.
    - `div`: Division of two `BitVec` values.
4. IO Operators
    - `write`: Write a `BitVec` value to a register.
    - `val`: Get the `BitVec` value stored in a register.
5. Bit-based Operators
    - `signExtend`: Signed extend a `BitVec` value.
    - `zeroExtend`: Zero extend a `BitVec` value.
    - `concat`: Concat two `BitVec` values.
    - `power`: Power a `BitVec` values with a constant.
6. Comparable Operators
    - `equal`: Compare two `BitVec` values, if they are equal, return `True`. Otherwise, return `False`.

- `notEqual`: Compare two `BitVec` values, if they are equal, return `False`. Otherwise, return `True`.
- `less`: Compare two `BitVec` values, if the first value is less than the second value, return `True`. Otherwise, return `False`.
- `greaterOrEqual`: Compare two `BitVec` values, if the first value is greater or equal than the second value, return `True`. Otherwise, return `False`.

7. Other

    - `signalException`: Raising an exception.

Fixed-size BitVector Theory

In this study, we propose BitVec based on using SMT format for representing path conditions in an effective way. The details of this format can be found on SMT-LIB website[1], we introduce briefly the set of operators as follows (note that `a` and `b` are two `BitVec` values):

1. Basic Bitvector Arithmetic

    - Addition: `(bvadd a b)`
    - Subtraction: `(bvsub a b)`
    - Unary Minus: `(bvneg a)`
    - Multiplication: `(bvmul a b)`
    - Unsigned Remainder: `(bvurem a b)`
    - Signed Remainder: `(bvsrem a b)`
    - Signed Modulo: `(bvsmod a b)`
    - Shift Left: `(bvshl a b)`
    - Logical Shift Right: `(bvlshr a b)`
    - Arithmetical Shift Right: `(bvashr a b)`

2. Bitwise Operations

    - Bitwise OR: `(bvor a b)`
    - Bitwise AND: `(bvand a b)`
    - Bitwise NOT: `(bvnot a)`

[1] http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml

- Bitwise NAND: `(bvnand a b)`
- Bitwise NOR: `(bvnor a b)`
- Bitwise XNOR: `(bvxnor a b)`

3. Predicates over Bitvectors

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

Chapter 5

Semantics Extraction

5.1 Pseudocode Specification

Table 5.1: The specification of instruction `ADDI`

Format	`ADDI rt, rs, immediate`
Purpose	To add a constant to a 32-bit integer. If overflow occurs, then trap.
Description	The 16-bit signed immediate is added to the 32-bit value in GPR `rs` to produce a 32-bit result. • If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs. • If the addition does not overflow, the 32-bit result is placed into GPR `rt`

Operation	`temp` ← `(rs[31]||rs[31..0]) + sign_extend(immediate)`
	`if temp[32]` ≠ `temp[31] then`
	` SignalException(IntegerOverflow)`
	`else`
	` rt` ← `sign_extend(temp[31..0])`
	`endif`

The specification of MIPS instructions was collected and extracted from the MIPS32 (release 2) instruction set manual after converting the PDF format of

this manual to the text format. There are four main sections including `format`, `purpose`, `description` and `operation`. Table 5.2 shows an example of the specification of instruction `ADDI`. Among four sections, `format` and `operation` are used to obtain Java methods representing MIPS32 formal semantics.

5.2 Three Steps of Semantics Extraction

The semantics extraction from the MIPS32 instruction set manual consists of three steps.

1. *Convert the pdf file to a text file*: After opening the pdf file by Microsoft WORD, we observe the keywords as `Format`, `Purpose`, `Description`, and `Operation` to divide it to the corresponding sections. By using these keywords, we obtain these sections of each instruction. Moreover, tables of instructions grouped by categories, which are CPU, FPU, Coprocessor, and Privileged, are extracted to select suitable instructions. Since IoT malware is mostly a sequential user-mode process, we target only on CPU category MIPS32 instructions, which are 63 instructions (listed in Appendix B).

2. *Format extraction*: Explained in Section 5.3. This step illustrates the process of format extraction from `format` section of specification to Java code.

3. *Operation extraction*: Explained in Section 5.4. This step shows the procedure of generating Java methods by proposing a context-free grammar.

They are summarized in Fig. 5.1.

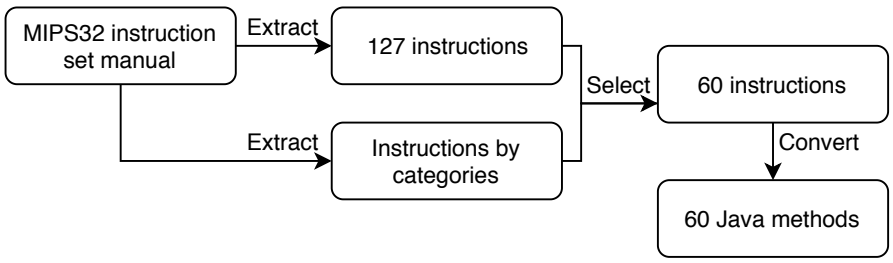

Figure 5.1: The overview of the specification extraction

5.3 Format Extraction

Table 5.2 shows the format section, which describes the operand and the opcodes of an instruction. MIPS instruction formats are divided into three types, hence, and three kinds of corresponding Java codes are prepared.

Table 5.2: The specification of the instruction ADDI

Format	**Java code**
Operand rd rs rt	void Operand(Character rd, Character rs, Character rt)
Operand rt rs immediate	void Operand(Character rt, Character rs, int immediate)
Operand rs immediate	void Operand(Character rs, int immediate)

The operands rd, rs, and rt describe the values of 32-bits registers Java class Character is adopted. The operand immediate is to store an integer and its Java class is int class. For instance, Table 5.3 shows the corresponding Java code of ADDI.

Table 5.3: The corresponding Java code of ADDI format

ADDI rt rs immediate	void ADDI(Character rt, Character rs, int immediate)

5.4 Operation Extraction

The operation section describes the pseudo-code. It is the most important field for automatically extracting MIPS formal semantics and generating Java executable code for the binary emulator of SyMIPS. However, MIPS Instruction Set document does not supply the definition of the syntax and the semantics of the pseudo-code. Hence, following to the automatic extraction of x86 formal semantics [11], we manually deduced and proposed a context-free grammar including 17 rules for parsing pseudo-code to abstract syntax trees as follows:

```
code: statement+ EOF;
statement:  structuredStatement|simpleStatement;
simpleStatement: assignmentStatement|expression;
structuredStatement: conditionalStatement|forLoopStatement;
parameterList: expression(',expression)*;
assignmentStatement: <assoc=right> expression''expression
    |'('parameterList')'''expression;
expression: factor
    |<assoc=right> expression('^')expression
    |expression('||')expression
```

```
    |expression(''|'/'|'*')expression
    |expression('div'|'mod')expression
    |expression('or'|'and'|'xor'|'nor'|'&')expression
    |expression('<<'|'>>')expression
    |expression('+'|'-')expression
    |expression('='|''|'>'|'<'|''|'')expression
    |expression('||'|''|'&&')expression
    |<assoc=right>expression'='expression;
indexing: identifier'['expression'..'expression']'
    |identifier'['expression']';
factor: funcCall
    |identifier
    |'('expression')'
    |unsignedConstant
    |indexing;
identifier: IDENT;
funcCall: <assoc=right> IDENT '('parameterList?')';
unsignedConstant: NUMBER;
conditionalStatement:   'if' expression 'then'
                            statement+
                        ('elseif' expression 'then' statement)*
                        ('else' statement)? 'endif';
forLoopStatement: 'for' IDENT 'in' '['expression'..'expression']'
                        statement
                  'endfor';
SPACE: [ \t\r\n]+ -> skip ;
NUMBER: [0-9]+ ;
IDENT: [a-zA-Z_] [a-zA-Z0-9_]* ;
```

From these rules, we used ANTLR (ANother Tool for Language Recognition)[1] to generate a parser for the pseudocode of the operation section. The abstract syntax trees are obtained by applying the parser. After preparing primitive Java methods, the abstract syntax trees are transferred to Java source code fragments by the depth-first traversal. Fig. 5.2 shows the result for the ADDI instruction.

5.4.1 Detecting and selecting primitive functions

The MIPS instruction set manual also provides a list of instructions grouped by categories including CPU, FPU, Coprocessor, and Privileged. The pseudocode description is firstly converted to an abstract syntax tree obeying to the prepared

[1] https://www.antlr.org

```
public void ADDI(Character rt, Character rs, int immediate){
    BitVec temp = add(concat(val(rs).get(31), val(rs).get(0, 31)),
                      signExtend(immediate));
    if(notEqual(temp.get(32), temp.get(31))){
        signalException(IntegerOverflow);
    } else {
        write(rt,signExtend(temp.get(0, 31)));
    }
}
```

Figure 5.2: The Java method of instruction `ADDI`

context-free grammar (Appendix A). Second, during the process of converting abstract syntax trees to Java methods, unknown primitive functions are detected. Fig. 5.4 shows a part of the abstract syntax tree represented the fist line `temp ← (rs[31]||rs[31..0]) + sign_extend(immediate)` of the `ADDI` instruction. Three primitive functions as `+`, `||`, and `sign_extend` (which are in dashed boxes) are detected during the parsing algorithm traversal through `expression` and `funcCall` nodes.

5.4.2 Representation of BitVector Theory

By using string variables for storing symbolic values in SMT format of BitVector theory, the primitive functions may update symbolic values. Fig. 5.3 shows an example of the pseudocode of a primitive function `and`. The symbolic values are updated during the execution of these instructions without further human effort.

```
BitVec and(BitVec m, BitVec n) {
    String symbolic = "(bvand "+ m.symbolic + " " + n.symbolic + ")";
    BitSet concrete = m.and(n);
    return new BitVec(concrete, symbolic);
}
```

Figure 5.3: The pseudo-code of the primitive function `and`

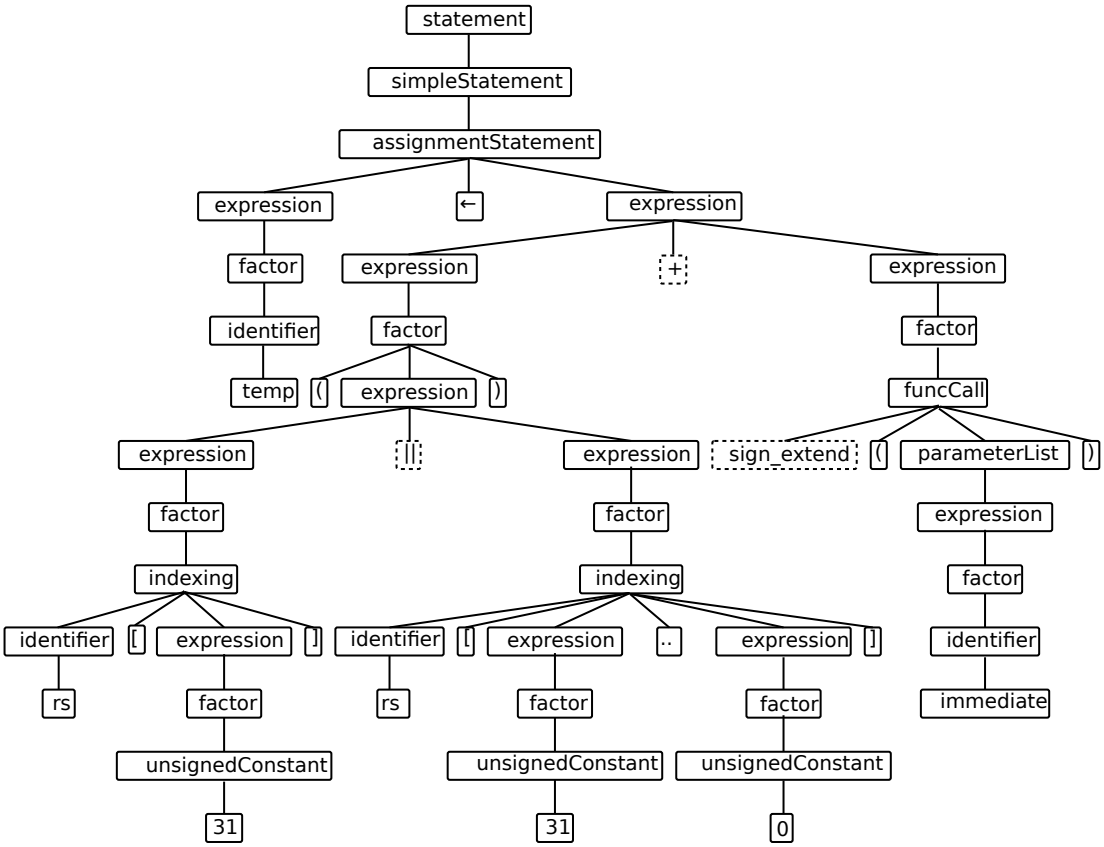

Figure 5.4: An example of an abstract syntax tree

Chapter 6

Conformance Testing

Software testing is a process to evaluate whether an application meets the specified requirements or not and to detect potential bugs or vulnerability to ensure the quality of products. There are many different types of testing from manual testing to automated testing such as unit testing, integration testing or functional testing. In this study, we conduct the conformance testing to validate the correctness of Java generated methods by comparing the output results generated by Java methods and a trusted MIPS emulator called MARS. This step has two main objectives including generating test cases automatically and then performing the conformance testing.

6.1 Test Cases Generation

To automatically generate test cases for full coverage, we applied a symbolic execution tool based on Java PathFinder named JDart[10] on the generated Java methods. It has been proposed in 2016 with two main goals. The primary goal has been to build a symbolic analysis framework that is robust enough to handle industrial-scale software. More precisely, it has to be able to execute industrial software without crashing, deal with long execution paths and complex path constraints. The second goal has been to build a modular and extensible platform that can be used for the implementation and evaluation of novel ideas in Dynamic Symbolic Execution. By applying JDart on generated methods, we can automatically obtain test cases.

6.2 Testing Environment

Conformance testing is executed by comparing the environment of the binary emulators between our method and MAR as a MIPS emulator. As mentioned at section 4.2, the environment consists of three components including `registers`, `memory`, and `stack`. The input and output of test cases are illustrated below:

1. Input
    - Source code: Java generated methods
    - Parameters: Generated by symbolic execution as mentioned at the above section.
    - Pre-Environment: the tuple $\langle$`registers`, `memory`, `stack`$\rangle$ before executing Java methods.
2. Ouput
    - Post-Environment: the tuple $\langle$`registers`, `memory`, `stack`$\rangle$ after executing Java methods.

6.3 Testing Procedure

Fig. 6.1 describes the overview of conformance testing that contains two steps.

1. Apply JDart for the symbolic execution on a generated Java method, and generate test cases to cover its all feasible branches.
2. Execute the generated Java method and the instruction on the trusted emulator MARS with all generated test cases, and compare their results.

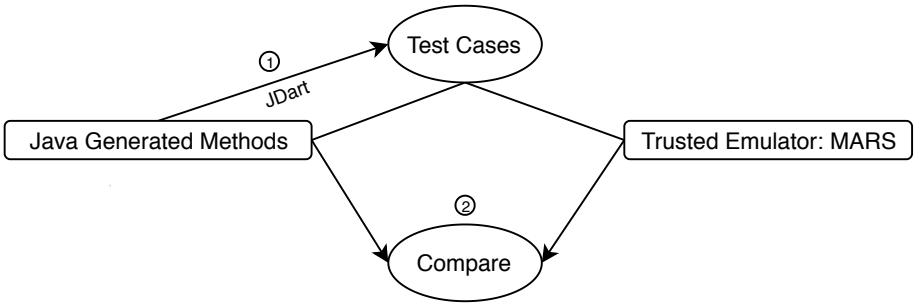

Figure 6.1: The workflow of conformance testing

For instance, JDart generates two test cases of the Java method for the instruction `ADDI`. Note that 32 characters from {`'0'..'9', 'a'..'v'`} are used to represent for 32 general registers referred by `rd`, `rs`, and `rt`.

In the following test cases, three characters `7, d, h` represent for three registers $\mathtt{r}_7, \mathtt{r}_{12}, \mathtt{r}_{15}$ respectively.

- Test case 1: `rt='7', rs='h', im=703`.
- Test case 2: `rt='d', rs='h', im=0`.

Chapter 7

Dynamic Symbolic Execution Tool

7.1 Overview

In this section, we introduce a preliminary version of a dynamic symbolic execution tool SyMIPS[1] (**Sy**mbolic Execution for **MIPS**). By exploiting several well-known methods including Capstone (as single-step disassembler) and Z3[2] (as a backend SMT solver), SyMIPS traces IoT Malware and generate its control flow graph under the presence of indirect jumps. Fig. 7.1 describes five steps of the workflow.

1. Capstone is used for the single-step disassembly to get an instruction $\texttt{inst}_\texttt{i}$
2. Interpret the instruction $\texttt{inst}_\texttt{i}$ to a Java method based on prepared primitive functions in the pseudocode.
3. Execute $\texttt{inst}_\texttt{i}$ and update the environment and the path conditions.
4. Generate the control flow graph based on step 3.
5. Repeating the process until reaching either an unsupported instruction or interrupted by timeout.

7.2 Environment Updates

SyMIPS updates the environment and the path condition when executing an instruction, based on the BitVec Java class and 27 primitive methods (Section 5.3).

[1] https://github.com/tracquangthinh/SyMIPS
[2] https://github.com/Z3Prover/z3

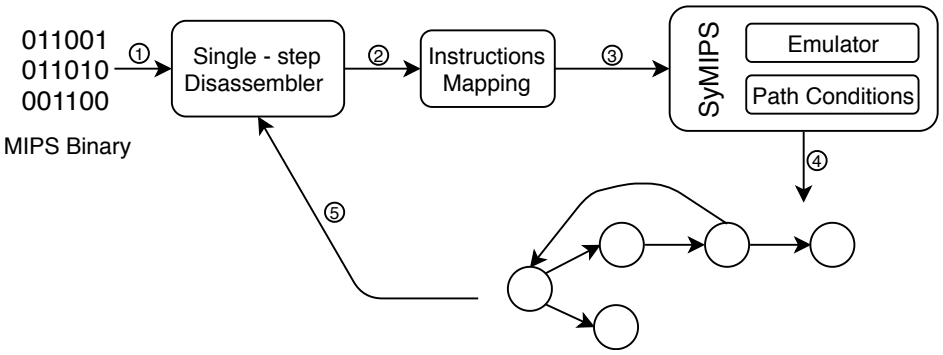

Figure 7.1: Tracing MIPS binary by SyMIPS

For instance, the instruction `AND r2, r3, r4` sets `r2` by `and(r3, r4)` and updates flags based on the primitive function `and`. Its updates occur only on the three registers `r2`, `r3` and `r4`.

For the `BitSet` value $\mathtt{c_i}$ and the symbolic value $\mathtt{s_i}$ with $\mathtt{i} \in \{2, 3, 4\}$, the pre-environment `preEnv` is:

$$\mathtt{r_2} : \langle \mathtt{c_2}, \mathtt{s_2} \rangle$$
$$\mathtt{r_3} : \langle \mathtt{c_3}, \mathtt{s_3} \rangle$$
$$\mathtt{r_4} : \langle \mathtt{c_4}, \mathtt{s_4} \rangle$$

The post-environment `postEnv` after executing the instruction `and` is:

$$\mathtt{r_2} : \langle \mathtt{and}(\mathtt{c_3}, \mathtt{c_4}), (\mathtt{bvand}\, \mathtt{s_3}\, \mathtt{s_4}) \rangle$$
$$\mathtt{r_3} : \langle \mathtt{c_3}, \mathtt{s_3} \rangle$$
$$\mathtt{r_4} : \langle \mathtt{c_4}, \mathtt{s_4} \rangle$$

while the `and` method is prepared in the BitSet class.

7.3 Path Conditions Generation

The path condition is updated when a conditional jump occurs. Returning to the example above , we assume that the next instruction is `beq r2 r3 offset` while `offset` is the destination of the jump instruction. This instruction `beq` compares the contents of two registers `r2` and `r3`, then if `r2` equals to `r3`, the instruction branches to the offset. The path condition of the true branch $\mathtt{pc_{true}}$ is updated as $\mathtt{pc_{true}} = \mathtt{pc_{true}} \wedge (= (\mathtt{bvand}\, \mathtt{s_3}\, \mathtt{s_4})\, \mathtt{s_3})$ while the path condition of the false branch $\mathtt{pc_{false}}$ is updated as $\mathtt{pc_{false}} = \mathtt{pc_{false}} \wedge (\mathtt{not}(= (\mathtt{bvand}\, \mathtt{s_3}\, \mathtt{s_4})\, \mathtt{s_3}))$. Although the above example is quite simple, more complicated instructions like `ADDI` also do

not require additional effort for updating path conditions due to the combination of environment update and `BitVec` computation.

7.4 Manual Implementation

From the supporting of 21 primitive functions, we successfully generate Java methods of 63 instructions. However, the binary emulator also requires load/store instructions. Hence, we manually implement 6 load/store instructions after observing MIPS32 Architecture Document as follows:

- `lb`: Load a byte value from the memory to a register.
- `sb`: Store a byte value from a register to the memory.
- `lw`: Load a word value from the memory to a register.
- `sw`: Store a word value from a register to the memory.
- `lh`: Load a half-word value from the memory to a register.
- `sh`: Store a half-word value from a register to the memory.

```
public void ADDI(Character rt, Character rs, int immediate){
    if(immediate <= power(2, 16)){
        BitVec temp = add(concat(val(rs).get(31), val(rs).get(0, 31)),
                          signExtend(immediate));
        if(notEqual(temp.get(32), temp.get(31))){
            signalException(IntegerOverflow);
        } else {
            write(rt,signExtend(temp.get(0, 31)));
        }
    }
}
```

Figure 7.2: The Java method of instruction `ADDI` after adding the constraint

The pseudocode section does not consist of the constraint of `immediate` parameter in arithmetic and logical instructions. During the process of parsing the abstract syntax trees of MIPS32 instructions to Java methods, we detect arithmetic and logical instructions to add the constraint into them. We also manually add the constraint $|\texttt{immediate}| \leq 2^{16}$ into the Java methods of `Operand rs rt immediate` and `Operand rs immediate` instructions. Fig. 7.2 shows the Java method of the ADDI instruction after adding the constraint of `immediate`.

7.5 SyMIPS versus Corana

Although SyMIPS and Corana [17] share the use of the `BitVec` class, there are several differences.

1. Unlike ARM that uses the flags and the conditional suffix to implement conditional executions, MIPS only uses general registers. Hence, the path condition updates for MIPS32 are only on registers.

2. Due to the characteristics of natural language specifications, ARM instructions consider 32-bit general registers as the word-size values and do not require to access single bits during its execution. Meanwhile, MIPS handles registers in the level of bits by producing `get` as a primitive function. For instance, the `ADDI` instruction uses a conditional statement to decide whether an overflow occurs (Fig. 5.2). By using the `get` function, `ADDI` accesses the 31^{st} and 32^{th} single bits of the temporary variable `temp`.

Chapter 8

Experiments

8.1 Semantics Extraction

From the requirements of MIPS instructions, arithmetic and logical instructions in the forms of `Operand rs rt immediate` and `Operand rs immediate` support only 16-bit value of `immediate`. This fact does not appear in the pseudocode description, but mentioned elsewhere in MIPS instruction manual. JDart may generate the value of `immediate` which its absolute value is larger than 2^{16} ($|\texttt{immediate}| > 2^{16}$). We simply add the constraint $|\texttt{immediate}| \leq 2^{16}$ into the Java methods of `Operand rs rt immediate` and `Operand rs immediate` instructions. All of 63 instructions have passed the conformance testing after adding the constraint of `immediate`.

8.2 Handling Jumps by SyMIPS

Although IoT malware rarely uses complex obfuscation techniques like self-modifying loops, identifying the destination of jumps is an essential task for understanding the control structure of IoT malware.

8.2.1 Indirect Jumps

Fig. 8.1 shows an example of indirect jump in `ELF:Tsunami-BE` taken from ViruSign. SyMIPS successfully finds the destination `0x401eb0` of the indirect jump at `0x400ee8` by concolic testing.

```
0x400edc    lw t9, -0x7fe4(gp)
0x400ee0    nop
0x400ee4    addiu t9, t9, 0xffc
0x400ee8    jalr t9
0x400eec    nop
0x400ef0    lw gp, 0x10(sp)
```

(a) Disassemble result by Capstone

```
0x400edc    lw t9, -0x7fe4(gp)
0x400ee0    nop
0x400ee4    addiu t9, t9, 0xffc
0x400ee8    jalr t9
0x401eb0    mflo a0
0x401eb4    lui v0, 0xb8c3
```

(b) Trace of the indirect jump by SyMIPS

Figure 8.1: An example of indirect jump from IoT malware

8.2.2 Conditional Jumps

The main task of the symbolic execution is to judge the feasibility of branches in a conditional jump. Fig. 8.2a shows a conditional jump `bnez` at `0x400190` in `ELF:DDoS-Y` collected from VirusShare[1]. SyMIPS judges both true and false branches are satisfiable, and when the condition is true, it jumps to `0x400208`.

Fig. 8.2b shows an example of another conditional jump `beqz` at `0x4004ec` in `ELF:DDoS-Y` taken from ViruSign. SyMIPS detects that the true branch is unsatisfiable, and only the false branch is executed. Thus, the next instruction from `0x4004ec` always goes to `0x4004f0` and the code fragment starting at the jump destination `0x40049c` is dead code.

```
0x400188    lbu v0, -0x2ca0(s1)
0x40018c    nop
0x400190    bnez v0, 0x400208
0x400208    lw ra, 0x20(sp)
0x40020c    lw s1, 0x1c(sp)
```

(a) Both true and false branches are SAT

```
0x4004e8    slti v0, v0, 2
0x4004ec    beqz v0, 0x40049c
0x4004f0    nop
0x4004f4    lw v1, 0x44(fp)
0x4004f8    addiu v0, zero, 1
```

(b) The true branch is UNSAT

Figure 8.2: Feasibility checking of conditional jumps by SyMIPS

8.3 SyMIPS Performance

We perform experiments on MIPS32 IoT malware (taken from ViruSign) to see the performance of SyMIPS. Note that current SyMIPS implementation is quite preliminary, and no smart optimizations are applied. We execute on 3219 samples

[1]https://www.virusshare.com

on Ubuntu 18.04 with Intel Core i5-6200U CPU, 2.30GHz and 8GB. The results are summarized in Table 8.1.

Table 8.1: The performance of SyMIPS on IoT malware

Types of Executions		**Number of samples**
Finished		2725
Interrupted	Out of Memory	415
	Jump to Kernel Space/ System Calls	79
Total		3219
Average Size		178.8 KB

Note that system calls or jump to kernel space raise an exception. First, the processor set the cause of exceptions into register `$13`. Then it changes user mode to kernel mode and disables further interrupts. Afterward, the processor saves current PC as the return address to return when done handling the exception and then, jump to exception handler code.

Table 8.2 shows the statistical result of execution time of SyMIPS on 3219 malware:

Table 8.2: The execution time of SyMIPS on IoT malware

Ranges(seconds)	**Number of Samples**
0 - 10	1658
10 - 20	941
20 - 30	155
30 - 40	36
40 - 50	154
50 - 60	74
>60	201
Average Time:	**17.46 seconds**

Chapter 9

Limitations

9.1 Exception Handler

From MIPS32 Architecture Document[1], there are four sets of causes for an exception in MIPS32:

1. **Caused by hardware malfunctioning**: e.g. machine check or bus error on a load or store instruction, or instruction fetch.
2. **Caused by external causes (to the processor)**: e.g. hardware interrupts.
3. **Caused by instruction problems**: e.g. address error such as jumping to Kernel address space, or integer overflow.
4. **Caused by executions of special instructions**: e.g. system calls or break.

Although there are several causes for an exception, we currently observe that only case 3 and 4 occur in IoT malware samples. However, the implementation of SyMIPS does not support the exception handler for catching exceptions like system calls or jump to kernel spaces. For these cases, SyMIPS currently terminates the execution and returns the output. Figure 9.1 shows an example of an interrupted execution caused by `syscall` instruction.

To request a service, programs load the system call code into register `$v0` and the arguments into registers `$a0`, `$a1`, `$a2`, `$a3`. System calls that return values put their result in register `$v0`. In the above example, `0xfa5` is the system code of opening a file.

[1] https://www.mips.com/products/architectures/mips32-2/

```
0x40426c      addiu sp, sp, -0x20
0x404270      sw ra, 0x1c(sp)
0x404274      sw s0, 0x18(sp)
0x404278      sw gp, 0x10(sp)
0x40427c      addiu v0, zero, 0xfa5
0x404280      syscall
```

Figure 9.1: An interrupted execution caused by `syscall`

9.2 Indirect Jumps to the Destination Stored in Memory

In present, we do experiments on IoT malware of Linux OS. The format of these samples is ELF, which is the abbreviation for Executable and Linkable Format and defines the structure for binaries, libraries, and core files. The formal specification allows the operating system to interpreter its underlying machine instructions correctly. ELF files are typically the output of a compiler or linker and are a binary format. ELF consists of two main sections including `.text` and `.data`. While `.text` section contains executable code, `.data` stores initialized data. From the experiments, we observe that IoT malware execute indirect jumps with the destination stored in the memory. For instance, Fig. 9.2 depicts an example of indirect jumps with destination stored in the memory from `ELF:Tsunami-BE`. At `0x400edc`, the jump address is loaded into register `t9` from memory at `0x44577c`, which belongs to `.data` section, before executing an indirect jump at `0x400ee8`.

```
0x400ed0      lui gp, 5
0x400ed4      addiu gp, gp, -0x3770
0x400ed8      addu gp, gp, ra
0x400edc      lw t9, -0x7fe4(gp)
0x400ee0      nop
0x400ee4      addiu t9, t9, 0xffc
0x400ee8      jalr t9
0x401004      addu gp, gp, t9
0x401008      addiu sp, sp, -0x20
```

Figure 9.2: An example of indirect jumps with destination stored in the memory

In current implementation, SyMIPS load `.data` section into the memory by using java-binutils[2] - a Java library. Otherwise, MARS does not support loading

[2]https://github.com/jawi/java-binutils

`.data` section into the memory. Hence, it will terminate when an load instruction from `.data` section is executed.

However, this library can not read binary format successfully in some samples. It leads to the inaccurate destination address of indirect jumps in SyMIPS. Fig. 9.3 illustrates an example of an inaccurate destination address of indirect jump traced by SyMIPS due to failed binary format reading form `ELF:Mirai-MLC`. At `0x401898`, the word at `0x451c904` belonged to `.data` section is loaded into register `r9`. However, SyMIPS can not read this section successfully and simply assigns 0 to `t9`. It leads to the inaccurate indirect jump to the entry point of the execution.

```
0x401890      addiu gp, gp, -0x7f9c
0x401894      addu gp, gp, ra
0x401898      lw t9, -0x8fe0(gp)
0x40189c      nop
0x4018a0      jalr t9
```

Figure 9.3: An example of an inaccurate jump

9.3 Out of Memory

SyMIPS or general Dynamic Symbolic Execution tools require a huge amount of memory for tracing binary files. Whenever SyMIPS executes a branch instruction, it has to back up the current environment by using a stack to roll back in further steps. Figure 9.4 describes the backup mechanism of SyMIPS on true branches. Because SyMIPS uses Depth First Search algorithm for tracing, it may lead to the out of memory error since SyMIPS has to store many environments in the stack.

```
if(Z3Solver.checkSAT(trueCond) != null){
   tracing(currentAddress, jumpAddress);
   Logs.infoLn("- True branch is SAT. Go to " +
       Arithmetic.intToHex(jumpAddress));
   stack.push(emulator.getEnv().clone());
   exec(emulator, nodes, jumpAddress, trueCond);
   emulator = new MIPS32(stack.pop());
}
```

Figure 9.4: The mechanism of environment backup of SyMIPS on true branches

Chapter 10

Conclusion and Future Work

10.1 Conclusion

We proposed a semi-automatic approach for extracting the formal semantics of MIPS32 instructions from MIPS instruction manual. We also introduced a preliminary version of a dynamic symbolic execution tool SyMIPS. By performing experiments on 3219 IoT malware collected from ViruSign, and SyMIPS successfully traced 2804 samples. Although the current implementation is preliminary, SyMIPS found the destinations of indirect jumps by concolic testing and discovers dead conditional branches in some samples. This thesis has contributed three essential objects:

1. **Semantics Extraction**: By proposing a context-free grammar for pseudocode of MIPS instructions, we generated successfully Java executable code of 63 CPU instructions from its pseudocode in MIPS32 instruction set manual. Currently, we ignored float points, co-processor, and privileged instructions to simplify the implementation.

2. **Path Conditions Generation**: The path conditions are updated during the execution of instructions under the support of primitive methods without extra-human efforts.

3. **SyMIPS - Dynamic Symbolic Execution Tool**: We utilized Java generated methods to develop SyMIPS and did an experiment on more than 3000 malware collected from ViruSign to observe the performance of SyMIPS.

10.2 Future Work

Although most IoT malware is a sequential user-mode process, it is necessary to cover remain instructions since the total number of MIPS instructions is not large. Hence, in the future, we intend to implement float points, co-processor, and privileged instructions to enhance the Dynamic Symbolic Execution tool. Furthermore, there are two future directions as follows:

1. **Cover other MIPS platforms**: As can be seen, MIPS Architecture Website also provides the instruction set manual of other platforms such as microMIPS or nanoMIPS. These documents have the same structures as MIPS32 manual. Hence, the process of formal semantics extractions can be conducted in the same manner.

2. **Extend to other architectures**: We have developed three methods for three different architectures including BE-PUM for x86, Corana for ARM, and SyMIPS for MIPS. In the future, it is possible to target to new architectures, e.g. SPARC.

For this study, our plans are:

1. Optimizing and maintaining Dynamic Symbolic Execution tool SyMIPS to reduce memory consumption and execution time.

2. Doing more experiments on IoT malware to observe and detect its characteristics.

3. Generalizing current implementation to other platforms.

Bibliography

[1] S. Anand and M. Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In *ASE*, pages 33–42, 2011.

[2] G. Bonfante, J. Fernandez, J. Marion, B. Rouxel, F. Sabatier, and A. Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *ACM SIGSAC*, pages 745–756, 2015.

[3] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446. IEEE, 2008.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2009.

[5] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE S&P*, pages 380–394, 2012.

[6] F. Desclaux. Miasm : Framework de reverse engineering. 2012.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. page 11, 2005.

[8] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *STTT*, pages 366–381, 2000.

[9] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, pages 385–394, 1976.

[10] K. Luckow, M. Dimjaevi, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamari, and V. Raman. JDart: A Dynamic Symbolic Analysis Framework. In *TACAS*, pages 442–459, 2016.

[11] H. Nguyen. Automatic Extraction of x86 Formal Semantics from Its Natural Language Description. In *Masters Thesis, School of Information Science, JAIST*, March 2018.

[12] M. Nguyen, M. Ogawa, and T. Quan. Obfuscation Code Localization Based on CFG Generation of Malware. In *FPS*, pages 229–247, 2015.

[13] C. Psreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing Nasa software. In *ISSTA*, page 15, 2008.

[14] K. Sen and G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*, pages 419–423. 2006.

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*, pages 138–157, 2016.

[16] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed Proof Generation for Machine Code. In *CAV*, pages 288–305. 2010.

[17] V. Vu and M. Ogawa. Formal Semantics Extraction from Natural Language Specifications for ARM. In *FM*, 2019, to appear.

Appendix

Instruction	Description
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word to Hi, Lo
MADDU	Multiply and Add Unsigned Word to Hi, Lo
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MUL	Multiply Word to GPR
MULT	Multiply Word
MULTU	Multiply Unsigned Word
SEB	Sign-Extend Byte
SEH	Sign-Extend Halftword
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word
NOP	No Operation
SSNOP	Superscalar No Operation
AND	And
ANDI	And Immediate
LUI	Load Upper Immediate

NOR	Not Or
OR	Or
ORI	Or Immediate
XOR	Exclusive Or
XORI	Exclusive Or Immediate
EXT	Extract Bit Field
INS	Insert Bit Field
WSBH	Word Swap Bytes Within Halfwords
MFHI	Move From HI Register
MFLO	Move From LO Register
MOVN	Move Conditional on Not Zero
MOVZ	Move Conditional on Zero
MTHI	Move To HI Register
MTLO	Move To LO Register
ROTR	Rotate Word Right
ROTRV	Rotate Word Right Variable
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable
BREAK	Shift Word Right Logical Variable
TEQ	Trap if Equal
TEQI	Trap if Equal Immediate
TGE	Trap if Greater or Equal
TGEI	Trap if Greater of Equal Immediate
TGEIU	Trap if Greater or Equal Immediate Unsigned
TGEU	Trap if Greater or Equal Unsigned
TLT	Trap if Less Than
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal
TNEI	Trap if Not Equal Immediate