Formal Semantics Extraction from MIPS Instruction Manual

Quang Thinh Trac and Mizuhito Ogawa

Japan Advanced Institute of Science and Technology
{tracthinh, mizuhito}@jaist.ac.jp

Abstract. This study 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 to a dynamic symbolic execution tool SyMIPS. We perform an empirical study on 3219 MIPS32 IoT malware collected from ViruSign and observe that SyMIPS successfully traces 2412 samples, in which SyMIPS finds the dead conditional branch, e.g., in `DDOS-Y`. The rest is interrupted by either timeout, stack overflow, or exceptions, which current SyMIPS does not cover.

Keywords: Dynamic Symbolic Execution, MIPS32, IoT malware

1 Introduction

Symbolic execution has been developed mostly for high-level programming languages, e.g., JPF-SE [1] for Java and Klee [4] for C. Recently, symbolic execution tools are extended to binary code. An early example is McVeto[11], followed by KLEE-MC[2], Mayhem[5], MiAsm[6], CoDisasm[3], BE-PUM[9], Angr[10], Corana[13]. Most of them are developed for x86 except Corana for ARM.

When we consider IoT devices, various architectures exist. Smaller CPUs, MPU (Micro Processor Unit), are either 32 bits or 64 bits, e.g., ARM Cortex-A, MIPS32, MIPS64, MC68000, Sparc (by Fujitsu), PowerPC, and x86. Controllers, MCU (Micro Controller Unit), are up to 32 bits, e.g., ARM Cortex-M7, Z80, PIC, AVR, MSP430 (TI), and RL78 (Runesas). When we develop binary symbolic execution tools, the large variation forces huge human effort. Good news is:

1. Each instruction set often has a concrete manual in rigid English.
2. MPUs and MCUs have shallow caches and mostly do not allow out-of-order execution. Avoiding multi-threads, weak memory models, and floating-point arithmetic, the operational semantics framework simply becomes the transitions on the environment consisting of *memory*, *stack*, *registers*, and *flags*.
3. Various debuggers and emulators are often available, which implement the semantics of instruction sets.

They suggest (semi-)automatic extraction of the formal semantics from English manuals. Furthermore, by comparing with the execution between existing debuggers/emulators and the generated symbolic execution tool, the conformance testing can resolve the ambiguity in natural language processing.

For extracting the semantics, the following three sections are essential.

- Format section shows the name of the instruction and its operands.
- Operation section shows how the environment is updated. Some instruction sets also have the pseudo-code descriptions, e.g., x86 and MIPS.
- Flag Update section shows the change of the boolean condition. Some instruction sets have no flags, e.g., MIPS, and the condition is set on registers.

Following to BE-PUM for x86 [8] and Corana for ARM [13]), this study investigates a semi-automatic extraction of the formal semantics of MIPS instructions. Among MIPS variations, we focus on MIPS32 (release 5) from MIPS32 instruction set manual[1], which has the emulator MARS. Among 127 collected MIPS32 instruction specifications, we focus on 63 of the CPU category. After preparing a Java template describing the operational semantics framework, we manually prepare 21 primitive functions in the pseudocode description, which successfully instantiate the Java template for all 63 instructions. The generated Java code is inserted into a dynamic symbolic execution tool SyMIPS[2]. We perform an empirical study on 3219 MIPS32 IoT malware in ViruSign[3] and observe that SyMIPS successfully traces 2412 samples. The rest is interrupted by either timeout, stack overflow, or exceptions, which current SyMIPS does not cover. Note that SyMIPS finds the dead conditional branch, e.g., in `DDOS-Y`.

Related Work

The first trial of a formal semantics extraction appears for x86 [8] for extending BE-PUM [9], which introduced the sentence-level similarity analysis to detect flag updates. The experiment shows that among 530 collected specifications from Intel Developer's Manual[4], Java method descriptions of 299 x86 instructions are successfully generated by manually preparing 30 primitive functions, which not only enlarged the BE-PUM support to the total 400 instructions but also found 5 human bugs in manually implemented 200 instructions.

The formal semantics extraction for ARM [13] is more challenging, since the ARM manual is described only in English. By manually preparing 228 semantics interpretation rules, the experiment shows that among 1039 collected ARM Cortex-M specifications from ARM manual[5], 662 instructions are successfully processed. Note that both apply the conformance testing by using the existing emulators, i.e., Ollydbg [6] for x86 and μVision[7] for ARM.

[1] https://www.mips.com/products/architectures/mips32-2
[2] `https://github.com/tracquangthinh/SyMIPS`
[3] https://www.virusign.com
[4] https://www.felixcloutier.com/x86
[5] https://developer.arm.com
[6] http://www.ollydbg.de
[7] http://keil.com/mdk5/uvision

2 Formal Semantics of MIPS

2.1 MIPS Architecture

MIPS is a RISC instruction set, which were introduced in 1985. MIPS assumes a load/store architecture (or known as register-register architecture, in which the memory access is limited to the load and store instructions. A conventional MIPS processor contains the following components:

1. **Registers**: is a small set of high-speed storage cells inside the CPU. MIPS provides 32 general-purpose registers.
2. **Memory**: is the 32-bits addressing space.
3. **Stack**: is taken as a special area of the memory.

In contrast to x86 and ARM, MIPS have no flags. Instead, it uses general registers for storing the boolean conditions. Furthermore, the MIPS instructions except for the load/store, `lb, sb, lw, sw`, cannot directly access memory.

2.2 MIPS Instruction Manual

The specification of the MIPS instructions is collected and extracted from the MIPS32 (release 5) instruction set manual. They are in the PDF format and consist of four prime sections including `format`, `purpose`, `description` and `operation`. Table 2.2 shows an example of the specification of instruction `ADDI`. Among four sections, `format` and `operation` are used to obtain Java methods.

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`

2.3 Java Methods as Formal Semantics

We describe the formal semantics of MIPS instructions by Java methods with a Java class `BitVec`, originally prepared for Corana [13]. The value of the `BitVec` class is a pair $\langle \texttt{bs}, \texttt{s} \rangle$, where `bs` is a 32-bit vector variable in the `BitSet` class and `s` is a string variable that stores a symbolic value in the BitVector theory. We manually prepare 21 primitive functions appearing in the pseudocode. An example below is a generated Java method of the instruction `ADDI`

```
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))); }
}
```

3 Specification Extraction

3.1 Operation Extraction

The operation section describes the pseudo-code. It is the most important field for extracting MIPS formal semantics and generating Java executable code. However, MIPS Instruction Set manual obeys general common knowledge on the syntax and the semantics of the pseudo-code. Following to x86 formal semantics extraction [8], we manually prepare a context-free grammar including 17 rules for parsing the pseudo-code. We used ANTLR (ANother Tool for Language Recognition)[8] to generate a parser, which results the abstract syntax tree.

Representation of BitVector Theory String variables are used to store values in BitVector theory of the SMT format and the primitive functions compute 32-bit values. Below is an example of a primitive function `and`.

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

3.2 Conformance Testing

JDart[7] is a dynamic symbolic tool built on the top of Java PathFinder[12]. After converting the pseudo-code to Java methods, we use JDart to generate the test cases of Java methods to cover all feasible execution paths of MIPS instructions. Then we apply the conformance testing by comparing the executed results of Java methods and MARS[9] - a trusted emulator of MIPS32.

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.

[8] https://www.antlr.org
[9] http://courses.missouristate.edu/KenVollmar/mars

4 Dynamic Execution Tool: SyMIPS

A preliminary version of a dynamic symbolic execution tool SyMIPS[10] (**Sy**mbolic Execution for **MIPS**) adopts Capstone (as a single-step disassembler) and Z3[11] (as a backend SMT solver),

4.1 Environment Updates

SyMIPS updates the environment and the path condition when executing an instruction, based on the BitVec class and 21 primitive methods (Section 2.3). For instance, `ADDI r2, r3, 3` set `r2` to `r3 + 3` and updates symbolic values. For the `BitSet` value $\mathtt{c_i}$ and the symbolic values $\mathtt{s_i}$ with $\mathtt{i} \in \{2, 3\}$, the pre-environment `preEnv` $\mathtt{r_2} : \langle \mathtt{c_2}, \mathtt{s_2} \rangle$; $\mathtt{r_3} : \langle \mathtt{c_3}, \mathtt{s_3} \rangle$ is updated to the post-environment `postEnv`

$$\begin{aligned}
\mathtt{r_2}: \quad & \langle c_3 + 3, ((_\ sign_extend\ 1)((_\ extract\ 30\ 0)(bvadd\ (concat \\
& \quad ((_\ extract\ 31\ 31)\ r_3)((_\ extract\ 30\ 0)\ r_3))\ \#x00000003)))\rangle \\
\mathtt{r_3}: \quad & \langle c_3 \quad , s_3 \rangle
\end{aligned}$$

4.2 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 r4 offset` while `offset` is the destination of the jump instruction. This instruction `beq` compares two registers `r2` and `r4`, then if `r2` equals to `r4`, it branches to the offset. The path conditions of the true and false branches are updated as:

$$\begin{aligned}
\mathtt{pc_{true}} = \quad & \mathtt{pc} \wedge (= ((_\ sign_extend\ 1)((_\ extract\ 30\ 0) \\
& \quad (bvadd(concat\ ((_\ extract\ 31\ 31)\ r_3) \\
& \quad ((_\ extract\ 30\ 0)\ r_3))\ \#x00000003)))\ r_4) \\
\\
\mathtt{pc_{false}} = \quad & \mathtt{pc} \wedge (not\ (= ((_\ sign_extend\ 1)((_\ extract\ 30\ 0) \\
& \quad (bvadd\ (concat\ ((_\ extract\ 31\ 31)\ r_3) \\
& \quad ((_\ extract\ 30\ 0)\ r_3))\ \#x00000003)))\ r_4))
\end{aligned}$$

4.3 SyMIPS versus BE-PUM, Corana

BE-PUM was originally implemented manually and later the formal semantics extraction of 299 x86 instructions extends BE-PUM [8]. Compared to BE-PUM, SyMIPS and Corana are generated from scratch and share the use of the `BitVec` class. However, there are several differences:

[10] https://github.com/tracquangthinh/SyMIPS
[11] https://github.com/Z3Prover/z3

1. ARM uses the flags and the conditional suffix to implement conditional executions. In contrast, MIPS only uses general registers.
2. ARM instructions treat 32-bit general registers as the word-size values and do not require to access single bits during the 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. By using the `get` function, `ADDI` accesses the 31^{st} and 32^{th} single bits of the temporary variable `temp`.

5 Experiments and Results

5.1 SyMIPS Performance

We perform experiments on MIPS32 IoT malware (taken from ViruSign) to see the performance of SyMIPS. Note that current SyMIPS implementation is preliminary. We try 3219 samples on Ubuntu 18.04 with Intel Core i5-6200U CPU, 2.30GHz and 8GB. The results are summarized below.

Types of Executions		**Number of samples**
Finished		2412
Interrupted	Out of Memory	415
	Jump to Kernel Space/ System Calls	79
	Fail to read binary format	313
Total		3219
Average Size		178.8 KB

Range(seconds)	**Number of Samples**	**Size(KBs)**			**Execution Time**		
		Min	**Max**	**Average**	**Min**	**Max**	**Average**
0 - 10	1658	0.5	638	165	1.21	991.22	17.46
10 - 20	941	30	763	111			
20 - 30	155	47	198	138			
30 - 40	36	59	240	153			
40 - 50	154	121	301	200			
50 - 60	74	142	1156	312			
>60	201	124	531	292			

5.2 Handling Dynamic Jumps by SyMIPS

Although IoT malware rarely uses obfuscation techniques, identifying the destination of indirect jumps is essential to understand the control structure.

```
0x401898    lw t9, -0x7fe0(gp)
0x40189c    nop
0x4018a0    addiu t9, t9, 0x19bc
0x4018a4    jalr t9
0x4019c8    addiu sp, sp, -0x20
0x4019cc    sw ra, 0x18(sp)
```

(a) Trace of the indirect jump

```
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

Indirect Jump Example (a) shows an indirect jump `jalr` at `0x4018a4` in `ELF:Mirai-ACL`. SyMIPS finds the destination `0x4019c8` by concolic testing.
Conditional Jump Example (b) shows a conditional jump `beqz` at `0x4004ec` in `ELF:DDoS-Y`. SyMIPS detects that the true branch is unsatisfiable. It always goes to `0x4004f0` and the code fragment starting at `0x40049c` is dead code.

6 Conclusion

We proposed a semi-automatic formal semantics extraction of MIPS32 instructions from their manual. Consequently, a preliminary version of a dynamic symbolic execution tool SyMIPS for MIPS32 was presented. The experiments on 3219 IoT malware taken from ViruSign successfully analyzed 2412 samples, including the detection of dead conditional branches, e.g., in `DDOS-Y`.
Acknowledgement This study is partially supported by JSPS KAKENHI Grant-in-Aid for Scientific Research (B)19H04083. The original content was accepted as the master thesis [14].

References

1. S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. *TACAS*, pp.134–138. 2007.
2. R. Anthony. Methods for Binary Symbolic Execution. *Ph.D Dissertation, Stanford University*, December 2014.
3. G. Bonfante, J. Fernandez, JY. Marion, B. Rouxel, F. Sabatier, and A. Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. *ACM SIGSAC*, pp.745–756, 2015.
4. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI*, 2009.
5. SK. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. *IEEE S&P*, pp.380–394, 2012.
6. F. Desclaux. Miasm : Framework de reverse engineering. 2012.
7. K. Luckow, M. Dimjasevic, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, and V. Raman. JDart: A Dynamic Symbolic Analysis Framework. *TACAS*, pp.442–459, 2016.
8. H.L.Y. Nguyen. Automatic Extraction of x86 Formal Semantics from Its Natural Language Description. *Master's Thesis, School of Information Science, JAIST*, March 2018.
9. M.H. Nguyen, M. Ogawa, and T.T. Quan. Obfuscation Code Localization Based on CFG Generation of Malware. *FPS*, pp.229–247, 2015.
10. 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. *IEEE S&P*, pp.138–157, 2016.
11. A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed Proof Generation for Machine Code. *CAV*, pp.288–305. 2010.
12. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *IEEE ASE*, pp.3–11, 2000.
13. V.A. Vu and M. Ogawa. Formal Semantics Extraction from Natural Language Specifications for ARM. *FM*, pp.465–483, LNCS 11800, 2019.
14. Q.T. Trac, Generating a Dynamic Symbolic Execution Tool from MIPS Specifiations. *Master's Thesis, School of Information Science, JAIST*, September 2019.