석 사 학 위 논 문
Master's Thesis

차등 테스트를 통한 Qualcomm Hexagon 에뮬레이터 및 디컴파일러의 정확성 분석

Analysis of the Correctness of Qualcomm Hexagon Emulators and Decompilers via Differential Testing

2022

정 현 식 (鄭 賢 植 Jeong, Hyunsik)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

차등 테스트를 통한 Qualcomm Hexagon 에뮬레이터 및 디컴파일러의 정확성 분석

2022

정 현 식

한 국 과 학 기 술 원

전산학부 (정보보호대학원)

차등 테스트를 통한 Qualcomm Hexagon 에뮬레이터 및 디컴파일러의 정확성 분석

정 현 식

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 12월 21일

Analysis of the Correctness of Qualcomm Hexagon Emulators and Decompilers via Differential Testing

Hyunsik Jeong

Major Advisor: Yongdae Kim
Co-Advisor: Insu Yun

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science (Information Security)

Daejeon, Korea
December 21, 2021

Approved by

Yongdae Kim
Professor of Electrical Engineering

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MIS 정현식. 차등 테스트를 통한 Qualcomm Hexagon 에뮬레이터 및 디컴파일러의 정확성 분석. 전산학부 (정보보호대학원) . 2022년. 20+iv 쪽. 지도교수: 김용대, 윤인수. (영문 논문)
Hyunsik Jeong. Analysis of the Correctness of Qualcomm Hexagon Emulators and Decompilers via Differential Testing. School of Computing (Graduate School of Information Security) . 2022. 20+iv pages. Advisor: Yongdae Kim, Insu Yun. (Text in English)

초 록

Qualcomm Hexagon은 범용 디지털 신호 프로세서를 위한 아키텍처로, 다양한 기기의 이동통신에 사용되고 있다. 그 대중성만큼 보안의 중요성이 대두됨에도 불구하고, Hexagon의 보안성 분석에 필수적인 에뮬레이터와 디컴파일러의 정확성에 대한 연구는 미비한 실정이다. 본 연구에서는 Hexagon 에뮬레이터와 디컴파일러로부터 버그를 자동으로 찾아내는 차등 테스트 도구를 제안한다. 이를 위해 Hexagon의 VLIW 구조로부터 기인하는 명령어의 제약 조건들을 알아보고, 이 제약 조건들을 준수하여 무작위의 명령어들을 생성할 수 있는 알고리즘을 고안하였다. 그리고 이를 통해 작성한 차등 테스트 도구를 Hexagon의 널리 알려진 두 에뮬레이터인 hexagon-sim과 quic/qemu, 그리고 디컴파일러인 binja-hexagon에 적용하였다. 그 결과 서로 다른 결과를 반환하는 약 15,000개의 명령어 입력들을 찾을 수 있고, 이를 분석하여 에뮬레이터와 디컴파일러의 정확성에 치명적인 4개의 버그를 발견했다.

핵 심 낱 말 퀄컴 헥사곤, 에뮬레이터, 디컴파일러, 차등 테스트, 아키텍처

Abstract

Qualcomm Hexagon is an architecture for general-purpose digital signal processors and is used for mobile communication of various devices. Despite the importance of security as much as its popularity, studies on the correctness of emulators and decompilers are insufficient, which are essential for Hexagon's security analysis. In this study, we propose a differential testing tool to automatically discover bugs in emulators and decompilers for Hexagon. To this end, we devise an algorithm to generate random instructions while ensuring constraints in Hexagon; Hexagon employs several constraints in its instruction because of its VLIW (very long instruction word) structure. Then, we applied our tool to two emulators for Hexagon (hexagon-sim and quic/qemu) and one decompiler, binja-hexagon from Google. As a result, we found about 15,000 instructions that cause inconsistent results among them. By analyzing these instructions, we could discover four bugs which are fatal to the correctness.

Keywords Qualcomm Hexagon, emulator, decompiler, differential testing, architecture

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Qualcomm Hexagon is an architecture for general-purpose digital signal processors (DSPs), designed for high performance and low power across a wide variety of multimedia and modem applications [14]. As Qualcomm has led the baseband market with a 52% revenue share in the second quarter of 2021 [3], it is clear that Hexagon is widely used in cellular machines and communications.

As the importance of Hexagon emerges, there have been studies on the Hexagon architecture: finding vulnerabilities in Qualcomm cDSPs [9], Qualcomm aDSPs [16], and Qualcomm basebands [2, 17, 19]. These studies focus on showing trials and errors and giving advices on analyzing and exploiting Qualcomm Hexagon chips, which are useful for vulnerability researchers.

However, it is difficult to find research on Hexagon on analyzing the correctness of emulators and decompilers on Hexagon. An emulator and a decompiler are two fundamental tools for binary analysis; Emulators can be used for testing programs before running on real Hexagon chips, and also for security analysis including fuzzing, dynamic testing and malware analysis. Decompilers are powerful tools for reverse engineers and security researchers, especially in unfamiliar architectures like Hexagon as it reduces time to understand the given instruction set architecture (ISA). If there is a bug in an emulator or a decompiler, it makes difficult to trust the result of emulation or decompilation. Therefore, there have been studies on checking whether an emulator runs as the given ISA [11, 10], or testing binary lifters [8]. Still, there was no such research on the Hexagon architecture before.

In this paper, we propose a differential testing tool to analyze the correctness of emulators and decompilers for Hexagon. This tool consists of two parts: generating random instructions and comparing the results from target emulators/decompilers. To generate random instructions, we analyze the Hexagon ISA and figure out constraints of instruction arrangement, which are derived from Hexagon's very long instruction word (VLIW) property. Based on the analysis, we devise an algorithm which generates random Hexagon instructions.

We applied this tool to two emulators and one decompiler on Hexagon: (1) hexagon-sim, the emulator included in the Hexagon SDK, (2) QEMU from Qualcomm Innovation Center (QUIC) [4], and (3) binja-hexagon, the decompiler from Google [7]. We chose these targets by usability in the real world. With this tool, it was able to find four bugs, which are previously unknown and fatal to the target emulators and the decompiler. These bugs are reported to the developers. The result shows that there may be incorrect implementations in Hexagon emulators and decompilers and it is able to find some of them with the tool.

Chapter 2. Background

2.1 Very Long Instruction Word

Instruction-level parallelism (ILP) is the concept of executing multiple instructions in parallel. This term has a different meaning from concurrency; ILP is limited to running multiple instructions in either one process or one thread in a single core. There have been several attempts to exploit the ILP, such as superscalar and very long instruction word (VLIW).

Superscalar is one of the most common microarchitectural techniques used in modern architectures. A superscalar CPU execute multiple instructions in parallel by using multiple execution units. To maximize the efficiency of using multiple execution units, superscalar CPUs usually have an issuer that passes each instruction to a proper execution unit which can handle it. One benefit of this structure is that it does not affect the instruction set architecture (ISA), which contains the definition of machine code. As a superscalar CPU automatically handles instructions to exploit ILP, programmers do not need to know how superscalar works to make a program.

Unlike superscalar, VLIW exploits ILP in a different way by changing the common ISA design. In a VLIW processor, there are multiple execution units like superscalar. However, a program must specify which operation will be executed on which execution unit. These operations are grouped into instructions; if there are n execution units in a processor, one instruction contains at most n operations which run in parallel. This idea moves a instruction scheduling problem from hardware to compilers and programmers.

The VLIW ISA has several restrictions on the arrangement of operations that does not exist in common ISAs such as IA-32 or ARM. These are called *constraints* in the Hexagon ISA. Constraints result from the structural limitations of processors; For example, it is impossible to put two unconditional branch operations in one instruction, as there is only one instruction pointer. Also, there can be other constraints as each execution unit may have specific roles. For example, if there is only one ALU execution unit in a processor, there can be at most one ALU operation in one instruction.

2.2 Hexagon ISA

In this section, we examine components of the Hexagon ISA. In the subsection 2.2.1, we look through the registers of Hexagon. In the subsection 2.2.3, we introduce and explain instruction packets and constraints.

2.2.1 General Registers

The Hexagon ISA has 32 general registers, and each register is 32-bit (`R0-R31`). Among the general registers, there are three registers used to support subroutines and stack. `R29` is used as the stack pointer, and has an alias `SP`. `R30` is used as the frame pointer, and has an alias `FP`. These can be compared to `sp` and `bp` registers in IA-32. `R31` stores the return address, and has an alias `LR`. For 64-bit computation, the general registers are grouped into register pairs. For example, the registers `R0` and `R1` form the register pair `R1:R0`. Table 2.1 shows such general register pairs for 64-bit computation.

Register	Register Pair
`R0`	`R1:0`
`R1`	
`R2`	`R3:2`
`R3`	
...	
`R28`	`R29:28`
`R29 (SP)`	
`R30 (FP)`	`R31:30 (LR:FP)`
`R31 (LR)`	

Table 2.1: General register pairs in Hexagon

2.2.2 Control Register

Hexagon also has 32 control registers (`C0-C31`) as well. Each control register has its own name, and `C`x is used as an alias. Table 2.2 shows the name of the control registers. The control registers provide access to processor features such as the program counter, hardware loops, and vector predicates. Among those, there are several control registers which are used frequently: (1) predicate registers, (2) addressing mode registers, and (3) the user status register.

Predicate Registers. The predicate registers (`P0-P3`, or `C4`) store the results of comparison instructions. Each predicate register is 8-bit, and stores one boolean (scalar predicate) or eight booleans (vector predicate). In the case of one boolean, the least significant bit (LSB) denotes true (1) and false (0). In the case of eight booleans, 1 denotes true and 0 denotes false, and it stores each boolean in one bit. The four predicate registers can be specified as a register quadruple (`P3:0`) which represents a 32-bit register.

Addressing Mode Registers. There are three types of control registers used for addressing mode: (1) modifier registers, (2) circular start registers, and (3) the global register. The modifier registers (`M0` (`C6`) and `M1` (`C7`)) are used in the following memory addressing modes:

- Indirect auto-increment register addressing (cf. `R3 = memw(R2++M1)`)
- Circular addressing (cf. `R0 = memb(R7++#4:circ(M0))`)
- Bit-reversed addressing (cf. `R0 = memub(R0++M1:brev)`)

Also, the circular start registers (`CS0` (`C12`) and `CS1` (`C13`)) are used in the circular addressing. The global register (`GP`, or `C11`) is used in GP-relative addressing (cf. `R2 = memw(GP+#200)`).

User Status Register. The user status register (`USR`) stores processor status control bits and flags which are accessible by user programs, including floating-point IEEE flags and a saturation overflow control bit.

2.2.3 Instruction Packets and Constraints

As mentioned in the section 2.1, the Hexagon ISA uses VLIW to exploit ILP but with different terms. In Hexagon, one single operation is called *an instruction*. A pack of instructions is called *an instruction packet*. Because Hexagon has four execution units, four slots exist in one instruction packet where each slot represents one execution unit. One instruction packet can contain from one instruction up to four

Register	Alias	Name
`SA0`	`C0`	Loop start address register 0
`LC0`	`C1`	Loop count register 0
`SA1`	`C2`	Loop start address register 1
`LC1`	`C3`	Loop count register 1
`P3:0`	`C4`	Predicate registers 3:0
`reserved`	`C5`	-
`M0`	`C6`	Modifier register 0
`M1`	`C7`	Modifier register 1
`USR`	`C8`	User status register
`PC`	`C9`	Program counter
`UGP`	`C10`	User general pointer
`GP`	`C11`	Global pointer
`CS0`	`C12`	Circular start register 0
`CS1`	`C13`	Circular start register 1
`UPCYCLELO`	`C14`	Cycle count register (low)
`UPCYCLEHI`	`C15`	Cycle count register (high)
`UPCYCLE`	`C15:14`	Cycle count register
`FRAMELIMIT`	`C16`	Frame limit register
`FRAMEKEY`	`C17`	Frame key register
`PKTCOUNTLO`	`C18`	Packet count register (low)
`PKTCOUNTHI`	`C19`	Packet count register (high)
`PKTCOUNT`	`C19:18`	Packet count register
`reserved`	`C20-29`	-
`UTIMERLO`	`C30`	Qtimer register (low)
`UTIMERHI`	`C31`	Qtimer register (high)
`UTIMER`	`C31:30`	Qtimer register

Table 2.2: Aliased control registers in Hexagon

instructions. Then, the instructions in a packet will run in parallel.

There are rules and restrictions on the arrangement of instructions in a packet, which are called *constraints*. In Hexagon, we have three constraints in the following.

Resource constraints. Resource constraints determine how many instructions of a specific type can appear in a packet. There are four execution units in Hexagon and each instruction can be executed on a particular type of unit. Table 2.3 shows what classes of instructions can appear in slot 0-3.

Slot	**Instructions**	
Slot 0	LD Instructions	ST Instructions
	ALU32 Instructions	MEMOP Instructions
	NV Instructions	SYSTEM Instructions
	Some J Instructions	
Slot 1	LD Instructions	ST Instructions
	ALU32 Instructions	Some J Instructions
Slot 2	XTYPE Instructions	ST Instructions
	J Instructions	Some J Instructions
Slot 3	XTYPE Instructions	ALU32 Instructions
	J Instructions	CR Instructions

Table 2.3: Available instruction classes in the slots

Grouping constraints. Grouping constraints restrict combination of specific instructions. For example, there are instructions called dot-new conditional instructions. The dot-new conditional instructions are a way to use the scalar predicate result from an instruction in the same packet. Check the following assembly code:

```
{
  P0 = cmp.eq(R2,#4)
  if (P0.new) R3 = memw(R4)
  if (!P0.new) R5 = #5
}
```

The dot-new conditional instructions presented in the code use `P0.new` which is the result from the first instruction `P0 = cmp.eq(R2,#4)`. `P0` is the *dot-new predicate register* of the given two instructions. The grouping constraint on dot-new conditional instructions ensures that there must be one instruction writing to the dot-new predicate register in the same packet.

Dependency constraints. Dependency constraints ensure that no more than one instruction writes on the same register. For example, `R0 = add(R2, R3)` and `R0 = sub(R4, R5)` cannot present in the same packet, as both writes `R0`.

Chapter 3. Design and Implementation

In this chapter, we discuss the design and implementation of the differential testing tool for emulators and decompilers on Hexagon. In the section 3.1, we describe the overview of the tool. In the section 3.2 and section 3.3, we devise an algorithm to generate random instruction packets which is used in the tool. In the section 3.4, we introduce the targets of the tool. Finally, we show challenges in implementation of the tool in the section 3.5.

3.1 Overview

The overall design of the tool is shown in Figure 3.1. The tool is written in Python, consists of three parts: (1) generating a random instruction packet, (2) running the instruction packet with the target emulators and decompilers, and (3) comparing the output CPU state.

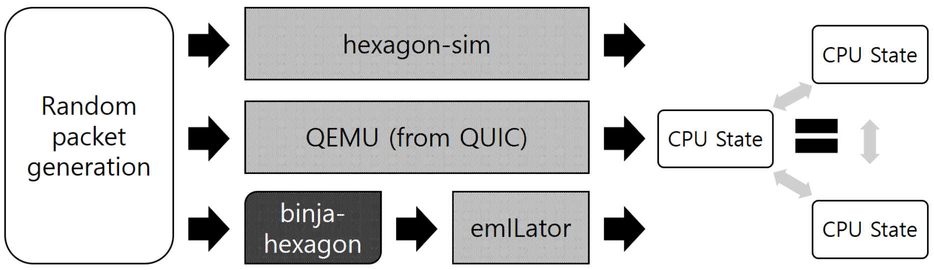


Figure 3.1: The overall design of the differential testing tool

The important part of the tool is the algorithm to generate random instruction packets. In the next two sections, we define an algorithm to generate a random instruction, then we extend the algorithm to generate a random instruction packet.

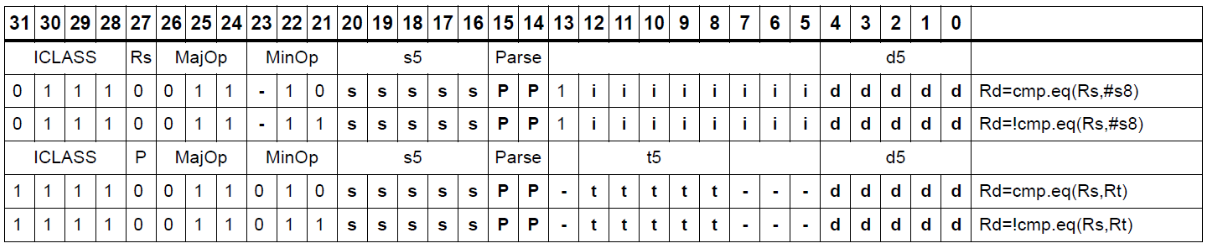
3.2 Random Instruction Generation Algorithm

To generate a random instruction, we first require the encoding of instructions. The encoding of all Hexagon instructions is defined in the manual [14]. For example, four "Compare to general register" instructions are interpreted as Figure 3.2.

There are total 12 symbols in the encoding, which represent operands and immediates in the instruction. Table 3.1 explains each symbol.

Any values can come to these symbols, except two cases. The first case is a parse bit `P`. Every instruction contains two parse bits, and `01` denotes a non-terminal instruction while `11` denotes a terminal instruction in the packet. `00` and `10` are for hardware loops, which is not covered in this paper. The second case is about register pairs. If the given operand denotes a register pair, the maximum value cannot come. For example, the maximum value in general registers is 31 and 31 cannot come to represent a register pair as there is no such pair `R32:31`.

Ensuring these two exceptions, it is possible to fill the symbols randomly. Thus, it completes the random instruction generation algorithm.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	1	1	-	1	0	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=cmp.eq(Rs,#s8)
0	1	1	1	0	0	1	1	-	1	1	s	s	s	s	s	P	P	1	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=!cmp.eq(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=!cmp.eq(Rs,Rt)

Figure 3.2: The encoding of "Compare to general register" instructions [14]. `s`, `d` and `t` denote the register operands `Rs`, `Rd` and `Rt`. `i` denote the intermediate value `#s8`. `P` denote the instruction position in the packet.

Symbol	Meaning
`P`	This symbol is called a parse bit, and denotes the instruction position in the packet. Every instruction contains two bits of `P`.
`s`, `t`, `u`, `v`	These symbols denote the source registers.
`d`, `e`	These symbols denote the destination registers.
`x`, `y`	These symbols denote the register which are both read and written.
`i`, `I`	These symbols denote the immediate values.
`N`	This symbol denotes the shifting amount of the result. (cf. `Rd=cmpy(Rs,Rt)[:<<N]:rnd:sat`)

Table 3.1: Symbols in instruction encodings

Python-like pseudocode of the algorithm is shown in Figure 3.3. Line 6-9 handles the parse bits `P`, and line 11 iterates the all 12 symbols except `P`. Line 13-16 handles the second exception about register pairs. Line 20-24 is used in random instruction packet generation.

3.3 Random Instruction Packet Generation Algorithm

Using the algorithm in the section 3.2, it is possible to design a naive algorithm for random instruction packet generation. However, there are the constraints discussed in the subsection 2.2.3. Nevertheless, we can define an algorithm ensuring the constraints step by step. We explain the algorithm with Python-like pseudocode shown in Figure 3.4 with the line numbers.

Resource Constraints. As each slot has the set of instructions which can come to the slot, it is possible to pass the slot information to the random instruction algorithm. In the random instruction generation algorithm, it handles the constraint in two steps: (1) Pick an instruction from the instruction set of the slot (Line 8), (2) Fill `11` in the parse bits if the slot is 0, and fill `01` if not (Line 10).

Dependency Constraints. To ensure dependency constraints, the random instruction algorithm must return the value of `d`, `e`, `x` and `y` which denote destination operands. If there is instructions writing to the same register, pick instructions again with the algorithm (Line 17-18).

Grouping Constraints. The process to ensure grouping constraints is similar to the process of the dependency constraints. Most of grouping constraints are related to instruction classes, so it is straightforward to check these constraints. For example, there is a constraint that the instruction in the

```
1  def generate_instruction(slot, instruction_set):
2      insn = pick_random_element(instruction_set)
3      written = {}
4      dot_new_predicate = null
5
6      if slot = 0:
7          insn.fill_field('P', 3)
8      else:
9          insn.fill_field('P', 1)
10
11     for symbol in [s, t, u, v, d, e, x, y, i, I, N]:
12         width = insn.field_length(symbol)
13         if insn.is_register_pair(symbol):
14             value = pick_random(range(0, 2^width - 2))
15         else:
16             value = pick_random(range(0, 2^width - 1))
17
18         insn.fill_field(symbol, value)
19
20         if symbol in [d, e, x, y]:
21             type = insn.get_register_type(symbol)
22             written = written |  {(type, symbol)}
23         if insn.is_dot_new_predicate(symbol):
24             dot_new_predicate = value
25
26     return insn, written, dot_new_predicate
```

Figure 3.3: The algorithm generating a random instruction

slot 0 must be ST-class if there is a ST instruction in the slot 1 (Line 21-22). The most tricky constraint is the constraint on dot-new conditional instructions, introduced in the subsection 2.2.3. To ensure this constraint, the random instruction algorithm also returns dot-new predicate registers (Line 14-15, 19-20).

```
def generate_instruction_packet():
    while True:
        written = {}
        written_num = 0
        required = {}
        packet = []
        for slot in range(0, 3):
            insn_set = get_instruction_set(slot)
            insn, written_part, dot_new_predicate = \\
                generate_instruction(slot, insn_set)
            packet.append(insn)
            written = written | written_part
            written_num += length(written_part)
            if dot_new_predicate != null:
                required = required | {(P, dot_new_predicate)}

        if length(written) != written_num:
            continue
        if length(required) != length(required | written):
            continue
        if not check_grouping_constraints(packet):
            continue

        return packet
```

Figure 3.4: The algorithm generating a random instruction packet

3.4 Targets

3.4.1 Target Emulators/Decompilers

We chose two emulators and one decompiler as the targets of the differential testing tool. The emulators are both from Qualcomm. The first emulator is hexagon-sim, which is included in the Hexagon SDK. It supports the full latest Hexagon ISA (Hexagon V67). The second emulator is QEMU from Qualcomm Innovation Center (QUIC) [4]. It is still under development, but supports a large portion of the latest Hexagon ISA.

The only decompiler, binja-hexagon from Google [7], is chosen by usability in the real world. There are few decompiler projects on Hexagon, and binja-hexagon is the only one supports instructions which are enough to represent basic programs such as printing `"Hello, world!"`. Still, it does not support advanced instructions including some `SYSTEM` instructions and vector computation.

As binja-hexagon emits the decompilation result in the Binary Ninja IL format, we use emILator [18] to emulate the given result.

3.4.2 Target Instructions

Table 3.2 shows the instruction classes of Hexagon and which classes are selected as the target of the tool. There are several unchosen classes because of difficulty of handling the instruction pointer (`J` and `JR`), running system calls (hardware loop related `CR` instructions and `SYSTEM`) and tracking memory addresses (`NV` and `MEMOP`).

Instruction Class	Target	Description
`ALU32` and `XTYPE`	O	Basic computing instructions
`CR` (except hardware loops)	O	Instructions for handling control registers
`ST` and `LD`	O	Basic memory instructions
`J` and `JR`	X	Branch instructions
`SYSTEM`	X	System-related instructions
`NV` and `MEMOP`	X	Advanced memory instructions
`CR` (hardware loops)	X	Instructions for handling the hardware loop feature

Table 3.2: Target instruction classes of the tool

3.5 Implementation

In this section, we look through challenges in implementation. In the subsection 3.5.1, we describe the way to get the information of all Hexagon instructions described in the programmers' manual. In the subsection 3.5.2, we describe challenges in program generation to execute the packet with the target emulators and decompilers.

3.5.1 Instruction Set Parsing

As mentioned in the section 3.2, the encoding of all Hexagon instructions is defined in the manual [14]. However, it is distributed in the `pdf` format, and it is a tough work to extract the encoding information from the `pdf` file to a programmable format. Fortunately, there is a project moved all the encoding information into `csv` files [6], using a table extraction tool called Tabula [15].

Also, we build regular expressions to extract information from the instruction syntax which are used in the random packet generation. For example, this is the regular expression to extract the addressing mode:

```
memu?b?[bhwd](?_fifo)?\(([^\(\)]*(?:\(Mu\))?)\)
```

`memu?b?[bhwd](?_fifo)?` handles the front part such as `memu`, `membh` or `memh_fifo`, which shows the length and the signedness. `([^\(\)]*(?:\(Mu\))?)` handles the inner part. Various operations can come here, such as `(##address)`, `(Rs+Ru<<#u2)` or `(Rx++I:circ(Mu))`.

3.5.2 Program Generation

To run an instruction packet with the emulators, we build a sample program shown in Figure 3.6. It is compiled with the llvm-musl-hexagon toolchain.

There are two difficulties in handling memory-related instructions: (1) the memory mapping of hexagon-sim and QEMU differs; (2) hexagon-sim does not allow to map a memory area in the specific address. In other words, `mmap` with `MAP_FIXED` does not work.

Therefore, we use one trick in the code. As the stack address of `main` always starts from `0x410eef4` in hexagon-sim, mmaping the area starting from `0x4100000` can create the area (`0x410eee0`-`0x410eef0`) which is both accessible from hexagon-sim (stack memory) and QEMU (mmaped memory). This trick is shown in line 7-15 of Figure 3.6.

To get the decompilation result of a packet in binja-hexagon, we write a simple assembly code shown in Figure 3.5. It is also compiled with the llvm-musl-hexagon toolchain.

```
.text
.globl main
.type main,@function
.type temp,@function
main:
    {
        nop, nop, nop, nop // Insert the packet here
    }
temp:
    {
        nop, nop, nop, nop
    }
```

Figure 3.5: The sample program code for binja-hexagon

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>

int main()
{
    // Only used in quic/qemu
    int *ptr = (int*) mmap(
        (void *)0x4100000, 65536, PROT_READ | PROT_WRITE,
        MAP_FIXED | MAP_PRIVATE | MAP_ANON,-1, 0
    );
    // Only used in hexagon-sim
    unsigned int mem_sim[4] = {0};

    unsigned int *mem = (unsigned int*) 0x410eee0;
    unsigned int inputs[21] = {};
    read(0, inputs, sizeof(inputs));
    print_state(inputs, mem);
    __asm(
        "{"
        "R10 = %0;"
        "} {"
        "R0 = memw(R10 + #0);"
        "R1 = memw(R10 + #4);"
        "} {"
        ... // omitted
        "} {"
        "C13 = R17;"
        "} {"
        " <target instruction packet> " // Insert the packet here
        "} {"
        "R11 = C4;"
        "} {"
        ... // omitted
        "} {"
        "memw(R10 + #32) = R8;"
        "memw(R10 + #36) = R9;"
        "}"
    : : "r" (inputs));
    print_state(inputs, mem);
}
```

Figure 3.6: The sample program code for hexagin-sim and QEMU

Chapter 4. Evaluation

In this chapter, we show our evaluation environment and results. In the section 4.1, we specify the environment including the version of the target emulators and decompiler. In the section 4.2, we show how many bugs are found and describe each bug.

4.1 Environment

We tested the target emulators and decompiler with the version:

- hexagon-sim (In HEXAGON Tools 8.3.07)
- QEMU (Nov 1, 2021, commit hash `94ca4341`)
- binja-hexagon (Oct 20, 2021, commit hash `31993a3a`)

We ran the tool for a day on 64-bit Ubuntu 20.04 system, with 20-core Intel i9-10900K CPU with 98GiB RAM.

4.2 Results

It was able to get about 15,000 inputs that generate different results with the tool. By analyzing the inputs, we found 4 bugs which are shown in Table 4.1.

Emulator/ Decompiler	**Condition**	**Bug**
hexagon-sim	Various memory instructions	hexagon-sim crashes
QEMU	`LD` instruction with the predicate register in the slot 0, `ST` instruction in the slot 1	The slot-1 instruction is not executed, when the predicate register of the slot-0 instruction is false.
QEMU	`Rd=convert_df2uw(Rss):chop` `Rd=convert_sf2uw(Rs):chop`	Does not set IEEE invalid sticky flag when the input causes integer overflow
binja-hexagon	Instructions with the predicate register	Regard all the non-zero values as true

Table 4.1: Bugs found with the differential testing tool

Memory Instruction Bug in hexagon-sim. About 100 memory instructions cause a crash in hexagon-sim. For example, `Rdd=membh(Rx++#s4:2)` causes a crash when `Rx=0x410eee0` and `#s4=5`, even though the address `0x410eee0` is accessible.

ST-LD Instruction Bug in QEMU. This bug occurs in certain conditions: (1) A `LD` instruction with the predicate register in the slot 0; (2) A `ST` instruction in the slot 1; (3) The predicate register used by the `LD` instruction must be false. Then, the `ST` instruction in the slot 1 is not executed even if it is supposed to be executed.

```
{
  if (P2)memh(R7++#-2s4:1)=R4
  if (!P2)R1=memw(R6+R8<<#2u2)
}
```

Figure 4.1: An example for ST-LD instruction bug in QEMU

See the example in Figure 4.1. When `P2` is true, the instruction `if (P2)memh(R7++#-2s4:1)=R4` must be executed. However, as `if (!P2)R1=memw(R6+R8<<#2u2)` is not executed, the instruction `if (P2)memh(R7++#-2s4:1)=R4` is not executed in QEMU. The PoC input for the example is in Table 4.2.

Register/ Memory	Input	Output (hexagon-sim)	Output (QEMU)
R1	0xc12e402a	0xc12e402a	0xc12e402a
R4	0x6a4f5bc1	-	-
R6	0x0838a31c	-	-
R7	0x0410eee0	-	-
R8	0xfef612f1	-	-
P3:0	0x00e167d5	-	-
Memory (0x410eee0)	0xd0afb236	**0xd0af5bc1**	**0xd0afb236**

Table 4.2: The PoC input for the packet from Figure 4.1

IEEE Invalid Sticky Flag Bug in QEMU. Both `Rd=convert_df2uw(Rss):chop` and `Rd=convert_sf2uw(Rs):chop` converts the floating-point value in `Rs`/`Rss` to the unsigned integer and stores it in `Rd`. When the floating-value is too large to be expressed in 32-bit integer, it must raise the IEEE invalid sticky flag [1]. However, it is not set in QEMU.

Predicate Register LSB Bug in binja-hexagon. As mentioned in the Table 2.2.2, the predicate register is true when the LSB is set to 1. However, the predicate register is considered true when the value is not zero in binja-hexagon.

Chapter 5. Discussion

Grouping Inputs. Differential testing was powerful enough to find inputs that generate different outputs from the target emulators and decompiler. However, some bugs generate numerous inputs that disturb to analyze the bugs from the inputs. For example, the memory instruction bug in hexagon-sim generated more than 100 inputs. This causes extra time to find bugs which emits smaller number of inputs such as the IEEE invalid sticky flag bug. If there is a way to organize the inputs with certain rules, it can be effective in future works.

Instruction Classes. In this paper, we did not cover several instruction classes. Among these classes, `NV` and `MEMOP` can be the possible targets to be tested next. An algorithm to track the target address from an instruction in these classes is required to test these classes. `J` and `JR` are challenging, as there is a problem with memory mapping in hexagon-sim which is mentioned in the subsection 3.5.2. As the stack is not executable, there should be another bypass to make a executable memory area with the same address. Hardware loop related `CR` instructions and `SYSTEM` instructions are special cases which are difficult to do differential testing. Another testing method is recommended for these instructions.

Input Generation. In the differential testing tool proposed by the paper, the value of registers is randomly picked from the possible values. If it is possible to consider edge cases, more bugs may be found by differential testing. For example, `NaN` and `inf` in floating point may cause bugs with floating-point instructions.

Chapter 6. Related Works

There has been studies on finding vulnerabilties from Hexagon DSPs: Qualcomm cDSPs [9], Qualcomm aDSPs [16], and Qualcomm basebands [2, 17, 19]. These studies focus on (1) reverse engineering of Hexagon firmwares and libraries, (2) understanding the properties of Hexagon DSPs (cf. ASLR), and (3) exploitation of found vulnerabilities. Some studies discuss a way to fuzz on Hexagon [9, 16] and doing return-oriented programming in Hexagon [19].

Differential testing is the term first used by McKeeman in 1998 [12]. To test C compilers, he made program generators ensuring the C standard, from level 1 (random ASCII characters) to level 7 (model-conforming C program). After his work, there has been research to improve differential testing using the coverage [5], or differences between the program [13].

In the area of binary lifting, there was an interesting attempt to use a symbolic equivalence check [8]. In this work, they generate a sequence of instructions from the given ISA, and find semantic differences between intermediate representations. This can be compared with differential testing. In differential testing, target programs run multiple inputs to find differences in outputs. In this work, it finds semantic differences in lifted outputs using a symbolic analysis.

Chapter 7. Conclusion

In this paper, we proposed the random instruction packet generation algorithm. This can be widely used in future Hexagon research, including random software testing and fuzzing. We also implemented the differential testing tool for Hexagon emulators and decompilers using this algorithm.

We evaluated our tool with two emulators and one decompiler: 1. hexagon-sim from the Hexagon SDK, 2. QEMU from Qualcomm Innovation Center (QUIC) [4], and 3. binja-hexagon from Google [7]. By using the tool, about 15,000 inputs causing different results were found, and 4 bugs were discovered by analyzing these inputs. These bugs are critical in real use cases. For example, several memory instructions can occur crashes in hexagon-sim. The bugs are all reported to the developers of the target programs.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Seamus Burke. A Journey Into Hexagon Dissecting a Qualcomm Baseband. In *DEF CON*, 2018.

[3] Business Wire. Strategy Analytics: Qualcomm Dominates With 52 Percent Revenue Share in Cellular Basebands in Q2 2021, 2021.

[4] Qualcomm Innovation Center. quic/qemu. `https://github.com/quic/qemu`.

[5] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 85–99, 2016.

[6] Comsecuris. Comsecuris/qemu. `https://github.com/Comsecuris/qemu-hexagon`.

[7] Google. google/binja-hexagon. `https://github.com/google/binja-hexagon`.

[8] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364, 2017.

[9] Slava Makkaveev. Pwn2Own Qualcomm Compute DSP for Fun and Profit. In *DEF CON*, 2020.

[10] Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, and Danilo Bruschi. A Methodology for Testing CPU Emulators. 22(4), oct 2013.

[11] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU Emulators. ISSTA '09, page 261–272, 2009.

[12] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[13] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[14] Qualcomm. Qualcomm Hexagon V67 Programmer's Reference Manual; 80-N2040-45 Rev. B, 2020.

[15] Tabula. Tabula: Extract Tables from PDFs. `https://tabula.technology/`.

[16] Dimitrios Tatsis. Attacking Hexagon: Security Analysis of Qualcomm's aDSP. In *Recon*, 2019.

[17] Tencent Blade Team. Exploring Qualcomm Baseband via ModKit. In *CanSecWest*, 2018.

[18] Josh Watson. joshwatson/emilator: A Low Level IL emulator for Binary Ninja. `https://github.com/joshwatson/emilator`.

[19] Ralf-Philipp Weinmann. Baseband exploitation in 2013: Hexagon challenges. In *Pacsec*, 2013.

Acknowledgments in Korean

석사과정의 2년은 저에게 있어서 정말 다사다난한 2년이었습니다. 그런 힘든 시간 속에서도 저에게 도움을 주신 분들이 있었기에 이렇게 결실을 볼 수 있었습니다. 그분들에게 감사의 인사를 드리고자 합니다.

먼저 석사과정 동안 이끌어주시고 도와주신 김용대 지도교수님께 마음속 깊이 감사드립니다. 교수님을 만났기에 다양한 주제를 접하고, 연구에 대해서 폭넓게 배울 수 있었습니다. 제가 힘든 시기에도 깊이 이해해주시고, 연구 외적으로도 많은 도움을 주셨기에 이 자리까지 올 수 있었습니다. 또한 윤인수 교수님께도 깊은 감사의 말씀을 드립니다. 교수님이 계셨기에 지금의 연구 주제를 정할 수 있었고, 연구 방향성에 대해서도 감을 잡을 수 있었습니다.

2년의 세월 동안 힘이 되어준 우리 SysSec 연구실 동료분들도 있습니다. 항상 질문을 받아주시고 연구 내적으로나 외적으로나 많은 도움을 주신 송창훈 선배, 날카로운 지적과 통찰력을 꼭 배우고 싶은 김동관 선배, 역분석 실력이 귀감이 되는 김은수 선배, 그리고 이분들을 비롯한 소프트웨어 분석팀 여러분과 함께 해서 석사과정을 보람차게 보낼 수 있었습니다. 비록 코로나바이러스로 인해 자주 만나 뵙지는 못했지만 도와주신 모든 연구실 동료 분들께도 감사의 말씀 드립니다.

마지막으로 항상 위기상황을 극복할 수 있게 도와준 구동욱, 마음의 버팀목이 되어준 김종호 두 친구에게 고마움을 전합니다. 또한 주말마다 같이 대회를 치르며 시간을 보내준 perfect blue 팀원들에게도 감사를 표합니다.

이외에도 미처 적지 못했지만 도움을 주신 많은 분이 더 계십니다. 도와주신 모든 분의 도움을 잊지 않고 꾸준히 정진하도록 하겠습니다. 감사합니다.

Curriculum Vitae in Korean

이　　　름:	정현식
생 년 월 일:	1997년 10월 26일
전 자 주 소:	me@rb-tree.xyz

학　　력

2020. 3. – 2022. 2.	한국과학기술원 정보보호대학원 (석사 예정)
2015. 3. – 2020. 2.	포항공과대학교 창의IT융합공학과 및 컴퓨터공학과 (학사)
2012. 3. – 2015. 2.	서울과학고등학교 (졸업)

연 구 업 적

1. **정현식**, 윤인수, 김용대, “차등 테스트를 통한 Qualcomm Hexagon 에뮬레이터 분석,” 한국정보보호학회 하계학술대회, 2021년 6월.