한국컴퓨터정보학회 하계학술대회 논문집 제26권 제2호 (2018. 7)

바이너리 분석을 위한 ARM 명령어 구조 분석

정승일[O], 류찬호[*]
[O*]한국과학기술원 사이버보안연구센터
e-mail: {sijung, choryu}@kaist.ac.kr[O*]

ARM Instruction Set Architecture Analysis for Binary Analysis

Seungil Jung[O], Chanho Ryu[*]
[O*]Dept. of Cyber Security Research Center, KAIST

● 요 약 ●

본 논문에서는 바이너리 분석을 위한 ARM의 구조를 분석한다. 바이너리 분석이란 0과 1로 이루어진 이진 값의 의미를 분석하는 것을 말한다. 바이너리 코드를 역어셈블(Disassemble)하여 값으로만 존재하는 데이터가 어떤 명령어(Instruction)이며 어떤 피연산자(Operand)를 의미하는지 알 수 있다. 소스코드를 컴파일하여 실행파일이 생성이 되면 바이너리 값으로 구성되며 이 실행파일을 바이너리 파일이라고도 한다. 바이너리 파일을 분석하기 위해서 CPU의 명령어 집합 구조(Instruction Set Architecture)를 알아야 한다. PC와 서버, 모바일 등에서 많이 사용되고 있는 ARM 중에서 64비트를 지원하는 AArch64(ARMv8)의 명령어 구조를 분석하여 효율적인 바이너리 분석의 기반을 마련하고자 한다.

키워드: 바이너리 분석(binary analysis), ARM 구조(ARM Architecture), ARM 명령어 집합(ISA)

I. Introduction

바이너리 분석은 정보보호에서 중요한 연구 분야 중 하나이다. 각종 위협이 되는 악성코드는 결국 실행파일이 실행이 되어야 피해를 줄 수 있다. 따라서 악성코드를 사전에 분석하게 되면 피해를 막을 수 있다. 실행파일의 동작을 분석하기 위해서 가장 쉬운 방법은 소스코드를 분석하는 방법이지만 소스코드를 확보하는 것은 어렵다. 그러나 바이너리 파일을 확보하는 것은 실행파일이 실행되기 위해서는 장치에 있어야하기 때문에 어렵지 않다. 따라서 실행파일을 확보하여 분석을 할 수 있는 기술이 있다면 사전에 악성코드를 탐지할 수 있다. 현재는 이러한 분석을 위해서 분석가가 파일을 분석하여 악성 여부를 검사하고 있다. 분석가들이 실행 파일 분석을 위해 널리 쓰이고 있는 'IDA Pro'는 이러한 분석을 돕는 도구이다. 그러나 'IDA Pro'는 어디까지나 분석가를 돕는 수동적인 도구이다. 바이너리 파일을 역어셈블(disassemble)하고 함수들의 흐름을 보여주는 등의 분석 지원 도구이다. 그러므로 이런 한계를 넘어서서 악성코드의 존재 여부를 알 수 있는 자동화된 분석 도구가 있다면 분석가가 직접 분석하는 것보다 빠르게 악성코드를 탐지하여 피해를 막을 수 있다. 이러한 도구 개발을 위해서 바이너리 분석은 필수적이다.

다양한 CPU 중에서 PC와 서버 및 모바일에서 사용되고 있는 ARM 아키텍처의 분석을 통하여 이러한 바이너리 분석 도구를 개발할 수 있는 기반을 다지고자 한다. ARM의 다양한 버전 중에서 주요 타깃은 64비트를 지원하는 AArch64(ARMv8)을 분석하고자 한다.

2장에서는 바이너리 분석 플랫폼 개발과 관련된 연구를 알아보고, 3장에서는 ARM의 구조를 분석하고 분석을 위해 구현한 함수를 소개한다.

II. Preliminaries

1. Related works

1.1 국내 동향

국내에서는 현재 공개된 바이너리 코드 분석 연구 및 도구 개발은 알려진 바가 없는 상황이다. 중요한 분야이지만 관심도 미미한 수준이다. 바이너리 코드 분석은 아니지만 취약점을 탐지하기 위하여 서울대학교의 Sparrow[1]와 고려대학교의 IoT Cube[2] 및 KISA 등에서 연구를 진행 중에 있다.

1.2 해외 동향

해외에서는 바이너리 코드 분석 연구가 활발하게 진행되고 있다. 미국을 비롯하여 유럽의 나라들이 바이너리 코드 분석을 연구하거나 도구를 개발하고 있다. 대표적으로 미국의 카네기 멜론 대학 (Carnegie Mellon University)의 BAP (Binary Analysis Platform)[3][5]과 미국의 UCSB (University of California, Santa Barbara)에서 개발한 PyVEX[4] 가 있다. 두 도구는 단순히 역어셈블만 하는 것이 아니라 중간 언어 (Intermediate Representation)를 표현하는 바이너리 코드 분석 플랫폼이다. 아래 그림은 '0x0001'에 대한 BAP과 PyVEX의 실행 예시 화면이다. 명령어의 동작을 중간언어로 표현한 것이다. 분석 플랫폼마다 중간언어로 변환하기 위한 레지스터나 메모

리 등의 표현 방법이 다르지만 동일한 동작을 표현한 것이다.

```
t0:Ity_I8 t1:Ity_I8 t2:Ity_I8 t3:Ity_I32 t4:Ity_I32 t5:Ity_I32 t6:Ity_I32

   00 | ------ IMark(0x400400, 2, 0) ------
   01 | t3 = GET:I32(ecx)
   02 | t2 = LDle:I8(t3)
   03 | t1 = GET:I8(al)
   04 | t0 = Add8(t2,t1)
   05 | STle(t3) = t0
   06 | PUT(cc_op) = 0x00000001
   07 | t4 = 8Uto32(t2)
   08 | PUT(cc_dep1) = t4
   09 | t5 = 8Uto32(t1)
   10 | PUT(cc_dep2) = t5
   11 | PUT(cc_ndep) = 0x00000000
   NEXT: PUT(eip) = 0x00400402; Ijk_Boring
```

```
ADD8mr(ECX,0x1,Nil,0x0,Nil,AL)
{
  v1 := mem[ECX]
  v2 := low:8[EAX]
  mem := mem with [ECX] <- mem[ECX] + v2
  CF := mem[ECX] < v1
  OF := high:1[v1] = high:1[v2] & (high:1[v1] ^ high:1[mem[ECX]])
  AF := 0x10 = (0x10 & (mem[ECX] ^ v1 ^ v2))
  PF := ~low:1[let v3 = mem[ECX] >> 4 ^ mem[ECX] in
    let v3 = v3 >> 2 ^ v3 in
    v3 >> 1 ^ v3]
  SF := high:1[mem[ECX]]
  ZF := 0 = mem[ECX]
}
```

Fig. 1. BAP (Top) vs PyVEX (Bottom)

III. The Proposed Scheme

AArch64를 분석하기 위해서 명령어의 구조와 피연산자의 구성요소를 알아야 한다. 그리고 그 구조를 파악하면 분석에 필요한 함수들을 구현할 수 있다. 명령어의 구조와 구현 함수를 소개하고자 한다.

1. AArhc64 명령어 구조

명령어의 구조는 Opcode(Operation Code)와 피연산자(Operand)로 구성된다. 명령어는 동작을 정의하고 피연산자는 그 동작에 필요한 정보를 가지고 있다.

아래 그림은 명령어와 구조와 예시이다. 명령어는 ADD이고 피연산자로는 두 개의 레지스터와 상수 값이 있다(LSL 왼쪽 시프트 연산자로 앞에 있는 상수 값에 적용되어 결국 상수 값이 된다).

[명령어] <Operand1>, <Operand2> ... <Operand5>

ex) ADD W26, W5, #0x371, LSL #12

Fig. 2. The instruction set

AArch64는 64비트 명령어를 지원하기 때문에 이전 버전과 레지스터 크기나 메모리 모드 및 인코딩된 비트의 필드 값을 표현하는 방법에 차이가 있다. 따라서 AArch64의 피연산자에는 어떤 종류가 있는지 알아야 한다. AArch64를 구성하는 피연산자(Operand)의 종류는 다음과 같다.

1.1 레지스터 (Register)

AArch64의 레지스터는 31개의 일반 목적 레지스터(General-purpose Regiser, GPR)와 SP(Stack Pointer) 레지스터, PC(Program Counter), 그리고 32개의 SIMD&FP 레지스터가 있다. 그 외에도 상태 관련 레지스터 및 시스템 레지스터 등이 존재한다. 아래 그림은 GPR 레지스터의 네이밍[6]에 대한 그림이다.

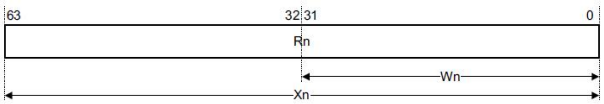

Fig. 3. General-purpose register naming

GPR의 경우 64비트의 Rn이라는 레지스터가 있고 표현을 할 때에는 32비트 레지스터는 Wn, 64비트 레지스터는 Xn으로 표현한다.

1.2 메모리 (Memory)

메모리는 주소에 따라 값을 읽거나 저장할 수 있다. 메모리의 동작에 따라 5가지의 어드레싱 모드가 존재하며, 각 어드레싱 모드는 3가지의 오프셋(immediate, register, extended register)이 존재할 수 있다. 구체적인 어드레싱 모드의[6] 형태는 아래 표와 같다.

Table 1. A64 Load/Store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	[base{, #0}]	-	-
Base plus offset	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S|U)XTW {#imm}]
Pre-indexed	[base, #imm]!	-	-
Post-indexed	[base], #imm	[base], Xm[a]	-
Literal (PC-relative)	label	-	-

1.3 SIMD and Floating-point

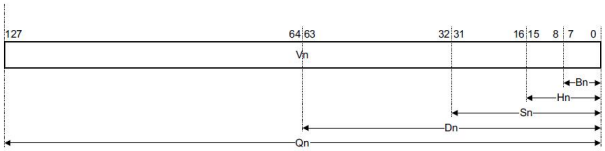

Fig. 3. SIMD and floating-point register naming

SIMD&FP 레지스터의 경우 128비트 공간의 크기에 따라서 8비트는 Bn, 16비트는 Hn, 32비트는 Sn, 64비트는 Dn, 128비트는 Qn으로 표현한다.

1.4 Immediate

Immediate는 양수 또는 음수인 값 이다. 레지스터에 저장될 수도 있고 주소 값이 되거나 주소 값에 더해지는 값이다.

2. Decode Instruction Set

2.1 Instruction Set

바이너리 코드를 분석하기 위해 기본적으로 바이너리가 어떤 포맷으로 인코딩되어 있는지 알아야 한다.

AArch64의 명령어 포맷은 32비트로 고정되어 있다. 이 중에서 [28:25](28비트에서 25비트 값을 의미)인 4비트 크기의 필드가 디코딩의 시작점이 된다. 예를 들어 4비트 필드가 'x1x0'이면 'Loads and stores' 관련 명령어라는 뜻이다.

Fig. 5. The encoding of an A64 instruction

4비트 디코드 필드에 따른 디코딩 그룹 및 명령어[6] 정보는 아래 표와 같이 구분된다.

Table 2. Main encoding table for the A64 instruction set

Decode fields op0	Decode group or instruction page
00xx	Unallocated.
100x	*Data processing - immediate* on page C4-193
101x	*Branches, exception generating and system instructions* on page C4-197
x1x0	*Loads and stores* on page C4-202
x101	*Data processing - register* on page C4-224
0111	*Data processing - SIMD and floating point* on page C4-233
1111	*Data processing - SIMD and floating point* on page C4-233

2.2 Decode Operation Code

Operation Code(이하 Opcode)는 CPU에 동작을 전달하는 명령어 코드이다. 예를 들어 'ADD', 'MOV'와 같이 값을 더하거나 옮기는 동작을 의미한다. Opcode를 디코딩하기 위해서는 위의 표와 같이 비트필드마다 의미를 정의한 표를 참고하여 순차적으로 디코딩 하면 최종적으로 Opcode를 찾게 된다.

2.3 Decode Operand

Opcode 정보를 디코딩하면 해당 명령어의 매뉴얼에[6] 피연산자에 대한 필드 정보를 확인 할 수 있다. 첫 번째 피연산자의 정보는 몇 번째 비트 값을 확인하면 되는지 그리고 그 값은 어떻게 해석되는지 확인 할 수 있다. 이런 과정을 거쳐 두 번째, 세 번째 피연산자의 정보를 확인하여 Opcode와 피연산자를 디코딩 할 수 있다.

2.4 Operation

Opcode와 Operand가 무엇인지 디코딩이 완료되면 해당 명령어가 어떤 동작을 하는지를 분석해야한다. 이 동작에 대한 설명 또한 'ARM Architecture Reference Manual'[6]에 설명되어 있다. Opcode와 피연산자 정보를 디코딩하는 역어셈블만으로는 명령어의 동작을 정확히 알 수 없다. 명령어가 어떤 동작을 하는지 까지 분석이 되어야한다. 아래 그림은 'ADD(immediate)' 명령어의 동작(Operation)을 설명하고 있다.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Fig. 6. Operation of ADD(immediate) instruction

3. Decode Function

앞서 분석된 내용을 기반으로 명령어 디코딩 관련 함수를 구현하였다. Opcode와 피연산자를 디코딩하는 함수, 그리고 각 비트 단위의 의미를 파악하기 위해 해당 비트 범위의 값을 가져오는 함수를 구현하였다. 구현은 함수형 언어인 F# 으로 구현하였다. 닷넷 프레임워크에서 동작하며 닷넷이 지원되면 다양한 플랫폼에서 동작되는 특징을 가지고 있다.

3.1 Decode function Opcode and Operand

Opcode와 피연산자를 디코딩하기 위해 op0[28:25] 필드의 값을 확인하는 메인 함수는 아래 그림과 같다.

```
let decodeAArch64 binary =
  let op0 = extractField binary 28 25
  match op0 with
  | op0 when op0 &&& 0b1100 = 0b0000 -> failwith "Unallocated."
  (* Data processing - immediate *)
  | op0 when op0 &&& 0b1110 = 0b1000 -> dataProcessingImmediate binary
  (* Branches, exception generating and system instructions *)
  | op0 when op0 &&& 0b1110 = 0b1010 -> branchAndExceptionAndSystem binary
  (* Loads and stores *)
  | op0 when op0 &&& 0b0101 = 0b0100 -> loadAndStores binary
  (* Data processing - register *)
  | op0 when op0 &&& 0b0111 = 0b0101 -> dataProcessingRegister binary
  (* Data processing - SIMD and floating point *)
  | op0 when op0 &&& 0b1111 = 0b0111 -> dataProcessingSIMDFP1 binary
  (* Data processing - SIMD and floating point *)
  | op0 when op0 &&& 0b1111 = 0b1111 -> dataProcessingSIMDFP2 binary
  | _ -> failwith "Invalid decode fields"
```

Fig. 7. Decode Main Function

decodeAArch64 함수는 [28:25]의 비트 필드 값을 확인하여 디코드 그룹(분류된 명령어 그룹)을 확인한다. Intel의 CISC(Complex Instruction Set Computer)와 순차적으로 파싱할 수 없다. 명령어마다 각각의 필드에 의미가 부여되며 모든 바이너리 전체를 확인해야 최종적으로 Opcode를 알 수 있다.

```
let parseAddSubtractImmediate binary =
  let decodeField = extractField binary 31 29 (* sf:op:S *)
  match decodeField with
  | value when value &&& 0b000 = 0b000
              && (extractField binary 23 22 >>> 1) = 0b1 ->
    failwith "Unallocated."
  | 0b000 -> ADD, decodeWdWnImmShf binary
  | 0b001 -> ADDS, decodeWdWnImmShf binary
  | 0b010 -> SUB, decodeWdWnImmShf binary
  | 0b011 -> SUBS, decodeWdWnImmShf binary
  | 0b100 -> ADD, decodeXdXnImmShf binary
  | 0b101 -> ADDS, decodeXdXnImmShf binary
  | 0b110 -> SUB, decodeXdXnImmShf binary
  | 0b111 -> SUBS, decodeXdXnImmShf binary
  | _ -> failwith "Invalid decodeFields"
```

Fig. 8. Decode function(Add/subtract (immediate))

위 그림의 ADD/ADDS/SUB/SUBS 명령어를 확인 할 수 있는 소스코드 이다. 명령어를 확인하면 어떤 형태로 피연산자가 구성되어 있는지 확인 할 수 있으며 또한 필드 값을 확인하여 피연산자가 레지스터인지 메모리인지 등을 확인 할 수 있다.

3.2 Extract Field function

명령어마다 각 비트 필드의 의미가 다르다. 따라서 비트 필드의 특정 범위의 값을 확인할 필요가 있다. 그러기 위해서 분석하고자 하는 32비트 바이너리와 확인하고자 하는 비트의 시작과 끝의 위치가 주어지면 해당 값을 가져오는 함수를 구현하였다. 함수의 소스코드는 아래 그림과 같다. extractField 함수의 사용 예시는 앞서 구현된 함수에서 사용되었다.

```
let extractField (binary: int) n1 n2 =
  let range = n1 - n2
  assert (range < 32)
  let mask = bigint.Pow (2I, range + 1) - 1I
  bigint (binary >>> n2) &&& mask |> int
```

Fig. 9. Extract Field function

IV. Conclusions

바이너리 분석은 정보보호 분야에서 핵심 기술 중 하나이지만 해결되어야할 난제들이[7] 존재한다. 그렇기 때문에 접근이 쉽지 않아 국내에서 연구가 활발히 진행되고 있지 못하는 이유 중 하나이다. 그러므로 바이너리 분석 연구가 발전이 되기 위해서 기초를 준비하는 것이 중요하다. 앞서 소개한 AArch64에 대한 구조의 이해와 동작을 분석하는 기술들이 더욱 발전하여 향후 바이너리 분석 플랫폼의 기반이 될 것으로 기대해 본다.

ACKNOWLEDGEMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2016-0-00102, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis).

REFERENCES

[1] “Sparrow,” http://ropas.snu.ac.kr/sparrow/.
[2] “IoTcube,” https://iotcube.korea.ac.kr/.
[3] “BAP,” https://github.com/BinaryAnalysisPlatform/bap.
[4] “PyVex,” https://github.com/angr/pyvex.
[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A binary analysis platform,” in Proceedings of the International Conference on Computer Aided Verification, 2011, pp. 463–469.
[6] “ARM architecture reference manual,” https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.
[7] Sang Kil Cha, “Software Security and Binary Analysis,” Communications of the Korean Institute of Information Scientists and Engineers 36(3), 2018.3, 11-16.