바이너리 분석을 위한 역어셈블러 검증 도구

정승일O1 류재철2
[1]한국과학기술원 사이버보안연구센터
[2]충남대학교 컴퓨터공학과
sijung@kaist.ac.kr, jcryou@cnu.ac.kr

Disassembler Verification Tool for Binary Analysis

Seungil JungO1 Jae-Cheol Ryou2
[1]Department of Cyber Security Research Center, KAIST
[2]Department of Computer Engineering, Chungnam National University

요 약

바이너리 분석을 시작하기 위해서 기본이 되는 과정 중 한 가지는 바이너리를 어셈블리 언어로 변환하는 역어셈블(Disassemble) 과정이다. 정확한 바이너리 분석을 위해서는 역어셈블이 정확하게 변환되었는지 검증할 필요가 있다. 이런 검증을 위해 현재 오픈소스로 제공되는 다양한 역어셈블러를 참고할 수 있다. 그러나 다양한 역어셈블러 도구에서 오피코드(Opcode)와 특히 오퍼랜드(Operands)를 표현하는 방식이 조금씩 차이가 있어 정확한 역어셈블 검증에 한계가 있다. 이를 해결하고자 현재 많이 사용되고 검증된 역어셈블러 도구인 'Objdump', 'Ndisasm', 'Capstone' 세 가지의 어셈블리(Assembly) 언어 표현 방식을 분석하여 통합된 타입으로 변환하여 비교하는 역어셈블러 검증 도구를 제안한다. 다양한 아키텍처 중에 인텔을 타겟으로 하였다. 이 검증 도구를 통하여 역어셈블의 정확성을 검증할 수 있으며, 또한 기존 역어셈블러 도구의 정확성도 확인할 수 있다.

1. 서 론

역어셈블러(Disassembler)는 바이너리 분석의 시작이라 할 수 있는 역어셈블(Disassemble) 과정을[1] 구현한 도구이다. 상용 분석 도구인 IDA Pro를 비롯하여 다양한 공개 역어셈블러 도구들이 있다. 이런 도구들을 활용하여 역어셈블의 결과를 얻기도 하고 결과의 정확성을 확인하기도 한다. 결과의 정확성을 검증하기 위해서 한 가지의 역어셈블러 도구와 비교하는 것보다 다양한 역어셈블러 도구의 결과를 비교하면 더욱 정확한 검증이 가능할 것이다. 역어셈블 과정은 해당 아키텍처 회사에서 제공하는 매뉴얼을 기반으로 해석할 수 있는데, 역어셈블러 도구를 개발하면서 역어셈블 결과의 표현 방식에서 조금씩 차이가 있는 경우가 있다. 바이너리 분석의 시작이라 할 수 있는 역어셈블 과정의 정확성은 전체 프로그램의 실행 결과가 달라질 수 있기 때문에 중요하다. 따라서 본 논문에서는 역어셈블러 도구의 표현 방식을 통합하는 타입을 정의하여 다양한 역어셈블러 도구의 역어셈블 결과를 정형화하여 비교하는 도구를 소개하고자 한다. 타겟으로 하는 아키텍처는 서버와 PC에서 많이 사용되고 있는 인텔 아키텍처이다.

2. 배경 지식

2.1 인텔 아키텍처

인텔 아키텍처는 CISC(Complex Instruction Set Computer) 방식으로 하나의 인스트럭션(Instruction)을 의미하는 바이너리 코드의 길이가 일정하지 않다. 바이너리 코드를 해석하기 위해서 인텔 인스트럭션 포맷을[2] 참조하여 바이트(Byte) 단위로 읽어 정의된 의미를 분석하면 어셈블리 언어로 변환할 수 있다. 인텔 인스트럭션 포맷은 아래 그림과 같이 정의되어 있다.

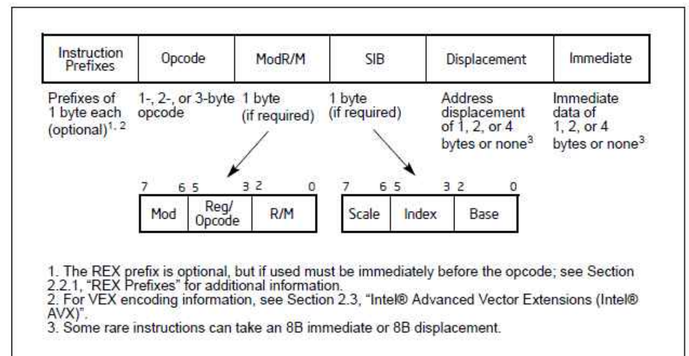

그림 1. Intel Architecture Instruction Format

오피코드(Opcode)는 명령코드(Operation Code)를 의미하며 수행할 명령어를 나타낸다. MOV, ADD, SUB 등이 오피코드이며 값을 옮기거나 더하거나 빼는 동작을 나타낸다.

오퍼랜드(Operand)는 명령어 실행에 필요한 피연산자를 의미한다. 위 그림에서 'ModR/M' 이후의 필드가 오퍼랜드를 나타낸다. 오퍼랜드 개수는 명령어에 따라서 다양하게 올 수 있다. 오퍼랜드 종류로는 레지스터(Register), 메모리(Memory), 이미디어트(Immediate) 등으로 분류할 수 있다.

2.2 역어셈블러 소개

역어셈블러 도구를 활용하여 오피코드와 오퍼랜드 정보

* 본 연구는 과학기술정보통신부 글로벌사이버보안기술연구(과제고유번호: 1711064090) 사업에서 지원하였습니다.

를 얻는다. 모든 역어셈블러 도구는 아키텍처 회사에서 제공하는 매뉴얼을 기반으로 개발되는데 매뉴얼 버전은 계속 업데이트되고 있으며 인텔은 제공되는 명령어의 수가 많아서 지원하는 범위가 도구마다 조금씩 다르기도 하다. 다음은 본 논문에서 역어셈블러 검증 도구에서 사용될 역어셈블러 도구들에 대한 설명이다.

- Objdump[3] : 리눅스에 기본적으로 제공되는 오브젝트 파일 분석 프로그램이다. 역어셈블 뿐만 아니라 섹션 및 심볼에 대한 분석이 가능하다.

- Capstone[4] : Capstone은 다양한 아키텍처를 지원하는 역어셈블러이다. 오픈 소스인 Capstone 엔진을 활용해 다양한 바이너리 분석 도구가 개발되고 있다.

- Ndisasm(Netwide Disassembler)[5] : NASM의 역어셈블러로 x86과 x86-64를 지원한다.

3. 도구 설계

3.1 역어셈블러 비교

역어셈블러의 실행 결과를 비교하기 위해 오피코드와 오퍼랜드의 표현 방식을 살펴볼 필요가 있다. 특별히 살펴보고자 하는 부분은 오퍼랜드에서 메모리의 디스플레이스먼트(Displacement) 표현 방식과 이미디어트 표현 방식이다. 숫자를 표현한 부분이기 때문에 표현 방식이 다소 차이가 있을 수 있다. 아래 표는 각 역어셈블러의 결과이다.

표 1. 역어셈블러 결과 비교

Binary Code	0x8b4708	0xff2524bd2100	0x83f801
MANUAL (64bit)	MOV r32, r/m32	JMP r/m64	CMP r/m32, imm8
Objdump	mov eax, DWORD PTR [rdi+0x8]	jmp QWORD PTR [rip+0x21bd24]	cmp eax, 0x1
Ndisasm	mov eax, [rdi+0x8]	jmp qword [rel 0x21bd2a]	cmp eax, byte +0x1
Capstone	mov eax, dword ptr [rdi + 8]	jmp qword ptr [rip + 0x21bd24]	cmp eax, 1

결과를 살펴보면 역어셈블러에 따라 MOV 명령어는 2번째 오퍼랜드인 메모리의 디스플레이스먼트를 16진수로 표현하거나 10진수 표현하였다. 또한, 메모리의 크기를 'DWORD'와 같이 나타내기도 하고 그렇지 않기도 하였다. 'JMP' 명령어의 경우에는 상대 주소(Relative Address)의 표현 방식이 다름을 볼 수 있다.
이와 같이 의미는 같지만, 결과를 출력하는 표현 방식이 달라서 정확한 비교를 할 수가 없다. 이러한 문제점을 해결하기 위해 통일된 표현 방식으로 나타낼 수 있는 통합된 타입이 필요하다. 역어셈블 결과를 통합된 타입으로 모두 변환 후에 비교하면 정확한 의미에 대한 비교가 가능하다. 다양한 IR(Intermediate Representation)을 통합된 타입으로 비교 검증하는[6] 개념을 참고하였다.

3.2 역어셈블러 통합 타입

각 역어셈블러의 결과를 통합된 형태로 변환하기 위해서 오피코드와 오퍼랜드를 통합할 수 있는 타입을 정의하였다. 아래 그림은 오피코드를 정의한 타입이다.

```
/// Chapter 5 Instruction Set Summary(Vol.1 5-1)
type Opcode =
  /// 5.1 General-Purpose Instructions
  (* 5.1.1 Data Transfer Instructions *)
  | MOV             // Move data between general-purpose registers; move data
                    // between memory and generalpurpose or segment registers;
                    // move immediates to general-purpose registers.
  | PUSH            // Push onto stack.
  | POP             // Pop off of stack.
  | PUSHA | PUSHAD  // Push general-purpose registers onto stack.
  | POPA | POPAD    // Pop general-purpose registers from stack.
  (* 5.1.2 Binary Arithmetic Instructions *)
  | ADD             // Integer add.
  | ADC             // Add with carry.
  | SUB             // Subtract.
  | SBB             // Subtract with borrow.
  | IMUL            // Signed multiply.
  | MUL             // Unsigned multiply.
  ...
```

그림 2. 오피코드 타입

오퍼랜드는 레지스터, 메모리, 이미디어트 세 가지로 정의하였다. 메모리에는 주소 값을 계산하기 위한 요소들이 존재한다. 아래 그림은 오퍼랜드 타입을 정의한 것이다.

```
/// Operand type
type Operand =
  | Register of Register
  | Memory of MemorySize option * BaseRegister option *
             ScaledIndex option * Displacement option
  | Immediate of int64
and MemorySize = uint32
and BaseRegister = Register
and ScaledIndex = Register * Index
and Index = uint8
and Displacement = int64
```

그림 3. 오퍼랜드 타입

```
/// Chapter 3.4 Basic Program Execution Registers(Vol.1 3-10)
type Register =
  /// General-Purpose Registers
  (* Quadword Registers *)
  | RAX | RBX | RCX | RDX | RDI | RSI | RBP | RSP
  | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15
  (* Doubleword Registers *)
  | EAX | EBX | ECX | EDX | EDI | ESI | EBP | ESP
  | R8D | R9D | R10D | R11D | R12D | R13D | R14D | R15D
  (* Word Registers *)
  | AX | BX | CX | DX | DI | SI | BP | SP
  | R8W | R9W | R10W | R11W | R12W | R13W | R14W | R15W
  (* Byte Registers *)
  | AL | BL | CL | DL | AH | BH | CH | DH | DIL | SIL | BPL | SPL
  | R8L | R9L | R10L | R11L | R12L | R13L | R14L | R15L
  /// Segment Registers
  | CS | DS | SS | ES | FS | GS
  /// EFLAGS Register
  | CF | PF | AF | ZF | SF | OF
  ...
```

그림 4. 레지스터 타입

위 그림은 오퍼랜드 타입 중 레지스터에 대한 타입 정의이다. 일반 사용 레지스터 (General-purpose registers)와 세그먼트(Segment) 레지스터, 이플래그(EFLAGS) 레지스터 등을 정의하였다.

4. 도구 검증

4.1 비교 검증 과정

비교 검증을 위한 순서는 다음과 같다. Objdump를 실행하여 파일 기반으로 역어셈블하고 그 결과 중에 주소 값과 바이너리 코드를 확인하여 다른 역어셈블러들이 역어셈블한다. 이 결과를 각각 비교한다.

① 바이너리 파일을 입력받으면 Objdump를 실행하여 주소 값(Address), 바이너리 코드(Binary Code), 오피코드, 오퍼랜드를 파싱하여 리스트로 저장한다.
② 주소 값과 바이너리 코드 정보를 이용하여 Capstone과 Ndisasm이 역어셈블을 실행한다.
③ 각 역어셈블러의 결과를 통합된 타입으로 변환한다.
④ 통합된 타입으로 변환된 결과를 비교한다.

아래 그림은 위의 변환 및 비교 과정을 그림으로 표현한 것이다.

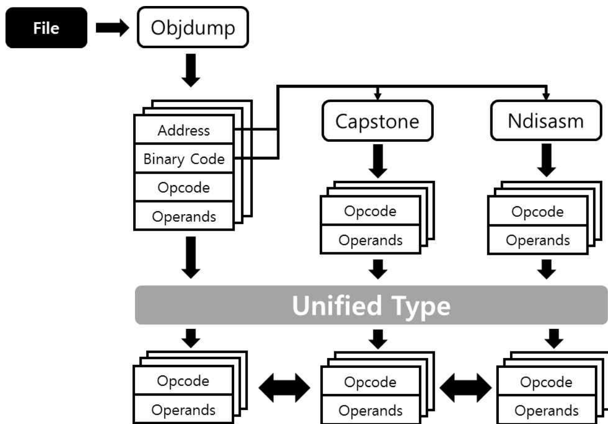

그림 5. 역어셈블러 결과 변환 및 비교 과정

4.2 개발 및 실험 환경

Coreutils의 'ls' 바이너리 파일을 대상으로 분석하고 검증하였다.

표 2. 개발 및 실험 환경

O/S 및 개발 언어		Ubuntu 16.04 x64 F# (.Net Core), python
역어셈블러 버전	Objdump	v2.26.1
	Capstone	V4.0.0
	Ndisasm	v2.11.08
ls (타겟 파일)	Version	(GNU coreutils) 8.25
	Arch	ELF 64-bit LSB x86-64
섹션 범위		.init, .text, .fini

4.3 테스트 결과

'ls' 파일을 분석한 결과 17,740개의 인스트럭션이 존재한다. 이 인스트럭션에서 바이너리 코드를 기준으로 중복을 제거하면 7,431개의 인스트럭션이 존재한다. 중복이 제거된 7,431개의 인스트럭션을 오퍼랜드 개수로 분류하여 각 역어셈블러와 비교한 결과는 아래 표와 같다.

표 3. 'ls' 파일 분석 결과

Operands	Objdump vs Capstone		Objdump vs Ndisasm		Capstone vs Ndisasm		All the same
	Unequal	Equal	Unequal	Equal	Unequal	Equal	
No Operand	0	6	0	6	0	6	6/6
One Operand	44	3011	2441	614	2463	592	592/3055
Two Operands	1392	2967	2626	1733	2975	1384	1382/4359
Three Operands	1	10	11	0	11	0	0/11
Total	1437	5994	5078	2353	5449	1982	1980/7431

위의 표와 같이 역어셈블 결과를 문자열로 단순 비교하면 7,431개의 인스트럭션 중에 1,980개만이 세 가지의 역어셈블러에서 동일한 결과를 출력하였다. 아래 표를 통해 통합되기 전의 표현 방법의 문제점과 그 문제점을 통합된 타입으로 어떻게 해결되는지를 정리하였다.

표 4. 역어셈블러 비교 분석

분류	문제점	예제	해결방법
이미디어트/디스플레이스먼트	10진수와 16진수 혼용	O, N: mov edi, **0x6** C: mov edi, **6**	16진수로 통일
메모리 크기	메모리 크기 표시 여부 및 표현 방법 상이	O: fld **TBYTE PTR** [rip+0xcb91] C: fld **xword** [rip+0xcb91] N: fld **tword** [rel 0xcb97]	메모리 크기 표시 및 표현 방법 통일
상대주소(Relative)	RIP와 계산 전/후 값 출력	O, C: sub rax,QWORD PTR [**rip+0x21bd35**] N: rax,[**rel 0x21bd3c**]	RIP와 계산 전 값으로 출력
오피코드	상태(Conditional) 오피코드, XCHG/NOP 혼용	1) O, C: **je** 0x873 N: **jz** qword 0x873 2) O, N: **xchg ax,ax** C: **nop**	한 가지 오피코드로 통일

※ O: Objdump, C: Capstone, N: Ndisasm

5. 결론 및 향후 연구

본 논문에서는 바이너리 분석을 위해 기본이 되는 역어셈블의 정확성을 검증하는 도구에 대해 기술을 검토하고 검증 방법을 제안하였다. 각 역어셈블러의 결과를 통합된 표현 방식으로 변환하여 비교 검증하는 방법이다. 이 도구를 통해 바이너리 코드가 정확한 어셈블리 언어로 표현되었는지 확인할 수 있다. 이번 논문에서는 대표적으로 사용되는 인텔 아키텍처를 타겟으로 하였는데 향후에 ARM과 MIPS에 대한 역어셈블러 검증 도구도 개발하고자 한다.

참고문헌

[1] 차상길. (2018). 소프트웨어 보안과 바이너리 분석. 정보과학회지, 36(3), 11-16.
[2] "Intel® 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/en-us/articles/intel-sdm
[3] "objdump," https://linux.die.net/man/1/objdump/.
[4] "Capstone," http://www.capstone-engine.org/.
[5] "NASM - The Netwide Assembler," https://www.nasm.us/doc/nasmdoca.html/.
[6] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 353-364, 2017.