# 효율적인 인텔 어셈블러 도구 설계

정승일[O], 류재철[*]
[O]한국과학기술원 사이버보안연구센터,
[*]충남대학교 컴퓨터공학과
e-mail: sijung@kaist.ac.kr[O], jcryou@cnu.ac.kr[*]

# An Efficient Intel Assembler Tool Design

Seungil Jung[O], Jae-Cheol Ryou[*]
[O]Department of Cyber Security Research Center, KAIST,
[*]Department of Computer Engineering, Chungnam National University

● 요 약 ●

소스 코드가 없는 악성코드를 분석하거나 소프트웨어 취약점 분석을 위해 바이너리 분석이 요구된다. 바이너리 분석을 위한 도구 중 어셈블러는 사용자의 입력 없이 컴파일러 내부에서 수행되기 때문에 사용자 관점의 연구는 많지 않다. 그러나 바이너리 분석 과정 중 역어셈블과 중간언어(Intermediate Representation)의 정확성을 검증하기 위해 사용자가 어셈블리어를 입력하여 결과를 확인할 수 있는 어셈블러가 요구된다. 본 논문에서는 어셈블리어를 바이너리 코드로 변환하는 어셈블러 도구를 함수형 언어인 F#으로 구현하여 어셈블리 과정을 효율적으로 설계한 어셈블러 도구를 제안한다. F#의 강력한 패턴 매칭 기능을 사용하여 수백 개의 명령어를 일괄적이고 직관적으로 처리하는 과정을 설계하고 구현하였다.

**키워드:** 어셈블러(Assembler), 함수형 언어(Functional Language), 바이너리 분석(Binary Analysis)

## I. Introduction

소프트웨어 보안 분야에서 바이너리 분석은 실행 파일을 분석하여 악성코드나 취약점을 찾는 등 다양한 연구로 확장될 수 있는 기반 기술이며, 실행 파일을 가장 신뢰할 수 있는 수준으로 분석할 수 있는 중요한 기술이다. 바이너리 분석 과정 중 역어셈블과 리프팅의 정확성은 프로그램 분석 전체에 영향을 주기 때문에 매우 중요하다. 따라서 역어셈블리 과정을 검증하는 방법이 필요하며, 이는 바이너리를 역어셈블한 결과와 어셈블리어를 바이너리로 어셈블한 결과를 비교하는 방법으로 검증할 수 있다. 이를 위해 어셈블리어를 바이너리 코드로 변환하는 어셈블러 도구가 요구된다. 또한, 어셈블러는 역어셈블리 검증뿐 아니라 사람이 이해하기 어려운 바이너리 코드를 직접 입력하지 않고 어셈블리어를 입력하여 중간언어(Intermediate Representation)를 검증하는 도구에 활용될 수 있다. 이러한 어셈블러는 수백 개의 명령어를 효율적으로 처리할 수 있도록 설계하는 것이 중요하다. 일반적인 프로그래밍 언어는 수백 개의 명령어를 처리하기 위한 분기문으로 IF문이나 SWITCH문을 사용하는데, 분기문을 더욱 간결하고 명확하게 처리할 수 있는 패턴 매칭과 강력한 타입 시스템을 지원하는 함수형 언어인 F#으로 이를 해결할 수 있다. 따라서 F#을 사용한 효율적인 어셈블러를 설계하고 구현하고자 한다.

## II. Preliminaries

### 1. Related works

#### 1.1 국내외 동향

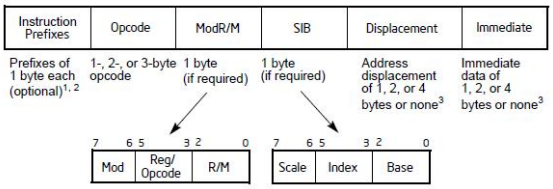


Fig. 1. Intel Architecture Instruction Format

바이너리 분석 연구는 해외에서 활발하게 진행되고 있으며 해외의 유수 학회에서는 바이너리 분석의 다양한 연구와 기술들이 발표되고 있다. 실제 2018년에는 해외의 탑 컨퍼런스인 NDSS(The Network and Distributed System Security Symposium)에서 바이너리 분석을 주제로 하는 워크숍(BAR)이 개최되기도 하였다. 지금까지 유럽과 미국에서 주도적으로 이루어지고 있는 바이너리 분석 연구는 최근 국내에서 발표된 바이너리 분석 플랫폼인 B2R2[1]를 비롯하여 아시아 지역에서도 퍼징, 메모리 취약점 분석, 익스플로잇 자동 생성

등 바이너리 분석 관련 연구가 활발히 진행되고[2] 있다.

### 1.2 인텔 아키텍처

인텔 어셈블러를 구현하기 위해서는 인텔 아키텍처에 대한 분석이 선행되어야 한다. 인텔 명령어 포맷[3]의 구조를 파악하여 어셈블러의 인코딩 과정을 설계할 수 있다. 인텔 명령어 포맷은 그림 1과 같다. 명령어 포맷은 6개의 필드로 구성되어 있으며, 6개의 필드 중 ModR/M과 SIB는 세부적으로 각각 3개의 필드로 구성된다. Prefix는 명령어의 부가적인 정보를 담고 있으며, Opcode 필드는 명령 코드(Operation Code)를 나타내며 수행할 명령어를 나타내는 부호이다. 나머지 ModR/M, SIB, Displacement, Immediate 필드는 Operand(피연산자)를 나타낸다. 각 필드에 대한 자세한 내용은 어셈블러 설계 과정에서 설명한다.

### 1.3 B2R2

B2R2는 KAIST에서 공개하고 지속적으로 연구가 진행되고 있는 바이너리 분석 플랫폼이다. 함수형 언어인 F#으로 구현되어 .NET 설치를 통해 외부 라이브러리의 의존성 없이 쉽게 실행 및 활용이 가능하며 함수형 언어의 특성에 따라 코드가 읽기 쉽고 간결하게 구현되었다. 또한, 다양한 아키텍처(Intel, ARM, MIPS 등)의 바이너리 분석을 지원하여 여러 시스템의 실행 파일의 분석이 가능하다. 현재 지원하는 아키텍처의 분석 정확성을 지속적으로 검증하고 있으며, 아키텍처의 추가를 위해 새로운 아키텍처 분석을 진행하고 있다.

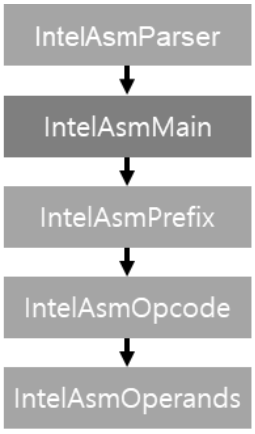


Fig. 2. Intel Assembler Modules

## III. The Proposed Scheme

### 1. 어셈블러 모듈 설계

어셈블러를 구현하기 위하여 인텔 명령어 포맷을 기반으로 세부 모듈을 구성하였다. 각 모듈은 명령어 포맷의 필드를 인코딩하기 위해 설계되었으며 전체적인 구성은 그림 2와 같다.

어셈블러 모듈을 설계하기 위해 인텔 아키텍처의 명령어 정보를 저장하는 타입이 필요하다. 이를 위해 B2R2[1]의 명령어 타입을 활용하였다. B2R2에서 정의한 인텔 명령어 정보 타입은 그림 3과 같으며 명령어 정보(Instruction Information)를 표현하기 위해 타입명은 InsInfo라 정의하였다. InsInfo 타입을 사용하여 어셈블러 과정에서 발생하는 정보를 처리하였다.

### 1.1 IntelAsmParser

입력받은 문자열을 InsInfo로 변환하는 역할을 담당한다. 이는 단순히 어셈블러어 문자열을 파싱하는 것이 목적이므로 Opcode와 Operand를 확인해야 알 수 있는 Prefix 등은 처리하지 않고 InsInfo를 생성한다.

```
type InsInfo = {
  /// Prefixes.
  Prefixes        : Prefix
  /// REX Prefix.
  REXPrefix       : REXPrefix
  /// VEX information.
  VEXInfo         : VEXInfo option
  /// Opcode.
  Opcode          : Opcode
  /// Operands.
  Operands        : Operands
  /// Instruction size information.
  InsSize         : InsSize
}
```

Fig. 3. Intel Assembler Modules

Table 1. Instruction Prefixes[3]

| Group | Description | Byte code |
|---|---|---|
| Group 1 | Lock and repeat | F0H, F2H, F3H |
| | Bound | F2H |
| Group 2 | Segment override | 2EH, 36H, 3EH, 26H, 64H, 65H |
| | Branch hints | 2EH, 3EH |
| Group 3 | Operand-size override | 66H |
| Gropu 4 | Address-size override | 67H |

### 1.2 IntelAsmMain

어셈블러의 메인 모듈이다. 메인 모듈의 역할은 IntelAsmParser가 문자열에서 변환한 InsInfo 타입을 바이트 코드로 인코딩하기 위해 Opcode를 매칭하고 필요한 해당 모듈을 호출한다. 또한 상대 주소의 경우 인코딩을 통해 명령어의 길이를 확인해야 PC(Program Counter) 주소를 알 수 있기때문에 이러한 계산을 처리하는 역할은 수행한다.

### 1.3 IntelAsmPrefix

인텔 명령어 포맷의 첫 번째 필드에 해당하는 Prefix를 인코딩하는 모듈이다. 인텔 Prefix는 표 1과 같이 4개의 그룹으로 나뉜다. Prefix는 옵션 필드이기 때문에 인코딩이 수행되지 않을 수 있다.

### 1.4 IntelAsmOpcode

Opcode를 확인하고 해당 바이트 코드로 변환하는 역할을 담당하는 모듈이다. 1,000개 이상의 Opcode를 처리하기 위해서는 효율적인 방식이 필요하다. 따라서 함수형 언어인 F#의 패턴 매칭을 사용하여 빠르고 효율적으로 Opcode를 바이너리 코드로 변환할 수 있다.

### 1.5 IntelAsmOperands

Operand는 피연산자라고 하며 Opcode의 연산을 위해 필요한 정보이다. Operand는 레지스터, 메모리 또는 상수 값을 나타내는 이미디어트(immediate)가 있다. 인텔 명령어 포맷에서 Operand를 정의한 필드는 ModR/M, SIB, Displacement, Immediate 이다. IntelAsmOperands 모듈은 앞서 언급한 4가지의 필드를 인코딩하는 역할을 담당한다. ModR/M은 Operand의 형태를 나타내며 SIB와 Displacement는 메모리 구성 및 추가적인 정보를 나타내며 Immediate는 상수를 저장하거나 연산하는데 사용된다.

```
let encodeInstruction ins ctxt =
  match ins.Opcode with
  | Opcode.AAA -> aaa ctxt ins.Operands
  | Opcode.AAD -> aad ctxt ins.Operands
  | Opcode.AAM -> aam ctxt ins.Operands
  | Opcode.AAS -> aas ctxt ins.Operands
  | Opcode.ADC -> adc ctxt ins
  | Opcode.ADD -> add ctxt ins
  | Opcode.ADDPD -> addpd ctxt ins
  | Opcode.ADDPS -> addps ctxt ins
  | Opcode.ADDSD -> addsd ctxt ins
  | Opcode.ADDSS -> addss ctxt ins
  | Opcode.AND -> logAnd ctxt ins
  | Opcode.ANDPD -> andpd ctxt ins
```

Fig. 4. Intel Instruction encoding function

## 2. 어셈블러 구현

### 2.1 패턴 매칭

F#에서 제공하는 강력한 패턴 매칭 기능을 이용하여 많은 Opcode와 다양한 형태의 Operand를 일괄적으로 처리하였다. 기존의 C나 Python 등의 언어에서는 IF문이나 SWITCH문과 같은 분기문을 이용하여 구현하였던 매칭 과정을 F#의 강력한 패턴 매칭 기능을 이용하여 쉽고 간결하게 구현하였다. 그림 4는 패턴 매칭을 사용하여 인텔 명령어 인코딩을 시작하는 단계로 Opcode를 매칭하는 함수이다.

### 2.2 상대 주소

주소를 표현하기 위한 두 가지 방법으로는 절대 주소(Absolute address)와 상대 주소(Relative address) 가 있는데, 절대 주소는 주소 값 0에서 부터 시작된 주소를 나타낸며, 상대 주소는 상대 값을 가진 주소를 의미한다. 현재의 위치에서 앞으로(Forward) 또는 뒤로(Backword) 이동할 수 있다. 어셈블러 구현 과정에서 절대 주소는 인코딩이 간단하지만 상대 주소의 경우는 모든 명령어의 길이를 확인해야 상대적인 주소를 계산할 수 있다. 따라서 모든 명령어의 길이를 1차적으로 인코딩한 이후에 상대 주소를 사용하는 Opcode에 대해서 주소를 계산한다.

## IV. Experiment

어셈블러 정확성을 검증하기 위하여 coreutils의 ls 바이너리를 타겟으로 선정하였다. ls.c의 소스 코드를 컴파일하는 과정에서 컴파일 옵션(gcc -S)을 사용하여 기계어인 어셈블리어로 변환하였고, 이 파일에서 어셈블리어를 대상으로 검증을 진행하였다. 어셈블러 검증을 위해 ls 바이너리의 명령어를 분석하여, 명령어 개수를 기준으로 정리한 결과는 표 2와 같다.

Table 2. 'ls' binary analysis

| | ls: coreutils 8.32 (not stripped) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ELF 64-bit | | | | ELF 32-bit | | | |
| 명령어 개수 | mov | 6,031 | nop | 928 | mov | 9,359 | test | 918 |
| | cmp | 1,441 | test | 825 | cmp | 1,543 | push | 874 |
| | jmp | 1,132 | xor | 786 | call | 1,408 | add | 703 |
| | call | 1,111 | pop | 686 | jz | 1,094 | jnz | 679 |
| | jz | 1,058 | *etc* | *4869* | jmp | 972 | *etc* | *6186* |
| | Total | | 18,966 | | Total | | 23,736 | |

표 2를 기준으로 파일 전체를 검증하기 전에 ls.c의 소스 코드를 어셈블리어로 컴파일한 일부 블록을 어셈블러로 테스트하였다. 테스트를 위한 샘플 어셈블리어는 그림 5와 같다.

어셈블리 파일에는 Opcode와 Operand 이외에도 어셈블과 상관없는 부가적인 정보들이 있는데, 스택 추적 및 소스코드 디버깅 관련된 정보를 제거하여 순수한 어셈블리어로 테스트를 진행하였다. 그림 6은 샘플 어셈블리어에 대한 어셈블러 실행 결과이다. 어셈블리어를 바이트 코드로 변환하였으며 결과 검증을 위하여 어셈블러에 의해 변환된 바이트 코드를 역어셈블러로 다시 출력하여 비교하였다.

표 3은 앞서 ls 바이너리에 대해 표 2의 분석 결과를 대상으로 전체를 검증한 결과이다. ls 바이너리 파일 전체 명령어를 검증하기 위하여 ls 바이너리를 역어셈블한 결과를 어셈블러로 실행하여 결과를 비교하였다. ls 바이너리에 대한 어셈블러는 아래 표와 같이 명령어를 기준으로 모두 동일하였다. 명령어 종류는 MOV, SUB와 같은 명령어 코드의 종류가 몇 개인지 나타내며, 명령어 총 개수는 중복을 제거하지 않은 전체 명령어의 개수를 나타낸다.

```
.LFB24:
    .cfi_startproc
    push    rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    mov rbp, rsp
    .cfi_def_cfa_register 6
    mov eax, edi
    mov BYTE PTR [rbp-4], al
    movzx   eax, BYTE PTR [rbp-4]
    pop rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
LFB24:
  push rbp
  mov rbp, rsp
  mov eax, edi
  mov byte ptr [rbp-4], al
  movzx eax, byte ptr [rbp-4]
  pop rbp
  ret
```

Fig. 5. Part of 'ls.c' assembly

```
00000000: FFF5                    push RBP
00000002: 488BEC                  mov RBP, RSP
00000005: 8BC7                    mov EAX, EDI
00000007: 8845FC                  mov byte ptr [RBP-0x4], AL
0000000a: 0FB645FC                movzx EAX, byte ptr [RBP-0x4]
0000000e: 488FC5                  pop RBP
00000011: C3                      ret
```

Fig. 6. 'ls.s' assembly results

Table 3. 'ls' binary assembler results

| ls | 분석도구 | 명령어 종류 | 명령어 개수 |
|---|---|---|---|
| x64 | 역어셈블러 | 97 | 18,966 |
| | 어셈블러 | 97 | 18,966 |
| | 일치율 | 100% | 100% |
| x86 | 역어셈블러 | 116 | 23,736 |
| | 어셈블러 | 116 | 23,736 |
| | 일치율 | 100% | 100% |

## V. Conclusions

함수형 언어인 F#을 사용하여 인텔 어셈블러를 구현하였다. 문자열 형태로 입력되는 어셈블리어를 바이너리 코드로 변환하기 위해서는 다양한 입력 형태를 분석하여 처리해야 한다. 이런 과정을 F# 언어에서 쉽게 사용할 수 있는 강력한 패턴 매칭 기능을 사용하여 직관적으로 구현하였다. 문자열에서 Opcode와 Operand를 파싱하여 바이너리 코드로 변환하는 기본적인 처리 과정을 구축하였기 때문에 ARM, MIPS 등 다양한 아키텍처의 어셈블러를 구현하는 것도 가능하다. 다양한 아키텍처를 지원하는 어셈블러를 통해 여러 바이너리 분석 플랫폼을 비교하여 중간언어를 검증[4]하는 방법과는 다른 인터렉티브한 중간언어 검증에 활용하는 등 바이너리 분석을 위한 연구를 계속 진행하고자 한다.

## REFERENCES

[1] B2R2 Project, https://github.com/B2R2-org/B2R2.

[2] Sang Kil Cha and Zhenkai Liang. 2020. "Asia's surging interest in binary analysis." Commun. ACM 63, 4 (April 2020), 86-88.

[3] "Intel® 64 and IA-32 Architectures Software Developer Manuals"

[4] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungil Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. "Testing intermediate representations for binary analysis." In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, 353-364.