Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.5, Oct. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.5.1233

기호 형태의 값-집합 분석을 이용한 ARM 위치 독립적 코드의 정교한 역어셈블리 기법*

하 동 수,† 오 희 국‡
한양대학교

A Disassembly Technique of ARM Position-Independent Code with Value-Set Analysis Having Symbol-Form Domain*

Dongsoo Ha,† Heekuck Oh‡
Hanyang University

요 약

스마트 모바일의 보급에 따라, 컴퓨터 보안에서 ARM 아키텍처 명령어로 구성된 위치 독립적 코드의 역어셈블리 기법이 중요해지고 있다. 그러나 대부분의 기존 기법들은 x86 아키텍처 대상으로 연구되었으며, 위치 종속적 코드의 문제 해결과 범용성에 초점이 맞추어져 있다. 따라서, ARM 아키텍처의 고정 길이 명령어와 위치 독립적 코드의 특징이 제대로 반영되지 않아, 바이너리 계측과 같이 바이너리 자체를 직접 수정하는 수준의 고도화된 응용 보안 기술에 적용하기에는 수집되는 주소 정보의 정확도가 낮다. 본 논문에서, 우리는 ARM 명령어로 구성된 위치 독립적인 코드의 특성을 반영한 역어셈블리 기법을 제안한다. 정확하고 추적 가능한 주소의 수집을 위해, 도메인이 기호화된 값-집합 분석을 설계하였다. 또한, 역어셈블의 주요 문제점을 해결하기 위해, 컴파일러가 생성하는 코드의 특징을 활용한 휴리스틱을 고안하였다. 우리 기법의 정확도와 유효성을 검증하기 위해, 안드로이드 8.1 빌드에 포함된 669개의 공유라이브러리 및 실행 파일을 대상으로 실험하였으며, 그 결과 완전한 역어셈블의 비율이 91.47%로 나왔다.

ABSTRACT

With the proliferation of smart mobiles, disassembly techniques for position-independent code (PIC) composed of ARM architecture instructions in computer security are becoming more important. However, existing techniques have been studied on x86 architecture and are focused on solving problems of non-PIC and generality. Therefore, the accuracy of the collected address information is low to apply to advanced security technologies such as binary measurement. In this paper, we propose a disassembly technique that reflects the characteristics of PIC composed of ARM instructions. For accuratly collecting traceable addresses, we designed value-set analysis having symbol-form domain. To solve the main problem of disassembly, we devised a heuristic using the characteristics of the code generated by the compiler. To verify the accuracy and effectiveness of our technique, we tested 669 shared libraries and executables in the Android 8.1 build, resulting in a total disassembly rate of 91.47%.

Keywords: Disassembly, Disassembler, Value-set analysis (VSA), Position-independent code (PIC), ARM architecture

Received(09. 10. 2018), Modified(10. 04. 2018), Accepted(10. 04. 2018)

* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성 지원사업의 연구결과로 수행되었음 (IITP-2018-2014-0-00636). 또한, 2015년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NRF-2015R1D1A1A09058200).

† 주저자, hds@hanyang.ac.kr
‡ 교신저자, hkoh@hanyang.ac.kr(Corresponding author)

I. 서 론

역어셈블리는 대상 바이너리에 포함된 기계어를 사람이 읽을 수 있는 형태로 복원하는 기법으로, 다양한 목적으로 활용된다. 일반적으로 대상 바이너리를 정적 및 동적으로 디버깅하거나 취약점을 찾기 위해 사용된다. 이뿐만 아니라 다른 기법의 기반 기술로도 중요하게 사용된다. 대표적으로 바이너리 계측 기법 (binary instrumentation)이 있으며, 이 기법은 CFI (Control Flow Integrity)[1-5], SFI (Software Fault Isolation)[6-9], 쉐도우 스택[10-14], 로우 레벨 난독화[15-17] 등의 보안 어플리케이션 작성과 프로그램 최적화, 프로파일링, 로우 레벨 버그 수정[19]을 위한 목적으로도 사용된다. 이처럼 역어셈블리 기법은 보안을 포함한 많은 영역에서 핵심 기술로 사용되고 있다.

특히, 바이너리 계측과 같은 기법에서는 역어셈블리의 정확도가 매우 중요하다. 바이너리 재작성(binary rewriting)이라고도 불리는 이 기법은 대상 바이너리의 특정 지점에 추가적인 명령어를 삽입한다[18-19]. 따라서 단순히 코드 세그먼트의 CFG(Control Flow Graph)를 복원하는 것뿐만 아니라, 특정 지점의 오프셋이 변경되었을 경우 영향을 받는 명령어의 오프셋(또는 주소)과 데이터(주소나 상대 주소)를 파악할 수 있도록 역어셈블리의 정교하고 정확한 결괏값을 요구한다. 이처럼 보안 영역에서 역어셈블리의 역할은 단순히 CFG를 복원하는 것이 아닌 정교한 수정 작업이 가능한 수준의 정보를 제공해야 하는 수준까지 도달하였다.

불행하게도 역어셈블리는 일반적으로 완벽하게 수행될 수 없다[20]. 소스 코드 수준에서 존재하는 변수나 함수의 이름, 타입, 심볼과 같은 정보는 구동 과정에서는 필요가 없으므로, 배포할 바이너리를 컴파일할 때에는 해당 정보는 지워지게 된다. 따라서 역어셈블리를 수행할 때, 계산의 결괏값이 어떤 종류인지 정확하게 파악하는 것이 어려우며, 이런 이유로 인하여 정확하고 정교한 역어셈블리가 어려워진다. 예를 들어, 부정확한 정보로 인하여 데이터 포인터를 명령어 포인터로 인식하여 잘못된 역어셈블리를 수행하거나, 반대로 명령어 포인터를 데이터 포인터로 인식하여 바이너리에 존재하는 모든 서브 루틴을 찾지 못할 수 있다. 이런 문제점을 보완하기 위해, 다양한 휴리스틱이 연구되고 있다[4,21-22].

현재까지 연구된 대다수의 역어셈블리 기법은 x86 아키텍처에 대한 것이며, 특히 위치 종속적인 코드에 집중되어 있다[4,20-22]. 반면 ARM 아키텍처와 위치 독립적인 코드 (PIC, Position Independent Code)는 깊게 다뤄지지 않고 있다[19]. 하지만, 근 10년 동안 스마트 모바일의 보급으로 인해, ARM 아키텍처가 중요해지고 있으며, 스마트 모바일의 특성상 높은 보안과 리소스의 효율적인 관리를 요구하기 때문에 위치 독립적 코드의 중요성 또한 같이 높아지고 있다. 스마트 모바일의 대표적인 운영체제인 안드로이드의 경우 5.0 버전 이후부터는 모든 서드 파티 앱의 네이티브 코드를 위치 독립적 코드로 강제하였으며, iOS의 경우 4.3 이후부터 사용을 권고하고 있다. 이런 추세를 고려해 볼 때, ARM 위치 독립적 코드의 정교한 역어셈블리에 대한 요구도 더욱 증가할 것으로 예상한다.

기존 x86 아키텍처와 위치 종속적 코드를 다루는 역어셈블리 기법으로 ARM 위치 독립적 코드를 정교하게 처리하기에는 한계가 있다. 고정 길이의 특성을 가진 ARM 아키텍처 명령어로 구성된 위치 독립적 코드는 모든 주소가 산술 연산 명령어를 통해 계산될 뿐만 아니라, 대다수의 경우 하나의 주소가 여러 단계를 거쳐서 생성되는 차별되는 특징을 가진다. 기존 기법들은 이 특징을 제대로 고려하지 않으므로 정확하게 주솟값을 수집할 수 없다.

대표적인 예로, 값-집합 분석을 사용하는 접근법을 들 수 있다. 명령어가 계산하는 주소를 정적으로 수집하기 위해 값-집합 분석을 사용하는데, 위치 종속적 코드가 대상일 경우, 상수와 주소의 구분이 어려워 단순히 정수의 집합이나 인터벌 형태로 분석 도메인을 설계한다. 따라서, 기법 자체가 범용적이며 실제 발생하는 값보다는 실제 값의 슈퍼셋을 수집하는 것에 초점이 맞춰져 있다[23,24]. 위치 독립적 코드는 주소와 상수의 구분이 가능하므로, 이를 고려할 경우 더 정교하게 주소를 수집할 수 있으며, 정확한 역어셈블리 기법을 만들 수 있다.

본 논문에서는 ARM 위치 독립적 코드의 특성을 활용한 특화된 역어셈블리 기법을 제안한다. 이 기법은 기존 상용 역어셈블러처럼 코드 세그먼트의 CFG를 단순히 나타내는 것이 아닌, 바이너리 계측 기법을 수행할 수 있을 정도의 풍부한 정보를 포함한 역어셈블리 결과를 도출할 수 있다. 이를 위해, 컴파일러의 특성과 ARM 위치 독립적 코드의 특성을 활용한 값-집합 분석을 설계 및 구현하였으며, 안드로이드 8.0 빌드에 포함된 669개의 바이너리를 대상으로

우리 기법의 유효성을 평가하였다.

본 논문의 구성은 다음과 같다. 2장에서 기본적인 배경 지식과 관련 연구에 대해 다룬다. 3장에서 역어셈블리의 어려운 점과 위치 독립적 코드에서 고려해야 하는 사항에 대해 알아본다. 4장에서 문제점을 해결하기 위한 기본적인 접근법을 다루며, 5장에서 자세한 내용을 소개한다. 6장에서는 우리가 디자인한 값-집합 분석을 다룬다. 7장에서 우리의 역어셈블리 기법을 평가하고, 8장에서 결론으로 마무리 짓는다.

II. 배경 지식 및 관련 연구

본 장에서는 ARM 명령어로 구성된 위치 독립적 코드의 특징과 역어셈블리의 기본적인 접근법에 관해 설명한다.

2.1 ARM 명령어로 구성된 위치 독립적 코드

위치 독립적 코드란, 로드 시 메모리상에 배치되는 절대주소에 상관없이 정상적으로 동작 가능한 코드를 의미한다. 반대로 위치 종속적 코드는 기대하는 메모리 주소에 종속적으로 코드가 구성되므로, 예상되는 주소가 아닌 다른 주소에 코드가 배치될 경우 정상적으로 동작하지 않는다. 즉, 위치 종속적 코드는 명령어의 즉시 값이나 데이터 세그먼트의 상숫값에 가상 메모리의 주솟값이 직접적으로 포함될 수 있음을 의미한다. 반대로 위치 독립적 코드는 가상 메모리의 주솟값이 명령어나 데이터 세그먼트에 직접 배치가 되지 않으며, 모든 주소는 상대 주소를 통해 계산된다. 따라서 바이너리가 로드되는 베이스 주소에 영향을 받지 않는다.

ARM 아키텍처의 명령어는 2 또는 4바이트로 고정 길이이다. ARM 아키텍처는 ARM 모드와 THUMB 모드의 두 가지 명령어 셋으로 구성되며, ARM 모드의 명령어는 4바이트 길이를 가지며 THUMB 모드의 명령어는 2 또는 4바이트 길이를 가진다.[1] 가변 길이를 가지는 x86 아키텍처의 명령어와 비교할 때, 고정 길이 명령어의 특징은 하나의 명령어가 포함할 수 있는 상숫값의 크기가 제한된다는 것이다. ADD 명령어의 경우를 보면, x86에서는 1, 2, 4바이트 크기의 즉시 값이 포함될 수 있지만, ARM에서는 1바이트 크기의 즉시 값만이 포함될 수 있다.

1) 세부적으로 구분하면, THUMB-1 명령어 셋은 2바이트, THUMB-2 명령어 셋은 4바이트 길이를 가진다.

```
x86 architecture

  __i686.get_pc_thunk.bx:  MOV   EBX, [ESP]
                           RET

  [Pattern 1]  CALL   __i686.get_pc_thunk.bx
               ADD    EBX, 0x1281180

ARM architecture

  [Pattern 1]  LDR   R2, [PC,#0x100]
               ADD   R2, PC, R2
               ...   (0x01281180) ...

  [Pattern 2]  MOVW   R2, #0x1180
               MOVT   R2, #0x0128
               ADD    R2, PC, R2

  [Pattern 3]  ADR    R2, #0x1280000
               ADD    R2, #0x1180
```

Fig. 1. Example of address calculation patterns with relative address 0x1281180 in PIC on x86 and ARM architectures

앞서 언급한 위치 독립적 코드와 ARM 아키텍처 명령어의 특징으로 인해, ARM 아키텍처 명령어로 구성된 위치 독립적 코드에서는 주소 계산이 여러 단계로 구성되는 특징이 존재한다. 많은 경우 상대 주소가 하나의 명령어에 완전히 포함되지 못하기 때문에 두 개 이상의 명령어를 통해 주소가 계산된다. 그리고 ARM 아키텍처는 PC 레지스터가 범용 레지스터이므로 상대 주소 사용 패턴이 매우 다양하며, 계산되는 주소를 올바르게 모두 찾기 위해서는 체계적인 접근법이 요구된다.

Fig. 1은 상대 주소 0x1281180을 주소 계산에 사용하기 위해 컴파일러가 생성하는 위치 독립적 코드의 예이다. x86에서는 PC 레지스터가 범용 레지스터가 아니므로, __i686.get_pc_thunk.bx 함수와 같이 스택 포인터가 가리키는 값을 레지스터로 저장한 다음 바로 리턴하는 형태의 함수를 통해 다음 명령어 주소를 얻는다. 그리고 하나의 ADD 명령어가 4바이트의 상대 주소를 완전히 포함할 수 있어, 하나의 산술 연산 명령어로 완전한 주소를 계산할 수 있다. 현재 gcc나 clang과 같은 컴파일러가 생성하는 x86 아키텍처에서의 주소 계산 패턴은 예시에서 언급한 것만 보고되고 있다[4,19].

반대로, ARM 아키텍처에서는 주소 계산을 위해 다양한 패턴이 존재한다. 첫 번째 패턴에서는 LDR 명령어를 통해, 4바이트 상대 주소를 읽어와 ADD 명령어의 PC 레지스터와 연산하여 주소를 구한다.

두 번째 패턴에서는 MOVW와 MOVT 명령어를 사용하여 4바이트 상대 주소를 계산한 다음 ADD 명령어와의 연산을 통해 주소를 구한다. 세 번째에서는 ADR 명령어를 통해 대상 주소의 일부분을 먼저 구한 다음 ADD의 즉시 값을 통해 대상 주소를 최종적으로 계산하여 얻는다. 이처럼 ARM 아키텍처에서는 고정 길이 명령어의 특성으로 인해, 상황에 따른 다양한 주소 계산 방법이 존재한다.

2.2 기본적인 역어셈블리 접근법

역어셈블리를 수행하는 방법은 크게 두 가지 방법으로, 선형 어셈블리 방법 (linear sweep)과 재귀 어셈블리 방법 (recursive traversal)으로 나뉜다. 첫 번째로, 선형 어셈블리 방법은 주어진 시작 주소부터 시작하여 끝 지점까지 순차적으로 명령어를 디코딩한다. 이 방법을 사용하는 대표적인 도구로 GNU Binutils의 Objdump가 있다.

선형 어셈블리 방법은 단순하지만, 일반적인 코드 세그먼트를 완벽하게 역어셈블리하지 못한다. 현대 컴파일러가 생성하는 코드 세그먼트는 명령어뿐만 아니라 명령어가 접근하는 리터럴풀을 포함하게 된다. 리터럴풀이란 상대 주소, 정수, 부동 소수점, 문자열 등과 같은 상숫값을 가지고 있는 데이터의 묶음이다. 리터럴풀은 코드 세그먼트 곳곳에 배치되며, 이런 이유로 인해 선형 역어셈블리 방법은 리터럴풀을 만나게 될 경우, 잘못된 역어셈블리 결과를 도출하게 된다. Fig. 2.는 리터럴풀의 예시를 보여준다.

재귀 어셈블리 방법은 주어진 시작 주소부터 시작하여 분기가 발생하는 명령어까지 역어셈블리를 수행하며, 분기 대상 지점부터 다시 역어셈블리를 수행한다. 따라서 이 방법은 선형 어셈블리의 문제점을 해결할 수 있다. 다이렉트 점프(direct jump)나 호출(direct call)같은 경우에는 대상 주소가 명확하여 문제가 되지 않지만, 간접 제어 흐름(indirect

```
0x2A8A   MOVS   R1, #0
0x2A8C   BL     label
0x2A90   MOVS   R0, #0
0x2A92   POP    {R4,PC}
0x2A94   DCD 0x333E
0x2A98   DCD 0x35CC
0x2A9C   PUSH   {R4-R6,LR}
0x2A9E   MOV    R6, R0
0x2AA0   MOV    R4, R2
```

Fig. 2. Example of the literal pool

control flow)의 경우에는 대상 주소를 따로 분석하여 찾아야 한다. 경우에 따라서는 모든 대상 주소를 찾기 어려우며, 이로 인해 완벽한 역어셈블리를 수행하지 못한다.

대다수 상용 분석 도구의 역어셈블러는 재귀 어셈블리 기반으로 동작하며, 선형 어셈블리 방법을 적절하게 결합하여 사용한다. 대표적인 예로, IDA Pro는 다양한 분석과 휴리스틱을 사용하여 간접 제어 흐름의 대상 주소를 찾아낸다. 그리고 이를 통하여 도달하지 못하는 코드 세그먼트의 빈 공간을 선형 역어셈블리를 사용하여 CFG 복원을 시도한다. 이뿐만 아니라 다른 도구들도 이 두 방법의 적절한 조합을 통해 분석을 시도하며, 그 결과 잘못된 역어셈블리 결과를 도출하는 경우가 종종 발생한다.

III. 역어셈블리의 문제점

본 장에서는 ARM 위치 독립적 코드를 역어셈블리하기 위해 해결해야 하는 사항을 설명한다. 이를 위해 먼저 일반적인 역어셈블리의 해결 사항을 다루도록 한다.

3.1 일반 역어셈블리의 문제점

일반적으로, 역어셈블리를 어렵게 하는 요소는 세 가지 요소로 정리할 수 있다. 첫 번째로, 간접 제어 흐름을 정확하게 모두 계산하는 문제이다. 두 번째로, 계산된 결괏값이나 데이터 섹션에 저장된 값이 포인터 값인지를 구분하는 문제이다. 세 번째로, 주어진 포인터가 데이터 영역을 가리키는지 명령어 영역을 가리키는지 구분하는 문제이다. 이 세 가지 문제를 완벽하게 해결하는 방법은 존재하지 않으며, 현재까지는 다양한 휴리스틱을 통해 해결하고 있다 [4,21,22].

첫 번째 문제점에서 가장 대표적인 문제는 점프 테이블을 사용하는 간접 점프(indirect jump)의 대상 주소를 정확하게 구하는 것이다. 일반적으로 테이블의 인덱스 값은 외부로부터 오거나 외부의 값과 같이 계산되는 경우가 많다. 따라서, 일반적인 데이터 흐름 분석을 통해 이 값을 정확하게 파악할 수 없다. 이뿐만 아니라 C++와 같은 언어로 컴파일된 바이너리는 가상 메소드 테이블을 통해 호출하는데, 이 때에도 메소드 테이블의 위치 또는 크기를 정확하게 파악하기 힘들다.

두 번째 문제점은 산술 연산을 통해 구해진 값의 주소 여부를 판단하기 어렵다는 것이다. 콜백 함수(callback function) 등에서 사용되는 주솟값은 간접 호출 명령어에 사용되지 않고 단순히 함수 호출 시 파라미터로 전달되거나 특정 필드에 저장된다. 따라서, 산술 연산 명령어 결괏값의 타입을 정확하게 구분하지 못할 경우, 올바른 역어셈블리를 수행할 수 없다. 이뿐만 아니라 데이터 세그먼트에 저장된 값 중에 주소가 포함될 수 있으며, 그 값을 식별하지 못하면 같은 문제가 발생한다.

세 번째는 문제점은 주솟값을 얻었다 하더라도 이 값이 명령어를 가리키는지 데이터를 가리키는지 구분하기 어렵다는 것이다. 데이터 영역을 가리키는 주소라 하더라도, 경우에 따라서는 정상적으로 역어셈블리가 수행된다. 이런 이유로 인해, 직접적으로 호출이나 점프가 되지 않는 주소의 타입을 판별하는 것은 어렵다. CISC인 x86의 경우 많은 종류의 명령어가 최대한 많이 인코딩되어 있으므로, 임의의 값이 주어졌을 때 역어셈블될 확률이 높다. RISC인 ARM 아키텍처도 ARM, THUMB-1, THUMB-2 등의 명령어 셋이 존재하므로, 94% 이상의 확률로 역어셈블이 성공된다.

3.1 ARM 위치 독립적 코드의 문제점

ARM의 위치 종속적 코드를 역어셈블리 할 때 가장 핵심이 되는 사항은 포인터 여부를 판단하는 문제이나, 위치 독립적 코드의 경우에는 포인터(또는 주소)를 판별하는 문제는 어렵지 않다. 즉, PC 레지스터와 연산 되지 않은 값은 주소가 아니며 데이터 세그먼트에는 주솟값이 바로 저장되지 못하므로, 판별하기가 상대적으로 쉽다. 반면, 여러 단계에 걸쳐 계산되는 주솟값을 정확하게 추적 및 수집하는 것이 어렵다.

Fig. 3.은 위치 독립적 코드의 예로, 정확한 주소 수집이 어려운 것을 보여준다. 케이스 1의 경우를 보면, 주소 0x1000과 0x1004에서 두 번의 SUB 명령어를 통해 주소 0x800에 위치한 배열의 지점을 계산한다. 그리고 주소 0x1100에서 반복적으로 4바이트씩 더하는 것을 볼 수 있다. 예시의 주석으로 처리된 부분은 각 프로그램 지점에서 해당 레지스터가 가질 수 있는 값의 집합을 나타낸다.

일반적으로 주석으로 처리된 것과 같은 결과를 얻기 위해서는 값-집합 분석을 수행한다. 케이스 1의

```
CASE 1
 0x0800    DCD  0x01234567
 0x0804    DCD  0x89ABCDEF
  ...
 0x1000    SUB  R2, PC, #0x408  // R2 → {0xC00}
 0x1004    SUB  R3, R2, #0x400  // R3 → {0x800}
 0x1008    LDR  R4, [R3]        // R3 → {0x800,0x804,...}
  ...
 0x1100    ADD  R3, #4          // R3 → {0x804,0x808,...}
 0x1104    CMP  arbitrary
 0x1108    BNE  0X1008

CASE 2
 0x1100    CMP    R3, #0x0F
 0x1102    BHI.W  label
 0x1106    TBH    [PC,R3,LSL#1]
 0x1108    DCW    relative address
  ...
 0x1128    DCW    relative address

CASE 3
 0x1100    CMP    R3, #0x02
 0x1104    ADDLS  PC, PC, R3, LSL#2
 0x1108    B      label
  ...      B      label
 0x1118    B      label
```

Fig. 3. Example of position-independent code that makes it difficult to accurately collect addresses

주소 0x1000과 0x1004는 문제가 되지 않지만, 주소 0x1100에서 문제가 발생한다. 일반적인 값-집합 분석은 주소를 수집하기 위해 도메인을 인터벌(interval)이나 집합 형태로 디자인한다. 이런 도메인 구조를 가지는 분석에서는 주소 0x1104와 같은 조건문을 올바르게 처리하지 못한다. 따라서, 일반적으로 수집되는 결괏값은 0x804부터 4의 배수를 가지는 모든 정수가 될 수 있다. 즉, (0x804,∞,4)와 같은 결과가 나올 수 있으며, 이 결괏값에는 배열 영역이 아닌 명령어 영역도 포함된다. 이런 결괏값은 약간의 휴리스틱을 사용하여, 데이터 주소를 분류할 수 있지만, 모든 케이스를 다루지 못하며, 바이너리 계측 기법과 같이 정교한 수집 데이터를 요구하는 곳에서는 사용하기가 어렵다.

케이스 2는 점프 테이블을 이용한 간접 제어 흐름 점프이다. 여기서 레지스터 R3는 점프 테이블의 인덱스로 외부 값이 포함돼서 전파된다. 따라서 이 값을 확실히 알 수 없으므로, 주소 0x1100과 0x1104의 정확한 값을 구하기가 어렵다. 휴리스틱

으로 점프 테이블의 위치를 찾는다 하더라도, 점프 테이블의 크기를 값-집합 분석을 통하여 정확히 구하기 어려우므로, 정확한 점프 테이블의 대상 주소를 수집하기 어렵다. 케이스 3은 점프 테이블이 B 명령어로 치환된 변형으로 케이스 2와 동일한 문제점을 가지게 된다.

정리하면, 위치 독립적 코드는 위치 종속적 코드와 다르게 데이터 세그먼트나 명령어 즉시 값(immediate value)에 주소가 직접적으로 배치되지 않는다. 따라서, 상수의 구분이 명확하다. 하지만, 경우에 따라 주소 계산이 여러 단계에 걸쳐서 이루어지므로, 주소 계산을 모두 추적하기 까다롭다. 특히 정적 역어셈블리의 경우, 요약 해석[22,23]에 기반을 둔 값-집합 분석의 한계로 인해 수집하는 값에 불필요한 결과가 많이 포함될 수 있다. 이런 이유로 인해, 계산되는 명령어 주소를 정확하게 알기 어렵다.

IV. 정확도 향상을 위한 접근법

ARM 명령어로 구성된 위치 독립적 코드는 다른 코드와 차별되는 특징으로 인하여 역어셈블리가 까다롭다. 하지만, 이 특징을 고려하여 값을 수집할 경우 기존 방법보다 정확하게 수집할 수 있다. 그리고 대표적인 gcc 및 clang 컴파일러의 코드 생성 특징을 활용하여, 정확도를 높인다.

4.1 주소 수집에 특화된 값-집합 분석

x86 아키텍처를 대상으로 개발된 값-집합 분석의 도메인은 위치 종속적 코드를 대상으로 디자인된 것이 대다수이다[25-27]. 따라서, 상수와 주소의 구분이 명확하지 않은 상태에서 값을 수집하게끔 디자인되어 있다. 하지만, 위치 독립적 코드는 구분이 명확하므로, 이점을 고려하여 값-집합 분석을 고려하면 더 정확하게 수집할 수 있다. 다음은 주소 수집을 위해 고려할 특징이다.

- PC 레지스터와 상수의 연산 결과는 주소이다.
- 주소와 주소의 연산 결과는 오프셋이다.
- 그 외의 값은 상대 주소(상수)이다.

x86 아키텍처를 포함하여[4], 상대 주소 계산에 연관되는 명령어는 한정적이다. 이론적으로는 많은 명령어가 사용될 수 있다. 하지만, 주요 컴파일러가 생성하는 코드 패턴을 보면, 10개 미만의 명령어만 주소 계산에 관련된다. 따라서 값-집합 분석을 수행할 때, 한정된 명령어의 결괏값만 전파 및 추적하면 된다. 그 외의 연산은 주소와 관련 없는 결과이므로 무시한다.

4.2 기호 형태의 도메인

우리의 목표는 단순한 역어셈블리를 위한 것이 아닌, 바이너리 계측과 같은 응용 분야의 활용을 염두에 둔다. 따라서, 여러 단계로 계산되는 주소의 생성 지점을 추적 가능해야 한다. 우리는 이를 위해 도메인을 기호(symbol) 형태로 수집한다. 기호 형태로 수집할 경우, 두 가지 이점이 있다. 첫 번째로, 어떤 값이 주어지면 그 값을 구성하는 값의 생성 지점을 확인할 수 있다. 두 번째로, 상수와 주소의 구분이 기호를 통해 바로 구분할 수 있다.

위치 종속적인 코드를 대상으로 하는 일반적인 값-집합 분석은 상수와 주소의 구분이 불분명하므로 단순히 연산 결괏값을 요약하여 수집한다. 반면, 위치 독립적인 코드는 분명하므로 기호로 수집하는 것이 유리하다. 그리고 생성되는 각각의 기호에 명령어 주소를 기록할 경우, 값 자체가 경로에 민감(path-sensitive)하게 되므로 프로그램 지점의 상태 값을 단순하게 디자인할 수 있다.

4.3 함수 내 분석 (intra-procedural analysis)

위치 독립적 코드에서 값-집합 분석을 통해 전파하는 값은 상대 주소(또는 상수)와 계산된 주소이다. 우리의 관찰 결과, 특정 주소를 생성하기 위해 사용되는 상대 주소는 그 주소를 계산하는 프로시저(또는 함수) 내에서만 얻어진다. 즉, 어떤 주소를 계산하기 위해 외부에서 얻은 값을 이용하여 주소 계산을 하지 않는다는 것을 의미한다. 따라서, 값-집합 분석을 함수 단위로만 분석하여도 모든 주소를 계산할 수 있다.

이해를 돕기 위해, 소스 수준에서 어떤 함수의 주소나 데이터 구조(또는 변수)의 주소가 명시된 것을 생각해보자. 이 코드를 컴파일러가 생성할 경우, 명시된 함수 내에서 어셈블리 코드를 생성할 것이다. 따라서, 함수 내 분석을 통할 경우, 최소한 변수의 시작 주소나 함수의 시작 주소는 구할 수 있음을 의

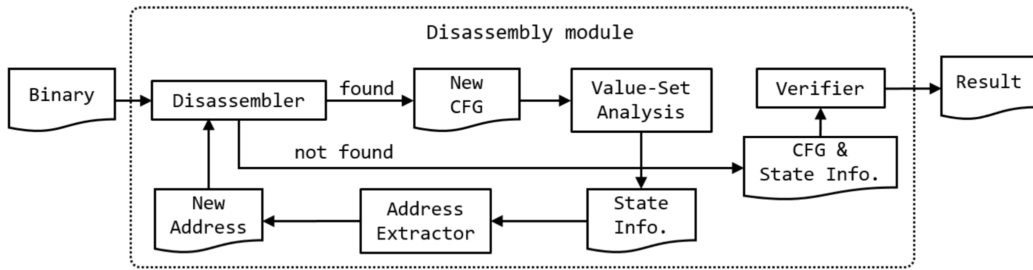

Fig. 4. Overview of disassembly process

미한다. 수작업으로 일반적인 형태를 벗어난 구현에서는 이 방법이 적합하지 않지만, 우리는 이런 경우는 고려하지 않는다.

계산된 특정 시작 주소가 다른 프로시저에 전달되어 추가적인 오프셋 계산을 수행할 수 있다. 대표적으로 배열이나 객체의 시작 주소가 이에 해당한다. 하지만, 우리는 접근하는 모든 주소를 구하는 것이 아니므로, 이런 경우는 무시한다. 또한, 자가수정 코드와 같이, 명령어 영역을 오프셋을 통해 접근할 수 있다. 일반적인 프로그램에서는 자가 수정 코드가 존재하지 않으며, 우리는 이런 경우는 다루지 않는다.

4.4 리터럴 풀의 배치 속성 활용

주소의 타입을 확인할 때, 문제가 되는 부분은 리터럴풀이다. 리터럴풀 때문에 코드 세그먼트를 가리키더라도 리터럴풀 속 데이터 주소일 가능성이 있어 함수의 주소라고 바로 판단하지 못하게 된다. 우리는 이 문제를 해결하기 위해, 리터럴풀이 배치되는 특성을 이용하고자 한다.

리터럴풀에는 불변하는 속성(immutable)의 값이 배치된다. ARM의 경우, 일반적으로 4바이트 상대 주소, 4바이트 이상의 정수 및 부동 소수, 배열, 문자열 등이 존재한다. 이 값들은 소스 수준으로 보았을 때, 특정 프로시저 내의 상숫값들에 해당하며, C언어 같은 경우 const 타입의 지역 변수(다른 프로시저의 스코프에 보이지 않는) 값들이 여기에 배치된다. 나머지의 값들의 경우, 읽기 전용 데이터 섹션이나 쓰기가 가능한 데이터 섹션에 배치가 된다.

이런 특징으로 인해, 컴파일러는 오브젝트를 생성할 때, 리터럴풀을 해당 서브루틴에 인접하게 배치시킨다. 우리의 관찰 결과에 따르면, gcc나 clang의 경우 보통 프로시저 하단에 배치하며, 프로시저의 크기가 크면 중단에도 배치한다. 그리고 간혹 배열과 같은 데이터를 함수 상단에 배치하는 경우도 존재하였다. 즉, 프로시저와 리터럴풀을 하나의 묶음으로 관리한다.[2] 이런 구조는 링킹 과정을 고려해 보더라도 자연스러운 방식이다.

우리는 더욱 정확한 확인을 위해, 갤럭시S8 (SM-G955N) 안드로이드 8.1 (R16NW.G955NKSU1CRD7)에 포함된 모든 공유라이브러리 및 실행 파일 669개를 조사하여, 리터럴풀의 배치 속성을 확인하였다. 그 결과, 우리의 가정에 어긋나는 사항을 확인하지 못하였다. 이 특징을 가정할 경우, 컴파일 시 사라진 정보를 어느 정도 대체할 수 있는 속성을 얻을 수 있으며, 다음과 같다.

> *프로시저 내에서 수집한 주소가 코드 세그먼트를 가리킬 경우, 해당 프로시저의 리터럴풀이거나 함수 주소이다.*

우리는 이 속성을 주소의 타입을 판별하는 데 활용하며, 자세한 내용은 5장에서 다루도록 한다.

V. 제안하는 역어셈블리 기법

Fig. 4.는 제안하는 역어셈블러의 기본적인 구조이다. 바이너리를 입력으로 받으며, 프로그램의 시작지점, 익스포트(export)된 심볼의 함수 주소, 그리고 재배치, 다이나믹, .init, .fini, 섹션 등에 포함된 함수 주소를 시작점으로 역어셈블리를 수행한다. 그리고 이를 통해 도출된 CFG를 기반으로 값-집합 분석을 수행하고 결과를 바탕으로 새로운 주소를 추출한다. 발견한 새로운 주소에서 다시 역어셈블리를 수행하며, 새로 발견되는 주소가 없을 때까지 이 작업을 반복한다. 마지막으로, 도출된 CFG와 값-집합

2) 오브젝트 속 함수의 심볼 정보를 보면, 리터럴풀을 함수의 단위로 같이 지정하고 있다.

분석의 결과를 검증함으로써 역어셈블리를 마친다.

5.1 기본적인 수행 구조

우리의 역어셈블리 기법은 정확성에 주요 목표를 둠으로, 선형 어셈블리 기법을 사용하지 않는다. 즉, 재귀 어셈블리 방법만을 사용하여 명령어 영역을 식별하며, 이를 통해 식별한 명령어가 실제 명령어임을 보장받을 수 있다. Fig. 5.는 우리 역어셈블러(Fig. 4.의 disassembler 파트)의 기본적인 알고리즘을 나타낸 것이다.

역어셈블리를 수행할 서브루틴의 주소가 주어지면, Disasm_Function을 통해 해당 서브루틴의 역어셈블리를 수행한다(#3). 이를 통해 함수 및 블록 정보(function_map, block_map)를 지속적으로 수집하며(#5,#21), 부가적으로 노리턴 함수(no_return_func_list)의 정보를 모은다(#7).

실질적인 역어셈블리는 Disasm_at 함수(#12)에서 이뤄진다. 시작 주소부터 엔트리 기본 블록을 구하고(#13), 새로운 분기 주소를 얻으면(#23-24), 그 지점부터 다시 기본 블록을 구한다. 더 이상 새로운 분기 주소를 구할 수 없을 때까지 수행한다(#15-26).

서브루틴에 간접 제어 흐름 전달 구문(indirect control flow transfer)이 있을 경우(#22), 각각의 주소를 역어셈블리 한다(#29). 여기서 간접 호출은 고려하지 않고, 점프 테이블을 사용한 간접 점프만 고려한다. 간접 점프도 해당 프로시저의 일부분이므로 결괏값을 앞선 결과와 병합하며(#30) 최종적으로 합쳐진 결괏값을 리턴한다(#33).

Disasm_Block(#36)은 기본 블록 시작 지점부터 분기 지점까지 명령어를 역어셈블 하는 함수로, 주어진 주소의 명령어를 디코딩하는 Decode_at 함수(#39)를 이용한다. Fig. 5.에 명시된 이 함수는 기본적인 구조만 언급되어 있다. 이전에 구한 기본 블록의 중간을 가리키는 경우와 같이 다양한 과정이 존재하지만, 전반적인 흐름의 설명을 위해 제외한다.

5.2 노리턴 함수 처리 방법

재귀 역어셈블리 방법만 사용할 경우, 노리턴 함수는 큰 문제가 된다. 일반적으로 BL 및 BLX와 같은 함수 호출 명령어는 기본 블록을 구할 때, 분기 명령어로 고려하지 않는다. 하지만, 노리턴 함수의

```
 1: global function_map, block_map, no_return_func_list
 2:
 3: procedure Disasm_Function(addr)
 4:     if addr ∉ function_map then
 5:         function_map[addr] ← Disasm_at(addr)
 6:         if function_map[addr].is_no_return() then
 7:             no_return_func_list.add(addr)
 8:         end if
 9:     end if
10: end procedure
11:
12: procedure Disasm_at(addr)
13:    queue.put(addr)
14:    iCFT ← {}
15:    while not queue.empty() do
16:        tgt_addr ← queue.pop()
17:        if tgt_addr ∈ block_map then
18:            continue
19:        end if
20:        disasm_result ← Disasm_Block(tgt_addr)
21:        block_map[tgt_addr] ← disasm_result.get_bk()
22:        iCFT.update(disasm_result.get_iCTF())
23:        for b ∈ disasm_result.get_branch_addr() do
24:            queue.put(b)
25:        end for
26:     end while
27:     for i ∈ iCFT do
28:         for a ∈ analysis_iCFT(i) do
29:             disasm_result' ← Disasm_at(a)
30:             disasm_result.merge(disasm_result')
31:         end for
32:     end for
33:     return disasm_result
34: end procedure
35:
36: procedure Disasm_Block(addr)
37:     disasm_result ← init()
38:     while true do
39:         result ← Decode_at(addr)
40:         if result.id = BL then
41:             tgt ← result.branch_target_addr
42:             if tgt ∈ no_return_func_list then
43:                 return disasm_result
44:             else if tgt ∉ function_map then
45:                 Disasm_Function(tgt)
46:             end if
47:             ... // something
48:         else if result.is_branch_instrunction() then
49:             return disasm_result
50:         else if result.is_instrunction() then
51:             ... // something
52:         else
53:             raise error
54:         end if
55:     end while
56: end procedure
```

Fig. 5. Disassembly algorithm

경우에는 해당 명령어를 마지막 지점으로 하는 기본 블록이 만들어져야 한다. 즉, 해당 명령어 다음 지점으로 역어셈블이 수행되어서는 안 된다.

대표적인 노리턴 함수로 exit와 abort 함수가 있다. 이런 함수는 호출한 서브루틴으로 돌아오지 않고 해당 함수에서 바로 종료한다. 컴파일러는 이런 함수의 호출 지점을 파악하여 기본 블록을 만든다. 따라서, 노리턴 함수를 고려하지 않으면 잘못된 CFG가 생성되며, 경우에 따라서는 데이터 영역을 역어셈블리 하는 경우가 생기기도 한다.

이뿐만 아니라, 노리턴 되는 함수가 끝 지점에 배치된 함수도 고려해야 한다. 예를 들어, A 함수가 exit나 abort로 종료되게끔 작성되어 있고, B 함수는 A 함수로 종료되게끔 작성되어 있다고 하자. 컴파일러는 이런 경우도 파악하여 코드를 작성한다. 즉, B 함수 같은 경우를 처리하지 않을 때도 역어셈블 에러가 발생할 수 있다.

우리는 응용 프로그램 이진 인터페이스 (application binary interface, ABI)를 참고하여 PLT 재배치 섹션의 심볼을 통해 C 및 C++의 노리턴 함수의 이름을 비교한다. 그리고 이런 노리턴 함수를 이용한 사용자 정의 노리턴 함수를 처리하기 위해, 우리는 서브루틴의 역어셈블을 재귀적으로 처리한다.

Disasm_Block 함수에서 BL 명령어가 디코딩된 경우(#40)를 보자. 대상 주소가 사전에 발견한 노리턴 함수일 경우(#42), 기본 블록 생성을 중지한다. 그렇지 않고 아직 발견한 함수도 아닐 경우에는(#44), 그 함수를 바로 역어셈블리 한다(#45). 호출 대상 함수가 역어셈블리되면, 노리턴 정보가 업데이트되며, 이 정보를 가지고 적절한 작업을 수행한다(#47).

5.3 점프 테이블의 크기 유추법

점프 테이블을 사용하는 간접 점프를 처리하는 것에 있어 가장 어려운 점은 점프 테이블의 크기를 파악하는 것이다. 우리는 이를 해결하기 위해, 코드의 특징을 이용한 방법을 제안한다. 이런 간접 점프는 C 언어나 C++의 스위치문을 통해 생성되며 gcc나 clang 컴파일러를 통해 생성할 경우, 일반적으로 인덱스(소스 수준에서는 스위치 상수)를 검증하는 비교 명령어가 붙게 된다. 우리는 이 비교 명령어의 상숫값과 뒤이어 나오는 분기문의 조건 접미사를 보고 테이블의 크기를 유추한다.

Fig. 3.의 케이스 2, 3은 ARM 위치 독립적 코드에서 자주 볼 수 있는 간접 점프 코드 패턴이다. 두 케이스 모두 레지스터 R3가 점프 테이블의 인덱스이며, 점프 테이블의 오프셋을 계산하기 전 CMP 명령어로 인덱스의 크기를 검증한다. 케이스 2의 경우에는 상수 15와 비교를 한 뒤 접미사 HI(15 초과)로 벗어난 경우를 필터링한다. 즉, 15 이하의 값만 오프셋 계산에 사용된다. 케이스 3의 경우 2와 비교를 하며, 접미사 LS(2 이하)를 통해 2 이하의 값만 오프셋 계산에 사용된다. CMP 명령어는 비부호 4바이트로 값을 확장하여 비교하므로 오프셋의 시작은 0부터이다.

제안한 방법은 대다수의 경우 적용할 수 있다. 4.4장의 실험 대상을 통해 확인해 본 결과, 세 가지 경우를 제외한 나머지 모든 인덱스의 크기를 유추할 수 있었다. 이 방법이 적용하기 어려운 경우, 우리는 BinCFI에서 사용한 휴리스틱[4]을 사용한다. 이 방법은 순차적으로 오프셋 테이블을 읽으면서 계산된 대상 주소가 올바른 지점을 가리키지 않을 때까지 수행한다.

5.4 주소 타입 유추법

우리가 제안하는 주소의 타입 판별 기법은 기본적으로 단순하다. 주소가 주어지면 그 지점을 시작으로 역어셈블리를 수행한다. 그 과정에서 발견되는 직접 점프 및 호출도 역어셈블리를 수행하며, 모든 과정에서 에러가 없을 경우 주소 타입이라고 판단한다. 그렇지 않을 경우에는 데이터 타입이라고 판단한다.

하지만, 불행하게도 이 방법은 잘못된 결과를 도출하는 경우가 생기며, Fig. 6.은 그 경우를 보여준다. 실험 결과에 따르면, 리터럴풀 끝 부근의 주소를 제시한 방법으로 수행할 경우 데이터를 정상적으로 역어셈블리하고 명령어 영역으로 넘어가게 되는 경우가 발생한다.

이 방법의 성공률을 가늠해보기 위해 4바이트 전 범위에서 정상적인 명령어의 개수를 확인하였으며, 그 결과는 다음과 같다:

Inst. set	Success	Failure	Succ.%
ARM mode	0xDCDC6637	0x232399C9	86.27%
Thumb mode	0xF33012C1	0xCCFED3F	94.99%

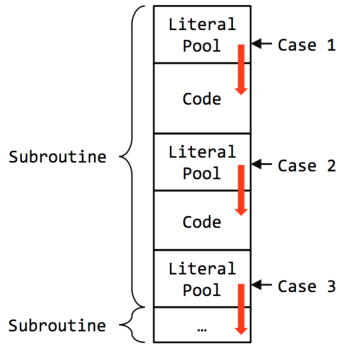

Fig. 6. Cases in which identification through disassembly produces incorrect results

이 결과는 임의의 데이터를 디코딩하였을 때 대략 90% 확률로 실패하는 것을 의미한다. 따라서, 리터럴풀 마지막 12바이트 정도를 우리의 방식으로 걸러내기에는 부적절하다. 또한, 리터럴풀의 끝 지점을 패딩할 때에 NOP 명령어를 사용하는 경우가 많으므로 실패율이 낮아지지 않는다.

우리는 이 제시한 방식의 약점을 보완하기 위해 리터럴풀의 특징을 이용한다. 4.4장에서 언급한 특징을 가정할 경우, 코드 세그먼트를 가리키는 주소가 데이터를 가리키면 Fig. 6.의 케이스 1,2,3 중 하나에 해당된다. 즉, 코드 세그먼트에서 데이터를 가리키는 것은 자신의 리터럴풀 밖에 없다는 것을 의미한다.[3] 우리는 주소 판별 과정을 Fig. 4.의 주소 추출(address extractor) 과정에서 하므로, 해당 서브루틴의 CFG를 구한 상태이다. 따라서, 케이스 1과 2가 발생할 경우 쉽게 판별할 수 있다.

케이스 3은 다음 서브루틴의 역어셈블리 여부가 보장되지 않으므로, 판별에 실패할 수 있다. 따라서 우리는 판별에 통과한 서브루틴의 엔트리 기본 블록을 따로 가지고 있으며, 주소 추출 단계 마지막에서 엔트리 블록의 가운데 지점을 가리키는 함수 시작 지점이 있는지 매번 검증한다. 모든 역어셈블리 과정이 끝날 때까지 검증에 통과한 주소를 명령어 주소 타입이라 판별한다.

VI. 기호 형태의 값-집합 분석

정확한 주소의 수집과 응용 기술을 위한 정보 제공을 위해, 우리는 도메인이 기호 형태로 구성된 값-집합 분석을 제안한다.

6.1 분석 도메인

Fig. 7.은 제안하는 값-집합 분석의 도메인을 나타낸다. 도메인을 보면 총 8 종류의 값 형태가 있으며, 명령어가 6개를 차지하고 있다. 대상으로 하는 컴파일러에 따라 주소 계산에 다른 명령어가 사용될 수 있으며, 그런 경우 도메인에 추가하면 된다. 우리가 대상으로 하는 컴파일러는 gcc와 clang이며, 해당 컴파일러에서는 언급된 명령어만이 주소 계산에 사용되었다.

각 값은 해당 값이 생성되는 주소를 포함하도록 디자인되어 있다. 이를 통해 어떤 값이 주어졌을 때, 그 값을 구성하는 값의 위치를 바로 알 수 있다. 또한, 주소 정보로 인해, 프로그램 각 지점의 상태를 경로에 따라 분리하여 수집할 필요가 없다. 값 자체가 경로에 민감(path-sensitive)하기 때문에, 하나의 상태로 레지스터의 값을 수집하더라도 값을 통해 구분할 수 있다.

ARM에서 산술 연산을 2의 보수를 기반으로 수행된다. 하지만, LDR와 STR와 같은 일부 명령어에서의 유효 주소(effective address)은 이를 따르지 않는다. 명령어에 포함된 즉시 값이 2의 보수로 이뤄져 있는 것이 아닌, 명령어 특정 필드 값에 따라 즉시 값의 부호가 결정된다. 따라서 이를 반영하기 위해 즉시 값의 형태는 정수 형태를 가지며, 기호 값

$$
\begin{aligned}
stt \in \quad & State &= \; & Registers + Stack \rightarrow 2^{Value} \\
reg \in \quad & Registers &= \; & \{\mathrm{R0}, \cdots, \mathrm{R12}, \mathrm{SP}, \mathrm{LR}, \mathrm{PC}\} \\
stk \in \quad & Stack &= \; & \{\mathrm{SP}\} \times off \\
imm \in \quad & Immediate &= \; & \left\{-2^{32}, \cdots, -1, 0, 1, \cdots, 2^{32}-1\right\} \\
off \in \quad & Offset &= \; & \mathbb{Z}
\end{aligned}
$$

$$
\begin{aligned}
addr \in \; & \left\{0, 1, 2, \cdots, 2^{32}-1\right\} \\
mode \in \; & \{\mathrm{ARM}, \mathrm{THUMB}\} \\
val \in \; & Value \\
val ::= \; & \mathrm{IMM}(imm \times addr) \\
& \mid \mathrm{PC}(mode \times addr) \\
& \mid \mathrm{MOV}(val \times addr) \\
& \mid \mathrm{MOVT}(val \times val \times addr) \\
& \mid \mathrm{ADD}(val \times val \times addr) \\
& \mid \mathrm{SUB}(val \times val \times addr) \\
& \mid \mathrm{LDR}(val \times val \times addr) \\
& \mid \mathrm{ADR}(mode \times val \times addr)
\end{aligned}
$$

Fig. 7. Domain of value-set analysis

3) 산술 연산 명령어가 계산한 값의 특징을 의미하며, 계산된 값이 파라미터로 전달되어 다른 곳에서 사용되는 것을 의미하지 않는다.

을 계산할 때에는 각 명령어의 정의에 따라 계산한다.

우리의 관찰에 따르면, 주소 계산에 사용되는 상대 주소나 중간 결괏값들은 경우에 따라 스택에 임시로 저장되기도 한다. 이를 반영하기 위해, 도메인에 레지스터뿐만 아니라 스택의 오프셋도 추가시킨다. 오프셋만으로도 스택에 저장되는 값을 구분하며, 우리는 함수 내 분석만 해서 문제가 되지 않는다.

6.2 전이 함수 (transfer function)

Table 1.은 해당 분석의 전이 함수를 보여준다. 기존 분석과 다른 점으로, 세 가지 전달 제약 사항이 있다. 첫 번째 제약 조건은 checkIter 함수로 나타난다. 이 제약 조건은 대상 명령어의 피연산자로 전달받은 값이 해당 지점에서 다시 사용되는 경우, 그 결괏값은 전파하지 않는 것이다. 우리의 목표는 정확한 명령어 주소를 수집하는 것이다. 일반적인 프로그램의 경우 계산된 주소가 다시 재계산에 사용되는 경우는 데이터 주소를 제외하고는 거의 없다. 또한, 우리는 자가수정 코드를 다루지 않는다. 따라서 이 제약 조건은 정확도를 낮추지 않는다.

이 제약 조건을 통해 우리는 분석을 유한시간 내에 종료할 수 있게 해주는 것과 더불어 정확도를 높여준다. 일반적으로 분석에서 반복문 안의 상태를 수집하기 위해, 상태를 요약하거나 일정 크기 이상의 상태를 버리기도 한다. 이런 이유로 분석의 정확도가 내려가게 되는데, 우리가 수집하고자 하는 대상은 반복적으로 계산되는 요소가 없으므로 이 제약 조건을 통해 정확하게 수집할 수 있게 된다.

두 번째 제약 조건은 isPcBased 함수로 나타난다. 이 제약 조건은 주소 계산에서 베이스 값 PC 레지스터를 기준으로 생성된 주소만 다루도록 한다. 이 제약 조건은 특정 지점에서 로드된 값이거나 상대 주소를 기반으로 계산되지 않은 값이 베이스로 사용되는 것을 저지한다. 이는 위치 독립적 코드 특성상으로 인해 존재하는 제약 조건이다. 경우에 따라 데이터 세그먼트에 주소가 배치될 수가 있지만, 이 정보는 재배치 심볼에 해당 정보가 있으며, 이를 베이스로 하는 주소 계산은 추적할 필요가 없다.

마지막으로 세 번째 제약 사항은 isCodeAddr 함수로 나타난다. 이 제약 조건은 베이스가 코드를 가리키는 주소가 아닌 경우, 즉 데이터를 가리키는 주소일 경우 더는 추적하지 않도록 한다. 우리의 목적은 접근하는 모든 주소를 추적하는 것이 아닌 함수와 데이터 구조(또는 청크)의 시작 주소를 수집하는 것이므로, 이 제약 조건을 둔다. 단, 이 조건이 없어도 동일한 분석을 수행할 수 있지만 수집하는 상태의 크기를 줄일 수 있어 분석의 규모를 키울 수 있고 후처리가 간단해진다.

VII. 실험 및 평가

우리는 역어셈블리의 정확도를 평가하기 위해 갤럭시S8(SM-G955N) 안드로이드 8.1(R16NW.G955NKSU1CRD7)에 포함된 공유라이브러리와 실행파일 669개를 대상으로 실험하였다. 역어셈블리의 정확도는 코드 세그먼트의 커버리지로 판단한다. 발견한 명령어 영역과 리터럴풀의 영역이 코드 세그먼트 전체 영역을 100% 커버할 경우 역어셈블리가 성공했다고 판단하며, 그렇지 않으면 실패했다고 판단한다.

풀 커버리지를 검증할 때, 리터럴풀이 문제가 된다. 이것은 데이터 영역이므로 경우에 따라서는 역참조가 되지 않을 수 있다. 문자열이나 배열 등의 경우는 주소만 계산되므로, 영역 식별에서 배제될 수 있

Table 1. Transfer function of value-set analysis

Instruction Format	Propagation
mov d, m	$stt(d) \leftarrow stt(m)$
mov d, i	$stt(d) \leftarrow \{\text{MOV}(\text{IMM}(i, a), a)\}$
movt d, i	$stt(d) \leftarrow \{\text{MOVT}(v, \text{IMM}(i, a), a) \mid stt(d) \neq \phi \wedge v \in stt(d)\}$
add d, m, n	$stt(d) \leftarrow \{\text{ADD}(v_1, v_2, a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge \neg checkIter(v_1, a) \wedge \neg checkIter(v_2, a) \wedge (isPcBased(v_1) \wedge isCodeAddr(v_1) \vee isPcBased(v_2) \wedge isCodeAddr(v_1))\}$
add d, m, i	$stt(d) \leftarrow \{\text{ADD}(v, \text{IMM}(i, a), a) \mid v \in stt(m) \wedge isPcBased(v) \wedge \neg checkIter(v)\}$
sub d, m, n	$stt(d) \leftarrow \{\text{SUB}(v_1, v_2, a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge isPcBased(v_1) \wedge isCodeAddr(v_1) \wedge \neg checkIter(v_1, a)\}$
sub d, m, i	$stt(d) \leftarrow \{\text{SUB}(v, \text{IMM}(i, a), a) \mid v \in stt(m) \wedge isPcBased(v) \wedge isCodeAddr(v) \wedge checkIter(v)\}$
ldr $d, [m, n]$	$stt(d) \leftarrow \{\text{LDR}(v_1, v_2, a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge isPcBased(v_1) \wedge \neg checkIter(v_1, a) \wedge \neg checkIter(v_2, a)\}$
ldr $d, [m, i]$	$stt(d) \leftarrow \{\text{LDR}(v, \text{IMM}(i, a), a) \mid v \in stt(m) \wedge isPcBased(v) \wedge \neg checkIter(v)\}$
ldr $d, [\text{SP}, i]$	$stt(d) \leftarrow stt(\text{SP}, i)$
str $d, [\text{SP}, i]$	$stt(\text{SP}, i) \leftarrow stt(d)$
adr i	$stt(d) \leftarrow \{\text{ADR}(getMode(), \text{IMM}(i, a), a)\}$
others	$stt(d) \leftarrow \phi$ for each d $\in getDestinationRegs()$

다. 또한, 리터럴풀의 정렬을 위해서 NOP 명령어가 삽입되는데, 우리의 기법은 재귀 어셈블리 방법만 사용하므로 이 영역 또한 식별하지 못한다. 이런 경우는 수작업으로 확인하였다.

우리 역어셈블리 기법의 목표는 정확도이므로, 100% 커버리지를 달성한 결과에 한해 추가로 검증 과정을 거친다. 명령어에서 역참조하는 리터럴풀의 영역과 식별된 명령어 영역이 겹치는 지점이 있는지 확인한다. 이는 리터럴풀과 명령어 영역이 중첩되지 않음을 가정한 것으로, 일반적인 컴파일러가 생성하는 코드에서는 자연스러운 것이다. 이런 경우는 일반적으로 노리턴 함수를 올바르게 처리하지 못하는 상황에서 발생한다. Table 2.는 해당 실험 결과를 보여준다.

우리의 기법은 COTS 바이너리를 대상으로 약 91.47%의 풀 커버리지 역어셈블리 성공률을 보인다. 주된 실패 요인은 두 가지로, 자동 생성된 코드와 데드 코드로 보이는 함수이다. 첫 번째로, 특정 라이브러리를 사용할 경우 예외처리 등의 목적으로 컴파일러 과정에서 일부 코드 조각이 자동으로 추가되는 것이 있다. 우리의 관찰 결과, 이런 코드는 따로 사용하지 않더라도 코드 세그먼트에 남아 있는 것으로 판단된다.

다른 이유로 사용자가 정의한 서브루틴으로 보이는 함수가 존재한다. 이런 코드가 존재하는 정확한 이유를 찾지 못하였으며, 개발 단계에서 버려지는 코드가 컴파일된 것으로 보인다. 검증을 위해, 대표적인 상용도구인 IDA Pro를 통하여 확인하였으며, 이 도구에서도 해당 영역을 참조하는 명령어를 발견하지 못하였다. 이 밖에도 점프 테이블을 사용하는 간접 점프를 올바르게 처리하지 못하여 실패하는 경우가 존재하였다.

Table 3.은 우리의 역어셈블러와 대표 상용 도구인 IDA Pro(Ver. 6.8)를 비교한 것이다. 참고로 대표적인 비상용도구인 Angr[28]도 실험하였으나, 이 도구는 점프 테이블을 이용한 간접 점프를 전혀 다루지 않아 정상적인 비교를 할 수 없었다. 그리고 IDA Pro는 실제 식별되지 않은 영역을 강제로 역어셈블리하여, 데드 코드로 보이는 영역도 역어셈블리한다. 따라서 정확한 비교를 위해 참조가 없는 영역은 발견하지 못한 것으로 두고 집계하였다.

이밖에도 다양한 성능 비교를 위해, 역어셈블러 Radare2[29]의 GUI 확장 도구인 Cutter[30]를 대상으로 실험하였다. 하지만, 일부 샘플에서 에러가 발생하고, ARM의 점프 테이블을 이용한 간접 점프를 대다수 인식하지 못하여 낮은 정확도를 보여주었다. 예로, Fig. 3.의 두 번째 케이스의 경우, 간접 점프의 식별은 하였으나 테이블의 크기를 유추하지 못하였으며, 세 번째 케이스의 경우에는 일반 명령어로 처리하여 완전히 잘못된 CFG를 도출하였다. 이 밖에도 thumb 모드에서 스위치문을 도와주는 함수 호출 등을 인식하지 못하여 상대적으로 낮은 정확도를 보여주었다.4)

Table 2. Rate of full coverage of disassembly

path	/system/lib/*	/system/bin/*	total	prop.
succ	590	22	612	91.47%
fail	56	1	57	8.53%
total	646	23	669	100%

Table 3. Accuracy comparison between IDA Pro and our disassembler

path	Our disassembler	IDA Pro
succ	612	615
fail	57	54
prop.	91.47%	91.92%

VIII. 결 론

스마트 모바일이 보급됨에 따라 ARM 위치 독립적 코드의 사용이 높아지고 있으며, 이에 따라 보안 응용 기술의 적용 요구가 점차 높아지고 있다. 하지만, 기존 역어셈블리 기법은 범용적이며 ARM 위치 독립적 코드에 바이너리 계측과 같은 보안 응용 기술을 적용할 수 있을 정도로 정확하지는 않다. 이를 해결하기 위해 우리는 특화된 값-집합 분석과 컴파일러가 생성하는 코드의 특징을 활용하여 정확도 높은 역어셈블리 기법 제안하였다.

우리의 기법은 바이너리 계측에 사용될 수 있을 정도로 신뢰 높은 분석 정보를 수집하면서도, 약 90% 이상의 완전한 역어셈블리 성공률을 보이는 것이 특징이다. 이 기법을 통해, CFI, SFI, 난독화, 프로파일링 등과 같은 응용 작업을 소스 코드 없이

4) Thumb mode toolchain helpers for compact switch (https://chromium.googlesource.com/chromiumos/platform/ec/+/refs/heads/master/core/cortex-m0/thumb_case.S)

수행할 수 있을 것으로 판단하며, 특히 개인정보가 중요한 스마트 모바일의 보안 강화에 큰 도움을 줄 것으로 기대한다.

References

〔1〕 M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in Proc. CCS, pp. 340-353, Nov. 2005.

〔2〕 J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code," in Proc. PLDI, pp. 355-366, Jun. 2011.

〔3〕 U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in Proc. OSDI, pp. 75-88, Nov. 2006.

〔4〕 M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in Proc. USENIX Security, pp. 337-352, Aug. 2013.

〔5〕 B. Niu and G. Tan, "Modular Control-flow Integrity," in Proc. PLDI, pp. 577-587, Jun. 2014.

〔6〕 R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in Proc. SOSP, pp. 203-216, Jan. 1993.

〔7〕 A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and Language-independent Mobile Programs," in Proc. PLDI, pp. 127-136, May. 1996.

〔8〕 S. L. Graham, S. Lucco, and R. Wahbe, "AdaptableBinary Programs," in Proc. USENIX Security, pp. 315-325, Jan. 1995.

〔9〕 B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in Proc. USENIX ATC, pp. 293-306, Jun. 2008.

〔10〕 T.-c. Chiueh and F.-H. Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks," in Proc. ICDCS, pp. 409-428, Apr. 2001.

〔11〕 L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal War in Memory," in Proc. SP, pp. 48-62, May. 2013.

〔12〕 M. Prasadand T.-c. Chiueh, "A BinaryRewriting Defense Against Stack based Buffer Overflow Attacks," in Proc. USENIX ATC, pp. 211-224, Jun. 2003.

〔13〕 A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-time Defense Against Stack Smashing Attacks," in Proc. pp. 251-262, Jun. 2000.

〔14〕 T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in Proc. ASIA CCS, pp. 555-566, Apr. 2015.

〔15〕 C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in Proc. CCS, pp. 290-299, Oct. 2003.

〔16〕 I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary Obfuscation using Signals," in Proc. USENIX Security, pp. 275-290, Aug. 2007.

〔17〕 H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P. Yew, "Control Flow Obfuscation with Information Flow Tracking," in Proc. MICRO, pp. 391-400, Dec. 2009.

〔18〕 Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary Component Extraction and Embedding for Software Security Applications," in Proc. ESORICS, pp. 200-218, Sep. 2013.

〔19〕 T. Kim, C. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D.

Xu, "RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications," in Proc. ACSAC, pp. 412-424, Dec. 2017.

〔20〕 D. Andriesse, X. Chen, V. van der Veen, A.Slowinska, and H. Bos, "An In Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in Proc. USENIX Security, pp. 583-600, Aug. 2016.

〔21〕 R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in Proc. DSN, pp. 201-212, Jun. 2017.

〔22〕 A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "MARX: Uncovering class hierarchies in C++ programs," in Proc. NDSS, pp. 1-15, Feb. 2017.

〔23〕 P. Cousot and R. Cousot, "Abstract Interpretation Frameworks," Journal of Logic and Computation, vol. 2, no. 4, pp. 511-547, 1992.

〔24〕 P. Cousot and R. Cousot, "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation," in Proc. PLILP, pp 269-295, Aug. 1992.

〔25〕 G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in x86 Executables," in Proc. CC, pp. 5-23, Apr. 2004.

〔26〕 J. Brauer, R. R. Hansen, S. Kowalewski, K. G. Larsen, and M. C. Olesen, "Adaptable value-set analysis for low-level code," in Proc. SSV, pp. 32-43, Aug. 2011.

〔27〕 Z. Zhang and X. Koutsoukos, "Generic value-set analysis on low-level code," in Proc. AVICPS, pp. 1-8, Dec. 2014.

〔28〕 C. Kruegel and Y. Shoshitaishvili, "Using Static Binary Analysis to Find Vulnerabilities And Backdoors in Firmware," presented at BlackHat USA, Las Vegas, NV, USA, Aug. 2015.

〔29〕 Radare2, "https://rada.re", 1. Oct. 2018.

〔30〕 Cutter, "https://github.com/radareorg/cutter", 1. Oct. 2018.

〈저자소개〉

하 동 수 (Dongsoo Ha) 학생회원
2010년 8월: 한양대학교 컴퓨터공학과 학사
2011년 3월～현재: 한양대학교 컴퓨터공학과 학사
〈관심분야〉 정보보호, 모바일 보안, 정적분석, 바이너리 분석

오 희 국 (Heekuck Oh) 종신회원
1982년: 한양대학교 전자공학과 학사
1989년: 아이오와주립대학 전자계산학과 석사
1992년: 아이오와주립대학 전자계산학과 박사
1993년～1994년: 한국전자통신연구원 선임연구원
1995년 3월～현재: 한양대학교 컴퓨터공학과 교수
〈관심분야〉 정보보호, 암호프로토콜, 시스템보안