Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.1, Feb. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.1.45

DEX와 ELF 바이너리 역공학 기반 안드로이드 어플리케이션 호출 관계 분석에 대한 연구*

안 진 웅,† 박 정 수, 응 웬 부 렁, 정 수 환‡
숭실대학교

Android Application Call Relationship Analysis Based on DEX and ELF Binary Reverse Engineering*

Jinung Ahn,† Jungsoo Park, Long Nguyen-Vu, Souhwan Jung‡
Soongsil University

요 약

DEX 파일과, SO 파일로 알려진 공유 라이브러리 파일은 안드로이드 어플리케이션의 행위를 결정짓는 중요한 구성요소이다. DEX 파일은 Java 코드로 구현된 실행파일이며, SO 파일은 ELF 파일 형식을 따르며 C/C++와 같은 네이티브 코드로 구현된다. Java 영역과 네이티브 코드 영역은 런타임에 상호작용할 수 있다. 오늘날 안드로이드 악성코드는 지속적으로 증가하고 있으며, 악성코드로 탐지되는 것을 회피하기 위한 다양한 우회 기법을 적용한다. 악성코드 분석을 회피하기 위하여 분석이 어려운 네이티브 코드에서 악성 행위를 수행하는 어플리케이션 또한 존재한다. 기존 연구는 Java 코드와 네이티브 코드를 모두 포함하는 함수 호출 관계를 표시하지 못하거나, 여러 개의 DEX을 포함하는 어플리케이션을 분석하지 못하는 문제점을 지닌다. 본 논문에서는 안드로이드 어플리케이션의 DEX 파일과 SO 파일을 분석하여 Java 코드 및 네이티브 코드에서 호출 관계를 추출하는 시스템을 설계 및 구현한다.

ABSTRACT

DEX file and share objects (also known as the SO file) are important components that define the behaviors of an Android application. DEX file is implemented in Java code, whereas SO file under ELF file format is implemented in native code(C/C++). The two layers – Java and native can communicate with each other at runtime. Malicious applications have become more and more prevalent in mobile world, they are equipped with different evasion techniques to avoid being detected by anti-malware product. To avoid static analysis, some applications may perform malicious behavior in native code that is difficult to analyze. Existing researches fail to extract the call relationship which includes both Java code and native code, or can not analyze multi-DEX application. In this study, we design and implement a system that effectively extracts the call relationship between Java code and native code by analyzing DEX file and SO file of Android application.

Keywords: Android, Malware, Static Analysis

Received(09. 27. 2018), Modified(12. 13. 2018), Accepted(12. 28. 2018)

* 본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 대학ICT연구센터육성지원사업(IITP-2018-2017-0-01633)과 2018년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2016-0-00078, 맞춤형 보안서비스 제공을 위한 클라우드 기반 지능형 보안 기술 개발)

† 주저자, anjinwoong@soongsil.ac.kr
‡ 교신저자, souhwanj@ssu.ac.kr(Corresponding author)

I. 서 론

전 세계에 스마트폰, 태블릿 PC와 같은 모바일 디바이스가 보급되고 있으며 Google의 Android와 Apple의 iOS가 가장 많이 사용되는 스마트폰 운영체제이다. 전 세계 안드로이드 사용자는 꾸준히 증가하고 있으며, 2019년에는 25억명에 달할 것으로 추정된다[1]. 2018년 9월 현재 State-Counter[2]에 따르면 안드로이드 운영체제의 점유율은 76.9%로 가장 점유율이 높으며, Android가 차지하는 높은 시장 점유율로 인해 안드로이드 악성 어플리케이션의 수도 매년 꾸준히 증가하고 있다[3]. McAfee의 보고서에 따르면 지난 2017년 하반기 안드로이드 악성 어플리케이션이 60% 증가하였다[4]. 이에 따라 안드로이드 악성 어플리케이션의 분석에 대한 중요성이 높아지고 있다. 악성 어플리케이션을 분석하는 방법으로 정적 분석과 동적 분석 두 가지 방법이 있다. 정적 분석은 안드로이드 어플리케이션 파일을 실제로 실행하지 않고 분석하는 방식으로, 바이트코드 혹은 디컴파일 된 소스코드를 이용하여 분석을 진행한다. 바이트코드를 이용한 분석 시, apk 파일을 디컴파일하여 AndroidManifest.xml 파일과 classes.dex 파일을 분석하여 악성 행위를 탐지한다. 그러나 최근 지능화된 일부 악성 어플리케이션은 분석이 어려운 네이티브(native) 코드로 구현된 SO (Shared Object)파일에서 악성행위를 수행하여 정적 분석을 우회하는 기법이 적용되고 있다. 이에 따라 자바 코드와 네이티브 코드를 모두 분석할 수 있는 시스템이 필요하다.

본 논문에서는 안드로이드 어플리케이션의 DEX(Dalvik EXecutable) 파일과 ELF(Executable and Linkable Format) 파일 구조 및 어플리케이션을 분석하기 위한 리버스 엔지니어링 툴과 기존 연구에 대해 소개하고, 기존 연구의 한계점을 보완하고 자바 코드와 네이티브 코드를 포함하는 정적 분석을 통해 함수의 호출관계를 추출하는 시스템을 제안 및 구현한다.

서론에 이어 본 논문의 2절에서는 관련 연구로 안드로이드 어플리케이션의 DEX와 ELF 파일 구조를 분석하기 위한 리버스 엔지니어링 툴과 기존 연구 및 한계점을 소개한다. 다음으로 3절에서는 안드로이드 SDK(Standard Development Kit)를 이용한 자바 코드로 구현된 실행파일인 DEX 파일의 구조와, NDK(Native Development Kit)를 이용하여 네이티브 코드로 구현된 실행파일인 ELF 파일의 구조 및 본 논문에서 제안하는 어플리케이션 분석 시스템에 대해 기술한다. 4절에서는 제안하는 시스템의 구현 및 결과와 기존 연구에 대해 비교하여 기술한다. 마지막으로 5절에서 결론 및 향후 연구 방향을 제시한다.

II. 관련 연구

안드로이드 어플리케이션은 SDK를 이용한 자바 코드와 NDK를 이용하는 C/C++와 같은 네이티브 코드로 구성된다. 자바 코드로 구현된 실행파일은 Dalvik Virtual Machine에서 동작할 수 있도록 DEX 파일 형식으로 최적화되어 classes.dex 파일로 저장된다. 스마트폰에서 사용하는 안드로이드 운영체제는 대부분 ARM 아키텍처를 채택하였으므로 네이티브 코드는 일반적으로 ELF 파일 형식의 Arm instruction 기반 SO 파일로 저장된다. 본 절에서는 안드로이드 DEX 파일과 SO 파일을 정적 분석할 수 있는 툴과 기존 연구에 대해 소개한다.

안드로이드 어플리케이션을 분석하기 위한 방법으로 정적 분석과 동적 분석이 존재한다. 동적 분석은 실제로 어플리케이션을 실행시키며 어플리케이션의 동작 정보를 수집하여 분석하는 반면, 정적 분석은 어플리케이션을 실행시키지 않고 기존 파일만으로 분석하는 방법이다. 안드로이드 어플리케이션을 정적 분석할 수 있는 툴이 존재하며, 대표적으로 Dex2jar와 Apktool이 있다.

2.1 정적 분석 툴

2.1.1 Dex2jar

Dex2jar는 안드로이드 어플리케이션의 실행파일인 classes.dex 파일을 jar 파일로 변환하는 툴이다[5]. 일반적으로 jar 파일을 자바 소스코드로 변환하여 GUI 형태로 볼 수 있는 기능을 제공하는 JD-GUI 툴과 같이 사용하여 안드로이드 어플리케이션을 분석하는데 사용된다.

2.1.2 Apktool

Apktool은 안드로이드 어플리케이션 파일인 apk 파일을 디컴파일하는 리버스 엔지니어링 툴이

```
                EXPORT Java_com_sklee_jnitest_HelloJNI_getString
Java_com_sklee_jnitest_HelloJNI_getString
                PUSH    {R4,LR}
                LDR     R2, [R0]
                LDR     R1, =(aThisIsJniTest - 0xC2E)
                MOVS    R3, #0x29C
                ADD     R1, PC          ; "This is JNI test :)"
                LDR     R3, [R2,R3]
                BLX     R3
                POP     {R4,PC}
; End of function Java_com_sklee_jnitest_HelloJNI_getString
```

Fig. 1. Disassemble example using IDA Pro

다[6]. Apktool은 apk 파일을 디컴파일하여 Dalvik 명령어를 사람이 볼 수 있도록 표현하는 smali 코드로 변환한다.

2.2 네이티브 분석 툴

2.2.1 IDA Pro

C/C++와 같은 네이티브 코드로 작성된 SO 파일을 분석하기 위한 대표적인 툴은 IDA Pro가 있다. 운영체제의 영향을 받지 않는 자바와는 다르게 네이티브 코드는 운영체제에 따라 컴파일 결과가 달라지며, IDA Pro는 안드로이드 디바이스에서 주로 사용되는 ARM 아키텍처를 지원하기 때문에 안드로이드에서 사용되는 네이티브 코드의 분석이 가능하다. IDA Pro는 어플리케이션 내에 포함된 SO 파일을 디컴파일하는 정적 분석을 수행하여 Fig.1.과 같이 ARM 명령어의 나열로 사용자에게 제공하며, 하나의 함수 내부에서의 분기를 그래프로 표시한다. 하지만 IDA Pro는 여러 개의 SO 파일을 동시에 분석할 수 없다. 또한 자바 코드 바이너리인 classes.dex 파일과 네이티브 바이너리인 SO 파일을 동시에 분석할 수 없으므로 두 영역의 상호작용은 알 수 없다는 단점을 지닌다.

2.2.2 Readelf

readelf 툴은 BFD(Binary File Descriptor Library) 라이브러리를 이용하지 않고 ELF 바이너리를 직접 읽어와 파일의 정보를 표시하며, ELF 파일 헤더와 program header, section header에 대한 정보와 section별 주소와 크기 정보를 제공한다. 하지만 섹션에 대한 기본적인 정보만을 제공하기 때문에 코드 섹션을 디스어셈블하는 기능이 존재하지 않는다는 단점을 지닌다.

2.2.3 Objdump

objdump 툴은 readelf와 같은 GNU 도구 모음 중 하나로 BFD 라이브러리를 이용하여 ELF 파일의 코드 섹션을 디스어셈블하여 제공하는 기능을 제공한다. 그러나 안드로이드에서 주로 사용되는 ARM 아키텍처에 대한 디스어셈블 기능을 지원하지 않아 어플리케이션의 SO 파일에 대해 분석할 수 없다는 문제점을 지닌다.

2.3 그래프 분석 툴

2.3.1 Smalica

안드로이드 어플리케이션의 Function Call Graph를 제공하는 툴로는 대표적으로 Smalisca가 있다[7]. Smalisca는 2015년 공개된 오픈소스로, 안드로이드 APK에 포함된 자바 클래스에 대한 정보를 제공한다. 클래스명, 클래스 속성, 클래스가 가진 메소드, 메소드의 호출 관계를 제공한다. 메소드의 호출 관계는 Fig.2.와 같이 자바 클래스의 메소드가 다른 메소드를 호출하는 관계를 화살표로 표시하여 그래프로 출력한다. Smalisca를 실행시키기 위해서 Python, Graphviz 등의 프로그램이 설치되어 있어야한다. Smalisca는 자바 코드에 대해서만 분석하므로 네이티브 코드에 대해서는 분석할 수 없다는 단점을 지닌다.

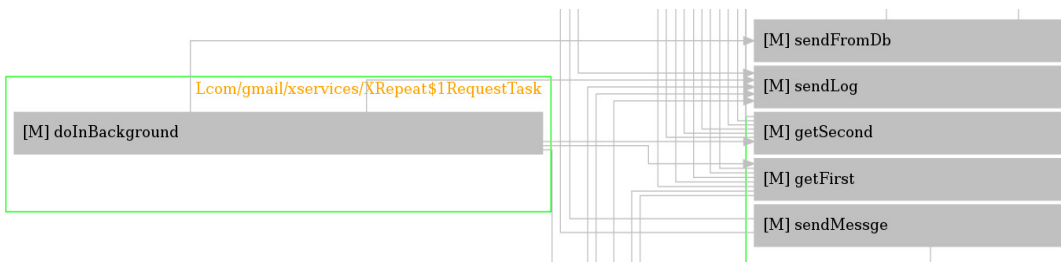


Fig. 2. Part of Function Call Graph using Smalisca[7]

2.3.2 FlowDroid

FlowDroid[8]는 안드로이드 어플리케이션을 위한 Flow Graph, 필드, 객체 및 라이프 사이클을 분석하는 연구로, Soot와 Heros를 기반으로 콜 그래프를 작성한다. Soot는 자바와 안드로이드 어플리케이션을 디컴파일 하거나 최적화하는 프레임워크이며, Heros는 자바 기반 프로그램 분석 프레임워크에 플러그인하여 데이터 흐름을 분석하는 IFDS/IDE solver이다. FlowDroid는 AndroidManifest.xml 파일과 classes.dex 파일, layout.xml 파일로부터 정보를 추출하여 분석을 진행하며, Main method의 첫 번째 라인부터 구문을 연속으로 분석하여 source에서 sink로 이어지는 흐름이 이어지는지 여부를 제공한다. 그러나 FlowDroid는 해당 논문에서 저자들이 자바 reflection 기법을 이용한 클래스 동적 로딩에 대응하지 못하며, DEX 파일이 여러 개 존재하는 어플리케이션에 대해서는 동적로딩 되는 DEX 파일에 대한 분석이 불가능하다는 점을 한계점으로 제시하였다. 또한 Flowdroid는 앱의 DEX 파일에 포함된 자바 코드를 디컴파일하여 분석하지만 네이티브 코드로 구현된 SO 파일에 대해서는 분석하지 않기 때문에 네이티브 코드의 함수 호출 관계는 분석하지 못한다는 단점을 지닌다.

2.3.3 DroidSafe

DroidSafe[9]는 안드로이드 어플리케이션에서 악성 코드를 확인하고 제거하는 기술에 대한 연구이다. 안드로이드 어플리케이션의 소스코드 혹은 자바 바이트코드를 통해 앱의 흐름을 분석하는 정적 분석에 대한 연구이다. AndroidManifest.xml 파일을 파싱하고 등록된 Intent filter에 대하여 매핑한 후, 콜백 컨텍스트, 라이프 사이클, 리소스와 컴포넌트 사이의 통신을 포착하고 source에서 sink로 이어지는 정보의 흐름을 추적하는 방식이다. DroidSafe는 저자들이 논문의 한계점으로 동적 로딩되는 DEX에 대해서 분석이 불가능하며, 자바 코드로 구현된 DEX 파일만을 분석하기 때문에 네이티브 코드에 대한 분석이 불가능함을 명시하였다.

NDK는 기존 네이티브로 작성된 코드의 재사용 혹은 높은 퍼포먼스를 요구하는 작업에 대응하기 위하여 제공되는 toolset이지만, 악성 어플리케이션 제작자가 어플리케이션의 정적 및 동적 분석을 회피하기 위하여 네이티브 코드가 활용하기도 한다. 이에 따라 지능화된 악성 어플리케이션의 정적 분석을 수행하기 위하여 네이티브 코드에 대한 분석 또한 이루어져야 한다.

2.3.4 Patrik Lantz 기법

Patrik Lantz가 제안한 기법[10]은 네이티브 라이브러리를 사용하여 정적 분석을 우회하는 앱을 분석하기 위해, JNI(Java Native Interface)를 통해 자바 코드가 네이티브 코드를 호출하거나 네이티브 코드가 자바 코드를 호출하는 방법에 대해 소개하고 데이터 흐름 분석을 위한 콜 그래프 생성 기법이다. 그러나 해당 기법을 제안한 저자는 네이티브 코드를 분석 시 네이티브 함수에서 자바 코드를 호출하는 관계를 추출하는 것은 가능하지만, 네이티브 함수에서 또 다른 네이티브 함수를 호출하는 경우는 분석하지 못한다는 점을 한계점으로 제시하였다. 또한, DEX 파일이 다수로 구성된 앱이 수행하는 기법인 DEX 동적 로딩에 대해서는 다루지 않는 문제점이 있다.

2.3.5 JN-SAF

JN-SAF는 안드로이드 앱의 콜 그래프를 생성하는 기존 연구를 통합하여 자바 코드와 네이티브 코드를 통합한 콜 그래프를 생성하는 연구이다[11]. 자바 코드의 콜 그래프 생성에는 Amandroid[12] 등 기존 도구를 사용하였으며, 네이티브 코드의 콜 그래프 생성에는 angr 도구를 사용하였다[13]. JN-SAF는 콜 그래프 생성에 이용한 기존 연구의 단점을 그대로 가진다. 저자는 자바 코드 분석 시 자바 inflection 기법에 대응하지 못하며 DEX 파일이 여러 개일 경우 대응할 수 없음을 명시하였다. 네이티브 코드 분석에서는 분석에 사용된 angr 도구의 단점을 승계하여 path explosion 문제가 발생한다. path explosion이란 symbolic execution으로 프로그램의 실행 가능한 경우의 수를 확인할 경우 발생하는 문제이다. 프로그램의 실행 가능한 모든 경우의 수가 프로그램의 크기에 따라 기하급수적으로 커지거나, 무한루프로 인해 무한한 경우의 수가 생기는 문제이다.

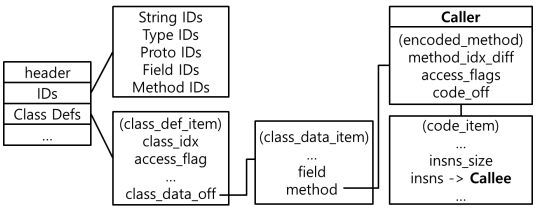


Fig. 3. Call relationship Extraction from DEX file

III. 제안 기법

2장에서 기존 연구에선 안드로이드 어플리케이션의 네이티브 코드에 대한 분석이 불가능하거나 DEX 파일이 다수인 경우, 분석이 불가능한 한계를 설명하였다. 본 논문에서는 자바 코드로 구현된 DEX 파일에 대한 정적 분석을 진행할 수 있는 분석 프로그램을 설계하고, 네이티브 코드로 구현된 ELF 파일 형식의 SO 파일에 대한 정적 분석을 진행하여 자바 코드와 네이티브 코드를 모두 포함한 함수 호출 관계를 분석하는 프로그램을 설계한다.

3.1 자바 코드 함수 콜 그래프

안드로이드 개발자가 작성한 자바 코드는 Dalvik Virtual Machine 에서 동작할 수 있는 바이너리 파일로 최적화되어 빌드 된다. 바이너리 파일은 어플리케이션 내에서 classes.dex 파일로 저장되며, DEX 파일 구조는 공식 웹페이지에 공개되어 있다 [14].

DEX 파일은 Table 1.과 같이 크게 String, Type, Proto, Field, Method, Class_Defs 필드로 나누어진다. Dex 파일의 header는 70바이트로 고정되어 있으며 checksum, 파일 크기, 해시값 등의 정보와 함께 각 필드의 크기와 offset 값을 저장하고 있으며, 따라서 header를 분석하여 각 필드의 데이터에 접근할 수 있다. IDs의 종류에는 문자열을 나타내는 String IDs, 앱 내에서 사용할 수 있는 변수와 클래스의 타입을 정의하는 Type IDs, 자바 메소드의 이름, 파라미터의 타입과 개수, 리턴 타입을 정의하는 Proto IDs, 클래스의 변수를 정의하는 Field IDs, 메소드의 클래스와 시그니처를 정의하는 Method IDs로 구분된다. String_IDs는 안드로이드 어플리케이션의 자바 코드에서 사용되는 모든 문자열을 저장하고 있다. 이 중에는 클래스명과 메소드명, 타입명등이 포함되어있으며 다른 필드에서 String_IDs를 참조하여 문자열을 표시한다. 특정 클래스의 메소드 호출 관계를 획득하기 위해선 아래와 같은 과정을 거친다.

Table 1. Dex File Structure

Name	Explanation
String_IDs	List of all strings
Type_IDs	Define the type of all classes, arrays, primitive types
Proto_IDs	Define return type and arguments of method
Field_IDs	List of field identifiers list
Method_IDs	List of method information
Class_Defs	List of class information
Call_site_IDs	List of call sites identifiers
Method_ handles	List of method handles
Data	Data area referenced in the above areas
Link_Data	Data used for statically linked files

3.1.1 Class_defs 필드 접근

DEX 파일은 Fig.3.과 같이 DEX 파일의 header와 문자열, 타입과 메소드 등을 정의하는 IDs, 클래스를 정의하는 Class Defs 영역 등으로 나뉜다. Fig.3.과 같이 DEX 파일의 header에서 class_defs 필드의 offset 값을 획득하여 class_defs 필드에 접근한다. Class_defs 필드는 class_def_item 포맷으로 정의되어 있으며 클래스명과 접근제한자, 부모 클래스 정보, 그리고 멤버변수와 메소드 정보에 대한 offset인 class_data_off가 존재한다.

3.1.2 멤버변수와 메소드 정보 획득

class_data_off를 통해 클래스에 정의된 멤버변수와 메소드에 대한 정보를 획득한다. class_data_item 포맷으로 정의되어있으며, 이 중 메소드에 대한 정보는 encoded_method_format의 리스트로 정의되어 있으며 method_idx_diff 변수가 Method_IDs를 참조하고, code_off 변수가 code_item 형식으로 data 필드를 참조한다. access_flags는 상수 값으로 public, private, final, native 등 메소드의 접근제한자를 나타낸다.

3.1.3 Method_IDs 참조

Method_IDs 필드는 Proto_IDs를 참조하여 메소드 파라미터의 종류와 개수, 메소드의 리턴 타입을 정의하며, 각 파라미터와 리턴 값의 타입은 Type_IDs를 참조하여 정의한다. 클래스명과 메소드명, 타입에 대한 명칭은 String_IDs를 참조하여 정의한다. Method_IDs 필드를 통하여 Caller에 대한 정보를 획득할 수 있다.

3.1.4 Dalvik bytecode 파싱

메소드의 실제 코드 영역은 code_item 형식으로 Data 영역을 참조하며, Dalvik bytecode 명령어로 저장되어있다. insns_size 변수의 값이 해당 메소드의 Dalvik instruction 개수를 의미하고, insns은 메소드에서 실행되는 Dalvik instruction의 리스트이다. Dalvik bytecode의 포맷은 웹페이지를 통하여 공개되어있다[15].

3.1.5 호출관계 추출

Data영역의 insns 리스트에서 다른 메소드를 호출하는 특정 Dalvik 명령어를 필터링한다. 메소드를 호출하는 Dalvik 명령어는 총 10개로 Table 2.에 정리하였다. 해당 Dalvik 명령어의 opcode 뒤의 2byte가 callee의 Method_IDs 에 대한 offset 값이다. 따라서 offset 인자 값을 통해 Method_IDs를 참조하여 callee 정보를 파싱해 메소드 간 호출 관계를 추출한다.

DEX 파일은 필드별로 클래스와 메소드 정보를 저장하고 있고, DEX 헤더에서 필드별 offset과 크기를 지정하고 있다. 따라서 DEX 파일의 헤더 정보를 바이너리 값으로 읽어 들인 후 분석하여 각 클래스와 메소드 정보를 획득하고, 메소드별 Dalvik bytecode 형태의 실행코드를 확인할 수 있다. 메소드를 호출하는 Dalvik 명령어는 Table 2.에서 명시한 총 10개로 명령어 이름이 모두 invoke-*로 시작한다는 특징을 지닌다. 이러한 Dalvik instruction의 인자 값을 통해 callee 메소드의 정보를 획득할 수 있다. 이를 통하여 메소드간 호출 관계를 획득할 수 있다. 이 중 어떤 메소드가 SO 파일에서 정의된 네이티브 메소드를 호출하는 경우, 접근제한자에 native가 포함된다. 따라서 접근제한자에 native가 포함된 메소드를 호출하는 경우 메소드가 네이티브 코드에 접근하는 것을 의미한다.

APK 내의 파일에 대해서 header를 통해 DEX 파일 여부를 확인하고, 모든 DEX 파일에 대해 동일한 과정을 수행하여 어플리케이션에서 사용하는 자바 코드의 호출 관계를 추출할 수 있다.

3.2 네이티브 함수 콜 그래프

ELF (Executable and Linkable Format) 형식은 유닉스 운영체제에서 사용되는 실행파일, 목적파일, 공유라이브러리 등을 위한 표준 파일 형식이다. ELF header에는 OS 및 instruction set 등에 대한 정보와 프로그램 헤더 테이블, 섹션 헤더 테이블에 대한 정보가 포함된다. 안드로이드 운영체제 기반 모바일 디바이스는 대부분 ARM 32-bit

Table 2. A set of Dalvik instructions that invoke method.

Instruction Name	Opcode	Explanation
invoke-virtual	6E	Invoke a normal virtual methods (not private, static, final)
invoke-super	6F	Invoke non-overridden method defined on interface
invoke-direct	70	Invoke constructor or private method
invoke-static	71	Invoke static method
invoke-interface	72	Invoke interface method
invoke-virtual/range	74	Invoke a normal virtual methods with more arguments
invoke-super/range	75	Invoke non-overridden method defined on interface with more arguments
invoke-direct/range	76	Invoke constructor or private method with more arguments
invoke-static/range	77	Invoke static method with more arguments
invoke-interface/range	78	Invoke interface method with more arguments

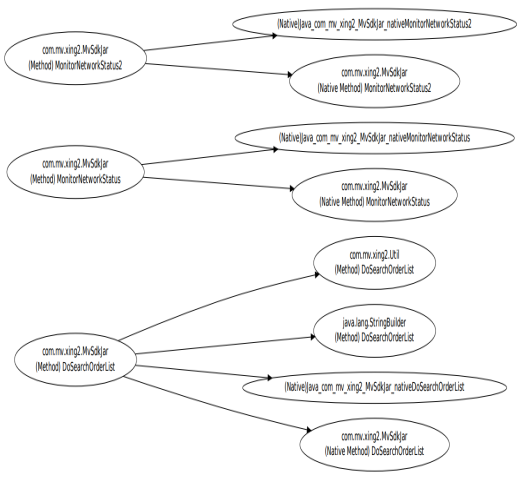


Fig. 4. Part of Function Call Graph Analysis result

Instruction Set 을 사용한다.

섹션 헤더 테이블은 파일의 섹션들에 대한 정보를 저장 한다. ELF 헤더의 e_shoff가 섹션 헤더 테이블에 대한 offset이며, e_shnum은 섹션 헤더 테이블 엔트리의 크기, e_shentsize가 섹션 헤더 테이블의 엔트리의 개수이다. E_shnum과 e_shentsize의 곱이 섹션 헤더 테이블의 전체 크기가 된다. 섹션 헤더 테이블의 각 엔트리에는 섹션의 이름, 타입, offset, size 등이 저장되어있으며, offset과 size를 통해 파일 내에서 저장된 주소를 추출해 섹션의 데이터를 파싱할 수 있다. 섹션은 symbol table, PLT(procedure linkable table), GOT(global offset table), instruction 등이 포함된다. 이 중 .text 섹션은 ARM 어셈블리 명령어로 구성되어 있으며 전체 코드를 나타낸다. ARM 아키텍처를 포함한 다양한 아키텍처의 디컴파일을 지원하는 디스어셈블러 프로젝트인 capstone 엔진[16]을 적용하여 .text 섹션을 디컴파일한다. 이후 opcode와 인자값을 추출하여 코드를 분석할 수 있으며, ARM instruction 중 branch 명령어를 추출하여 호출 관계를 정의할 수 있다.

안드로이드 운영체제에서는 미리 빌드 된 라이브러리를 제공하는 Dynamic Link 방식을 채택하고 있다. Dynamic Link는 공유 라이브러리 방식을 사용하여 실행파일이 라이브러리 코드를 포함하지 않는 방식이다. 라이브러리를 하나의 메모리 공간에 매핑하고 여러 프로그램에서 공유하여 파일의 크기가 작고 실행 시 적은 메모리를 사용하지만 라이브러리에 대한 의존성이 존재한다는 단점이 존재한다. PLT와 GOT는 Dynamic Link 방식에서 사용되는 섹션으로, PLT는 프로그램이 호출하는 모든 함수를 나열하고 있으며, GOT는 PLT에서 참조하는 테이블로 호출되는 함수의 실제 주소를 저장하는 테이블이다. 함수를 호출할 때에는 PLT 섹션을 참조하여 다른 라이브러리의 함수를 호출한다. 따라서 SO 파일을 리버싱하여 branch 명령어에서 PLT 섹션을 참조하는 경우 라이브러리 함수를 호출하는 관계를 추출할 수 있다. 이러한 방식을 이용하여 네이티브 함수 내의 호출 관계를 획득할 수 있다.

3.3 자바와 네이티브 함수 콜 그래프 결합

안드로이드 내에서 자바와 네이티브 코드가 상호 호출하기 위한 방법으로는 아래 과정을 거친다.

3.3.1 JNI_Onload

JNI_Onload는 자바에서 SO library를 load하는 과정으로, 자바에서 제공하는 System.loadLibrary() 혹은 System.load() 함수를 호출하여 어플리케이션에 포함된 SO 파일을 로딩하고 네이티브 메소드를 등록하는 과정이다.

3.3.2 JNI Export

JNI_Export 방식은 JNI_Onload 과정을 통해

Table 3. Comparison with previous research

	Our proposal	FlowDroid [8]	DroidSafe [9]	Patrik Lantz's Proposal[10]	JN-SNF [11]
Analyze Dex file	O	O	O	O	O
Analyze SO file	O	X	X	△	O
Analyze Multi-DEX APK	O	X	X	X	X
Call graph combining DEX & SO	O	X	X	O	O

로딩 된 SO 파일 내의 네이티브 메소드를 호출하는 과정이다. Java_className_methodName의 형식으로 네이티브 메소드를 정의하고, 자바에서 native 접근 제한자를 가진 네이티브 메소드를 선언한다. 자바에서 네이티브 메소드를 호출시 일반적인 자바 메소드를 호출하는 것과 동일하게 네이티브 메소드를 호출하는 방식이다.

따라서 네이티브 메소드를 호출하기 위해서는 자바 코드에서 System.loadLibrary() 혹은 System.load() 함수가 호출되어야하며, 접근 제한자에 native 로 정의된 네이티브 메소드를 호출하게 된다. 본 논문에서 제안하는 시스템은 네이티브 메소드 호출시 ELF 파일 내의 Dynamic Symbol Table을 참조하여 Java_className_methodName 형식의 호출 대상 함수를 검색 후, 해당 함수의 코드 영역인 .text 섹션을 참조하여 자바 코드와 네이티브 코드의 호출 관계 그래프를 연결한다. 시각화에는 그래프를 그리는데 사용되는 오픈소스인 Graphviz 모듈[17]을 사용하였으며, 안드로이드 어플리케이션의 caller-callee 관계를 표현하였다. Fig.4.는 본 논문에서 구현한 시스템으로 실제 악성 어플리케이션에서 추출한 호출 관계의 일부를 표시한 것이다. 해당 앱은 네이티브 코드에서 네트워크 연결 상태를 확인한 후 앱을 다운받는 downloader 앱이며, Fig.4.에 나타나는 바와 같이 앱을 다운받기 전 네이티브 코드로 구현된 네이티브 함수인 Java_com_mv_xing2_MvSdkJar_nativeMonitorNetworkStatus 함수에서 네트워크 상태를 모니터링한 후 앱을 다운받는다. 이와 같이 본 시스템을 이용하여 자바 코드와 네이티브 코드를 포함한 전체 어플리케이션의 행위와 흐름관계를 파악할 수 있다.

IV. 기존 연구와 비교

기존 연구와의 비교를 Table 3.에 정리하였다. 단일 DEX 파일은 동일하게 가능하나, FlowDroid는 네이티브 코드에 대한 분석이 불가능하며, 자바 reflection 기법을 이용하여 동적으로 DEX 파일의 클래스를 로딩하는 Multi-Dex 어플리케이션은 분석하지 못한다는 한계점을 지닌다. 이는 FlowDroid가 앱의 메인 DEX인 classes.dex 파일에 대해서 분석하고, 메인 DEX가 아닌 다른 DEX 파일의 분석을 하지 못하며 자바 reflection을 이용한 클래스 동적 로딩에 대한 분석이 불가능함을 한계점으로 가지기 때문이다. DroidSafe는 DEX 파일의 분석만 가능하며 네이티브 코드와 Multi-Dex 어플리케이션을 분석하지 못한다는 한계점을 지닌다. IDA Pro는 DEX 파일, SO 파일을 각각 분석할 수 있다는 장점을 가지나, Multi-Dex 어플리케이션을 분석하지 못하고 자바 코드와 네이티브 코드를 결합하여 분석하지 못한다는 단점을 지닌다. Patrik Lantz의 제안 기법은 자바 코드와 네이티브 코드를 분석한 콜 그래프를 그릴 수 있다는 장점을 지니지만, 네이티브 함수가 다른 네이티브 함수를 호출하는 경우에 대해서는 분석이 불가능하다는 단점을 지닌다. JN-SAF는 안드로이드 앱의 자바 코드와 네이티브 코드를 분석하여 통합된 콜 그래프를 그릴 수 있으나, 자바 reflection 기법과 Multi-DEX 앱을 분석하지 못하고, 네이티브 코드 분석 시 path explosion으로 인한 문제가 존재한다.

본 논문에서 제안하는 시스템은 DEX 파일의 헤더를 검사하여 bytecode 로 저장된 모든 DEX 파일을 읽은 후 정적 바이너리 분석하여 자바 코드 분석이 가능하며, 자바 reflection 기법이 적용된 Multi-Dex 어플리케이션을 분석할 수 있다. 또한, 네이티브 코드 분석을 위해 어플리케이션 내에 저장된 파일의 헤더를 검사하여 SO 파일인지 확인하고, SO 파일을 분석하여 네이티브 코드 내에서의 함수 호출 관계를 분석한다. 네이티브 코드 분석에는 symbolic execution 기법을 사용하지 않고 경우의 수에 상관없이 코드 내에 존재하는 모든 호출 관계를 추출하기 때문에 symbolic execution으로 인해 발생하는 path explosion 문제가 발생하지 않는다. 또한, 자바 코드와 네이티브 코드가 상호간 호출하는 관계를 추출하고 이를 결합하여 자바 코드와 네이티브 코드까지 모두 포함된 함수 호출 관계를 추출할 수 있다. 지능화된 악성 앱의 경우, 자바 코드에서 민감 정보를 획득하고 네이티브 코드에서 민감 정보를 전송하는 등 자바 코드 혹은 네이티브 코드를 개별로 분석 시 파악할 수 없는 악성행위 사례가 존재한다. 이러한 사례에서 본 논문의 연구를 이용하여 자바 코드와 네이티브 코드의 상호간 연결되는 호출 관계를 추출하여 행위를 파악하는 것이 가능하다.

V. 결 론

최근 지능화된 안드로이드 악성 어플리케이션은 정적 분석되는 것을 막기 위하여 정적 및 동적 분석

이 어려운 네이티브 코드로 구현된 SO 파일을 활용하여 악성 행위를 숨긴다. ELF 파일 형식의 Arm Instruction 기반 SO 파일을 분석하여 네이티브 코드에 대해 분석할 수 있으며, 어플리케이션의 DEX 파일을 추출하여 정적으로 Dalvik Instruction을 이용하여 자바 코드에 대해 정적 분석을 수행할 수 있다.

본 논문에서는 자바 코드로 구현된 DEX 파일과 네이티브 코드로 구현되는 SO 파일에 대해서 정적 분석을 수행하여 자바 코드와 네이티브 코드 모두를 포함하여 함수 호출 관계를 추출할 수 있는 시스템을 제안 및 구현하였다. 기존 연구에서는 악성 앱 분석 시 자바 코드 혹은 네이티브 코드를 개별로 분석한다는 문제점을 지녀, 자바 코드와 네이티브 코드에서 상호간 민감 정보가 전달되는 경우 분석에 어려움이 존재하였다. 본 논문에서 제안 및 구현한 시스템에서는 자바 코드와 네이티브 코드에서 수행되는 앱의 호출 관계를 추출하여 악성행위 동작 시퀀스를 획득하는 것이 용이해진다. 이에 따라 코드 상에서 실제로 전달되는 민감 정보 컨텐츠의 흐름을 파악하여 데이터 추적 시 자바 코드와 네이티브 코드 모두에서 사용되는 데이터에 대해 분석할 수 있도록 활용할 수 있다.

References

[1] Suman R Tiwari, "A survey of android malware detection technique," Journal of Network Communications and Emerging Technologies (JNCET), vol. 8, no. 4, pp. 332-334, Apr. 2018.

[2] StateCounter, "Mobile operation market" http://gs.statcounter.com/os-market-share/mobile/worldwide, 2018-09-05.

[3] McAfee Labs, "mcAfee mobile threat" https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf, 2018-09-06.

[4] McAfee Labs, "mcAfee mobile threat" https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2017.pdf, 2018-09-06.

[5] dex2jar, "dex2jar github" https://github.com/pxb1988/dex2jar, 2018-09-10.

[6] apktool, "apktool" https://ibotpeaches.github.io/Apktool, 2018-09-10.

[7] Smalisca, "smalisca github" https://github.com/dorneanu/smalisca, 2018-09-10.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Acm Sigplan Notices, vol.49, no.6, pp. 259-269, Jun. 2014.

[9] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard, "Information Flow Analysis of Android Applications in DroidSafe," NDSS, vol.15, pp. 110, Feb. 2015.

[10] Patrik Lantz and Bjorn Johansson, "Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications," Wireless Communications and Mobile Computing Conference (IWCMC), IEEE, pp. 587-593, Aug. 2015.

[11] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang, "JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1137-1150, Oct. 2018.

[12] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel and Giovanni Vigna, "Sok:(state of) the art of war: Offensive techniques in binary ana-

lysis," 2016 IEEE Symposium on Security and Privacy, pp 138-157, May. 2016.

[13] Dalvik Executable format, "DEX format" https://source.android.com/devices/tech/dalvik/dex-format, 2018-09-15.

[14] Fengguo Wei, Sankardas Roy, and Xinming Ou, "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps," Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329-1341, Nov. 2014.

[15] Dalvik Bytecode, "dalvik bytecode" https://source.android.com/devices/tech/dalvik/dalvik-bytecode, 2018-09-15.

[16] Capstone Project, "disassembly framework" https://www.capstone-engine.org, 2018-09-23.

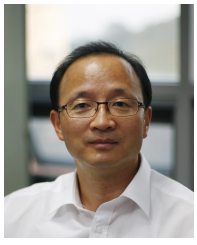
[17] John Ellson, Emden Gansner, Lefteris Koutsofios, North Stephen C, and Gordon Woodhull, "Graphviz—open source graph drawing tools," In International Symposium on Graph Drawing, Springer Berlin Heidelberg, vol.2265, pp. 483-484, Feb. 2002.

〈저 자 소 개〉

안 진 웅 (Jinung Ahn) 학생회원
2017년 2월: 숭실대학교 정보통신전자공학부
2017년 3월~현재: 숭실대학교 정보통신공학과 석사과정
〈관심분야〉 정보보호, 모바일 보안, 클라우드 보안

박 정 수 (Jungsoo Park) 학생회원
2013년 2월: 숭실대학교 정보통신전자공학부 졸업
2015년 2월: 숭실대학교 융합소프트웨어학과 석사
2015년 3월~현재: 숭실대학교 융합소프트웨어학과 박사과정
〈관심분야〉 클라우드 보안, 무선 네트워크 보안

응웬부렁 (Long Nguyen-Vu) 학생회원
2012년 9월: Vietnam National University of Information Technology
2016년 2월: 숭실대학교 정보통신공학과 석사
2016년 3월~현재: 숭실대학교 정보통신공학과 박사과정
〈관심분야〉 클라우드 보안, 모바일 보안, 빅 데이터



정 수 환 (Souhwan Jung) 종신회원
1985년 2월: 서울대학교 전자공학과 졸업
1987년 8월: 서울대학교 전자공학과 석사
1996년 6월: University of Washington 박사
1988년~1991년: 한국통신 전임 연구원
1997년~현재: 숭실대학교 전자정보공학부 교수
〈관심분야〉 클라우드 보안, 모바일 보안, 네트워크 보안