



저작자표시-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.
- 이 저작물을 영리 목적으로 이용할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

공학석사 학위논문

정적 분석을 통한 안드로이드 기반
스마트폰의 악성코드 탐지 기법

Malware Detection Technique of Android-based
Smartphone using Static Analysis



지도교수 이 장 세

2011년 2월

한국해양대학교 대학원

컴퓨터공학과
윤진식



공학석사 학위논문

정적 분석을 통한 안드로이드 기반
스마트폰의 악성코드 탐지 기법

Malware Detection Technique of Android-based
Smartphone using Static Analysis



2011년 2월

한국해양대학교 대학원

컴퓨터공학과

윤진식

이 논문을 윤진식의 공학석사 학위논문으로 인준함.

위원장 류길수



위원 김재훈



위원 이장세



2010년 12월 24일

한국해양대학교 대학원

목 차

제 1 장 서론	1
제 2 장 관련 연구	3
2.1 안드로이드 구조 및 보안 모델	3
2.1.1 안드로이드 플랫폼	3
2.1.2 안드로이드 보안 모델	6
2.2 악성코드 분석 및 탐지 기법	9
2.2.1 악성코드 정의 및 특징	9
2.2.2 악성코드 분석 기법	11
2.2.3 파일 기반 악성코드 탐지 기법	12
제 3 장 MDA(Malware Detection for Android) 시스템 설계 및 구현	15
3.1 MDA 시스템 구조	15
3.2 dex 분석기	16
3.2.1 dex 추출기	16
3.2.2 dex 파서	18
3.3 악성코드 탐지 엔진	30
3.3.1 시그니처 탐지기	30
3.3.2 휴리스틱 탐지기	31
제 4 장 MDA 시스템 실험	35
4.1 dex 분석기 실험	36
4.1.1 dex 추출	36
4.1.2 dex 파싱	37
4.2 악성코드 탐지 엔진 실험	39
4.2.1 SHA-1 시그니처 탐지	40
4.2.2 휴리스틱 탐지	41
4.2.3 허용 어플리케이션 시그니처 추가	43
제 5 장 결론 및 향후 과제	45
참고 문헌	46

표 목차

표 2.1	안드로이드 보안 기법	7
표 3.1	dex 파일에서 사용되는 타입들	18
표 3.2	dex 파일 구조	20
표 3.3	dex 헤더 구조	22
표 3.4	클래스 정의 아이템 구조	23
표 3.5	타입 ID 아이템 구조	24
표 3.6	문자열 ID 아이템 구조	24
표 3.7	문자열 데이터 아이템 구조	25
표 3.8	클래스 데이터 아이템 구조	25
표 3.9	메소드 데이터 아이템 구조	26
표 3.10	메소드 ID 아이템 구조	26
표 3.11	코드 아이템 구조	27
표 3.12	메소드 호출 관련 명령어	28
표 3.13	명령어 형식 크기	29
표 3.14	악성 행위 관련 클래스와 API	34

그림 목차

그림 2.1 안드로이드 구조	3
그림 2.2 dex 파일로의 변환 과정	5
그림 2.3 PE(Portable Executable) 형식의 악성코드 파일	13
그림 3.1 MDA 시스템 구조	15
그림 3.2 apk 파일의 구조	17
그림 3.3 dex 파일 추출 알고리즘	17
그림 3.4 Signed LEB128 디코딩 알고리즘	19
그림 3.5 Unsigned LEB128 디코딩 알고리즘	20
그림 3.6 허용 어플리케이션 시그니처 데이터베이스 구조	30
그림 3.7 메소드 호출 스택 가공 알고리즘	32
그림 3.8 API 코드 정보 데이터베이스 파일 구조	33
그림 3.9 악성 행위 휴리스틱 시그니처 데이터베이스 파일 구조	33
그림 4.1 MDA 시스템 순서도	35
그림 4.2 kr.co.onepiece.oppa-1.apk 구조	36
그림 4.3 dex 추출기 결과 배열 크기	37
그림 4.4 dex 파일 헤더 추출 결과	37
그림 4.5 클래스 별 메소드 호출 결과	38
그림 4.6 허용 어플리케이션 시그니처 데이터베이스 내용	40
그림 4.7 API 코드 정보 데이터베이스 내용	41
그림 4.8 악성 행위 휴리스틱 시그니처 데이터베이스 내용	42
그림 4.9 악성코드 탐지 알림창	43
그림 4.10 추가된 허용 어플리케이션 시그니처 데이터베이스 내용	44

Malware Detection Technique of Android-based Smartphone using Static Analysis

Jin-Sik Yun

*Department of Computer Engineering
Graduate School of
Korea Maritime University*

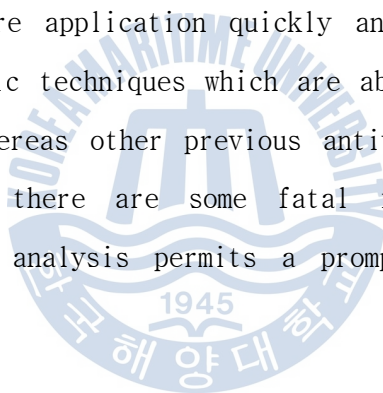
Abstract

As Google's Android shows the fastest growing mobile smartphone operating system in the world, also some vulnerability issues in Android browser which could allow an attacker to remotely steal the user's local data or to make spoof applications that could silently download a mobile malware application in the background have been increasing today. Due to the increase, many Android applications which may contain mobile malware recently started to show up on the Android Market(Google's app store for Android).

Most mobile antivirus applications are for Windows Mobile and Symbian devices so far; However with the increase of threats also for Android, many mobile antivirus companies are trying to treat this OS in their product. Nevertheless a lot of security issues in mobile circumstance arise continuously. The important thing is that both neither Google nor Microsoft handle and approve, in fact, apps the same way for example

Apple does.

The motivation for this thesis was to extract and detect an Android malware efficiently and more quickly. The static analysis is used to achieve this. Further, the signature detection technique with a few heuristic techniques shows the brilliant results. In particular, the proposed solution extracts a header of malware and a operation data with so called static analysis. Then, 'SHA-1 signature' and 'API Call Combination signature' are extracted - each function separately identifies a specific application and detects the specific factors that affect to the operation system directly. Finally the proposed solution senses a mobile malware application quickly and correctly with these signatures and heuristic techniques which are able to detect a new and mutant efficiently, whereas other previous antivirus applications show good performance but there are some fatal flaws to detect them. Futhermore the static analysis permits a prompt detection before an execution.



제 1 장 서 론

최근 스마트폰 시장에서 가장 큰 화두 중 하나인 안드로이드(Android)는 구글(Google)에서 발표한 오픈소스(open source) 기반 모바일 운영체제이다. 안드로이드는 오픈소스 기반 플랫폼이기 때문에 발전 가능성이 크다는 장점을 가지고 있지만 유통 방식이 개방적이기 때문에 보안상 취약한 플랫폼으로 지적받고 있다. 이러한 우려와 같이, 최근 해외에서부터 시작된 안드로이드 기반 악성코드는 국내에서도 발견되어 여러 사용자에게 큰 피해를 주고 있다[1,2].

모바일 악성코드는 기존의 PC 플랫폼 기반 악성코드와는 다르게 스마트폰의 배터리 소모 기능과 기기 및 위치 정보, 전화 번호, 메시지 등의 개인 정보 유출 기능 등이 있다[3]. 이 때문에 PC 플랫폼 기반의 백신을 개발하는 여러 기업이 뒤늦게 스마트폰 기반 백신을 개발하고 있으며 최근에는 안드로이드 기반 백신도 발표하였다. 하지만 급격한 스마트폰의 발달과 보급으로 인해 스마트폰은 이미 노트북을 능가하는 대중적인 모바일 컴퓨팅 기기가 되었고 다양한 기법의 악성코드가 등장하고 있다[4].

기존의 PC 플랫폼 기반 악성코드 탐지 기법에는 시그니처(signature) 기반 탐지 기법과 휴리스틱(heuristic) 기반 탐지 기법, 행위 기반 탐지 기법 등이 사용되는데 PC 플랫폼에서 악성코드 예방을 위한 백신은 이와 같은 여러 기법을 복합적으로 사용하여 다양한 악성코드를 탐지한다[4]. 하지만 이런 방법들의 복합적인 사용은 스마트폰에 비해 상대적으로 높은 PC의 하드웨어 사양을 고려한 것으로 상대적으로 낮은 하드웨어 사양을 가진 스마트폰에 도입하기 힘들다.

현재 안드로이드 기반 스마트폰 백신은 기본적으로 시그니처 기반 탐지 기법을 사용하며 이에 추가적으로 안드로이드 플랫폼의 권한 특징을 이용한 행위 기반 탐지 기법을 통해 악성코드를 검사하고 있다. 따라서 변종 또는 신종 악성코드를 탐지할 수 없는 시그니처 기반 탐지 기법으로는 새롭게 등장하는 모

파일 악성코드를 탐지하기 어렵고 행위 기반 탐지 기법은 오탐률이 높다[5].

본 논문에서는 악성코드가 실행되는 기반 환경인 안드로이드 플랫폼의 구조에 대해 알아보고 안드로이드 내부 보안 모델에 관해 고찰하며 기존의 악성코드 분석 기법과 탐지 기법들의 특징에 대해 분석한다. 이러한 내용을 바탕으로 정적 분석을 통한 시그니처 추출 및 안드로이드 기반 어플리케이션 내 API 호출 스택 추출 기법을 제안하고 안드로이드 기반 시그니처 탐지 기법과 휴리스틱 탐지 기법을 적용하여 악성코드를 탐지하는 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서 안드로이드의 구조와 보안 모델, 기존 악성코드 분석 및 탐지 기법에 대해 살펴본다. 3장에서는 제안하는 탐지 기법을 적용한 전체 시스템의 구조를 제안하여 설계하고 구현한다. 4장에서 테스트를 한 후, 마지막으로 5장에서 결론과 향후 과제를 제시한다.



제 2 장 관련 연구

모바일 악성코드도 운영체제를 기반으로 하는 어플리케이션이다. 제안하는 안드로이드 기반 악성코드 탐지 기법을 설명하기에 앞서 본 장에서는 모바일 악성코드의 기반 플랫폼인 안드로이드의 구조와 보안 모델을 소개하고 기존 악성코드 분석 및 탐지 기법을 설명한다.

2.1 안드로이드 구조 및 보안 모델

2.1.1 안드로이드 플랫폼

안드로이드는 휴대용 장치를 위한 운영체제, 미들웨어, 핵심 어플리케이션을 포함하고 있는 소프트웨어 스택을 뜻한다. 2007년 11월 구글에서 발표한 안드로이드 플랫폼은 비 독점 개방 플랫폼을 지향하여 만들어진 것으로, 독점적인 운영체제 플랫폼인 심비안, 아이폰, 윈도우 모바일과는 대조적이다[5]. 2010년 12월 현재, 안드로이드 2.3(Gingerbread) 버전이 발표되어 있는 상태이다[6].

(가) 시스템 구조

안드로이드 전체 시스템 구조는 그림 2.1과 같다[5,6].

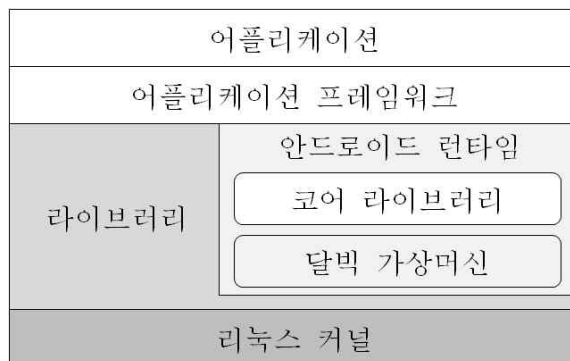


그림 2.1 안드로이드 구조

Fig. 2.1 Architecture of Android

- 어플리케이션(application) : 이메일 클라이언트, SMS 문자 메시지 프로그램, 캘린더, 맵, 브라우저 등과 같은 핵심 어플리케이션을 탑재하고 있는 계층이다. 모든 어플리케이션은 자바(java) 프로그래밍 언어를 사용하여 작성되어 있다.
- 어플리케이션 프레임워크(application framework) : 안드로이드의 API 집합으로 이루어진 계층이다. 어플리케이션들은 하위의 커널이나 시스템 라이브러리를 직접적으로 호출할 수 없으며 API를 통해서 기능을 요청해야 한다. 어플리케이션을 만드는 데 사용될 수 있는 풍부하고 확장이 가능한 뷰(view) 집합, 어플리케이션이 다른 어플리케이션의 데이터에 접근하는 것을 가능하게 하거나, 자신의 데이터를 공유하는 것을 가능하게 하는 콘텐츠 제공자(content provider), 문자열, 그래픽, 레이아웃 파일과 같은 비 코드 리소스에 대한 접근을 제공하는 자원 관리자(resource manager), 모든 어플리케이션이 상태 바에 알림 메시지를 표시하는 것이 가능하게 하는 알림 관리자(notification manager), 어플리케이션의 생명주기를 관리하며, 사용자의 일반적인 어플리케이션 네비게이션 히스토리를 관리하는 액티비티(activity)를 포함하는 집합이 존재한다. 어플리케이션과 마찬가지로 자바 프로그래밍 언어를 사용했다.
- 라이브러리(library) : 어플리케이션들이 공통적으로 사용하는 시스템 라이브러리의 집합으로 이루어진 계층이다. 라이브러리는 장비의 전반적인 속도를 결정하는 중요한 요소여서 자바가 아닌 C/C++로 작성되어 있다. 라이브러리 계층을 구성하는 핵심 라이브러리는 시스템 C 라이브러리, 미디어 라이브러리, 서비스 매니저, LibWebCore, SLG, 3D 라이브러리, FreeType, SQLite이다.
- 안드로이드 런타임(android runtime) : 안드로이드 런타임은 코어 라이브러리(core library)와 달빅 가상머신(dalvik virtual machine)로 구성된다. 코어 라이브러리는 자바 프로그래밍 언어의 핵심 라이브러리에서 사용가능한 대부분의 기능을 포함하고 있다. 안드로이드는 자바 가상머신

을 직접 사용하지 않으며 모바일 환경에 최적화된 달빅 가상머신을 사용한다. 각 프로세스별로 별도의 달빅 가상머신이 할당되므로 안정성이 높고 메모리 사용량을 줄여 복수개의 가상머신도 효율적으로 동작하도록 설계되었다. 레지스터 기반인 달빅 가상머신은 안드로이드 전용의 가상머신이므로 자바 클래스를 바로 실행할 수 없으며 클래스 파일을 dex 포맷으로 변환해야만 실행이 가능하다. dex 포맷으로의 변환 과정은 그림 2.2와 같다[6].

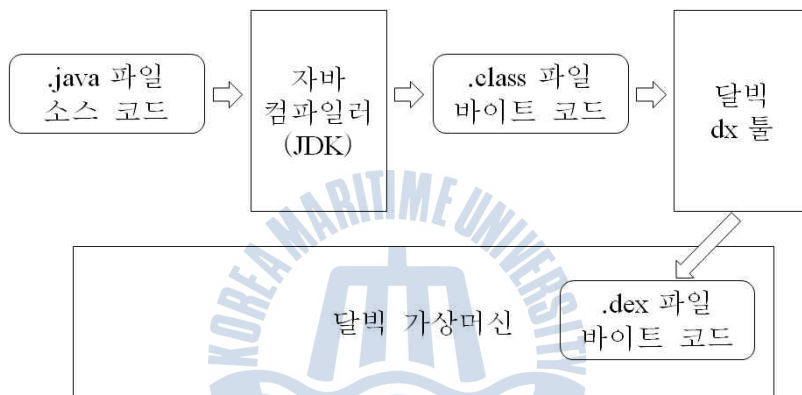


그림 3.2 dex 파일로의 변환 과정

Fig. 2.2 Processing steps of conversion to dex file

- 리눅스 커널(linux kernel) : 안드로이드는 보안, 메모리 관리, 프로세스 관리, 네트워크 스택, 드라이버 모델과 같은 핵심 시스템 서비스에 대해서는 리눅스 버전 2.6에 의존한다. 안드로이드가 리눅스를 채용한 주된 이유는 디바이스 드라이버의 지원이 광범위하기 때문이다.

(나) 실행 파일 구성 요소

안드로이드 실행파일을 이루는 4개의 컴포넌트는 다음과 같다[7,8].

- 액티비티 : 사용자 인터페이스를 구성하는 기본 단위이다. 권한을 부여한 후 외부에서 프로그램 내 특정 액티비티를 호출하여 직접 사용가능하므로 프로그램의 시작점이 특별히 정해져 있지 않고 타 프로그램 개발 시 재사용이 용이하다.
- 서비스 : UI(User Interface)가 없어 사용자 눈에 직접적으로 보이지 않으며 백그라운드에서 무한히 실행되는 컴포넌트이다.
- 브로드캐스트 수신기(broadcast receiver) : 시스템으로부터 전달되는 이벤트를 대기하고 이벤트 발생 시 수신하는 역할을 한다.
- 콘텐츠 제공자 : 다른 어플리케이션을 위해 자신의 데이터를 제공한다. 어플리케이션 간에 데이터를 공유할 수 있는 합법적인 유일한 장치이다.

2.1.2 안드로이드 보안 모델

안드로이드는 각각의 어플리케이션과 시스템 내의 각 영역들이 자신의 고유한 프로세스 상에서 실행되는 멀티 프로세스 시스템이다. 어플리케이션과 시스템간의 보안은 어플리케이션에 할당된 UID(User ID)나 GID(Group ID)와 같은 표준 리눅스 설비를 통해 프로세스 레벨에서 차단된다. 추가적인 보안 기능은 권한(permission) 메커니즘에 의해 제공된다[7].

(가) 보안 모델 구조

안드로이드 보안 모델 구조에서 기본적인 핵심 개념은 어떠한 어플리케이션도 다른 어플리케이션과 운영체제, 또는 사용자에게 나쁜 영향을 미칠 수 있는 임의의 행동을 수행할 수 있는 권한을 가지지 않도록 한다는 것이다[9].

이는 사용자의 개인적인 데이터에 대한 읽고 쓰기 및 다른 어플리케이션의 파일에 대한 읽고 쓰기, 네트워크 접근, 디바이스에 대한 활성 상태 유지 등을

포함하며 이는 어플리케이션의 보안 샌드박스(sandbox)에 의해 이루어진다. 기본적인 샌드박스에 의해 제공되지 않는 추가적인 기능들을 위해 필요한 권한을 명시적으로 선언하지 않고는 다른 어플리케이션에 영향을 줄 수 없다. 어플리케이션이 요구하는 권한들은 다양한 방법들로 운영체제에 의해 제어될 수 있으며 전형적으로는 인증서를 기반으로 자동으로 허용하거나 불허되고 사용자에게 확인을 요청하는 것에 의해 제어된다. 어플리케이션이 시스템에 요청하는 권한은 어플리케이션 내에 정적으로 선언된다. 따라서 그 권한은 어플리케이션 설치 전에 알 수 있으며 설치 후에는 변경될 수 없다[8-11].

(나) 안드로이드 보안 기법

세부적인 안드로이드 내부 보안 기법은 전체적으로 표 2.1과 같다[10,12].

표 2.1 안드로이드 보안 기법
Table 2.1 Security mechanisms in Android

분류	기법
리눅스 기법	POSIX 사용자
	파일 접근
환경적 특징	메모리 관리 장치
	안전한 형식
	휴대폰 보안 특성
안드로이드 특유 기법	권한
	구성요소 캡슐화
	서명
	달빅 가상머신

표 2.1에서 각각의 기법은 리눅스 고유의 기법과 모바일 기기의 환경적 특징, 안드로이드 특유 기법으로 분류되고 기법에 대한 설명은 다음과 같다[12].

- POSIX(Portable Operating System Interface) 사용자 : 다른 어플리케이션의 교란을 막기 위한 목적으로 각 어플리케이션이 서로 다른 UID와 GID를 할당 받아 독립된 계층에서 실행되는 방법이다.
- 파일 접근 : 다른 어플리케이션의 접근을 막기 위해 어플리케이션이 자체 폴더 내부만 접근이 가능하도록 제한한 방법이다.
- 메모리 관리 장치 : 권한 상승, 정보 유출, 서비스 거부 등을 막기 위해 각 프로세스가 자신의 고유한 주소 공간을 가지는 방법이다.
- 안전한 형식 : 버퍼 오버플로우(buffer overflows)와 스택 스매싱(stack smashing) 등을 막기 위해 어플리케이션 컴파일과 실행 시 안전하다고 보증된 특정 포맷만 사용하도록 강제하는 방법이다.
- 휴대폰 보안 특성 : 휴대폰 분실을 막기 위해 스마트폰에서 사용하는 SIM 카드를 인증하는 방법이다.
- 권한 : 어플리케이션이 악의적 행위를 하지 못하도록 기능을 제한하기 위해 설치 시 운영체제에 필요한 권한을 요청하고 그것을 사용자에게 표현하도록 하는 방법이다[13].
- 구성요소 캡슐화 : 어플리케이션이 다른 프로그램을 교란시키거나 비공개 요소 또는 API에 접근하는 것을 막기 위해 어플리케이션 내의 각 요소들이 공개 레벨을 가지도록 하는 방법이다.
- 서명 : 둘 이상의 어플리케이션이 같은 출처로부터 개발된 것인지 비교하기 위해 개발자가 어플리케이션을 배포 시 자체 서명 알고리즘을 통해 서명하도록 하는 방법이다.
- 달빅 가상머신 : 버퍼 오버플로우, 원격 코드 실행, 스택 스매싱 등의 메모리 관련 문제를 막기 위해 어플리케이션이 각각 독립적인 가상머신 안에서 실행되도록 하는 방법이다[14].

2.2 악성코드 분석 및 탐지 기법

2.2.1 악성코드 정의 및 특징

악성코드는 악성 또는 악용 가능한 소프트웨어의 집합으로, 바이러스, 웜, 스파이웨어, 악성 애드웨어 등 사용자와 컴퓨터에게 잠재적으로 위험이 되는 모든 소프트웨어를 총칭하는 말이다. 사전적 의미로 멀웨어(malware)는 ‘malicious software(악의적 소프트웨어)’의 약자로, 사용자의 의사와 이익에 반해 시스템을 파괴하거나 정보를 유출하는 등 악의적 활동을 수행하도록 의도적으로 제작된 소프트웨어를 말한다. 국내에서는 악성코드로 번역되며 자기 복제와 파일 감염이 특징인 바이러스를 포함하는 더 넓은 개념이다[4]. 본 논문에서는 악성코드의 한 분류인 모바일 악성코드의 탐지를 연구한다.

모바일 악성코드란 기존 PC 환경에서 발생하는 악성코드와 유사하게 모바일 단말기를 대상으로 개인정보 유출, 시스템 파괴, 원격지 접속 등의 악의적인 행위를 수행하기 위해 제작된 악성 프로그램을 의미한다[15]. 모바일 악성코드를 분류하면 대표적으로 전파를 목적으로 다른 파일을 감염시키는 바이러스, 유용한 프로그램을 가장한 트로이 목마(trojan horse), 자신의 복사본을 생성하여 전파하는 웜(worm)이 있으며, 이중에서도 사회공학기법을 이용하여 사용자의 설치를 유도하는 트로이 목마가 가장 큰 비율을 차지하고 있다. 이들은 주로 PC와의 동기화, 메모리 카드, 블루투스, MMS(Multimedia Message Service), 모바일 인터넷을 통해 전파된다. 모바일 악성코드의 대표적인 특징은 다음과 같다[16].

- 파일실행 차단
- 파일 감염 및 덮어 쓰기
- 휴대폰의 원격 제어
- 어플리케이션 혹은 아이콘의 변경
- 블루투스 혹은 MMS를 통한 확산

- SMS(Short Message Service) 메시지 전송을 통한 부당 요금 발생
- 메모리 카드 차단
- 사용자 데이터 은닉 및 도난
- 유료 서비스 무단 접속 및 국제전화 무단 발신으로 부당 요금 발생
- 사용자의 SMS 훔쳐보기
- 휴대폰 정보의 유출
- 다른 악성코드의 설치
- 휴대폰을 이용해 PC에 악성코드 설치

위와 같은 특징을 가지는 모바일 악성코드에 의한 단말기 사용자의 실제 피해 유형은 다음과 같다[17].

- 파일 조작 : 어플리케이션 및 시스템 프로그램에 특정 내용을 덮어 쓰거나 다른 파일로 교체하여 프로그램의 실행 및 단말기 사용이 불가하게 된다.
- 정보 유출 : 전화번호부나 주소록, 사진 등의 개인정보 및 스마트 폰 사용자의 수신 메시지 내용과 통화 내역이 외부로 유출되게 한다.
- 서비스 과금 : 감염된 스마트 폰을 통해 SMS와 MMS 메시지를 무단으로 발송하게 하여 금전적인 피해 유발한다.
- 장치 사용 불가 : 서비스 거부 공격을 통해 스마트 폰의 배터리를 방전시키거나, 메모리 카드의 패스워드를 임의로 변경하여 사용 불가하게 된다.

2.2.2 악성코드 분석 기법

악성코드의 탐지를 위해서 선행으로 해당 악성코드를 분석해야 한다. 악성코드의 분석은 악성코드 탐지의 기준이 되는 특성을 추출하는 과정으로서 분석기법은 크게 정적 분석과 동적 분석으로 구분된다[4,18,19].

(가) 정적 분석 기법

악성코드 분석에 사용되는 주요 기법 중 하나인 정적 분석 기법은 소스 코드를 연구하여 프로그램이 하는 행동을 이해하는 방법이다. 하지만 대부분 악성코드는 바이너리 형식으로 배포가 이루어져 소스 코드를 수집하기 어렵다. 바이너리 코드는 디버거(debugger)와 디셈블러(disassembler)를 이용하여 분석이 가능하지만, 이런 툴들을 사용해도 바이너리가 특유의 인코딩을 가질 때는 적절치 않다. 물론 충분한 시간만 주어지면 아무리 크고 복잡한 바이너리도 코드 분석 기법으로 완벽하게 분석 가능하다[20].

(나) 동적 분석 기법

동적 분석 기법은 악성코드의 행위적 성향에 초점을 맞춘 기법으로 바이너리를 특정 통제된 환경에 가둬 두고 실행하여 그것의 행동을 자세히 관찰하는 방법이다. 주로 악성코드의 영향을 제어할 수 있는 가상 운영체제를 사용하여 이루어지며 악성코드의 영향으로 변하는 환경(파일 시스템, 레지스트리, 네트워크 등)을 자세히 감시하고, 정보를 수집하여 기존 환경과의 미세한 차이점을 분석한다.

동적 분석 기법의 장점은 전문가부터 일반 사용자까지 분석 가능하다는 것이다. 동적 분석 기법을 이용한 리버스 엔지니어링(reverse engineering)은 실제 바이너리 코드를 생성하지 않으면서도 충분한 분석 능력을 가진다[19].

2.2.3 파일 기반 악성코드 탐지 기법

악성코드를 탐지하는 기법으로는 시그니처 기반 탐지 기법, 행위 기반 탐지 기법, 휴리스틱 탐지 기법 등의 많은 탐지 기법이 있다[4,18].

현재까지 알려진 대부분의 안티 바이러스(anti virus) 소프트웨어는 파일 기반의 진단법을 사용한다. 이는 대부분의 악성코드가 실행되기 위해서는 해당 운영체제에서 실행이 가능한 특정한 파일 형태로 되어 있어야하기 때문이다. 예를 들면, 악성코드가 윈도우 시스템에서 실행되기 위해서는 윈도우 시스템에서 실행 가능한 파일 포맷인 PE(Portable Executable) 형식을 가지고 있어야함을 의미한다.

그림 2.3과 같은 윈도우 기반 악성코드를 진단하기 위해서는 안티 바이러스 소프트웨어 역시 이러한 파일 형식을 인식하고 악성코드로 판단을 내릴 수 있는 특정 형식의 시그니처를 가지고 있어야 된다. 이러한 진단법이 대부분의 안티 바이러스 소프트웨어가 사용하는 시그니처 기반 또는 스트링 검사 방식 진단법이다. 이러한 시그니처 기반의 진단법은 악성코드로 분류된 파일의 특정 부분 또는 고유한 부분을 검사의 대상으로 하기 때문에 오탐(false positive)과 미탐(false negative)을 최소화하는 정확한 진단이 가능하다는 것과 파일 검사 시에 파일들의 특징적인 부분들만 비교하기 때문에 빠른 스캐닝(scanning)을 장점으로 들 수 있다[21].

그러나 이러한 시그니처 기반 진단법은 악성코드의 파일 자체가 몇 백 바이트만 바뀌어도 진단이 되지 않는 미탐이 발생함으로 파일이 조금만 변경된 새로운 변형에 대해서는 대응을 할 수가 없게 된다. 그리고 기존에 알려진 악성코드에 대해서만 대응을 할 수 있으므로 새로운 형태의 알려지지 않은 악성코드에 대해서는 대응을 할 수 없다는 단점을 가지고 있다[4,18].

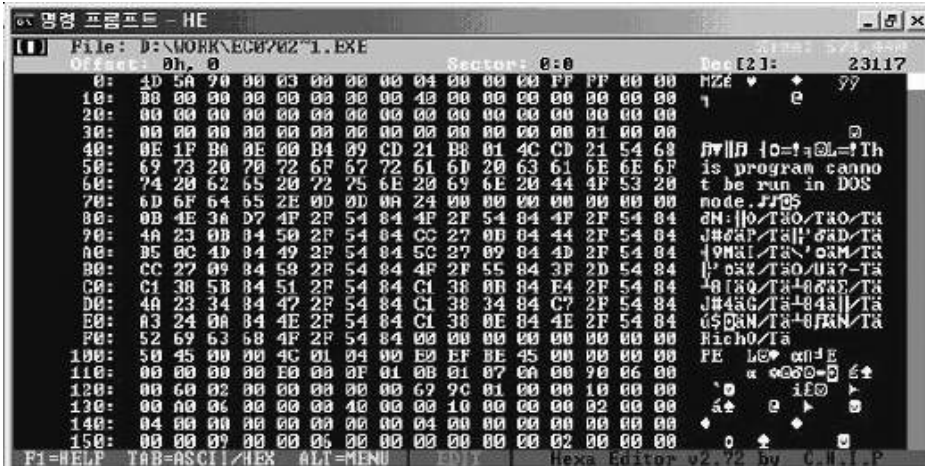


그림 2.3 PE(Portable Executable) 형식의 악성코드 파일

Fig. 2.3 Malicious code file of PE format

시그니처 기반의 진단법이 가지고 있는 한계를 극복하고자 개발한 탐지 기법 중 하나가 휴리스틱 탐지 기법이다. 휴리스틱 탐지 기법은 일반적인 악성코드가 가지고 있는 특정 폴더에 파일 쓰기와 특정 레지스트리 부분에 키 생성과 같은 명령어들을 스캐닝 엔진(scanning engine)에서 시그니처화 하여 파일을 검사할 때 이를 사용한다. 검사 대상 파일을 휴리스틱 시그니처(heuristic signature)와 비교하여 일반적으로 알려진 악성코드와 얼마나 높은 유사도를 가지고 있는지를 판단하여 알려지지 않은 새로운 악성코드를 탐지하는 기법이다[4,21,22].

이러한 휴리스틱 탐지 기법은 검사 대상이 되는 파일에 대해서 휴리스틱 검사 시에 스캐닝 엔진이 가지는 형태에 따라서 아래와 같이 크게 2가지 형태로 나누어서 볼 수 있다[21,23,24].

- 동적 휴리스틱 탐지 기법 (dynamic heuristic detection technique) : 동적 휴리스틱 탐지 기법은 검사 대상이 되는 파일을 가상화된 영역 또는 운영체제와 분리된 특정 공간에서 파일 실행을 통해 수집한 다양한 정보들을 휴리스틱 시그니처와 비교를 통하여 검사를 수행하는 기법이다. 즉 검사 대상이 되는

파일이 어떠한 형식으로든 실행이 되어야 하며 이러한 점으로 인해 런타임 휴리스틱 탐지 (runtime heuristic detection) 기법이라고도 불리고 있다.

- 정적 휴리스틱 탐지 기법 (static heuristic detection technique) : 정적 휴리스틱 탐지 기법은 동적 휴리스틱 탐지 기법과는 반대로 파일에 대해서 어떠한 형식의 파일 실행 없이 파일 자체를 그대로 스캐닝 엔진에서 읽어 들여 휴리스틱 시그니처와 비교를 수행하는 탐지 기법이다.

이 외에도 네거티브 휴리스틱(negative heuristic) 탐지 기법도 있다. 이는 이제까지 기술한 휴리스틱 탐지 기법과는 반대로 악성코드에서 전혀 사용하지 않은 윈도우 팝업 창 생성과 같은 명령어들이 존재 할 경우에는 악성코드로 판단하지 않는 기법이다. 즉, 휴리스틱 시그니처와 동일한 명령어들이 존재하지 않는 부정의 경우에만 악성코드로 판단한다[23].

이러한 형태들의 휴리스틱 탐지 기법은 기존에 알려진 악성코드도 탐지가 가능하며 알려지지 않은 새로운 악성코드 역시 탐지가 가능한 장점을 갖는다. 그러나 이 탐지 기법은 정상 파일을 악성코드라고 판단하게 되는 오탐의 발생이 가장 큰 단점이다. 특히 파일을 삭제하고 치료하는 안티 바이러스 소프트웨어의 특성상 오탐률이 높게 된다면 컴퓨터 시스템 자체에 큰 문제를 유발할 수 있게 됨으로 이는 치명적인 문제점이라고 볼 수 있다[24,25].

기존 안드로이드 기반 백신 어플리케이션에서 기본적으로 사용하는 시그니처 탐지 기법은 가볍고 정확성이 높지만 앞서 설명한 바와 같이 변종 및 신종 악성코드를 발견하지 못하는 단점이 있고 권한 특성을 이용한 행위 기반 탐지 기법도 오탐률이 높은 단점을 가진다[26]. 따라서 기존의 안드로이드 기반 백신 어플리케이션에서 사용한 시그니처 탐지 기법과 행위 기반 탐지 기법의 단점을 보완할 수 있는 정적 분석을 이용한 파일기반의 시그니처 탐지 기법과 휴리스틱 탐지 기법을 제안한다.

제 3 장 MDA(Malware Detection for Android)

시스템 설계 및 구현

본 장에서는 제안하는 탐지 기법을 적용한 MDA 시스템의 구조와 흐름을 설명한다. 그림 3.1은 MDA 시스템의 구조를 나타내며 시스템의 흐름 순서에 따라 구성요소들을 자세히 살펴본다.

3.1 MDA 시스템 구조

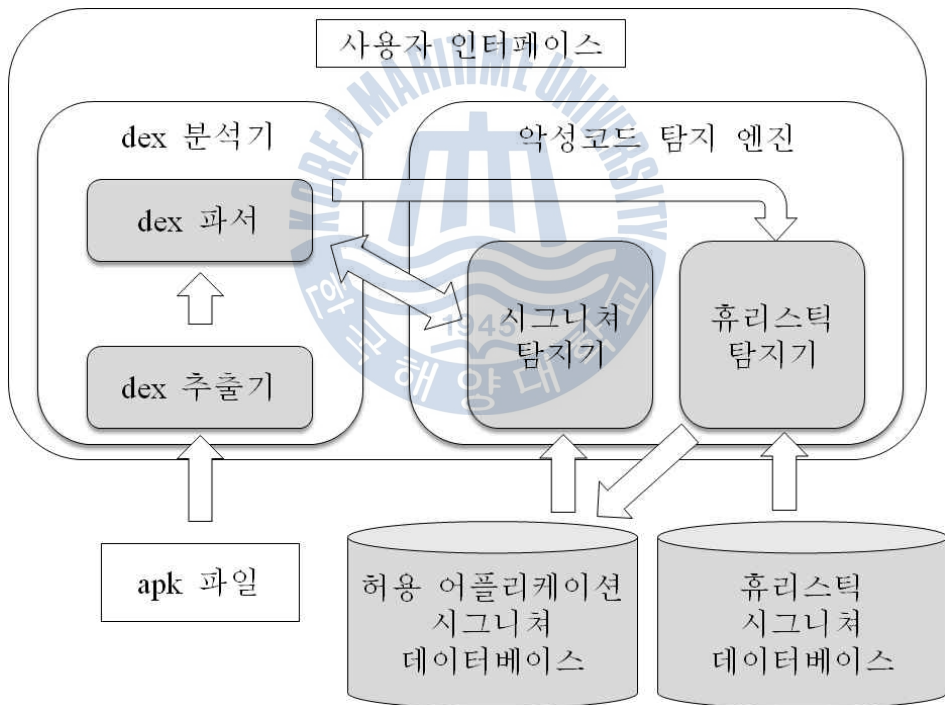


그림 3.1 MDA 시스템 구조

Fig. 3.1 System architecture of MDA

MDA 시스템은 그림 3.1과 같이 사용자 인터페이스, dex 분석기, 악성코드 탐지 엔진(engine), 휴리스틱 시그니처 데이터베이스, 허용 어플리케이션 시그니처 데이터베이스로 구성된다. dex 분석기는 다시 dex 추출기와 dex 파서(parser)로 구성되고 악성코드 탐지 엔진은 허용 어플리케이션 시그니처 데이터베이스를 참조하는 시그니처 탐지기와 휴리스틱 시그니처 데이터베이스를 참조하는 휴리스틱 탐지기로 구성된다.

3.2 dex 분석기

dex 분석기는 apk 파일에서 dex를 추출하는 dex 추출기와 분리한 dex를 구문 분석하여 탐지기에 전달하는 dex 파서로 구성된다.

apk는 Android Package의 약자로 안드로이드 기반 어플리케이션 설치파일의 확장자이다. dex 분석기는 검사할 apk 파일을 분석하여 dex 파일을 추출한 후, dex 헤더를 분리하고 헤더에서 시그니처를 추출하여 시그니처 탐지기로 전달한다. 또한 시그니처 탐지기로부터 허용되지 않은 어플리케이션이라는 값을 받으면 정적 분석 기법을 사용해 dex 바이트 코드를 구문 분석하여 호출 순서에 따라 함수들을 클래스별로 추출한 후 휴리스틱 탐지기에 전달한다.

3.2.1 dex 추출기

dex 추출기는 apk 파일에서 실행 파일에 해당하는 dex 확장자 파일을 추출하여 dex 파서에 전달한다.

안드로이드 기반 악성코드도 하나의 어플리케이션이므로 apk 파일을 분석하는 과정이 선행되어야 한다. 안드로이드 응용 프로그램을 생성하게 되면 모든 소스 코드와 자원들(resources), 자산들(assets) 그리고 하나의 메니페스트 파일이 apk 파일에 “META-INF” 폴더, “res” 폴더와 “AndroidManifest.xml”, “classes.dex”, “resources.arsc” 파일로 압축되어 포함된다. apk 파일 내부 구조는 그림 3.2와 같다.

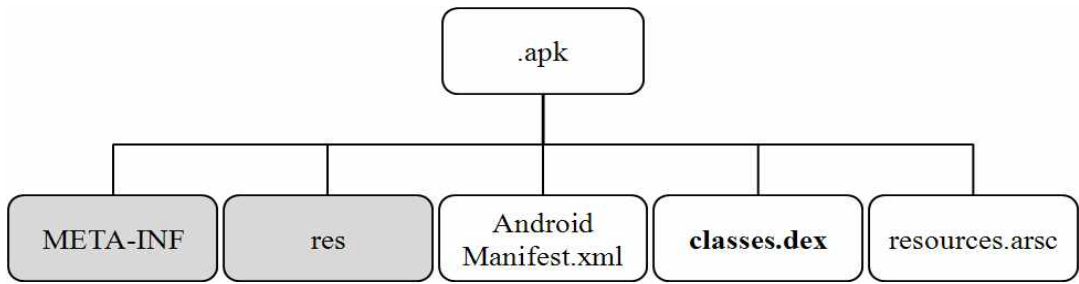


그림 3.2 apk 파일의 구조

Fig. 3.2 Structure of apk file

앞서 2장에서 설명한 바와 같이, 달빅 가상머신 상에서 실제 구동되는 실행 파일은 dex 파일이고 dex 파일 내부에 java 소스코드가 컴파일되어 바이트코드 (bytecode) 형식으로 저장되어 프로그램의 전체 정보를 가지므로 apk 파일 내부에서 dex 파일만을 추출하여 검사에 사용한다. 안드로이드 응용 프로그램을 생성하면 apk 내부에 dex 파일의 이름이 “classes.dex” 로 자동 생성되므로 모든 apk 파일 내에서 dex 파일의 이름은 동일하다. apk 파일 생성에 사용되는 압축 알고리즘은 zip 파일에 사용된 deflate 압축 알고리즘으로, 압축 및 해제 알고리즘이 공개되어 있다. java의 zip 파일 관련 라이브러리를 사용하여 dex 파일을 추출하는 방법은 그림 3.3과 같다. 추출이 끝나면 dex 파일의 내용이 바이트 배열에 저장되고, dex 파서에 전달된다.

1. 절대 위치를 포함한 파일명으로 ZipFile 객체 생성
2. “classes.dex” 이름으로 ZipFile 객체 내부의 Entry 검색해서 ZipEntry 객체 생성
3. 생성한 ZipEntry 객체의 실제 바이트 크기만큼 바이트 배열 생성
4. ZipEntry에서 InputStream을 가져와 DataInputStream 객체 생성
5. DataInputStream 객체를 이용해 생성한 바이트 배열에 저장

그림 3.3 dex 파일 추출 알고리즘

Fig. 3.3 Extract algorithm for dex file

3.2.2 dex 파서

dex 파서는 추출한 dex 파일의 정보가 담긴 바이트 배열을 정적 분석 기법을 이용하여 dex 헤더(header)를 추출하고 헤더의 정보를 이용하여 고유 시그니처를 추출한 후 시그니처 탐지기에 전달한다. 시그니처 탐지기로부터 허용되지 않은 어플리케이션 값을 전달 받으면, 실제 data 영역의 클래스 정보, 함수 정보, 함수 내 코드 정보 등을 정적 분석 기법을 이용해 구문 분석한다. 또한 dex 바이트코드 형식으로 이루어져 있는 코드 정보에서 함수를 호출하는 명령어(opcode)를 순차적으로 추출하여 전체 프로그램 내의 클래스 별 함수 호출 스택을 생성한다. 생성한 함수 호출 스택들을 휴리스틱 탐지기에 전달한다.

(가) dex 파일 내부 타입 구조

dex 파일의 정보는 표 3.1과 같은 타입들로 표현된다.

표 3.1 dex 파일에서 사용되는 타입들
Table 3.1 Data types using in dex file

이름	설명
byte	8-bit signed int
ubyte	8-bit unsigned int
short	16-bit signed int, little-endian
ushort	16-bit unsigned int, little-endian
int	32-bit signed int, little-endian
uint	32-bit unsigned int, little-endian
long	64-bit signed int, little-endian
ulong	64-bit unsigned int, little-endian
sleb128	signed little-endian base 128, 가변길이
uleb128	unsigned little-endian base 128, 가변길이
uleb128p1	unsigned little-endian base 128 plus 1, 가변길이

dex 파일의 바이트 구조는 기본적으로 little-endian을 기준으로 이루어진다. 예를 들어, 0x56 0x34 0x12 0x00와 같은 4바이트 배열은 0x00123456 이라는 값으로 인식해야 한다.

- LEB128(Little-Endian Base 128)

LEB128은 임의의 signed 또는 unsigned 정수를 가변 길이로 인코딩한 형식이다. DWARF3 형식을 기반으로 한 이 형식은 dex에서 32-bit 수에만 사용된다.

sleb128과 uleb128 타입의 디코딩 알고리즘은 그림 3.4와 그림 3.5와 같다 [27]. uleb128p1은 uleb128의 실제 값에 1을 뺀 값을 나타낸다.

```
long DecodingSignedLeb128() {
    int bitpos = 0;
    long vln = 0L;
    while(true){
        int inp = bis.read();
        vln |= ((long) (inp & 0x7F)) << bitpos;
        bitpos += 7;
        if ((inp & 0x80) == 0)
            break;
    }
    if (bitpos < 32 && ((1L << (bitpos - 1)) & vln) != 0)
        vln -= (1L << bitpos);
    return vln;
}
```

그림 3.4 Signed LEB128 디코딩 알고리즘

Fig. 3.4 Decoding algorithm for signed LEB128

```

long DecodingUnsignedLeb128() {
    long value = 0L;
    int count = 0;
    while (true) {
        int b = bis.read();
        value |= (b & 0x7f) << count;
        if((b & 0x80) == 0)
            break;
        count += 7;
    }
    return value;
}

```

그림 3.5 Unsigned LEB128 디코딩 알고리즘

Fig. 3.5 Decoding algorithm for unsigned LEB128

(나) dex 구조

dex 파일의 전체 구조 및 순서는 표 3.2와 같다.

표 3.2 dex 파일 구조

Table 3.2 Structure of dex file

이름	형식
헤더	헤더
문자열 ID 테이블	문자열 ID 아이템 []
타입 ID 테이블	타입 ID 아이템 []
프로토 ID 테이블	프로토 ID 아이템 []
필드 ID 테이블	필드 ID 아이템 []
메소드 ID 테이블	메소드 ID 아이템 []
클래스 정의 테이블	클래스 정의 아이템 []
데이터	ubyte[]
링크 데이터	ubyte[]

각 ID 테이블에는 각 형식의 정보들을 가리키는 ID 아이템 배열 인덱스나 dex 파일의 오프셋을 담은 아이템들이 나열되어 있고, 해당 형식의 실제 값들은 데이터 영역에 ubyte 배열로 저장되어 있다.

(다) dex 헤더

dex 파일의 0번째 주소부터 시작하여 파일의 가장 앞부분에 위치하며, 파일의 전체 구조를 개략적으로 담고 있는 dex 헤더는 dex 파일의 오프셋 0x00부터 0x6F까지 총 112 바이트 고정된 크기로 이루어진다. dex 헤더의 내부 구조는 표 3.3과 같다.

추출한 dex 헤더 정보에서 SHA-1 시그니처 정보는 어플리케이션 고유의 정보를 나타내기 때문에 데이터 영역을 분석하기 전에 시그니처 탐지기에 시그니처 정보를 전달하여 허용 어플리케이션 여부 검사를 하게 한다. 시그니처 탐지기로부터 허용되지 않은 어플리케이션이라는 값을 전달 받으면, 헤더 정보에 있는 오프셋들을 따라가며 정적 분석을 통해 클래스의 세부 내용을 분석한다. 본 논문에서 제안한 파서가 사용하는 헤더 정보는 표 3.3에 음영으로 표시된 부분과 같다.

표 3.3 dex 헤더 구조

Table 3.3 Structure of dex header

오프셋	길이	내용
0x0000	8	매직, 버전 정보
0x0008	4	체크섬
0x000c	20	SHA-1 시그니처
0x0020	4	dex 파일 크기
0x0024	4	헤더 크기
0x0028	4	엔디안 설정 값
0x002c	4	링크 크기
0x0030	4	링크 오프셋
0x0034	4	맵 리스트 오프셋
0x0038	4	문자열 ID 테이블 크기
0x003c	4	문자열 ID 테이블 오프셋
0x0040	4	타입 ID 테이블 크기
0x0044	4	타입 ID 테이블 오프셋
0x0048	4	프로토 ID 테이블 크기
0x004c	4	프로토 ID 테이블 오프셋
0x0050	4	필드 ID 테이블 크기
0x0054	4	필드 ID 테이블 오프셋
0x0058	4	메소드 ID 테이블 크기
0x005c	4	메소드 ID 테이블 오프셋
0x0060	4	클래스 정의 테이블 크기
0x0064	4	클래스 정의 테이블 오프셋
0x0068	4	데이터 크기
0x006c	4	데이터 오프셋

(라) 클래스 정보 추출

앞서 2장에서 설명한 바와 같이, 정적 분석은 소스 코드 레벨에서 분석하여 프로그램의 행위를 판단하는 것이다. dex 파일 내부의 클래스 정의 내용을 분석하면 소스 코드 레벨의 클래스 내부 메소드 정보를 알아낼 수 있다.

제안한 파서에서 사용되는 클래스의 타입 이름과 메소드 정보를 추출하기 위해서는 표 3.3의 dex 헤더 정보에서 클래스 정의 테이블 오프셋 값을 추출한 후 dex 파일의 시작부터 오프셋 값만큼 이동하여 클래스 정의 아이템 배열이 있는 클래스 정의 테이블 영역에 접근한다. 클래스 내부 정보들의 인덱스와 오프셋을 가지는 클래스 정의 아이템의 구조는 표 3.4와 같고 클래스 정의 아이템 내부 정보들 중에서 제안한 파서가 분석에 사용하는 항목은 음영으로 표시된 부분과 같다.

표 3.4 클래스 정의 아이템 구조

Table 3.4 Structure of class definition item

이름	형식	내용
클래스 타입 인덱스	uint	클래스 타입의 타입 ID 아이템 인덱스
접근 기호	uint	클래스 접근 기호 값
상위클래스 타입 인덱스	uint	상위 클래스 타입의 타입 ID 아이템 인덱스
인터페이스 오프셋	uint	인터페이스의 시작 오프셋
파일명 문자열 인덱스	uint	클래스가 속한 파일 이름의 문자열 ID 아이템 인덱스
주석 오프셋	uint	주석의 시작 오프셋
클래스 데이터 오프셋	uint	클래스 데이터 아이템의 시작 오프셋
정적 변수 오프셋	uint	정적 변수들의 시작 오프셋

클래스 타입 이름을 추출하기 위해서는 표 3.3의 dex 헤더 정보에서 타입 ID 테이블 오프셋 값을 추출한 후 파일에서 오프셋 값 위치로 이동하여 타입 ID 아이템 배열이 있는 타입 ID 테이블 영역으로 접근한다. 그리고 표 3.4의 클래

스 타입 인덱스 값을 추출하여 타입 ID 테이블 영역에서 인덱스 값 번째 타입 ID 아이템에 접근한다. 타입명 문자열 인덱스를 가지는 타입 ID 아이템의 구조는 표 3.5와 같다. 타입 ID 아이템은 실제 문자열 값이 아닌 문자열 ID 아이템의 인덱스 값을 가진다.

표 3.5 타입 ID 아이템 구조

Table 3.5 Structure of type ID item

이름	형식	내용
타입명 문자열 인덱스	uint	타입명 문자열의 문자열 ID 아이템 인덱스

클래스 타입 이름의 실제 문자열 값을 구하기 위해서는 표 3.3의 dex 헤더 정보에서 문자열 ID 테이블 오프셋 값을 추출한 후 파일에서 오프셋 값 위치로 이동하여 문자열 ID 아이템 배열이 있는 문자열 ID 테이블 영역으로 접근하고 표 3.5의 타입명 문자열 인덱스 값을 추출하여 문자열 ID 테이블 영역에서 인덱스 값 번째 문자열 ID 아이템에 접근한다. 문자열 데이터 아이템의 오프셋을 담고 있는 문자열 ID 아이템의 구조는 표 3.6과 같다.

표 3.6 문자열 ID 아이템 구조

Table 3.6 Structure of string ID item

이름	형식	내용
문자열 데이터 오프셋	uint	문자열 데이터 아이템의 시작 오프셋

파일에서 표 3.6의 문자열 데이터 오프셋의 값으로 이동하면 실제 문자열의 내용이 담긴 문자열 데이터 아이템에 접근한다. 문자열 데이터 아이템의 구조는 표 3.7과 같다. UTF-16 크기 값만큼 데이터 항목의 바이트 배열을 읽어 문자열로 변환하면 클래스 타입의 이름 문자열 추출이 완료된다.

표 3.7 문자열 데이터 아이템 구조

Table 3.7 Structure of string data item

이름	형식	내용
UTF-16 크기	uleb128	해당 문자열 크기(UTF-16 형식 크기)
데이터	ubyte[]	실제 UTF-8 문자열 코드 배열(끝은 NULL)

다음으로 클래스 내부의 메소드 이름을 추출하기 위해서는 표 3.4의 클래스 데이터 오프셋 값을 사용하여 상기에 설명한 방법과 동일하게 클래스 데이터 아이템에 접근한다. 클래스의 내부 정보를 담고 있는 클래스 데이터 아이템의 구조는 표 3.8과 같고 제안한 파서가 분석에 사용하는 항목은 음영으로 표시된 부분과 같다.

표 3.8 클래스 데이터 아이템 구조

Table 3.8 Structure of class data item

이름	형식	내용
정적 변수 크기	uleb128	정적 변수 개수
인스턴스 변수 크기	uleb128	인스턴스 변수 개수
일반 메소드 크기	uleb128	일반 메소드 개수
가상 메소드 크기	uleb128	가상 메소드 개수
정적 변수	변수 아이템[]	정적 변수 아이템 배열
인스턴스 변수	변수 아이템[]	인스턴스 변수 아이템 배열
일반 메소드	메소드 데이터 아이템[]	일반 메소드 데이터 아이템 배열
가상 메소드	메소드 데이터 아이템[]	가상 메소드 데이터 아이템 배열

일반과 가상 메소드의 메소드 데이터 아이템 배열 크기는 각 메소드 크기 항목의 값과 같다. 클래스 내부의 모든 메소드 정보를 분석하기 위하여 모든 메소드 데이터 아이템을 반복하여 분석한다. 메소드의 정보와 내부 코드 아이템

인덱스 등을 가지는 메소드 데이터 아이템의 구조는 표 3.9와 같고 제안한 파서에서 분석에 사용하는 항목은 음영으로 표시된 부분과 같다.

표 3.9 메소드 데이터 아이템 구조

Table 3.9 Structure of method data item

이름	형식	내용
메소드 인덱스	uleb128	메소드 ID 아이템 인덱스
접근 기호	uleb128	메소드 접근 기호 값
코드 오프셋	uleb128	코드 아이템의 시작 오프셋

메소드의 이름을 추출하기 위해서는 상기에 설명한 방법으로 메소드 ID 테이블에 접근한 후 표 3.9의 메소드 인덱스 값을 이용하여 해당 메소드 ID 아이템에 접근한다. 메소드 정의 정보들의 인덱스를 가지는 메소드 ID 아이템의 구조는 표 3.10과 같고 제안한 파서에서 분석에 사용하는 항목은 음영에 표시된 부분과 같다.

표 3.10 메소드 ID 아이템 구조

Table 3.10 Structure of method ID item

이름	형식	내용
클래스 타입 인덱스	ushort	메소드가 속한 클래스 타입의 타입 ID 아이템 인덱스
프로토 인덱스	ushort	메소드 프로토타입의 프로토 ID 아이템 인덱스
메소드명 문자열 인덱스	uint	메소드 이름 문자열의 문자열 ID 아이템 인덱스

표 3.10의 메소드명 문자열 인덱스 값을 이용하여 상기에 설명한 문자열 추출 방법으로 메소드 이름을 추출하면 클래스 내부 메소드의 이름 추출이 완료

된다. 이와 같은 방법으로 실제 내부 메소드가 있는 클래스들 즉, 클래스 데이터 오프셋 값이 0이 아니고, 메소드 크기의 값이 0이 아닌 클래스를 모두 분석하면 각 클래스별 메소드 이름들을 연관하여 추출할 수 있다.

(마) 코드 정보 추출

위에서 설명한 클래스 정보를 추출하면 메소드의 정보를 함께 추출할 수 있다. 이때, 메소드의 실제 코드 정보를 분석해야 실제 실행 코드에서 어떤 API를, 또는 어떤 메소드를 순차적으로 호출 하는지 알 수 있다. 즉, API의 이름과 호출 순서를 아는 것이 해당 메소드의 행위를 판단 할 수 있는 정보가 된다. 이를 위해서 표 3.9의 메소드 데이터 아이템에서 코드 오프셋 값으로 이동하면 메소드의 소스 코드 정보를 담은 코드 아이템에 접근하게 된다. 코드 아이템의 구조는 표 3.11과 같고 제안한 파서에서 구문 분석을 위해 사용하는 부분은 음영으로 표시된 부분과 같다.

표 3.11 코드 아이템 구조

Table 3.11 Structure of code item

이름	형식	내용
레지스터 크기	ushort	레지스터 개수
매개변수 크기	ushort	매개변수 개수
리턴변수 크기	ushort	리턴변수 필요 공간
try 크기	ushort	try 회수
debug 정보 오프셋	uint	debug 정보 시작 오프셋
명령어 크기	uint	명령어 배열 크기
명령어	ushort[]	바이트 코드 배열
여백	ushort(선택적)=0	2바이트 NULL
try	try 아이템[(선택적)]	try 아이템 배열
핸들러	catch_handlers(선택적)	catch 형식과 핸들러들

코드 아이템에서 명령어 항목의 값이 실제 달빅 가상머신에서 구동되는 명령어 배열이다. 명령어 배열은 2바이트 단위로 이루어져 있으며, 명령어는 1바이트를 사용하여 $2^8(0\sim 255)$ 개가 사용되지만 중간에 사용되지 않는 번호도 있다. 그리고 실제 분석이 필요한 명령어는 메소드를 호출하는 명령어기 때문에 관련 명령어들만 정리하면 표 3.12와 같다.

표 3.12 메소드 호출 관련 명령어
Table 3.12 Opcode for method call

명령어	이름
0x6E	invoke-virtual
0x6F	invoke-super
0x70	invoke-direct
0x71	invoke-static
0x72	invoke-interface
0x74	invoke-virtual/range
0x75	invoke-super/range
0x76	invoke-direct/range
0x77	invoke-static/range
0x78	invoke-interface/range

메소드 호출 관련 명령어들은 공통적인 형식을 가진다. 명령어 1바이트 뒤에 인수의 크기가 1바이트 나오고 이어서 메소드 인덱스가 2바이트 정수로 나온다. 이어서 2바이트는 인수들이 저장된다. 여기서 메소드 인덱스를 추출해서 메소드 테이블을 따라가면 호출한 메소드의 정보를 추출할 수 있다.

명령어들의 기본 형식은 정해져 있기 때문에 메소드 호출 관련 명령어를 제외하고는 명령어 형식만 분석하여 해당 명령어 형식의 크기만큼 오프셋을 넘기고 분석하면 메소드 호출 관련 명령어만 순차적으로 추출할 수 있다. 명령어들

의 기본 형식 크기는 표 3.13과 같다.

표 3.13 명령어 형식 크기

Table 3.13 Size of opcode format

명령어	이름	명령어 형식 크기(바이트)
0x00	nop	2
0x01	move	
...	...	
0xCF	rem-double/2addr	4
0x05	move-wide/from16	
0x08	move-object/from16	
...	...	
0xE2	ushr-int/lit8	6
0x03	move/16	
0x06	move-wide/16	
...	...	
0x78	invoke-interface/range	
0x18	const-wide	10

이와 같이 메소드 내부 코드 분석을 통하여 순차적으로 추출한 호출 메소드 이름을 스택에 쌓으면 메소드별 메소드 호출 스택이 만들어진다. 클래스 이름과 메소드 이름을 키(key)로 하여 검색하면 해당 메소드 내부 코드의 메소드 호출 스택이 빠르게 탐색되어 반환될 수 있도록 Map<클래스 이름, Map<메소드 이름, Array<클래스 이름. 메소드 이름>>>으로 설계하였다.

3.3 악성코드 탐지 엔진

시그니처 탐지기와 휴리스틱 탐지기로 구성된 악성코드 탐지 엔진은 각각의 탐지기에서 관련 데이터베이스의 정보와 비교하여 악성코드를 탐지한다. 먼저 시그니처 탐지기를 거쳐 허용된 어플리케이션인지 확인하고, 허용되지 않은 어플리케이션의 경우 휴리스틱 탐지기에서 과거의 정적 분석된 내용과 데이터베이스의 정보를 비교하여 악성 행위를 탐지한다.

3.3.1 시그니처 탐지기

시그니처 탐지기는 휴리스틱 탐지기를 수행하기 전에 1차 검사를 담당하는 탐지기로서 dex 파서에서 전달 받은 SHA-1 시그니처와 허용 어플리케이션 시그니처 데이터베이스의 정보를 비교하여 어플리케이션의 SHA-1 시그니처가 사용자가 허용한 어플리케이션의 SHA-1 시그니처인지 검사한다. dex 파일의 특정 위치에 있는 20바이트로 고정된 크기의 시그니처를 문자열 비교 방식으로 데이터베이스와 비교하므로 정확하고 빠른 검사가 이루어진다. 시그니처 탐지기가 참조하는 허용 어플리케이션 시그니처 데이터베이스는 하나의 dat 파일로 저장되며, 내부 구조는 그림 3.6과 같다.

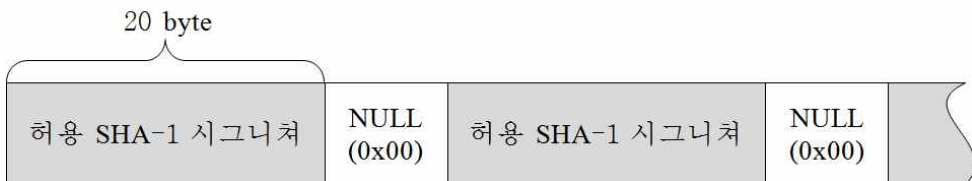


그림 3.6 허용 어플리케이션 시그니처 데이터베이스 구조

Fig. 3.6 DB structure of allow application signature

3.3.2 휴리스틱 탐지기

시그니처 탐지기를 통과하지 못한 경우, dex 파서에서 추출한 클래스와 메소드 단위의 메소드 호출 스택을 휴리스틱 탐지기가 전달 받는다. 휴리스틱 시그니처 데이터베이스는 코드 정보 데이터베이스 파일과 악성 행위 조합 정보 데이터베이스 파일로 구성되며, 휴리스틱 탐지기는 전달 받은 메소드 호출 스택을 코드 정보 데이터베이스 파일을 통해 API를 코드화 시킨 후, 코드의 조합을 악성 행위 조합 정보 데이터베이스 파일을 이용해 문자열 비교 방식으로 비교한다. 휴리스틱 탐지기에서 악성행위의 위험을 탐지한 어플리케이션을 사용자가 수동으로 허용하면 허용 어플리케이션 시그니처 데이터베이스에 해당 어플리케이션의 SHA-1 시그니처가 추가된다.

(가) 함수 호출 스택 가공

dex 파서에서 전달 받은 메소드 호출 스택은 클래스별 메소드가 나열되어 있고, 그 메소드별 내부 코드에서 호출된 메소드들이 호출되는 순서대로 “클래스 이름.함수 이름” 형식으로 나열된다. 하지만 악성행위의 시그니처는 안드로이드 API들의 조합으로 표현했기 때문에, 클래스 내부 메소드의 호출이나 다른 클래스의 메소드를 호출하는 형식은 메소드 내부의 안드로이드 API 호출들로 구성되도록 가공해야 한다. 가공 방법은 그림 3.7과 같으며 가공 결과 값은 클래스별 “클래스 이름.API 이름” 배열이 생성된다.

```

map<ClassName, map<MethodName, APIList>> stack_process_func
(map<ClassName, map<MethodName, MethodList>> stack) {
    map<ClassName, map<MethodName, APIList>> APIstack;
    class_itr = stack.begin;
    while(class_itr != stack.end) {
        method_itr = class_itr.begin;
        while(method_itr != class_itr.end) {
            for(int i=0; i<method_itr.value.size; ++i) {
                if(isAPI(method_itr.value[i]) == true)
                    APIstack[class_itr.key][method_itr.key].
                    add(method_itr.value[i];
                else{
                    APIstack.add(stack[method_itr.value
                                [i].class]
                                [method_itr.value[i].method])
                }
            }
            method_itr++;
        }
        class_itr++;
    }
    return APIstack;
}

```

그림 3.7 메소드 호출 스택 가공 알고리즘
 Fig. 3.7 Method call stack process algorithm

(나) 휴리스틱 시그니처 데이터베이스

안드로이드에서 사용하는 API들을 정수 코드 값으로 치환하기 위한 API 코드 정보 데이터베이스 파일과 치환한 정수 코드 값의 조합으로 악성 행위를 표현한 휴리스틱 시그니처들이 나열되어 있는 악성 행위 휴리스틱 시그니처 데이터베이스 파일로 구성된 데이터베이스이다. 2개의 dat 파일로 이루어져 있으며,

“APIcodes.dat” 파일에는 “클래스 이름.API 이름:n”의 형식으로 안드로이드에서 제공하는 클래스 이름과 API 이름을 키로 하여 콜론 뒤의 정수 코드 값의 쌍으로 나열되어 있다. “Heuristic Signatures.dat” 파일에는 악성 행위가 되는 API의 조합을 코드정보 데이터베이스의 정수 코드 값으로 치환하여, ‘a,b,c’ 형식으로 나열하고 ‘.’으로 구분하여 저장한다. 휴리스틱 시그니처 데이터베이스를 구성하는 각각의 데이터베이스 파일 구조는 그림 3.8과 그림 3.9와 같다.

“클래스 이름.API 이름”	:	^[1-9][0-9]*\$	NULL	“클래스 이름.API 이름”	:	^[1-9][0-9]*\$	NULL
-----------------	---	----------------	------	-----------------	---	----------------	------

그림 3.8 API 코드 정보 데이터베이스 파일 구조

Fig. 3.8 DB structure of API code data

^[1-9][0-9]*(. [1-9][0-9]*){2,5}\$.	^[1-9][0-9]*(. [1-9][0-9]*){2,5}\$.	^[1-9][0-9]*(. [1-9][0-9]*){2,5}\$.
------------------------------------	---	------------------------------------	---	------------------------------------	---

그림 3.9 악성 행위 휴리스틱 시그니처 데이터베이스 파일 구조

Fig. 3.9 DB structure of malicious acts heuristic signature

본 논문에서는 표 3.14와 같이 안드로이드 기반 모바일 악성코드의 악성 행위 중 개인정보 유출과 배터리 소모에 관련된 클래스와 API에 대하여 3개 이상 6개 이하의 API 조합으로 이루어진 154개의 악성행위 조합을 적용하였다.

표 3.14 악성 행위 관련 클래스와 API

Table 3.14 Malicious acts related class and APIs

악성 행위	클래스	API
개인정보 유출	GpsSatellite	getAzimuth
		getElevation
		getPrn
	Location	getAccuracy
		getAltitude
		getLongitude
		getLatitude

	ContactsContract	lookupContact
		getLookupUri
배터리 소모	BluetoothAdapter	enable
		startDiscovery
	WifiManager	enableNetwork
		reconnect
		startScan
		getScanResults

	NetworkInfo	getDetailedState
		isAvailable
		isConnected

제 4 장 MDA 시스템 실험

본 논문에서 제안한 MDA 시스템은 안드로이드 어플리케이션 패키지 파일인 apk 파일 하나를 그림 4.1과 같은 순서로 검사한다.

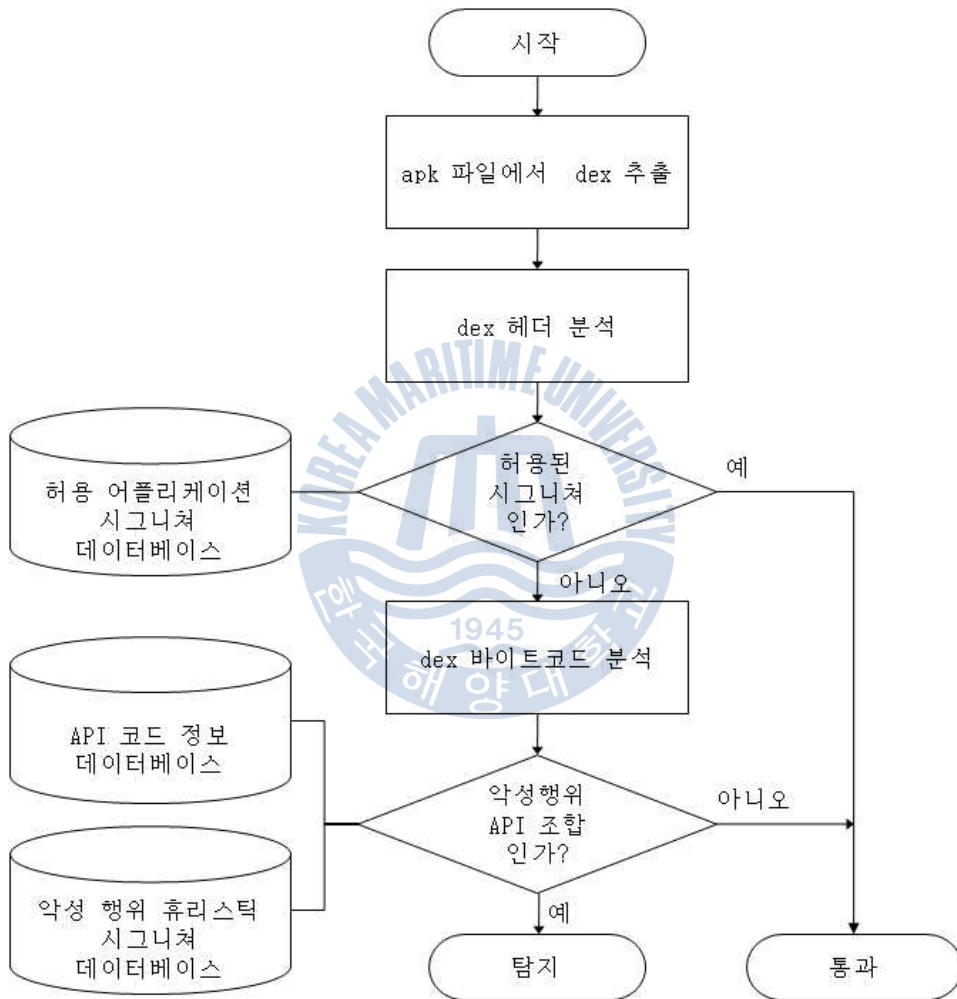


그림 4.1 MDA 시스템 순서도

Fig. 4.1 Flowchart of MDA System

4.1 dex 분석기 실험

사용자가 시스템 내부에 apk 파일을 다운로드하면 /data/app 위치에 파일이 생성된다. 분석기는 /data/app 내부의 모든 apk 파일을 검사하게 되는데, 먼저 dex 추출기가 실행되면서 apk 파일에서 실행파일에 해당하는 dex 파일을 추출한다. 추출이 완료되면 dex 파서가 실행되며 추출한 dex 파일에서 헤더를 분석한다. 헤더에 포함된 SHA-1 시그니처를 탐지엔진의 시그니처 탐지기에 전달한다. 검사가 통과하지 못하면, dex 파서는 정적 분석을 통한 dex 바이트 코드 분석을 시작한다. 분석을 완료하면 어플리케이션이 호출한 메소드가 순서대로 나오게 되는데 이를 휴리스틱 탐지기에 전달한다.

최근 개인 정보 유출의 가능성이 크다는 “오빠민지” (kr.co.onepiece.oppa-1.apk) 어플리케이션을 예로 실제 분석기 동작 과정을 설명하겠다.

4.1.1 dex 추출

apk 파일을 압축해제 툴로 살펴보면 내부 구조는 그림 4.2와 같다.

파일명	압축크기	원본크기	압축률	종류
assets				로컬 디스크
META-INF				로컬 디스크
res				로컬 디스크
AndroidManifest.xml	1,420	5,480	74%	XML 문서
classes.dex	39,007	91,980	58%	DEX 파일
resources.arsc	50,524	50,524	0%	ARSC 파일

그림 4.2 kr.co.onepiece.oppa-1.apk 구조

Fig. 4.2 structure of kr.co.onepiece.oppa-1.apk

dex 추출기를 통해 뽑아내면 압축이 해제된 원본 크기의 바이트 배열을 가진다. 그림 4.2의 dex 파일의 원본 크기와 그림 4.3의 count 크기가 동일한 것을 알 수 있다.

dr	DataReceiver (id=830085168080)
bis	DataReceiver\$XByteArrayInputStream (id=...
buf	(id=830085271392)
count	91980
mark	0
pos	0

그림 4.3 dex 추출기 결과 배열 크기

Fig. 4.3 Result of dex extractor array size

4.1.2 dex 파싱

dex 파서가 그림 4.3의 dex 파일에서 헤더를 추출하면 그림 4.4와 같다.

class_defs_off	19884
class_defs_size	73
data_off	22220
dr	DataReceiver (id=830085168080)
method_ids_off	14708
method_ids_size	647
methodList	LinkedHashMap (id=830085364816)
SHA_1_signature	(id=830085365008)
string_ids_off	112
string_ids_size	1426
type_ids_off	5816
type_ids_size	213

[-61, 91, -10, -66, -32, 63, 86, 39, 4, 57, 43, -87, 52, -123, -59, -98, 96, -14, 2, -44]

그림 4.4 dex 파일 헤더 추출 결과

Fig. 4.4 Extract result of dex file header

그림 4.4의 결과 값은 십진수 값이다. 헤더의 값들 중 SHA-1 시그니처가 추출되는 것을 확인할 수 있다. SHA-1 시그니처를 시그니처 탐지기에서 검사한 후 통과를 하지 못하면, 정적 분석을 통해 dex 바이트 코드 분석이 이루어진다. 그림 4.5는 분석 결과인 클래스별 메소드 호출 스택 내용 중 클래스 하나의 메소드 호출 스택 내용을 나타낸 것이다.

```

ActivityMapView$FunctionGetFianceLocation={
    finish=[XmlPullParserFactory.newInstance,
        XmlPullParserFactory.newPullParser,
        URL.openStream,
        XmlPullParser.setInput,
        XmlPullParser.getEventType,
        XmlPullParser.getEventType,
        XmlPullParser.next,
        XmlPullParser.getName,
        String.equals,
        XmlPullParser.getText,
        String.contains,
        ActivityMapView$FunctionGetFianceLocation$MyData.setLongitude,
        Double.parseDouble,
        ActivityMapView$FunctionGetFianceLocation$MyData.setLongitude,
        Exception.printStackTrace,
        String.equals,
        String.contains,
        ActivityMapView$FunctionGetFianceLocation$MyData.setLatitude,
        XmlPullParser.getText,
        Double.parseDouble,
        ActivityMapView$FunctionGetFianceLocation$MyData.setLatitude,
        String.equals,
        XmlPullParser.getText,
        ActivityMapView$FunctionGetFianceLocation$MyData.setLast_update,
        String.equals,
        XmlPullParser.getText,
        ActivityMapView$FunctionGetFianceLocation$MyData.setOnLocation,
        XmlPullParser.getName,
        String.equals],
    doFinalAction=[String.contains,
        ActivityMapView.access$0,
        Toast.makeText,
        Toast.show,
        ActivityMapView$FunctionGetFianceLocation$MyData.getLatitude,
        ActivityMapView$FunctionGetFianceLocation$MyData.getLongitude,
        ActivityMapView$FunctionGetFianceLocation$MyData.getLatitude,
        ActivityMapView$FunctionGetFianceLocation$MyData.getLongitude,
        ActivityMapView$FunctionGetFianceLocation.getGeoPoint,
        ActivityMapView$FunctionGetFianceLocation.addFiancePin,
        ActivityMapView.access$2,
        MapController.animateTo,
        ActivityMapView.access$3,
        ActivityMapView$FunctionGetFianceLocation$MyData.getLast_update,
        StringBuilder.append,
        StringBuilder.toString,
        TextView.setText]
}

```

그림 4.5 클래스 별 메소드 호출 결과

Fig. 4.5 Method call results of each class

이름이 “ActivityMapView\$FunctionGetFianceLocation” 인 클래스의 호출 메소드를 확인 할 수 있다. 내부 메소드는 “finish”, “doFinalAction” 두 가지를 가지고, 각 메소드 내부의 코드에서는 그림 4.5와 같은 순서로 메소드 호출이 이루어진 것을 알 수 있다. 호출된 메소드의 형식은 “클래스 이름.메소드 이름” 의 형태임을 알 수 있다.

4.2 악성코드 탐지 엔진 실험

2개의 탐지기는 시그니처 탐지기, 휴리스틱 탐지기 순서로 실행된다.

먼저 시그니처 탐지기는 dex 파서에서 SHA-1 시그니처를 전달받아 허용 어플리케이션 시그니처 데이터베이스에서 NULL을 단위로 구분하여 한 문자씩 문자열 비교 형식으로 검사를 실행한다. 데이터베이스의 값과 일치하면, 허용된 어플리케이션이므로 허용 값을 dex 파서에 반환하여 해당 검사를 종료시킨다. 휴리스틱 탐지기는 시그니처 탐지기가 허용하지 않은 경우 실행되며 dex 파서에서 전달받은 메소드 스택을 휴리스틱 시그니처 데이터베이스의 악성 행위 휴리스틱 시그니처 데이터베이스 파일과 비교하기 쉽게 가공한 후 휴리스틱 시그니처와 비교하여 검사한다.

마지막으로 휴리스틱 탐지기에서 악성코드 탐지 판단된 경우, 사용자에게 알림 메시지를 보내게 된다. 사용자가 허용을 선택하면 허용 어플리케이션 시그니처 데이터베이스에 해당 어플리케이션의 SHA-1 시그니처가 추가되며 추후 검사에서 해당 어플리케이션을 통과하게 된다.

4.2.1 SHA-1 시그니처 탐지

앞 절에서의 예와 같이 “오빠 믿지” 어플리케이션에서 추출한 SHA-1 시그니처는 {-61, 91, -10, -66, -32, 63, 86, 39, 4, 57, 43, -87, 52, -123, -59, -98, 96, -14, 2, -44} 의 값을 가지는 크기가 20바이트인 배열이다.

이와 같은 시그니처를 허용 어플리케이션 시그니처 데이터베이스의 값들과 비교하게 되는데 처음 어플리케이션을 검사하거나 사용자가 등록을 선택하지 않은 경우에는 데이터베이스에 추가되지 않기 때문에 검사에서 허용하지 않은 어플리케이션으로 판단하고 dex 파서에 허용 불가 값을 반환한다.

제안한 MDA 시스템 어플리케이션이 설치된 직후 허용 어플리케이션 시그니처 데이터베이스는 초기에 비어있는 상태로 설정되며 어플리케이션을 반복적으로 사용하면서 데이터베이스의 내용이 추가되어 저장 및 유지된다. 10가지의 어플리케이션을 허용한 상태의 허용 어플리케이션 시그니처 데이터베이스 내용은 그림 4.6과 같다.

```

-0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -A -B -C -D -E -F
00000000- CA FE 30 97 ED ED 86 95 D1 C5 A4 ED 98 C3 41 F4 [...D.....A.]
00000001- ED 9F 1C 67 00 0F 3B 93 79 CF FA 1B E5 AA 9C C0 [...g...y.....]
00000002- 77 FA 28 1C 12 B4 48 DC 50 00 1C 13 90 2A 63 43 [w.(...H.P...*cC]
00000003- 1A 35 82 57 76 A7 98 AC E1 CB 5B 63 FC D7 00 C8 [.5.Wv....[c....]
00000004- F6 6F 2D FD C6 3C ED FD EF 9B 34 46 74 D6 14 D7 [.o-...<...4Ft...]
00000005- 20 BA 79 00 92 4E 01 24 E3 F8 69 62 D2 85 98 52 [...y..N$...ib...R]
00000006- DA 84 C4 A0 CC 32 45 C2 00 76 52 CF 47 90 A0 CE [...2E...vR.G...]
00000007- FE 00 7F 8E E4 9E 2F 03 D4 C8 08 EA 58 00 56 D9 [...d.../...X.V.]
00000008- 98 A8 28 29 B7 D6 A7 9E 41 21 C0 DB BE 84 1E AB [...()]...A!.....]
00000009- 9B B1 00 00 79 C6 35 B8 3D AA 29 51 4D DF 2A 2D [...y.5.=.)QM.*-]
0000000A- 8B 81 AE 4E AB A1 38 00 92 59 F9 A6 EE 0E E3 E9 [...N..8..Y.....]
0000000B- 77 1B BF 17 8D 90 8A 43 F3 60 EE EC 00 3C 12 94 [w.....C.....<...]
0000000C- 9C 41 9F 7D FA BB 8E EE 9F D3 23 84 AF 09 A1 6C [...A.].....#. ...]
0000000D- 93 00

```

그림 4.6 허용 어플리케이션 시그니처 데이터베이스 내용
 Fig. 4.6 DB contents of allow application signature

그림 4.6은 데이터베이스 파일인 “AllowSignatures.dat” 의 내용을 16진수 값으로 덤프시키는 *hexdump*라는 프로그램을 통해 직접 분석한 결과이다. 20바이트의 시그니처와 구분자 NULL(0x00)을 포함한 21바이트가 10개 쌓여 210바이트

트가 만들어진 모습이다.

4.2.2 휴리스틱 탐지

시그니처 탐지기에서 허용되지 않은 어플리케이션으로 판단되면 dex파서에서 완성한 메소드 호출 스택을 휴리스틱 탐지기로 전달한다. 휴리스틱 탐지기에서는 클래스 내부 함수를 제외한 API만을 뽑아내는 과정을 거치고 API들의 조합을 악성행위로 판단하기 위해 먼저 API들을 휴리스틱 시그니처 데이터베이스에서 사용하는 정수 코드로 치환한다. 정수 코드로의 치환 작업은 API 코드 정보 데이터베이스 파일을 참조한다. 이전 장에서 설명한 구조와 같은 API 코드 정보 데이터베이스 파일인 “APIcodes.dat”의 내용은 그림 4.7과 같다.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
00000000-	48	74	74	70	55	52	4C	43	6F	6E	6E	65	63	74	69	6F	[HttpURLConnection]
00000001-	6E	2E	64	69	73	63	6E	6E	6E	65	63	74	3A	01	00	55	[n.disconnect:...U]
00000002-	52	4C	2E	6F	70	65	6E	43	6F	6E	6E	65	63	74	69	6F	[RL.openConnection]
00000003-	6E	3A	02	00	44	65	66	61	6C	75	74	48	74	74	70	43	[n:...DefalutHttpC]
00000004-	6C	69	65	6E	74	2E	65	78	65	63	75	74	65	3A	03	00	[lient.execute:...]
00000005-	48	74	74	70	52	65	73	70	6F	6E	73	65	2E	67	65	74	[HttpResponse.get]
00000006-	45	6E	74	69	74	79	3A	04	00	57	65	62	56	69	65	77	[Entity:...WebView]
00000007-	2E	6C	6F	61	64	55	72	6C	3A	05	00	4C	6F	63	61	74	[.loadUrl:...Locat]
00000008-	69	6F	6E	2E	67	65	74	4C	61	74	69	74	75	64	65	3A	[ion.getLatitude:...]
00000009-	06	00	4C	6F	63	61	74	69	6F	6E	2E	67	65	74	4C	6F	[.Location.getLo]
0000000A-	6E	67	69	74	75	64	65	3A	07	00	4C	6F	63	61	74	69	[ngitude:...Locat]
0000000B-	6F	6E	2E	67	65	74	41	6C	74	69	74	75	64	65	3A	08	[on.getAltitude:...]

그림 4.7 API 코드 정보 데이터베이스 내용
Fig. 4.7 DB contents of API integer code

그림 4.7에서 표시된 부분이 하나의 API 부분이다. 클래스 이름과 API 이름 문자열이 바이트 값으로 저장되어 있고, API와 정수 코드를 구분하는 ‘:’ (0x3A) 뒤에 정수 코드 값 1이 있다. 끝부분에 구분자인 NULL(0x00)이 붙는다. 이전 장에서 설명한 바와 같이 개인정보 유출과 배터리 소모에 관련된 클래스와 API를 데이터베이스에 저장하였고 표 3.14의 일부가 그림 4.7에 나타난 것을 확인 할 수 있다.

API 호출 스택의 내용이 모두 정수 코드로 바뀌면 코드가 나열되고 그 안에

서 악성행위가 되는 정수 코드의 조합을 검사한다. 악성행위 정수 코드 조합들은 휴리스틱 시그니처 데이터베이스의 악성 행위 휴리스틱 시그니처 데이터베이스 파일에 저장되어 있으며 이전 장에서 설명한 구조와 같이 데이터베이스 파일인 “HeuristicSignatures.dat”의 내용은 그림 4.8과 같다.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
00000000-	87 2C 6A 2C	44 2E 3A 2C	6E 2C 06 2C	12 2C 05 2E	[, j, D, :, n, , , , ,]												
00000001-	08 2C 54 2C	91 2E 3E 2C	02 2C C6 2C	64 2E 90 2C	[, T, , , >, , , , d, ,]												
00000002-	44 2C BD 2C	71 2E A5 2C	02 2C 87 2C	4E 2E B4 2C	[D, , , q, , , , , N, ,]												
00000003-	8A 2C 1A 2E	54 2C 04 2C	87 2C AE 2E	0F 2C B7 2C	[, , , T, , , , , ,]												
00000004-	24 2C 77 2C	6C 2E 60 2C	A1 2C 81 2C	22 2C 3C 2E	[\$, w, l, ' , , , , " , < ,]												
00000005-	94 2C 91 2C	81 2C 14 2C	A9 2E 06 2C	07 2C 02 2E	[, , , , , , , , , ,]												
00000006-	0A 2C C5 2C	C1 2C 7B 2C	75 2E 03 2C	46 2C 1F 2E	[, , , , , {, u, , , F, ,]												
00000007-	B7 2C B9 2C	8A 2E 0F 2C	66 2C A2 2E	2B 2C 35 2C	[, , , , , f, , , +, 5,]												
00000008-	AC 2C C0 2C	BA 2E 4B 2C	82 2C 28 2C	B5 2E 01 2C	[, , , , , K, , , (, , ,]												
00000009-	C6 2C 29 2C	88 2E AF 2C	35 2C 74 2E	7B 2C 65 2C	[, ,) , , , , 5, t, {, e,]												
0000000A-	50 2C 2D 2C	77 2E 62 2C	1B 2C 07 2C	10 2C 6A 2C	[P, -, w, b, , , , , j,]												
0000000B-	69 2E 14 2C	83 2C 56 2C	39 2E 6B 2C	21 2C 38 2E	[i, , , , y, 9, k, l, 8,]												
0000000C-	7C 2C 17 2C	3A 2C 5B 2C	13 2C 3E 2E	6B 2C 26 2C	[l, , , : , [, , , >, k, & ,]												
0000000D-	4D 2C 4F 2E	08 2C 02 2C	87 2C 42 2E	61 2C B7 2C	[M, 0, , , , , , B, a, ,]												
0000000E-	B1 2C 84 2E	79 2C C2 2C	0F 2C C0 2E	9A 2C 1A 2C	[, , , , y, , , , , ,]												
0000000F-	15 2C 8E 2E	61 2C 66 2C	2A 2C 0F 2E	50 2C 28 2C	[, , , a, f, +, , , P, (,]												

그림 4.8 악성 행위 휴리스틱 시그니처 데이터베이스 내용
 Fig. 4.8 DB contents of heuristic signature

그림 4.8의 표시된 부분이 하나의 조합을 나타낸 부분이다. 정수 코드를 구분하는 ‘,’ (0x2C)가 정수 코드 사이에 있고 조합 끝부분에는 구분자인 ‘.’ (0x2E)가 붙는다. 검사를 하는 “오빠 믿지” 어플리케이션은 표시된 부분의 코드 조합 값인 “6,7,2”에 해당하는 “Location.getLongitude”, “Location.getLatitude”, “URL.openConnection” API 호출의 조합이 탐지되었다. GPS 위치 정보 수집과 네트워크 접속에 사용되는 상기 API들은 이전 장에서 설명한 바와 같이 개인정보 유출로 분류되는 악성행위의 위험이 있는 것으로 판단한다.

4.2.3 허용 어플리케이션 시그니처 추가

악성코드로 확인되면 그림 4.9과 같이 알림창이 출력되고 사용자에게 해당 어플리케이션을 치료할 것인지, 허용 어플리케이션으로 등록할 것인지 묻는다.



그림 4.9 악성코드 탐지 알림창
Fig. 4.9 Malware detection messagebox

삭제(치료) 버튼을 선택하면 해당 어플리케이션이 삭제되며 치료가 이루어진다. 등록 버튼을 선택하면 SHA-1 시그니처를 허용 어플리케이션 시그니처 데이터베이스에 추가한다. 추가된 데이터베이스의 내용은 그림 4.10과 같다.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
00000000-	CA	FE	30	97	ED	ED	86	95	D1	C5	A4	ED	98	C3	41	F4	[.0.....A.]
00000001-	ED	9F	1C	67	00	0F	38	93	79	CF	FA	1B	E5	AA	9C	C0	[...g...;y.....]
00000002-	77	FA	28	1C	12	B4	48	DC	50	00	1C	13	90	2A	63	43	[w.(...H.P....+cC)]
00000003-	1A	35	82	57	76	A7	98	AC	E1	CB	58	63	FC	D7	00	C8	[.5.#\v.....[c....]
00000004-	F6	6F	2D	FD	C6	3C	ED	FD	EF	98	34	46	74	D6	14	D7	[.o-...<...4Ft...]
00000005-	20	8A	79	00	92	4E	01	24	E3	F8	69	62	D2	85	98	52	[.y.N.\$...ib...R]
00000006-	DA	84	C4	A0	CC	32	45	C2	00	76	52	CF	47	90	A0	CE	[.....2E...vR.G...]
00000007-	FE	00	7F	8E	E4	9E	2F	03	D4	C8	08	EA	58	00	56	D9	[.0.../...X.V.]
00000008-	98	A8	28	29	B7	D6	A7	9E	41	21	C0	0B	BE	84	1E	AB	[...()...A!.....]
00000009-	98	B1	00	00	79	C6	35	B8	3D	AA	29	51	4D	DF	2A	2D	[...y.5.=.)QM.*-]
0000000A-	8B	81	AE	4E	AB	A1	38	00	92	59	F9	A6	EE	0E	E3	E9	[...N..8..V.....]
0000000B-	77	1B	BF	17	8D	90	8A	43	F3	60	EE	EC	00	3C	12	94	[w.....C.....<..]
0000000C-	9C	41	9F	7D	FA	BB	8E	FE	9E	D3	23	84	AF	09	A1	6C	[.A.).....#.....l]
0000000D-	93	00	C3	5B	F6	BE	E0	3F	56	27	04	39	2B	A9	34	85	[...[...?V'.9+.4.]
0000000E-	C5	9E	60	F2	02	D4	00										[.....]

그림 4.10 추가된 허용 어플리케이션 시그니처 데이터베이스 내용
 Fig. 4.10 DB contents of added allow application signature

끝부분에 추가된 시그니처는 “오빠 믿지” 어플리케이션의 SHA-1 시그니처 20 바이트와 구분자인 0x00이다. 앞의 그림 4.4에서 확인한 바와 같이 {-61, 91, -10, -66, -32, 63, 86, 39, 4, 57, 43, -87, 52, -123, -59, -98, 96, -14, 2, -44}가 16진수 표현인{0xC3, 0x5B, 0xF6, 0xBE, 0xE0, 0x3F, 0x56, 0x27, 0x04, 0x39, 0x2B, 0xA9, 0x34, 0x85, 0xC5, 0x9E, 0x60, 0xF2, 0x02, 0xD4}로 추가된 것을 확인 할 수 있다. 추가된 시그니처의 내용은 다음에 검사할 때 시그니처 탐지기에 의해 검사되어 허용 어플리케이션으로 판단하게 된다.

제 5 장 결론 및 향후 과제

본 논문에서는 정적 분석을 사용한 안드로이드 기반의 악성코드 탐지 기법을 다루었다. 악성코드 탐지 기법으로는 파일 기반의 시그니처 탐지 기법과 휴리스틱 탐지 기법을 사용하였다. 기존의 안드로이드 기반 백신 어플리케이션에서는 시그니처 탐지 기법만을 사용하거나, 권한을 검사하는 행위 탐지 기법을 선택적으로 사용하여 변종이나 신종 악성코드를 탐지하지 못하거나 어플리케이션의 실제 행동을 파악하지 않고 단순히 요청 권한만으로 판단하여 오탐률이 높았다. 하지만 본 논문에서 제안한 탐지 기법은 정적 분석을 통해 어플리케이션에서 호출되는 API를 추출하여 어플리케이션이 시스템에 직접 접근하는 행위를 파악하고, 악성행위에 해당하는 API 호출의 조합을 비교하여 변종, 신종 악성코드의 탐지가 이루어진다. 또한 기존의 시그니처 탐지 기법을 선행 처리하여 악성코드의 빠른 탐지가 가능하다.

향후에는 휴리스틱 시그니처에 해당하는 안드로이드 API를 모든 범위에서 탐지 가능하도록 데이터베이스에 추가하고, API 조합을 여러 단계에서 복합적으로 검사 하여 오탐률이 더 낮아지도록 탐지 방법을 개선해야 할 것이다. 휴리스틱 기법 외에 다른 탐지 기법도 도입하여 성능을 확인해 보고 융합할 수 있는 탐지 기법의 연구도 필요하다.

참고 문헌

- [1] 김재생, “스마트 폰의 기술 소개 및 활성화방안”, 한국콘텐츠학회지, 제8권, 제2호, pp. 34-38, 2010.
- [2] 남기효, 박상중, 강형석, 길지호, “스마트폰 보안 기술 및 솔루션 동향”, 정보통신산업진흥원 주간기술동향, 제1466호, pp. 1-7, 2010.10.
- [3] 김지훈, 조시행, “사이버 환경에서의 보안위협”, 정보보호학회지, 제20권, 제4호, pp. 11-20, 2010.
- [4] 장영준, “알려지지 않은 악성코드 탐지를 위한 기법”, 안철수연구소 전문가 칼럼, http://www.ahnlab.com/kr/site/securityinfo/secunews/secuNewsView.do?curPage=1&menu_dist=3&seq=10894&columnist=15&dir_group_dist=0&dir_code=, 2007.
- [5] 김상형, 안드로이드 프로그래밍 정복, 한빛미디어, 2010.
- [6] Android Developers, What is Android?, <http://developer.android.com/guide/basics/what-is-android.html>.
- [7] Shane Conder and Lauren Darcey, 안드로이드 프로그래밍, 위키북스, 2010.
- [8] Jesse Burns, “Developing Secure Mobile Applications for Android”, <https://www.isecpartners.com/white-papers/2010/7/22/developing-secure-mobile-applications-for-android.html>, 2010.
- [9] 정승일, “안드로이드 플랫폼과 스마트폰 기술 발전 동향”, 대한전자공학회 학술대회, 제33권, 제1호, pp. 2000-2001, 2010.
- [10] 김도현, 오요시, 최희철, 백승현, 박종혁, “안드로이드 스마트폰에서의 강화된 정보보안 연구”, 한국정보기술융합학회논문지, 제3권, 제2호, pp. 110-118, 2010.
- [11] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi

- Dolev, and Chanan Glezer, "Google Android:A Comprehensive Security Assessment", IEEE Security & Privacy, Vol.8, Issue.2, pp. 35-44, 2010.
- [12] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev, "Google Android:A State-of-the-Art Review of Security Mechanisms", Computing Research Repository, 2009.
- [13] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka, "Towards Formal Analysis of the Permission-based Security Model for Android", International Conference on Wireless and Mobile Communications, pp. 87-92, 2009.
- [14] William Enck, Machigar Ongtang, and Patrick McDaniel, "Understanding Android Security", IEEE Security & Privacy, Vol.7, Issue.1, pp. 50-57, 2009.
- [15] 김기영, 강동호, "개방형 모바일 환경에서 스마트폰 보안기술", 한국정보보호학회지, 제19권, 제5호, pp. 21-28, 2009.
- [16] 강동호, 한진희, 이윤경, 조영섭, 한승완, 김정년, 조현숙, "스마트폰 보안 위협 및 대응 기술", ETRI 전자통신동향분석, 제25권, 제3호, pp. 72-80, 2010.
- [17] 김익수, 정진혁, 이형찬, 이정현, "모바일 악성코드 분석 방법과 대응 방안", 한국통신학회논문지, 제35권, 제4호, pp. 599-609, 2010.
- [18] Markus Schmall, "Bulding an Anti-Virus Engine", <http://www.symantec.com/connect/articles/building-anti-virus-engine>, 2002.
- [19] Malware Analysis, <http://www.malwareinfo.org/files/MalwareAnalysisHow2.pdf>.
- [20] 이형준, 김철민, 이성욱, 홍만표, "정적 분석을 이용한 다형성 스크립트 바이러스의 탐지기법 설계", 한국정보과학회 학술발표논문집, 제30권, 제1호, pp. 407-409, 2003.

- [21] 하우리 보안칼럼, “악성코드의 악성 행위와 탐지 기법”, http://www.hauri.co.kr/customer/security/colum_view.html?intSeq=93&page=1&keyfield=strSubject&key=%BE%C7%BC%BA%C4%DA%B5%E5, 2009.
- [22] Igor Muttik, “STRIPPING DOWN AN AV ENGINE”, virus bulletin, 2000.
- [23] Symantec White Paper Series, “Understanding Heuristics”, <http://symantec.com/avcenter/reference/heuristc.pdf>, 1998.
- [24] Markus Schmall, “Heuristic Techniques in AV Solutions”, <http://symantec.com/connect/articles/heuristic-techniques-av-solutions-overview>, 2002.
- [25] 김은영, 오형근, 배병철, “바이러스 탐지를 위한 휴리스틱 스캐닝 기법 및 행위 제한 기법 분석”, 한국정보과학회 학술발표지, 제29권, 제2호, pp. 577-579, 2002.
- [26] Asaf Shabtai, “Malware Detection on Mobile Devices”, Mobile Data Management, pp. 289-290, 2010.
- [27] LEB128, <http://en.wikipedia.org/wiki/LEB128>

