Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.25, NO.1, Feb. 2015 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2015.25.1.17

안드로이드 환경에서 자바 리플렉션과 동적 로딩을 이용한 코드 은닉법*

김 지 윤,[1†] 고 남 현,[2] 박 용 수[1‡]
[1]한양대학교, [2]한국폴리텍2대학

A Code Concealment Method using Java Reflection and Dynamic Loading in Android*

Jiyun Kim,[1†] Namhyeon Go,[2] Yongsu Park[1‡]
[1]Hanyang University, [2]Korea Polytechnic II College

요 약

본 논문은 기존에 널리 사용되는 바이트코드(bytecode) 중심의 안드로이드 어플리케이션 코드 난독화 방법과 달리 임의의 안드로이드 어플리케이션의 DEX 파일 자체를 추출하여 암호화하고, 암호화한 파일을 임의의 폴더에 저장한 후 코드를 수행하기 위한 로더 앱을 만드는 방법을 제시한다. 이벤트 처리 정보를 은닉하기 위하여, 로더 앱 내부의 암호화된 DEX 파일은 원본 코드와 Manifest 정보 일부를 포함한다. 로더 앱의 Manifest는 원본 앱의 Manifest 정보 중에서 암호화된 클래스에 포함되지 않은 정보만을 기재하였다. 제안기법을 사용시, 첫째로 공격자는 백신을 우회하기 위해 난독화된 코드를 포함한 악성코드 제작이 가능하고, 둘째로 프로그램 제작자의 입장에서는 제안기법을 이용하여 저작권 보호를 위해 핵심 알고리즘을 은폐하는 어플리케이션 제작이 가능하다. 안드로이드 버전 4.4.2(Kitkat)에서 프로토타입을 구현하고 바이러스 토탈을 이용하여 악성코드 난독화 능력을 점검해서 제안 기법의 실효성을 보였다.

ABSTRACT

Unlike existing widely used bytecode-centric Android application code obfuscation methodology, our scheme in this paper makes encrypted file i.e. DEX file self-extracted arbitrary Android application. And then suggests a method regarding making the loader app to execute encrypted file's code after saving the file in arbitrary folder. Encrypted DEX file in the loader app includes original code and some of Manifest information to conceal event treatment information. Loader app's Manifest has original app's Manifest information except included information at encrypted DEX. Using our scheme, an attacker can make malicious code including obfuscated code to avoid anti-virus software at first. Secondly, Software developer can make an application with hidden main algorithm to protect copyright using suggestion technology. We implement prototype in Android 4.4.2(Kitkat) and check obfuscation capacity of malicious code at VirusTotal to show effectiveness.

Keywords: Java reflection, Malware, Malicious code, DES, Data encryption standard, Bytecode, Copyright protection, Obfuscation, Intent, Intent filter, Class encryption, Dynamic keys, AndroidManifest

접수일(2014년 8월 25일), 수정일(1차: 2014년 10월 24일, 2차: 2014년 12월 01일), 게재확정일(2014년 12월 11일)

* 본 논문은 2014년 한국컴퓨터정보학회 제50차 하계학술대회에서 발표한 논문("안드로이드 환경에서 클래스 반사와 예외 처리를 이용한 임의 코드 수행 방법 및 코드 은닉 방법")을 확장한 것임.

* 이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구입니다(No. 2012R1A1A2007263)

† 주저자, kjycslab@gmail.com
‡ 교신저자, yongsu@hanyang.ac.kr(Corresponding author)

I. 서 론

스마트폰은 이제 전 세계인의 보편적인 통신 기기로 자리 잡고 있다. 스트레티지 애널리틱스[1]에 따르면, 스마트폰의 운영체제(OS) 중에서, 시장 점유율이 가장 높은 운영체제는 안드로이드이다. 안드로이드 환경은 개방성이 높은 장점이 있지만, 어플리케이션이 역공학 공격에 취약한 단점이 있다 [2-6].

이러한 취약점 때문에 안드로이드 앱의 원본 코드를 변조하여 금융사기를 포함한 각종 악성 행위를 하는 어플리케이션을 개발 및 배포하거나 불법 복제를 하는 행위가 빈번히 일어나고 있다[6]. 역공학 공격 이외에도 최근에 WebView API를 활용한 공격[2]이 이슈가 되는 등 안드로이드 운영체제의 취약점을 이용한 다양한 공격이 발생되고 있다.

안드로이드 앱에 대한 역공학 공격을 막기 위하여 식별자 변환(Renaming), 제어 흐름 변환(Control Flow), 문자열 암호화(String Encryption), API 은닉(API Hiding), 클래스 암호화(Class Encryption) 등의 다양한 방법[7](관련 연구 2.4.1 참조)이 사용되고 있으나 이런 방법들은 대부분 분석가가 앱의 소스 코드를 보는 것 자체를 차단하는 것이 아니라 소스 코드를 좀 더 복잡하게 만드는 것에 집중한다.

본 논문에서는 기존 방식과는 달리, 소스 코드가 아닌 DEX 파일[8]을 분석가가 보기 매우 힘들도록 세 개의 키를 활용하는 Triple DES 알고리즘(TDEA)[9-11](관련연구 2.5.2 참조)으로 암호화하는 방법을 제안한다. 제안 방법은 기존에 널리 사용되는 바이트코드 중심의 안드로이드 어플리케이션 코드 난독화 방법과 달리 암호화하려는 앱 내부의 DEX 파일을 암호화하고, 이 파일을 복호화하여 실행하는 로더 앱을 만들 때에 암호화한 DEX 파일을 리소스 파일로 앱 내부에 포함함으로써 로더 앱 실행시 런타임에서 동적으로 암호화된 바이트코드를 복호화 및 동적 로딩하고 자바 리플렉션 기법으로 실행한다. 또한, 분석을 더욱 어렵게 하기 위하여, 원본 앱의 Manifest에 기재된 리시버, 서비스 컴포넌트의 인텐트, 인텐트 필터 정보를 삭제하고 나머지 리시버 컴포넌트 정보도 삭제하여, 암호화할 원본 앱의 DEX 파일 내부에 새로운 클래스를 만들어 삭제한 정보를 기재한다. 새롭게 추가된 클래스는 로더 앱이 암호화된 DEX를 복호화하고 로드한 후에 리플렉션으로 실행되도록 만든다. 따라서 제안 기법은 관련 연구 2.4.1에 소개된 기존의 바이트코드 변환 방식의 난독화 기법[7]과 함께 활용할 수 있다.

본 논문의 방법과 유사한 방식을 취한 관련 논문[4]에서, 본 방법론에 대해 저장된 파일이 탈취되거나 권한 문제로 삭제할 수 없는 문제점이 재기되었다[4]. 그러나 본 연구에서는 앱 내부의 데이터 영역을 활용하여 자유자재로 복호화한 데이터를 기록 및 삭제하고 보호할 수 있도록 하였다.

제안 방법은 원본 안드로이드 어플리케이션 내부 DEX 파일을 디컴파일하여 새로운 코드를 넣은 후에 다시 컴파일한다. 이후 이 DEX 파일 전체를 암호화하고 실행을 위한 로더 앱 안에 포함시켰다. 이 때문에 원본 앱의 코드 전체가 난독화 되어서 원본 앱과 난독화된 앱의 유사도를 안드로가드[12]의 안드로심으로 비교시 상당히 낮은 유사도를 얻는 효과가 있었고 악성 앱을 은닉하여 바이러스 토탈[13]에서 진단 결과 단 1개의 백신도 원본 앱과 같은 진단 결과를 내놓지 않았다. 또한 분석가가 복호화 키를 획득하는 방법 등으로 암호화된 파일을 복호화하지 못하면 원본 코드를 전혀 볼 수 없는 효과가 있다.

본 논문의 구성은 2장에서 관련 연구를 설명하고 3장에서 Triple DES를 이용한 암호화 및 코드 은폐 기법, 자바 리플렉션과 클래스 로더를 이용한 복화화 및 임의 코드 수행 방법을 설명 후 제안 기법의 안전성을 분석한다. 이후 실용성을 보이기 위해 4장에서 프로토타입 구현 결과와 악성 앱 및 정상 앱을 이용한 바이러스 검사와 유사도 측정 결과 및 기존 연구와의 비교를 다루었고 5장에서 결론으로 마무리 한다.

II. 관련 연구

2.1 일반적인 안로드이드 어플리케이션

2.1.1 안드로이드 어플리케이션 구조 및 런타임 환경

안드로이드 어플리케이션은 자바 프로그래밍 언어로 작성되며, APK라고 불린다. 안드로이드 어플리케이션은 zip 알고리즘으로 압축된 파일 형태와 같은 형태이다[4]. 안드로이드 어플리케이션을 zip 알고리즘으로 압축해제를 하면 크게 그림 1의 ①~④의 요소를 얻을 수 있다. ①은 소스 코드가 포함된 DEX 파일이고 ②는 컴파일된 리소스 파일이다. ③은 컴파일할 필요가 없는 리소스 파일이며, ④는 어플리케이션 실행에 필요한 정보들이 있는 파일이다[14]. 안드로이드 어플리케이션은 암호로 서명된다. 이는 보안성을 높이

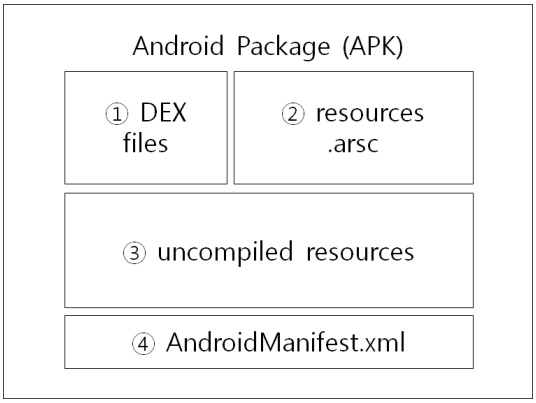

Fig. 1. Android application structure.[14]

지는 못한다. 하지만 안드로이드 어플리케이션의 개발자를 확인하거나 구분하는데 사용된다[4].

안드로이드 어플리케이션 실행을 위한 요소는 Dalvik 바이트코드, Dalvik Virtual Machine(DVM), Native Code, 안드로이드 어플리케이션 Framework의 4가지로 구성된다. DVM은 Dalvik 바이트코드를 읽어서 실행하고 Native Code의 함수를 호출할 수 있는 역할을 한다. Native Code는 라이브러리 요청에 의해 DVM이 아닌 프로세서에서 직접 실행된다. 안드로이드 어플리케이션 Framework는 운영체제의 시스템 운영과 어플리케이션의 상호 작용을 위한 API와 같은 기반을 제공한다[4].

2.1.2 안드로이드 어플리케이션 실행시 일반적인 클래스 등록 절차

제안 방식은 기존방식과 달리 암호화할 원본 앱의 DEX 파일 전체와 Manifest 일부를 암호화하기 때문에 원본 앱의 실행 형태를 그대로 복원시켜 주어야 한다. 따라서 자바 리플렉션을 활용하여 수동으로 프로세스에 클래스를 등록하는 절차가 필요하다.

안드로이드는 Zygote를 통해 프로세스를 관리한다. 모든 안드로이드 앱의 액티비티와 클래스는 Zygote에 의해 fork(생성)된 프로세스 위에서 동작한다. 프로세스를 fork할 때 어플리케이션의 실행 요청이 담긴 Binder IPC를 받는 액티비티 관리자는 액티비티 스레드라고 명명되는 표준 클래스를 사용한다. 이 클래스는 시스템의 리눅스 프로세스와 어플리케이션을 묶는(binding) 작업을 수행한다.

이러한 프로세스 생성 과정을 거치고 나서 안드로이드의 어플리케이션 계층에서는 실제 실행이 가능하도록 액티비티 스레드 클래스를 호출하고 Instrumentation 정보 등록을 한다[15].

구체적으로 클래스 기재 후 실행중인 안드로이드 시스템과 기재한 클래스의 상호작용을 위해 Instrumentation 형태의 클래스를 생성한 뒤 LoadedApk 클래스의 makeApplication를 호출하여 등록한다. 앞서 리플렉션 기법으로 액티비티 스레드에 등록된 어플리케이션 정보와 등록한 Instrumentation 정보를 이용하여 어플리케이션 초기화 설정을 한다.

제안 기법에서는 클래스가 로드될 때 리플렉션 기법으로 안드로이드 액티비티 스레드에 임의의 어플리케이션 정보를 등록할 클래스를 생성한다. 이를 바탕으로 로더 앱에서 로드한 DEX의 원본 어플리케이션 정보를 담고 있는 클래스와 액티비티 스레드의 앱 바인딩 데이터로부터 임의대로 실행하고자 하는 클래스를 기재할 수 있는 엑티비티 스레드 내부의 공간을 할당할 수 있다. 이 공간은 어플리케이션의 정보가 있다.

이를 활용하여 클래스 네임 필드에 사용하고자 하는 클래스의 정보를 임의대로 기재한다. 리플렉션 기법을 이용하여 어플리케이션 정보가 저장된 액티비티 스레드 공간에 리소스 정보를 등록하여 차후 은닉된 DEX에서도 내부 리소스 영역에 접근하여 활용할 수 있는 여지를 제공한다.

요약하면 본 논문에서는 안드로이드 시스템이 어플리케이션이 실행할 때 수행하는 일련의 과정 중에서 액티비티 스레드 클래스를 이용하여 어플리케이션 정보를 등록하고 실행하는 시스템의 기본 동작을 자바 리플렉션을 활용하여 개발자가 임의대로 수행하도록 구현하였다.

2.2 안드로이드 어플리케이션 공격 기법

2.2.1 역공학 공격

안드로이드 어플리케이션(APK 파일) 내부에는 클래스 파일을 포함하는 DEX 파일이 존재한다. 파일의 확장자를 apk에서 zip으로 바꾸고 압축 해제하면 DEX 파일과 리소스 파일 등을 획득할 수 있다. 이렇게 얻은 DEX 파일은 공개 툴인 dex2jar[16] 등을 이용하여 jar 파일로 변환할 수 있다. 이렇게 얻은 jar 파일을 이용하면 공개 툴인 JD-GUI[17] 등으로 쉽게 바이트코드를 확인하고 자바 코드로 변환할 수

있고, 소스 파일에서 취약점을 분석 할 수 있다[2][3][6]. 또한, apktool[18] 등으로 앞서 얻은 jar 파일을 클래스 파일로 다시 디컴파일하고 바이트 코드를 변조하여 악성코드를 삽입하여 컴파일하는 등 리패키징 앱 제작이 가능하다[5].

2.3 자바 리플렉션을 활용한 어플리케이션 공격 기법

자바 리플렉션은 일반적으로 자바 가상 머신에서 구동되는 어플리케이션의 런타임 동작을 검사하거나 조작하려는 프로그램에 의해 사용된다. 자바 리플렉션은 강력한 기능을 가지고 있어 어플리케이션을 원래 동작과 다른 방향으로 동작하게 하거나 정상적인 사용을 불가능하게 만들 수 있다는 점을 주의해야 한다[19].

2.3.1 WebView 위협 모델

실행중인 앱은 WebView를 활용하여 내부에서 브라우저를 호출하여 웹 페이지를 표시하고 제어할 수 있다. 특히 WebView API를 활용하면 웹페이지에 포함된 자바스크립트를 실행하거나 이벤트를 모니터링 할 수 있다. 이 때문에 악성 코드가 삽입된 웹페이지를 로드 시키고 자바 리플렉션을 활용하여 시스템 API를 웹에서 호출하는 방식으로 안드로이드 시스템을 공격할 수 있었다. 이 방법은 최근 안드로이드 운영체제에서는 차단되었다. 다른 방법으로 WebView를 활용하여 악성 앱을 실행시켜 웹 애플리케이션을 공격하는데도 이용될 수 있다[2].

2.3.2 Points-to 정적 위반 탐지 방법

자바는 클래스를 동적으로 생성 및 리플렉션을 사용하여 호출하는 것을 지원한다. 이를 이용하여 기존의 정적 분석 방법과 유사한 코드 취약점 분석 작업을 동적 분석으로 진행할 수 있다. 아래는 이런 방식으로 취약점을 정적으로 탐지하는 방법인 문맥기반 Points-to 분석 방법이다. Points-to 분석 결과는 pointsto(v, h) 관계를 가지며 v는 존재하는 변수를 뜻하며 h는 힙(Heap) 영역에 적재된 변수를 뜻한다. 이를 식으로 표현하면 아래와 같다[20].

$1 \le i < k$:
$pointsto(v_i,h_i)$ and $pointsto(v_{i+1},h_{i+1})$ [20]

2.3.3 자바 리플렉션을 이용한 윈도우즈 보안 위협

자바는 안드로이드 이외에도 다양한 플랫폼에서 활용하고 있는 프로그래밍 언어이다. 최근 윈도우즈에서도 리플렉션을 이용해 자바의 보안 관리자(SecurityManager)를 우회할 수 있는 취약점을 이용하여 임의의 PE 코드를 실행하는 공격이 발생하였다. 이처럼 리플렉션 기법은 시스템의 기반을 자바로 두고 있는 안드로이드는 물론 자바를 활용하는 다른 플랫폼에서도 충분한 보안 위협이 될 수 있다[21].

2.4 안드로이드 어플리케이션 보호 기법

2.4.1 안드로이드 난독화 기술

널리 쓰이는 자바 난독화 기술인 (1) ~ (5) 기술이 안드로이드에서도 쓰이고 있다. 본 논문에서는 기존에 사용되고 있는 클래스 암호화 기술 보다 더 효율적인 방법을 제안한다.

(1) 식별자 변환 : 클래스, 메소드 등의 이름을 의미 없는 이름으로 대체하는 기법이다.
(2) 제어 흐름 변환 : 클래스 파일 내부에 일부 코드의 실행 순서를 바꾸고 더미 코드를 삽입하는 기법이다.
(3) 문자열 암호화 : 소스 코드 내의 문자열 일부 또는 전부를 암호화된 문자열로 대체하는 기법이다.
(4) API 은닉 : 특정 라이브러리 또는 메소드 호출을 감추는 기법이다. 자바 리플렉션 활용이 필요하다.
(5) 클래스 암호화 : 특정 단일 클래스 또는 복수의 클래스가 포함된 파일 내용 전체를 암호화 하여 저장한 뒤 동적으로 복호화하고 클래스 로더로 로드하여 필요한 클래스를 실행하는 기법이다[7].

Table 1. Obfuscation techniques of tools[7]

	PG	DO	AT	DG
RN	O	O	O	O
CF	X	O	O	O
SE	X	O	O	O
AH	X	X	X	O
CE	X	X	X	O

RN=Renaming
CF=Control Flow
SE=String Encryption
AH=API Hiding
CE=Class Encryption
PG=Proguard
DO=DashOPro
AT=Allatori
DG=DexGuard

2.4.2 안드로이드 인텐트

안드로이드 환경에서는 서로 다른 어플리케이션의 상호 작용을 인텐트를 통해서 한다. 안드로이드 인텐트는 3가지로 나뉜다. 첫째, 명시적 인텐트는 같은 어플리케이션 내에 속해있는 컴포넌트의 이름을 입력하여 직접 메시지 객체를 전달한다. 둘째, 인텐트 필터에서 액션, 카테고리, 데이터를 참고하여 해당되는 컴포넌트에게 인텐트를 전달하는 암시적 인텐트가 있다. 마지막으로 특정 이벤트 발생 시점에 모든 어플리케이션에 인텐트를 전달하여 각 어플리케이션의 리시버가 이벤트를 수신할 수 있는 브로드캐스트 인텐트가 있다[22]. 안드로이드에서 인텐트 필터에 의한 컴포넌트 접근은 Manifest 설정으로 차단 또는 허용할 수 있다. 일반적으로 개발자들은 다른 어플리케이션의 접근을 막기 위해 컴포넌트 이름을 공개하지 않는다. 인텐트 필터에 의한 엑티비티 접근 과정은 아래와 같다[23].

(1) 액티비티 A는 StartActivity로 통과하는 액션을 포함한 인텐트를 생성한다.
(2) 안드로이드 시스템은 인텐트 필터를 이용하여 인텐트에 알맞은 모든 앱을 검색한다.
(3) 안드로이드 시스템은 알맞은 액티비티 B의 onCreate() 메소드를 호출하고 인텐트로 이것을 통과시킨다[23].

인텐트 필터를 통해 여러 어플리케이션이 발생시킨 인텐트 중 자신이 받을 인텐트만을 받아서 처리하도록 되어있다. 이러한 과정을 거치지 않고 다른 어플리케이션의 자원을 사용하고자 할 경우 CONTEXT_IGNORE_SECURITY 설정으로 인증서를 무시하여 보호 시스템 우회가 필요하다[24].

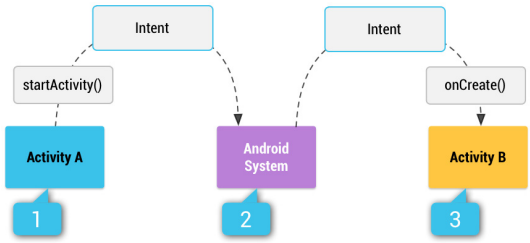

Fig. 2. Overview of Android Intent[23]

2.5 DES 알고리즘

본 논문에서는 Triple DES 알고리즘을 활용하여 DEX 파일을 암호화하였다. Triple DES는 2개의 키를 이용한 방식과 3개의 키를 이용한 방식이 있다. 본 논문에서는 3개의 키를 사용하였다.

2.5.1 DES 알고리즘

DES(DEA) 알고리즘은 암호화와 판독을 위한 64비트의 키로 고안되었다. 키를 활용하여 같은 크기의 평문을 암호문으로 바꾸는 대칭키 블록 암호 알고리즘이다. 키는 랜덤으로 생성된 암호화를 위한 56비트 문자열과 오류 탐지를 위한 8비트의 문자열로 이루어져 있다. 입력되는 평문을 초기 치환하고, Feistel 구조의 16라운드 연산을 수행한 후, 최종 치환을 수행하여 암호문이 생성된다. 또한, 암호화와 같은 구조로 수행되는 복호화 과정은 암호화에 사용했던 키를 반대 순서로 사용하여 진행된다. 각 라운드는 2개의 32비트 블록으로 나누고 왼쪽과 오른쪽으로 구분한다. 오른쪽 블록은 다음 라운드의 왼쪽 블록으로 이동되고 왼쪽 블록은 라운드 함수를 수행한 오른쪽 블록과 XOR된 결과 값이 다음 라운드의 오른쪽 블록으로 이동된다. 단, 마지막 라운드 이후에는 두 블록의 위치를 교환하지 않고 출력한다[9-11].

2.5.2 Triple DES 알고리즘

Single DES와 Double DES 알고리즘은 안정성이 낮다고 널리 알려져 있다. 따라서 Triple DES 알고리즘이 널리 사용되고 있다. Triple DES의 가장 큰 특징은 총 3번의 DES 연산으로 최종 암호문 값을 출력하고, 복호화도 3번의 연산을 거치는 것이다. 구체적으로 암호화는 DES를 이용하여 암호화, 복호화, 암호화 순서로 진행되며 복호화는 DES로 복호화, 암호화, 복호화 순서로 진행된다. 이 때 각 단계마다 다른 키를 사용하여 세개의 키를 사용하는 방식과 첫 번째와 세 번째 단계에서 같은 키를 사용하여 암호화함으로써 두 개의 키를 활용하는 방식이 있다[9-11].

III. 제안 기법

관련 연구 2.4.1에 소개된 것과 같이 기존의 안드로이드 어플리케이션 암호화 기법은 바이트코드에 중점을 두고 있다. 하지만 이러한 기법을 채택하면 암호화 및 복호화를 위한 알고리즘이 매우 복잡하고 구현이 어려우며 사용자가 체감할만한 실행 속도 저하가 발생할 수 있다. 반면, 제안 기법은 암호화하고자 하

는 앱을 압축해제 했을 때 루트 디렉토리의 모든 클래스가 저장된 DEX 파일을 추출하여 추출한 DEX 파일 자체를 암호화 및 복호화하기 때문에 복호화 시간을 제외하고는 실행 속도가 원본 앱과 거의 같으며 암호화하려는 코드를 포함한 원본 앱과 암호화된 코드를 바탕으로 실행되는 로더 앱, Manifest 정보를 포함한 클래스의 패키지 명이 같다는 가정 하에 DEX 파일 내부의 바이트코드 특성을 고려할 필요가 없다. 다만, 원본 앱의 실행 형태를 그대로 복원시켜 주어야 하기 때문에 자바 리플렉션을 활용하여 수동으로 프로세스에 클래스를 등록하고 원본 앱의 Manifest 정보에 기재된 행위가 로더 앱에서 진행되도록 만들기 위한 작업이 필요하다.

우선, 3.1절에서(그림 3 참고) Triple DES를 이용한 암호화 및 코드 은폐 기법을 다루며 3.2절에서(그림 4 참고) 자바 리플렉션과 클래스 로더를 이용한 복호화 및 임의 코드 수행 방법을 소개하고 3.3절에서 복호화 키 관리 방법의 안정성을 분석한다.

3.1 Triple DES를 이용한 암호화 및 코드 은폐 기법

이 절에서는 세 개의 키를 사용하는 Triple DES 및 동적 로딩을 이용한 코드 은폐 기법을 제안한다. 안드로이드 앱은 주로 마켓을 통해 배포되며 각 단말기 제조사에 의해 맞춤화된 다양한 운영체제 환경에서 구동된다. 본 연구에서는 제안 기법이 적용된 앱의 범용성을 마켓 등에서 자유롭게 배포할 수 있는 수준이 되도록 높이기 위하여, 암호화된 DEX 파일을 복호화하는 키를 얻는 절차를 서버와 연결하거나 사용자에게 키 값을 입력받는 등의 외부 입력 없이 독립적으로 수행하는 방법을 제안한다. 이 때문에 공격자가 리버싱 및 디버깅 등의 과정을 수행하여 키 값을 찾아낼 수 있는 단점이 존재한다. 따라서 본 연구에서는 공격자가 복호화 키를 가능한 한 추측하기 어렵게 하기 위하여 동적 키 생성을 포함하는 복잡한 복호화 키 생성 과정을 거친다. 하지만 복잡한 과정을 거쳐도 공격자가 무차별 대입(Brute Force) 이외의 다른 방법으로 키를 획득할 수 있는 여지는 남아있다. 이와 같은 공격 가능성을 줄이기 위해서 키를 서로 다른 장소에 다른 방식으로 저장하여 키 값의 유출 가능성을 최소화하였다. 제안 기법의 안정성은 3.3절에서 상세히 다룬다.

제안 기법은 그림 3의 flowchart와 같은 형태로 아래의 순서대로 원본 앱의 정보를 암호화한다.

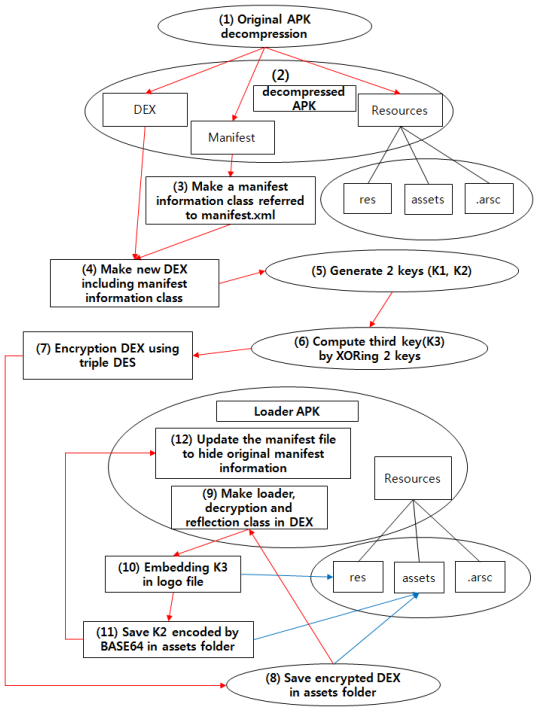

Fig. 3. Flowchart of DEX encryption process.

(1) APK 파일은 zip 압축 파일과 같은 형태를 가진다. 따라서 APK 파일은 zip 알고리즘으로 압축해제할 수 있다.

(2) 암호화 대상 APK 파일의 압축을 해제하면 classes.dex, AndroidManifest.xml과 그 밖의 리소스 파일을 얻을 수 있다.

(3) 이벤트 처리 정보를 은닉하기 위해 (2)의 과정에서 얻은 AndroidManifest.xml의 정보에서 리시버, 서비스 컴포넌트의 인텐트, 인텐트 필터 정보와 나머지 리시버 컴포넌트 정보를 추출하여 이를 포함한 class 파일을 만든다. 이렇게 만들어진 class 파일에 포함된 Manifest 정보는 이후 (8)의 과정에서 만들어지는 로더 앱의 AndroidManifest.xml에는 삭제된다.

(4) (3)의 과정에서 만들어진 class 파일을 (2)의 과정에서 획득한 classes.dex 파일에 포함시킨다.

(5) 임의의 64비트 길이를 가진 키 스트링을 두 개 K1, K2를 생성한다.

(6) 직전 과정에서 만들어진 K1과 K2를 XOR 연산하여 3번째 키 K3를 만든다.

(7) 높은 보안성을 위해 본 논문에서는 암호화 알고리즘으로 Triple DES를 사용한다. Triple DES

에는 56비트 키 3개가 필요하다. (5)~(6) 과정에서 얻은 3개의 키는 각 64비트이다. 공격자에게 혼동을 주기 위하여 키 길이를 임의로 늘렸으나 실제로는 56비트만 필요하다. 따라서 각 64비트의 키에서 필요한 상위 56비트를 잘라서 암복호화에 필요한 키를 얻는다. 그 다음에 (4)의 과정에서 만들어진 classes.dex 파일을 zip 형식으로 압축하고 Triple DES 알고리즘을 활용하여 암호화한다. 암호화된 파일은 분석가의 리버싱을 어렵게 하는 효과가 있다. 제작자의 판단에 따라 Triple DES 대신 AES 등의 다른 암호 알고리즘을 채택할 수도 있다.

(8) 암호화한 DEX 파일을 동적 로딩하여 원본 앱과 똑같이 실행할 수 있도록 만들어 주는 로더 앱을 만든다. 앞서 만들어진 암호화된 DEX 파일은 앱 내부 데이터 영역 assets 폴더에 저장된다.

(9) 로더 앱의 소스 코드인 DEX 파일은 3개의 클래스로 구성된다. assets 폴더 안에 있는 암호화한 DEX 파일을 복호화하는 역할을 하는 복호화 클래스가 있고, 복호화된 DEX 파일을 로드한 다음에 자바 리플렉션을 이용하여 원본 앱과 똑같이 로더 앱이 동작할 수 있도록 돕는 로더 클래스 그리고 자바 리플렉션을 사용할 수 있도록 해주는 리플렉션 클래스가 있다.

(10) 앱 내부 데이터 영역 res 폴더에 저장된 로고 이미지 파일에 K3 키를 임베딩(Embedding)한다.

(11) assets 폴더에 K2를 BASE64로 인코딩하여 저장한다. 남은 키는 동적으로 앞에서 얻은 2개의 키를 XOR하여 만들어지므로 파일로 저장되지 않는다.

(12) 로더 앱의 Manifest는 원본 앱의 동일한 파일에서 (3)의 과정에서 만들어진 클래스 내부에 저장된 정보가 삭제된 상태이다.

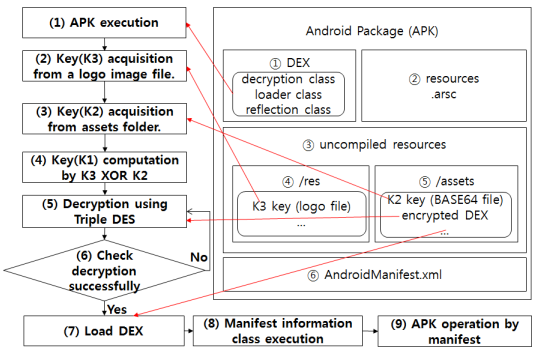

Fig. 4. Flowchart of runtime process.

이 절의 과정에 따라 생성된 APK 파일 실행 과정은 그림 4의 flowchart 및 3.2절과 같다.

3.2 자바 리플렉션과 클래스 로더를 이용한 복호화 및 임의 코드 수행 방법

기존 안드로이드 앱은 루트 디렉토리의 DEX 파일에 모든 클래스가 저장되어 있고 앱 실행 즉시 한 번에 로드된다. 반면에 본 논문에서 소개하는 방법을 활용하면 제안 기법이 적용된 앱은 원본 앱의 클래스 모두를 암호화한 파일을 포함하고 있기 때문에 암호화된 파일을 복호화하기 위한 코드가 담긴 DEX 파일의 클래스가 실행된 이후에 암호화된 파일을 복호화하고 그 다음에 원본 앱의 클래스가 원본 앱의 Manifest 정보와 동일한 과정을 거쳐 실행된다. 이러한 런타임 과정의 개요는 그림 4와 같다.

제안 기법이 적용된 앱은 자바 리플렉션 기법으로 안드로이드 시스템에 접근하여 어플리케이션 내부 리소스 영역(assets, res 폴더)인 assets 폴더에 저장된 제 3의 난독화된 DEX 파일을 로드하고 파일 내부의 클래스를 자유자재로 활용할 수 있다. 따라서 보안을 무시하는 설정(CONTEXT_IGNORE_SECURITY)[24]이 필요 없게 되며, 제안 기법을 활용하고자 하는 소프트웨어 개발자가 Manifest에 인텐트 필터 정보나 일부 컴포넌트 정보를 기재하지 않는 대신에 DEX 파일 내부의 클래스에 기재하여 어플리케이션 실행 정보 노출을 피할 수 있다. 이후 리플렉션 기법으로 시스템에 접근한 뒤 복호화된 클래스를 액티비티 스레드에 등록하면서 은닉했던 내용을 실행할 수 있다.

구체적인 실행 과정은 아래와 같다.

(1) 제안 기법이 적용된 APK 파일이 사용자 또는 시스템에 의해 실행된다.

(2) 실행 과정 (1)이후에 암호화된 DEX 파일을 복호화 하기 위해서 암호키를 획득하는 과정을 거친다. res 폴더 내부의 키가 임베딩(Embedding) 되어있는 이미지 파일에서 K3 키를 추출한다.

(3) assets 폴더에 BASE64로 암호화된 K2 키를 추출한다.

(4) 위의 과정에서 얻은 두 개의 키를 XOR 연산하여 K1 키를 얻는다. 나머지 키는 동적으로 획득함으로써 키 탈취 가능성을 낮춘다.

(5) K1, K2, K3는 각 64비트의 키로 앞 56비트를

잘라서 사용한다. 획득한 키로 난독화된 DEX 파일을 Triple DES 알고리즘으로 복호화하여 로더 앱의 assets 폴더에 저장하는 과정을 거친다.

(6) 복호화된 파일이 정상적이지 않을 경우에 자바 코드의 예외 핸들링 부분을 이용하여 성공적으로 복호화하는 과정을 거친다.

(7) 복호화가 성공했으면 복호화한 DEX 파일을 클래스 로더로 동적 로딩하는 과정을 진행한다.

(8) 원본 앱의 Manifest 정보와 똑같이 로더 앱이 실행되기 위하여 (7)의 과정이 성공하면 Manifest information 클래스 등록 절차가 필요하다. 자바 리플렉션 기법으로 프로세스에 등록하기 위해 필요한 클래스 등록 절차(관련 연구 2.1.2 참조)를 수작업으로 진행한다.

(9) (8)의 과정에서 Manifest information 클래스가 등록되면, 원본 앱의 Manifest와 같이 로더 앱이 동작하게 된다.

3.3 복호화 키 관리 방법의 안정성

3.1절과 3.2절에서 제안된 방법으로 임의의 앱을 암호화했을 경우에 공격자가 Triple DES 알고리즘으로 원본 DEX 파일을 암호화한 사실을 인지한다고 가정하면 크게 3가지 공격 형태가 예상된다:

(1) 공격자가 키 2개를 파악했을 경우: 공격자는 2개의 키를 XOR 연산으로 조합하거나 2^{55}회 대입하여 3개의 암호키를 모두 얻을 수 있다. 따라서 해커가 K2를 알아낼 확률을 P2, K3를 알아낼 확률을 P3라고 할 때 공격자가 암호를 해제할 확률은 아래와 같다. (P2와 P3는 $0 \le P2, P3 \le 1$)

K2, K3로 XOR 연산하여 K1을 알아낼 확률: $P2 \cdot P3$

K1을 Brute force 공격으로 알아낼 평균 확률 : $P2 \cdot P3 \cdot \frac{1}{2^{55}}$

그러므로 이 경우에 키가 모두 드러날 확률은 $P2 \cdot P3$ 또는 평균 $P2 \cdot P3 \cdot \frac{1}{2^{55}}$ 이다.

(2) 공격자가 1개의 키를 파악했을 경우: 공격자는 $2^{110}(2^{55} \cdot 2^{55})$회 대입하여 공격하거나 2^{55}회 대입하여 한 개의 키를 더 얻은 후에 두 키를 XOR 연산하면 암호화된 코드를 원본 코드로 복호화할 수 있다. 이 경우 해커가 K2, K3 중에서 1개의 키를 알아낼 확률을 P라 할 때 공격자가 암호를 해제할 확률은 아래와 같다. ($0 \le P \le 1$)

K2, K3 중에서 하나의 키를 얻었을 때 남은 하나를 Brute force 공격으로 얻고, K2과 K3를 XOR 연산하여 K1을 구하는 평균 확률 : $P \cdot \frac{1}{2^{55}}$

K2, K3 중에서 하나의 키를 얻었을 때 나머지 2개의 키를 모두 Brute force 공격으로 획득할 평균 확률 : $P \cdot \frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$

그러므로 이 경우에 키가 모두 드러날 평균 확률은 $P \cdot \frac{1}{2^{55}}$ 또는 $P \cdot \frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$ 이다.

(3) 공격자가 어떤 키도 알지 못할 경우: $2^{165}(2^{55} \cdot 2^{55} \cdot 2^{55})$회 대입하거나 $2^{110}(2^{55} \cdot 2^{55})$회 대입하여 2키를 얻은 후에 XOR로 나머지 키를 만들어 공격할 수 있다.

2개의 키를 Brute force 공격으로 획득한 후에 두 키를 XOR 연산으로 조합할 때 평균 확률 : $\frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$

3개의 키를 모두 Brute force 공격으로 획득할 평균 확률 : $\frac{1}{2^{55}} \cdot \frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$

그러므로 이 경우에 키가 모두 드러날 평균 확률은 $\frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$ 또는 $\frac{1}{2^{55}} \cdot \frac{1}{2^{55}} \cdot \frac{1}{2^{55}}$ 이다.

(1)~(3)을 요약하면, 공격자가 3개의 키 중 어떠한 키도 얻을 수 없다면 Brute force 공격으로 암호화된 DEX 파일을 복호화하기 어렵다. 하지만 앱 내부에 숨긴 키가 유출될 경우 Brute force 공격으로도 복호화될 가능성이 충분히 존재한다. 이와 같은 공격 가능성을 줄이기 위하여 외부 입력 받은 값을 바탕으로 Key를 생성하여 Triple DES나 AES 알고리즘을 적용하면 해당 알고리즘이 공격당할 확률에 제안 기법이 적용된 앱이 공격당할 확률이 거의 수렴한다. 따라서 본 연구에서 가정한 것처럼 마켓에서 앱을 배포하여야하는 등 범용성이 필요한 경우가 아니라면 외부에서 키 입력을 받는 것을 권장한다.

IV. 실험 결과

4.1 프로토타입 구현

실험을 위해 안드로이드 버전 4.4.2(Kitkat)의

API 19 SDK를 이용하여 APK 파일을 제작하였다. 우선 Manifest의 리시버 컴포넌트 기재로 사용자 이벤트 발생(USER_PRESET)시 특정 메시지가 출력되는 간단한 앱(앱1)을 제작하고 DEX 파일을 추출하였다. 추출된 DEX 파일을 apktool[18]을 이용하여 분해한 뒤 기존 Manifest에 기재했던 리시버와 같은 실행 정보를 담은 별도의 클래스를 저장하였다. 저장한 DEX 파일을 zip 형식으로 압축한 뒤 DES 알고리즘을 활용하여 암호화하였다. 그 후 앱1의 코드를 집어넣을 앱(앱2)를 제작하였다. 실험을 위해 만든 APK 파일의 Manifest에는 은닉된 액티비티 정보 등을 기재하였다. 레이아웃 요소는 은폐된 클래스를 로드할 APK 파일이 담고 있어야 한다. 실행 후 로드될 앱1의 코드 중에서 앱2와 중복된 클래스는 공격자에게 숨기고자 하는 코드이므로 앱2에서는 빌드 정보만 남기고 제거하였다.

이 앱2는 안드로이드의 getAssets API를 이용하여 암호화된 파일을 로드하고 복호화 루틴을 담은 메소드를 이용하여 복호화한다. 이 과정에서 복호화 된 데이터는 배열에 저장한 뒤 파일로 생성한다.

이후 loadDex[8]로 복호화된 데이터를 불러오고 실패한 경우에는 예외 처리에 의해 재시도를 한다. 클래스 로더로 은폐했던 클래스를 불러오고 나서 리플렉션으로 은닉되었던 컴포넌트들을 실행시켜줄 클래스를 스레드에 기재 후 실행한다.

구현한 APK 파일 실행으로 패키지 내부에 존재하지 않았던 액티비티가 생성되어 내부의 텍스트가 출력됨을 확인하였다. 또한 Manifest에 리시버 컴포넌트를 기재하는 대신 별도의 클래스 파일 추가로 대체한 것이 정상적으로 동작하는 것을 증명하는 안드로이드 토스트(Toast)[25] 메시지 박스가 출력됨을 확인할 수 있었다.

Table 2. System Environment

List	Version
Android	4.4.2 (Kitkat)
SDK	API 19

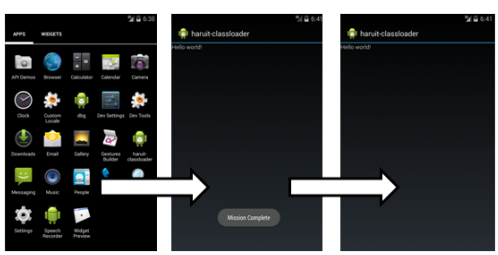

Fig. 5. Prototype implement

앱1과 앱2의 실행 속도 차이를 알아보기 위하여 앱2의 실행 직후와 암호화된 DEX 파일 로드 직후 시점의 시간을 측정한 뒤 그 차이를 구하였다. 실험은 삼성전자 SKT용 갤럭시 팝(모델명 : SHV-220S, AP : 엑시노스 4412 쿼드코어 1.4GHz, 메모리 : 2GB DDR2) 단말기를 활용하여 100회 실시되었다. 그 결과는 표 3과 같다.

앱2에서 암호화된 DEX 파일을 실행한 이후에는 앱1을 곧바로 실행한 것과 거의 동일한 방법으로 앱이 작동한다. 따라서 원본 앱과 제안 기법을 적용한 앱은 실행 속도 측면에서만 의미 있는 차이가 있을 것으로 예상된다. 표13을 볼 때 원본 앱과 제안 기법을 적용한 앱의 실행 속도 차이가 2초 미만이고 최근 국내에 시판중인 단말기 대부분이 실험한 단말기 보다 빠른 AP를 탑재하고 있으므로 제안 기법이 적용된 앱을 사용하는 사용자가 느끼는 불편은 거의 없을 것으로 예상된다.

Table 3. Figures of average, maximum, minimum about DEX loading measured time.

Average	Maximum	Minimum
1.863 s	1.945 s	1.799 s

4.2 악성코드 은폐로 활용될 위험성 관련 실험

악성코드의 은폐 기능을 실험하기 위해 7개의 악성 앱 및 3개의 정상 앱에 제안 기술을 적용하여 (표 4, 5) 난독화한 뒤 바이러스 토탈에서 진단하였다. 사용된 악성코드는 공통적으로 사용자에게는 보이지 않는 백그라운드에서 악의적인 행위를 하는 기능이 포함되어 있다.

아래의 그림 6와 그림 7은 실험 악성 앱 순번 1(진단명 Android-Malicious/DougaLeaker)의 원본 앱과 이 앱에 제안 기법을 적용한 앱을 바이러스 토탈에서 진단한 결과이다.

순번 1 악성 앱의 경우 원본 앱은 54개의 백신 중 42개의 백신에서 진단된 것과 달리 난독화 기술을 적용한 앱은 3개의 백신 프로그램에서 진단되었다. 표 6는 난독화된 앱을 진단한 3개 백신의 진단명을 나열한 것이다. 3가지 백신 모두 원본 악성코드의 진단명과

Table 4. Malicious apps list using concealment examination

No.	Diagnosis(AhnLab-V3) & MD5
1	Android-Malicious/DougaLeaker 00e74c118fa3902e5c85fd8e37f3d084
2	Android-Malicious/HijackBank 0b7212244f7d9d25a2d8af73de009b31
3	Android-Malicious/PNStealer 7e8d8b7ad55e6f8771025116eb9c5aa1
4	Android-Malicious/SMStealer 2d626744507735028858df9d5305610e8
5	Android-Malicious/MsgIntercept 66c49214650ced56a6dc5bdba543f55e
6	Android-Malicious/Narut 9fccb87a8f1d7480d7a73c221ffdfdc2
7	Android-Malicious/SMStealer f9cf4ab581ec929b3f456823d92d7a08

Table 5. Commercial apps list used by concealment examination

No.	Application & MD5
1	AndroZip 4.7.1 24bdd3f81e38c5b5f6319726415334e0
2	Opera Max 3eb17a2a208e15039442260bfb870cf9
3	HackRun 1.3 7adc62598d5588e7be8d698f24cc224c

Fig. 6. Diagnosis results of original malware 1

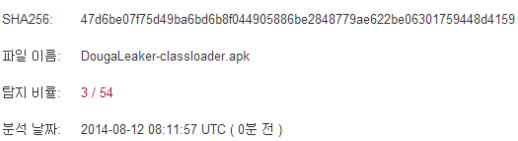

Fig. 7. Diagnosis results of app concealed DEX about malware 1

다른 진단명으로 진단하는 것을 알 수 있다. 순번 1을 포함한 실험 앱 10개에 제안 기법을 적용하고 바이러스 토탈에서 진단한 결과 모두 같은 진단 결과가 나왔다.

10개의 앱(악성 앱 7개, 상용 앱 3개)을 바이러스 토탈에서 백신 프로그램으로 진단한 결과를 표로 정리하였다. True Positive Rate(TPR)는 악성 앱을 악성이라 판단한 비율(Detection rate)이고 True Negative Rate(TNR)는 정상 앱을 정상이라 진단한 비율이다. False Positive Rate(FPR)는 정상을 악성으로 오탐한 경우이며 False Negative Rate(FNR)는 악성을 정상으로 미탐한 경우이다.

Table 6. detection results of diagnostic anti-virus Software both original apps and concealed apps

Anti-Virus software	Original app diagnosis Concealed app diagnosis
NANO-AV	Trojan.Android.Douga.bdokvn Trojan.Android.Agent.dcspod
Avast	Android:Dougalek-B [Trj] Android:Agent-DSU [Trj]
DrWeb	Android.Douga.1.origin Android.MulDrop.19.origin

표 7과 8은 실험 앱 원본의 탐지 결과이고, 표 9와 10은 제안 기법을 적용한 악성 앱과 정상 앱의 탐지 결과이다.

표 9과 표 10의 진단 결과를 보면 일부 백신이 난독화한 앱을 진단하는 것을 확인할 수 있다. 그러나 그림 8과 같이 악성 바이트코드를 암호화한 파일은 진단하지 못했다. 표 6과 같이 난독화된 앱을 악성으로 진단한 3개의 백신 진단명은 Agent 또는 Drop이다. 그림 8과 같이 난독화된 바이트코드 파일은 진단하지 못하고 실행 앱만을 Agent 또는 Drop 진단명으로 진단하는 점을 감안하면 백신 프로그램 엔진이 휴리스틱 기능으로 제 3의 DEX 파일을 로드하는 부분만을 진단하는 것으로 추정된다.

안드로가드[12]에는 유사도 비교도구인 안드로심이 포함되어 있다. 안드로심은 2개의 앱을 입력 받아

Table 7. Detection rate and figures of original malicious apps in 54 anti-virus

No.	True Positive Rate	False Negative Rate
1	77.8% (42/54)	22.2% (12/54)
2	55.6% (30/54)	44.4% (24/54)
3	16.7% (9/54)	83.3% (45/54)
4	63.0% (34/54)	37.0% (20/54)
5	59.3% (32/54)	40.7% (22/54)
6	46.3% (25/54)	53.7% (29/54)
7	50.0% (27/54)	50.0% (27/54)
Avg.	52.6%	47.4%

Table 8. Detection rate and figures of original commercial apps in 54 anti-virus

No.	True Negative Rate	False Positive Rate
1	100% (54/54)	0% (0/54)
2	100% (54/54)	0% (0/54)
3	100% (54/54)	0% (0/54)
Avg.	100%	0%

Table 9. Detection rate and figures of concealed malicious apps in 54 anti-virus

No.	True Positive Rate	False Negative Rate
1	5.6% (3/54)	94.4% (51/54)
2	5.6% (3/54)	94.4% (51/54)
3	5.6% (3/54)	94.4% (51/54)
4	5.6% (3/54)	94.4% (51/54)
5	5.6% (3/54)	94.4% (51/54)
6	5.6% (3/54)	94.4% (51/54)
7	5.6% (3/54)	94.4% (51/54)
Avg.	5.6%	94.4%

Table 10. Detection rate and figures of concealed commercial apps in 54 anti-virus

No.	True Negative Rate	False Positive Rate
1	94.4% (51/54)	5.6% (3/54)
2	94.4% (51/54)	5.6% (3/54)
3	94.4% (51/54)	5.6% (3/54)
Avg.	94.4%	5.6%

Fig. 8. Detection results of obfuscated bytecode about malware 1

앱의 DEX 파일을 추출한 뒤 앱 내부의 개별 메소드에 대해 상대 앱의 개별 메소드와 비교하는 방법으로 유사도를 비교한다. 비교한 데이터를 바탕으로 아래 식과 같은 방법으로 유사도를 산출하여 완전 다른 코드를 0%로하고 완전히 같은 코드를 100%로 결과를 도출한다. 아래 식에서 *비유사도*는 안드로심이 유사하다고 판단하는 메소드의 *비유사도*만을 의미한다.

$$\frac{1-(\sum_{1}^{i}\text{비유사도}+0*\sum_{1}^{j}\text{동일 메소드 수}+\sum_{1}^{k}\text{추가된 메소드 수})}{\text{총 메소드 수}+\text{추가된 메소드 수}-\text{없는 메소드 수}} * 100$$

안드로가드 1.9에 포함된 안드로심 도구를 이용하여 앞서 실험한 10개의 악성 및 정상 앱을 대상으로 원본 앱과 난독화된 앱의 유사도를 비교하였다. 그 결과를 표 11에 나열하였고 최대, 최소 그리고 평균으로 정리하여 표 12에 표기하였다.

안드로심은 각 앱의 DEX 파일을 추출하여 DEX에 포함된 클래스 내의 메소드를 대상으로 분석을 할 뿐 메소드의 실행 순서를 고려하지 않는다. 이 때문에 어느 정도 유사한 메소드가 있으면 전혀 다른 기능을 하더라도 유사도 산정에 포함시켜서 실제보다 유사도가 높게 산출되는 문제점이 있다. 그 밖에도 악성 앱에 비해 상용 앱의 유사도가 상당히 높은 이유는 안드로이드 라이브러리 중 하나인 Android Support Library[26] 때문으로 추정된다. 이 라이브러리는 여러 버전의 안드로이드 호환과 관련된 기능을 담당한다. 앱의 실행 성능을 올리기 위해 상용 앱과 원본 앱을 난독화한 앱에는 Support Library가 사용되었으나 실험에 사용된 악성 앱들은 이 라이브러리가 사용

Table 11. Results of test app similarity evaluation

Target	Similarity
Malware 1	0.358744 %
Malware 2	1.505498 %
Malware 3	19.790074 %
Malware 4	0.224215 %
Malware 5	13.619469 %
Malware 6	1.287516 %
Malware 7	2.031199 %
Commercial app 1	32.970870 %
Commercial app 2	25.553674 %
Commercial app 3	30.844352 %

Table 12. Figures of average, maximum, minimum about table 10 data.

Average	Maximum	Minimum
12.818561 %	32.970870 %	0.224215 %

되지 않았다.

실험 결과를 보면 백신 프로그램은 본 연구와 같은 방법으로 악성코드를 은폐시켰을 경우 악성코드를 정확하게 진단하지 못한다. 이에 따라 기존의 진단 방법으로는 정상 앱과 악성 앱에 대한 구분이 어려워 악성 앱 진단에 상당한 어려움을 겪을 것으로 예상된다. 다만 안드로이드 앱의 사인을 검사하여 특정 사인에 대해 예외 처리를 하면 백신 프로그램이 진단하지 않도록 할 수도 있다.

4.3 코드 은닉 관점에서 기존 도구와의 비교

제안 기법은 식별자 변환, 제어 흐름 변환, 문자열 암호화, API 은닉, 클래스 암호화 등의 기존 바이트코드 변환 형태의 난독화 기술과 달리 바이트코드가 기록된 파일 전체를 암호화한 기술이다. 즉, 기존의 바이트코드 변환 형태의 기술을 이미 적용한 상태에서 제안 기법의 활용이 가능하며 권장한다. 유사한 기술이 적용된 DexGuard에서는 동적으로 암호화 된 파일을 생성하는 과정을 수행한 뒤에 복호화 및 로드를 하는 과정을 진행하였으나 본 논문에서는 동적으로 암호화 된 파일을 생성하는 과정이 없이 코드를 난독화하였으며, 모든 DEX 파일 내부의 클래스를 암호화하고 복호화 루틴만 남김으로써 DEX 파일 구조 파악을 어렵게 하였다. 또한, DexGuard에서는 암호키를 동적으로 생성되는 파일에 포함시켜서 활용하지만 본 논문에서는 생성된 키를 메모리상에서만 저장 및 활용하여 유출 가능성을 줄였다.

Table 13. A comparative table of obfuscation techniques between existing tools and our scheme[7].

	PG	DO	AT	DG	**OS**
RN	O	O	O	O	**X**
CF	X	O	O	O	**X**
SE	X	O	O	O	**X**
AH	X	X	X	O	**X**
CE	X	X	X	O	**O**

RN=Renaming PG=Proguard
CF=Control Flow DO=DashOPro
SE=String Encryption AT=Allatori
AH=API Hiding DG=DexGuard
CE=Class Encryption **OS=Our Scheme**

V. 결 론

제안 기법은 제 3의 앱 내부 DEX 파일 전체를 난독화하고 일반적으로 Manifest에 기재되는 리시버와 인텐트, 인텐트 필터 정보 등을 난독화된 DEX 파일 내부에 포함시켜 앱의 내부 정보를 대부분 은닉하였고, 이러한 상태에서 실행 앱이 난독화된 제 3의 앱을 로드 시킨 뒤 구동할 수 있다. 이런 특징을 이용해서 제 3의 앱 내부 DEX 파일을 난독화한 후에 난독화된 파일을 loadDex API를 활용하여 동적로딩 시키고 자바 리플렉션 기법을 통해 로딩된 DEX의 클래스를 활용하였다. 이 연구 결과는 개발자가 난독화하고 싶은 앱을 난독화 한 뒤에 복호화 앱을 통해 정상적으로 구동할 수 있음을 의미한다. 본 논문에서는 이러한 과정을 단일 앱 내부에서 가능할 수 있도록 구현했다.

이 방법을 이용하여 악성 앱을 난독화하고 바이러스 토탈에서 검사하여 백신 프로그램이 정확하게 진단하지 못하는 것을 확인하였다. 일부 백신에서 실험 앱을 휴리스틱과 관련된 진단명으로 진단하였으나 정상 실험 앱과 악성 실험 앱 모두에 동일한 진단 결과를 보인다. 백신 프로그램의 특성상 본 기술이 정상 앱에서 활용될 경우 정상 앱의 진단을 피해야 하므로 진단에 상당한 어려움을 겪을 것으로 추정된다,

이처럼 실험을 통해 본 기술을 적용할 경우 저작권 보호에 유리하지만 악성코드에 활용될 경우 피해 예방이 매우 어려울 것으로 보인다.

References

[1] STRATEGY ANALYTICS, http://www.strategyanalytics.com/

[2] Joonhyouck Jang, Seunghwan Han, Yookun Cho, U jin Choe and Jiman Hong, “Survey of Security Threats and Countermeasures on Android Environment,” *Journal of Security Engineering*, Vol.11 No.1, pp. 01-12, Feb. 2014.

[3] Alexandrina KOVACHEVA, ″Efficient Code Obfuscation for Android,″ *Advances in information Technology's Communications in Computer and Information Science*, Vol. 409, pp.104-119, Aug. 2013.

[4] Patrick Schulz, "Code Protection in Android," *Institute of Computer Science 4 Communication and Distributed Systems in Bonn University*, June 2012.
[5] Hao Hao, Vicky Singh, and Wenliang Du, "On the effectiveness of API-level access control using bytecode rewriting in Android," *ASIA CCS '13 Proceedings of the 8th ACM SIGSAC symposium on Information of computer and communications security*, pp. 25-36, 2013.
[6] W. Zhou, Y. Zhon, X. Jiang and P. Ning, "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces," *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, pp. 317-326, Feb. 2012.
[7] Yuxue Piao and Jin-hyuk Jung and Jeong Hyun Yi, "Structural and Functional Analyses of ProGuard Obfuscation Tool," *Networks and Services*, Vol. 38B. No. 08, pp. 654-661, Aug. 2013.
[8] DexFile, http://developer.android.com/reference/dalvik/system/DexFile.html
[9] William M. Daley and Raymond G. Kammer, "DATA ENCRYPTION STANDARD (DES)," *FIPS PUB 46-3*, Oct. 1999.
[10] William C. Barker and Elaine Barker, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher," *NIST Special Publication 800-67*, Jan. 2012.
[11] Doo-Sik Choi, Doo-Hwan Oh, Jeong-Soo Park and Jae-Cheol Ha, "An Improved Round Reduction Attack on Triple DES Using Fault Injection in Loop Statement," *Journal of The Korea Institute of Information Security & Cryptology*, Vol. 22. No. 4, pp. 709-717, Aug. 2012.
[12] Androguard, https://code.google.com/p/androguard/
[13] Virus Total, http://virustotal.com
[14] Building and Running, http://developer.android.com/tools/building/index.html
[15] Alessandro Armando, Alessio Merlo. Mauro Migliardi and Luca Verderame, "Breaking and Fixing the Android Launching Flow," *Computers & Security*, Vol. 39, pp. 104-115, Nov. 2013.
[16] dex2jar, https://code.google.com/p/dex2jar/
[17] JD-GUI, http://jd.benow.ca/
[18] android-apktool, https://code.google.com/p/android-apktool/
[19] Trail: The Reflection API (The Java™ Tutorials), http://docs.oracle.com/javase/tutorial/reflect/
[20] V. Benjarmin Livshits and Monica S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proceedings of the 14th USENIX Security*, Aug. 2005.
[21] MARIUS POPA, "Analysis of Zero-Day Vulnerabilities in Java," *Journal of Mobile, Embedded and Distributed Systems*, Vol. 5. No. 3, pp. 108-117, Sep. 2013.
[22] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner, "Analyzing Inter-Application Communication in Android," *MobiSys '11 Proceedings*, Vol. 9, pp. 239-252, 2011.
[23] Intents and Intent Filters, http://developer.android.com/guide/components/intents-filters.html#iobjs
[24] CONTEXT_IGNORE_SECURITY, http://developer.android.com/reference/android/content/Context.html
[25] Toast, http://developer.android.com/reference/android/widget/Toast.html
[26] Support Library, http://developer.android.com/tools/support-library/index.html

〈저자소개〉

김 지 윤 (Jiyun Kim) 학생회원
2013년 2월: 한양대학교 자원환경공학과 졸업
2014년 3월~현재: 한양대학교 컴퓨터소프트웨어학과 석사과정
〈관심분야〉 컴퓨터시스템, 모바일시스템, 네트워크 보안, 웹 보안, 금융 보안, 보안 정책

고 남 현 (Namhyeon Go) 학생회원
2015년 2월: 한국폴리텍2대학 컴퓨터정보과 졸업
〈관심분야〉 인터넷 보안, 컴퓨터 보안, 임베디드 보안, 금융 보안, 보안 정책

박 용 수 (Yongsu Park) 종신회원
1996년: KAIST 전산학과 졸업(학사)
1998년: 서울대학교 대학원 컴퓨터공학과 졸업(석사)
2003년: 서울대학교 대학원 전기컴퓨터공학부 졸업(박사)
2003년~2004년: 서울대학교 자동제어특화연구센터 연수연구원
2005년~현재: 한양대학교 소프트웨어전공 부교수
〈관심분야〉 컴퓨터 보안, 인터넷 보안