Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.6, Dec. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.6.1553

패킹된 안드로이드 어플리케이션의 안전한 실행을 위한 실행 환경 무결성 검증 기법*

하 동 수,† 오 희 국‡
한양대학교

A Method Verifying Execution Environment Integrity for Secure Execution of Packed Android Application*

Dongsoo Ha,† Heekuck Oh‡
Hanyang University

요 약

안드로이드의 소스 코드는 공개되어 있어 목적에 따라 수정이 용이하다. 최근 이런 점을 악용하여 런타임 보호 기법을 우회하는 원본 실행 코드 탈취 기법이 등장하였다. 불행하게도, 안드로이드 디바이스는 파편화가 심해 시스템의 무결성을 검증하기가 어렵다. 이 문제를 해결하기 위해, 본 논문에서는 어플리케이션 권한의 특징을 이용하여 간접적으로 실행 환경의 무결성을 검증하는 기법을 제안한다. 우리의 기법은 원본 실행 코드를 실행하기 전 더미 실행 코드를 로드 및 실행하여 비정상적인 이벤트를 모니터링하며, 이를 통해 무결성 여부를 판단한다. 우리의 기법은 현재까지 공개된 우회 기법을 탐지할 수 있으며, 평균적으로 약 2초 내외의 성능 오버헤드를 가진다.

ABSTRACT

The source code for Android is open and easy to modify depending on the purpose. Recently, this charateristic has been exploited to bypass the runtime protection technique and extract the original executable code. Unfortunately, Android devices are so fragmented that it is difficult to verify the integrity of the system. To solve this problem, this paper proposes a technique to verify the integrity of the execution environment indirectly using the features of the application permission. Before executing the original executable code, it loads and executes the dummy DEX file to monitor for abnormal events and determine whether the system is intact. The proposed technique shows a performance overhead of about 2 seconds and shows that it can detect the bypassing technique that is currently disclosed.

Keywords: Android, Packer, Execution environment, Integrity verification, Runtime packer

I. 서 론

안드로이드(Android)는 다양한 기기와 폼 팩터에 사용될 수 있도록 제작된 리눅스 기반의 운영체제로, 자바 기반으로 프레임워크가 구성되어 있다. 이러한 특징으로 인하여, 서드 파티는 어플리케이션을 개발할 때 디바이스의 아키텍처를 크게 고려하지 않아도 되며 제공되는 자바 API를 이용하여 호환성 높은 어플리케이션을 개발할 수 있다. 최근 10년간 안드로이드는 오픈 소스로 인한 이점과 높은 호환성

Received(09. 10. 2018), Accepted(10. 10. 2018)
* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구센터육성 지원사업의 연구결과로 수행되었음 (IITP-2018-2014-0-00636). 또한, 2015년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NRF-2015R1D1A1A09058200).
† 주저자, hds@hanyang.ac.kr
‡ 교신저자, hkoh@hanyang.ac.kr(Corresponding author)

을 통해 빠르게 성장할 수 있었다.

바이트코드의 높은 호환성은 역공학에 취약하다. 즉, 바이트코드는 네이티브코드와 비교하여 더 추상적이고 이해하기 쉬운 구조이다. 안드로이드 어플리케이션은 덱스 (DEX, Dalvik Executable) 포맷[1]의 바이트코드로 구성되며, 이 포맷은 공개되어 있다. 따라서 사람이 읽을 수 있는 형태로 쉽게 변환될 수 있으며 (예로, DexDump, APKTool, Dex2Jar 등[2-4]), 약간의 문법만 익힌다면 쉽게 어플리케이션의 의미를 쉽게 파악할 수 있다. 이뿐만 아니라 APKTool과 같은 도구를 이용할 경우 수정 후 재배포도 가능하다.

바이트코드로 구성된 어플리케이션을 보호하기 위해, 구글은 시스템 API를 제외한 사용자 정의 클래스, 필드, 변수 등의 문자열을 컴파일할 시에 임의의 문자열로 변환해 주는 문자열 난독화 도구(Proguard)를 제공한다. 하지만, 여전히 분석이 쉬우며. 최근 난독화된 문자열을 머신러닝을 통해 복원하는 기법[5]을 사용할 경우 많이 무력화시킬 수 있다. 다른 접근법으로 덱스 파일의 포맷과 안드로이드에서 이를 디코딩하는 방식의 차이를 이용하여, 올바르게 바이트코드를 디코딩할 수 없도록 하는 방식도 고안되었다[6]. 하지만, 이런 트릭들은 쉽게 파훼 되며, 역공학을 원천적으로 막지는 못한다.

우리의 조사에 따르면, 현재 가장 많이 사용되는 방식은 패킹 기법(packer)으로, 런타임 실행 압축을 이용한 방식이다[7]. 안드로이드는 자바 기반이지만, 높은 연산을 요구하는 작업들을 네이티브코드로 처리할 수 있게끔 JNI(JAVA native interface)를 제공한다. 패킹 기법은 이를 이용하여 보호 및 복호화 루틴을 네이티브코드로 작성한다. 즉, 네이티브코드의 복잡성에 안전성을 두게 만든다. 또한, 안드로이드는 리눅스 기반이므로 각각의 어플리케이션이 독립적으로 샌드박싱되어 있는 점을 제외하고는 리눅스의 시스템 함수 등을 그대로 사용할 수 있다. 패킹 기법은 ptrace, inotify 등의 시스템 함수를 이용하여 런타임 보호 기법을 구현한다. 결과적으로, 패킹된 어플리케이션을 역공학하기 위해서는 네이티브코드를 분석해야 한다.

안드로이드의 소스 코드는 공개되어 있어 공격자는 시스템을 쉽게 수정할 수 있다. 최근 이를 통해 패킹 기법을 우회하는 기법이 등장하였다[9,16,17]. 2015년 안드로이드 달빅 머신의 수정을 통하여, 런타임 시에 클래스 로더가 로드하는 클래스 정보를 수집하는 기법[16]이 발표되었다. 이를 통해 공격자는 패킹 기법의 보호 기법을 직접 무력화하지 않고도, 단순히 어플리케이션의 모든 동작을 수행하는 것만으로 원본 덱스에 포함된 클래스 정보를 수집할 수 있다. 이와 유사하게 2016, 2017년에도 안드로이드 시스템의 일부를 수정하여 런타임 보호 기법을 우회하여 원본 덱스를 추출하는 방식이 등장하였다[9,17].

Fig. 1. Android device fragmentation (2015.8)

따라서 실행 환경의 무결성을 보장하지 못한다면 런타임 보호 기법 자체가 완전히 무력화되며, 원본 덱스가 손쉽게 노출되는 상황이 불가피하다. 2017년도 BlackHat USA의 브리핑에 따르면[9], Baidu, Bangcle, Tencent, Ali, 360 Jiagu 등과 같은 유명 패커 기법에서 실행 환경의 무결성이 올바르게 이뤄지지 않음이 보고되었다. 그리고 실행 환경 수정을 통해 런타임 보호 기법을 우회하여 원본 덱스를 쉽게 탈취하는 데모가 시연되었다. 우리가 알고 있는 한, 아직 안전한 언패킹 동작 환경을 위해 실행 환경의 무결성을 검증하는 기법은 없다.

불행하게도, 안드로이드 실행 환경의 무결성을 검증하는 것은 어렵다. 무결성을 검증하기 위해서는 검증 기준이 필요하지만, 여러 제조사를 통해 다양한 디바이스와 빌드 버전이 존재하므로 비교 대상이 많다. 더군다나 제조사에 따라 실행 환경이 수정되기 때문에 더욱 어렵다. 오픈 시그널의 조사에 따르면, 2013년 8월 11,868대, 2014년 8월 18,796대, 2015년 8월 24,000여대 이상의 안드로이드 디바이스가 존재하는 것으로 발표되었으며[8], 이 추세로 보아 현재에는 5만여대 이상이 존재할 것으로 예상한다(Fig. 1. 참고). 따라서, 안드로이드 디바이스에서는 비교 대상을 통한 무결성 검증은 어려울 것으로 판단된다.

이 문제를 해결하기 위해, 본 논문에서는 안드로이드 어플리케이션이 실행되는 환경의 특징을 이용하여 무결성을 검증하는 기법을 제안한다. 실행 환경의 무결성을 검증하기 위한 다양한 요소들이 있으며, 우리는 그중에 안드로이드 런타임의 수정을 통하여 런타임 보호 기법을 우회하고 덱스를 추출하는 것을 탐지하는 기법에 관해 다룬다. 따라서, 루팅된 환경에서의 동작, 주요 함수의 후킹, 이벤트 무력화 등의 무결성은 기존의 다른 기법으로 검증된 상태라 가정한다.

제안하는 기법은 시스템의 무결성을 직접 확인하는 방식이 아닌 간접적으로 유추하는 기법으로, 공격자가 원본 덱스를 탈취할 때 발생하는 이벤트를 확인함으로써 실행 환경의 무결성 여부를 판단한다. 또한, 이를 보완하는 방안으로, 주요 시스템 파일의 해시 값을 통해 검증하는 화이트 리스트 기법을 추가로 제안한다.

남은 논문의 구성은 다음과 같다. 2장에서 무결성 검증에 관한 연구와 우회 기법을 소개한다. 그리고 3장에서는 안드로이드 어플리케이션과 패커의 기본 구조를 다룬다. 4장에서는 실행 환경의 무결성을 검증하는 기법 두 가지를 제안한다. 그리고 5장에서는 제안한 기법의 성능 오버헤드를 평가한다. 6장에서는 제안하는 기법의 한계를 다루며, 7장에서 결론을 지으며 마무리 짓는다.

II. 관련 연구

본 장에서는 먼저 어플리케이션 및 시스템 무결성에 관한 연구를 다루며, 이어서 언패킹 과정의 특징을 이용한 우회 기법에 대해 다루도록 한다.

2.1 무결성 검증에 관한 연구

바이트코드 기반으로 구성된 안드로이드 어플리케이션의 특징을 이용하여 역공학을 수행할 때, 어플리케이션의 실행 흐름을 수정하여 각 지점의 수행 정보를 수집하거나 보호 모듈을 분리하는 식의 접근법이 많이 이용된다. 이 문제를 해결하기 위해 앱 자체의 무결성을 검증하는 연구가 많이 이뤄졌었다. 국내 연구로 2014년 심형준 등의 연구에서는 안드로이드 매니패스트에 기록된 서명을 통해 원본 덱스의 무결성을 검증하는 기법을 제안하였다[10]. 같은 해 이광형 등의 연구에서는 제3기관을 통해 서명된 원본 덱스을 런타임 시에 전송받아 실행하는 기법을 소개하였다[11]. 이 밖에도 어플리케이션의 무결성 및 리패키징 여부를 확인하는 기법들은 국내외 특허를 통해서도 많이 공개되었다[12-14].

언패킹 과정에서 실행 환경의 안전을 보장하기 위해 무결성 검증을 수행하는 연구는, 우리가 아는 한 아직 없는 것으로 보인다. 이는 2017년 BlackHat USA의 브리핑[9]을 통해, 간접적으로 확인할 수 있다. 이 시연에서 발표자는 DEX2OAT 컴파일러가 사용하는 안드로이드 런타임 공유라이브러리를 수정하여, 원본 덱스의 파일 디스크립터를 통해 원본을 파일 시스템으로 추출한다. Baidu, Bangcle, Tencent, Ali, 360 Jiagu 등과 같은 유명 상용 패커가 시스템 무결성 검증에 실패하였다. 또한 DexHunter[16], CrackMe[17] 등과 같이 안드로이드 런타임을 수정하여 덱스를 추출하는 기법과 관련된 연구(참조 문헌 포함)를 조사하였지만, 언패킹 과정에서 실행 환경의 무결성을 검증하는 연구는 발견하지 못하였다.

2.2 실행 환경을 이용한 우회 기법

2015년부터 패킹된 안드로이드 어플리케이션을 역공학 하지 않고 우회를 통해 원본 덱스를 추출하는 기법이 많이 공개되었다. 2015년 3월 박영웅의 브리핑[15]에서 덱스클래스로더의 특징을 이용하여 원본 덱스를 추출하는 기법이 소개되었다. 언패킹 과정에서 원본 어플리케이션의 실행 흐름으로 전환하기 위해서는 복호화된 원본 덱스의 덱스클래스로더를 생성해야 한다. 그리고 덱스클래스로더가 생성될 때, 원본 덱스가 컴파일되어 캐시 폴더에 저장되며 그것을 로드하여 최종적으로 객체가 생성된다. 따라서 이를 이용하여 덱스클래스로더 객체가 생성되는 과정에서 파일 시스템에 떨어지는 컴파일(또는 최적화)된 덱스를 수집하기만 하면 컴파일된 덱스를 추출할 수 있다. 또한, 컴파일된 덱스는 원본 덱스로 쉽게 복원될 수 있다. ART 모드의 OAT 파일도 내부에 최적화된 덱스가 존재하며, 쉽게 복원할 수 있다.

이후의 패커는 이 취약점을 보완하여, 컴파일된 덱스가 원본 형태 그대로 파일 시스템에 떨어지도록 하지 않는다. 우리의 분석 결과, 2016년도 5월경 서비스된 Bangcle의 경우, 경량화 암호 알고리즘인 RC4로 암호화하여 컴파일된 덱스를 파일 시스템에 보관하는 것을 볼 수 있었다. 그리고 매 실행 시에는

암호화된 캐시(컴파일된 덱스)를 로드 시에 (후킹을 통해) 복호화하여 사용하도록 하였으며, 최종적으로 이 취약점을 보완하였다.

2015년 11월에 달빅 머신을 수정하여 덱스 클래스 정보를 수집하는 DexHunter[16]가 발표되었다. 그리고 2017년 7월에 이와 유사한 기법인 CrackDex[17]가 발표되었다. 두 연구 모두 달빅 머신의 실행 흐름을 수정하기 위해, 공유라이브러리 libdvm.so(또는 libart.so)를 수정하였다. 이와 조금 다른 접근법으로 2017년 8월에 덱스 컴파일러를 수정한 기법이 소개되었다[9]. 원본 덱스를 실행하기 위해, 설치 이후에 최초로 실행할 때에는 무조건 DEX2OAT 컴파일러를 거쳐야 한다. 따라서, 컴파일러가 원본 덱스에 접근하는 지점(libart.so)에 그 덱스를 복사하는 코드를 추가함으로써 원본을 추출한다(Fig. 2. 참고). Fig. 3,은 안드로이드 런타임을 수정하여 덱스를 추출하는 기본적인 공격 지점을 도식화한 것이다.

```
1: DexFile::Dexfile(const uint8_t *base,
2:                  size_t size,
3:                  const std::string& location,
4:                  uint32_t location_checksum,
5:                  MemMap* mem_map,
6:                  const OatDexFile* oat_dex_file)
7: :begin_(base), size_(size),  ... {
8:   ...
9:   std::ofstream dst(location+"__unpacked", \
10:                     std::ios::binary);
11:  dst.write(reinterpret_cast<const char*>(base),\
12:            size);
13:  dst.close();
14:  ...
15: }
```

Fig. 2. Example of modifications for extracting original DEX (Bold font is inserted code in order to extract original DEX)

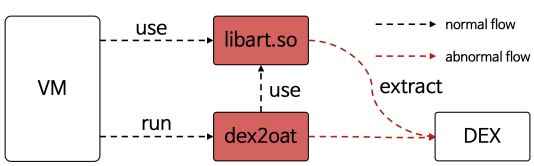

Fig. 3. Diagram of attack point

III. 배경 지식

본 장에서는 제안하는 방법을 이해하는 데 필요한 기본적인 지식을 다룬다.

3.1 안드로이드 어플리케이션 구조

안드로이드 어플리케이션은 크게 네 가지 요소로 구성되며, 이 구성요소들은 ZIP 알고리즘으로 묶여 *.apk 확장자로 놓이게 된다. 첫 번째 구성요소는 어플리케이션의 실행 파일로 덱스 파일이다. 확장자는 *.dex이며, 사용자가 자바 수준으로 작성한 클래스 정보들이 달빅 바이트코드로 인코딩되어 여기에 포함되어 있다. 두 번째 구성요소는 매니페스트(Manifest) 파일이다. 이는 인코딩된 xml 파일로, 어플리케이션이 사용하고자 하는 권한, 외부에 서비스하고자 하는 컴포넌트 이름과 속성 등이 포함되며 어플리케이션의 역할을 나타낸다. 세 번째 구성요소는 리소스이다. 리소스는 어플리케이션이 부가적으로 사용하고자 하는 읽기 전용의 파일이 포함된다. 예를 들어, 그림이나 음성과 같은 멀티미디어뿐만 아니라 부가적으로 사용하고자 하는 네이티브 라이브러리나 실행 파일이 포함되기도 한다. 네 번째 구성요소는 서명 파일로 어플리케이션을 서명한 정보와 공개키가 포함된다.

안드로이드 어플리케이션은 구동되면 안드로이드는 Application 타입의 객체와 패키지 속 덱스의 클래스로더를 생성한다. 그리고 수신한 이벤트에 따라 onCreate 메소드를 시작으로 어플리케이션 생명주기에 따라 각 메소드가 일련의 순서에 따라 호출되면서 구동된다. 여기서 중요한 점은 매니패스트에 명시된 클래스 정보가 생성된 클래스로더에 존재하지 않더라도 해당 클래스가 로드되기 전까지는 에러가 발생하지 않는 점이다. 패커 구현에 타입 검사가 동적으로 수행되는 점이 이용된다.

3.2 기본적인 패커 구조

안드로이드 어플리케이션을 패킹하는 방식은 다양할 수 있으나 그 기본 골격은 거의 같다. Fig. 4.은 이를 보여준다. 리패키징(또는 패킹)이 적용되면, 원본 덱스는 암호화되어 리소스에 포함된다. 그리고 원본 덱스 자리에 언패킹을 수행할 스텁 덱스가 들어가게 된다. 매니패스트는 원본의 형태를 유지하며, 스

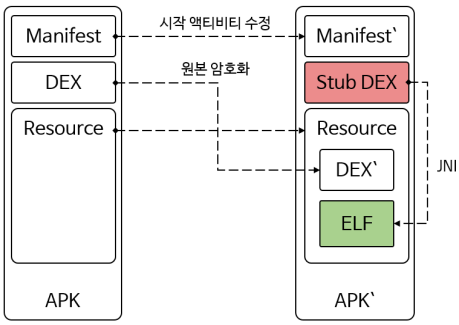

Fig. 4. Basic structure of packing android application

텁 덱스의 시작 클래스를 추가로 설정해 준다. 구현에 따라, 암호화된 덱스는 리소스에 포함되지 않을 수 있으며, 시작 클래스를 설정하는 방법은 원본 매니페스트의 구성에 따라 조금 달라질 수 있다.

패킹된 어플리케이션이 구동되면, 스텁 덱스의 실행 흐름에 따라 진행된다. 언패킹의 일반적인 과정은 다음과 같다. 우선, 런타임 보호 기법을 실행한 뒤 원본 덱스를 복호화한다. 다음으로 복호화된 덱스를 가지고 클래스로더를 생성하며, 스텁 클래스로더를 원본의 클래스로더로 치환한다. 마지막 과정으로, 원본 클래스로더에서 원본의 Application 클래스 정보를 불러와 객체를 생성하고 수행 흐름에 맞는 메소드를 호출하면서 언패킹 과정을 마친다.

스텁 덱스 또한 바이트코드이므로 역공학에 취약하다. 따라서, 이를 보완하기 위해 대다수의 패킹 기법은 앞서 언급한 과정을 JNI를 통하여 네이티브 수준에서 수행한다. 따라서 스텁 덱스는 실행 흐름의 큰 틀을 구성하며, 세부 동작은 공유라이브러리(Fig. 4.의 ELF)의 함수에서 수행된다.

3.3 안드로이드 어플리케이션 권한

기본적으로 안드로이드 어플리케이션은 샌드 박스에서 동작한다. 따라서, 운영체제나 다른 어플리케이션에 영향을 줄 수 없다. 단, 매니페스트에 명시된 권한에 한하여 자바 어플리케이션에 권한이 부여되며, 자바 API를 통해 외부로 요청할 때 부여받은 권한을 사용할 수 있다.

JNI를 통해 수행되는 네이티브 코드에서의 권한은 다른 방식으로 제어된다. 안드로이드는 리눅스 기반이므로, 각 어플리케이션은 하나의 프로세스이며 각 프로세스의 권한은 리눅스 권한 체계와 SEAndroid (SELinux의 확장된 버전)를 통하여 관리된다. 그리고 네이티브 코드에서 JNI를 통해 자바 API를 호출할 때에는 매니페스트의 권한이 동일하게 적용된다.

제조사와 안드로이드 빌드에 따라 해당 프로세스의 권한이 조금씩 차이가 있으나, 기본적으로 해당 프로세스의 디렉토리(/proc/*PID*/*)와 어플리케이션의 디렉토리 (/data/data/*app_name*/*)는 접근이 가능하다. 그리고 쓰기 작업은 기본적으로 샌드박싱된 어플리케이션 디렉토리에서만 가능하다. 즉, 쓰기 작업의 경우 외부 저장소 접근 권한을 받지 않을 때는 외부에 영향을 주지 못한다.

IV. 제안하는 기법

우리 기법의 목표는 현재 발표된 우회 기법과 이것의 변형된 기법을 검증하는 것에 있다. 본 장에서는 이를 위한 실행 환경의 무결성 검증 기법 두 가지를 제안한다.

4.1 검증 시점

일반적인 언패킹 과정은 Fig. 5.와 같다. 매니페스트가 가리키는 시작 클래스의 생명 주기에 따라 메소드가 실행되며, 핵심 루틴은 JNI를 통해 실행된다. 네이티브 수준에서 가장 먼저 이뤄지는 작업은 런타임 보호 모듈을 구동시키는 것이다. 그리고 보호 모듈이 안전하게 동작하면, 암호화된 덱스를 복원하

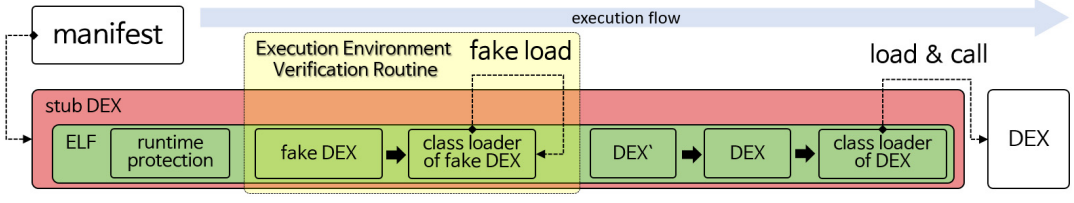

Fig. 5. Overview of execution environment verification for secure unpacking

고 클래스로더를 생성한다. 마지막으로 생성된 클래스로더의 시작 클래스를 로드하여 메소드를 호출하는 것으로 언패킹 과정이 끝나게 된다. 검증 루틴은 런타임 보호 모듈이 정상적으로 구동된 이후와 원본 덱스를 복원하기 전에 이뤄진다. Fig. 5.에서는 실행 환경 검증 루틴이라 명시된 부분이 이에 해당한다.

4.2 기법1: 더미 덱스를 통한 이벤트 검사

현재 발표된 공격기법의 핵심은 어플리케이션의 실행 흐름을 빼는 것이다. 수정되는 libdvm.so이나 libart.so는 어플리케이션의 프로세스가 사용하는 공유라이브러리로 달빅 머신이나 컴파일러를 구동시킬 때 실행된다. 따라서 추가 및 수정되는 코드의 권한은 프로세스의 권한을 따르게 된다. 이런 이유로, 수정된 코드로 원본 덱스에 접근하여 외부로 저장하기 위해서는 제한이 따르게 된다.

4.2.1 아이디어

추가된 코드가 원본 덱스의 데이터를 외부로 유출하는 작업은 해당 프로세스의 권한 아래에서 이뤄진다. 따라서, 데이터를 쓸 수 있는 대상은 매우 한정적이며, 우리는 제한된 지점의 이벤트를 모니터링함으로써 실행 환경의 무결성 여부를 간접적으로 판단한다.

현재 발표된 우회 기법은 크게 두 가지로 요약할 수 있다. 런타임 시 클래스를 로드하는 시점에 접근하는 방법 그리고 컴파일하기 위해 대상 덱스의 파일 디스크립터나 mmap 메모리에 접근하는 방법이다. 이 두 접근법 모두 클래스로더를 생성하거나 사용할 때 수행된다. 따라서, 원본 덱스의 클래스로더를 생성하기 전 더미 덱스의 클래스로더를 생성하고 여기에 포함된 클래스가 로드되도록 클래스 객체를 생성한다. 그리고 이때 발생하는 이벤트를 모니터링한다. Fig. 5.의 노란 영역이 이에 해당한다.

안드로이드 어플리케이션의 기본적인 권한을 기준으로, 유출 시나리오는 세 가지로 볼 수 있다. 첫 번째로, 어플리케이션에 할당된 공간에 파일로 복사하는 방법이 존재한다. '/data/data/*app_name*/*'의 하위 경로에 해당한다. 두 번째로, 외부 저장소 쓰기 권한(WRITE_EXTERNAL_STORAGE)이 존재할 경우, SD card와 같은 곳에 저장할 수 있다. 세 번째로, 네트워크 권한(INTERNET)이 존재할 경우 소켓을 통해 외부로 전송할 수 있다.

일반적으로 더미 덱스의 클래스로더를 생성 및 사용할 때에는 런타임 보호 모듈 및 DEX2OAT 컴파일러(5.0 이상 기준)가 생성하는 이벤트를 제외한 나머지 이벤트가 발생하였을 경우 실행 환경이 오염된 것으로 판단한다. 6.0 이하에서는 캐시 디렉토리에 OAT 파일 하나가 생성되며, 7.0에서는 OAT와 프로파일링 파일, 그리고 8.0에서는 OAT, 프로파일링 파일, VDEX 파일이 생성된다. 이렇듯 빌드에 따른 특성을 이용하여 판별 기준을 작성한다.

4.2.2 가정 사항

제안하는 기법은 샌드박싱된 안드로이드 어플리케이션의 권한 특성을 이용한 것이다. 따라서 이 기법은 공격자가 안드로이드를 수정할 때 기본적인 특성은 수정하지 않는 것을 가정한다. 이 가정은 현재 제안된 기법을 고려할 때 타당하다고 본다. 우리의 기법은 현재 발표된 공격기법의 방어를 목표로 한다. 따라서, DexHunter, CrackDex를 비롯하여 다른 여러 기법의 공통된 부분을 다루기 위해 이 가정은 엄격하지 않으며 충분하다고 판단된다.

4.2.3 지연 공격 대처 방안

본 항에서는 제안하는 기법이 우회될 수 있는 문제점에 대해 다루도록 한다. 제안하는 기법은 원본 DEX를 추출하기 위해 발생하는 이벤트를 간접적으로 확인하는 것이다. 따라서, 약간의 트릭에 검증 루틴이 무력화될 수 있다. 예를 들어, fork를 통해 새로운 프로세스를 생성하여 일정 기간의 지연을 준 다음 DEX를 복사할 수 있다. 유사하게 새로운 쓰레드를 생성해서 딜레이를 줄 수도 있다.

이런 경우를 대비하기 위해, 무결성 검증이 끝난 직후 최종적으로 프로세스와 쓰레드 리스트를 검사해줘야 한다. 프로세스의 경우 ps를 통해 자식 프로세스의 여부를 통해 검증할 수 있다. 그리고 쓰레드의 경우 /proc/*pid*/task/ 하위 경로의 쓰레드 리스트의 전후를 비교하거나 그 시점에 존재해서는 안 되는 쓰레드를 검출하는 식으로 검증할 수 있다.

4.2.4 구현 방안

네트워크 소켓을 통한 유출을 탐지하는 방식은

Table 1. Fake DEX verification time (Galaxy S7, Android 7.0)

App \ Time	1	2	3	4	5	6	7	8	9	10	Avg.
Melon (1.1MB)	1.83s	1.86s	1.81s	1.83s	1.88s	1.92s	1.98s	1.81s	1.95s	1.82s	1.87s
Duna (3.3MB)	3.88s	3.82s	3.95s	3.98s	3.91s	3.82s	3.94s	3.91s	3.93s	3.87s	3.90s
CGV (4.1MB)	3.97s	4.04s	3.92s	4.12s	3.93s	3.97s	4.12s	3.91s	4.11s	3.94s	4.00s
U+Mem (5.2MB)	5.03s	5.32s	5.21s	4.98s	5.13s	5.33s	5.01s	5.24s	5.06s	5.14s	5.15s

Table 2. Other integrity verification time (Galaxy S7, Android 7.0)

Time	1	2	3	4	5	6	7	8	9	10	Avg.
Verification time	0.31s	0.29s	0.30s	0.31s	0.32s	0.29s	0.30s	0.32s	0.31s	0.31s	0.31s

/proc/*pid*/net/tcp 파일의 비교를 통해 구현될 수 있다. 일반적으로 언패킹 과정에서는 새로운 소켓이 열리지 않으므로, 앞선 방식과 유사하게 해당 파일을 통해 비교하거나 해당 파일의 업데이트 유무를 통해 검증할 수 있다.

파일 시스템의 이벤트는 inotify를 이용하여 감지한다. 대상 경로의 하위 모든 파일에 대해 생성 및 쓰기 이벤트를 등록시킴으로써 검증할 수 있다. 간편한 구현을 위해 오픈 소스인 inotify-tools[18]와 같은 라이브러리를 이용할 경우 쉽게 구현할 수 있다.

4.3 기법2: 화이트 리스트를 이용한 검증

어플리케이션의 목적에 따라, 제한된 디바이스를 대상으로 서비스하거나 매우 높은 수준의 보안을 요구하는 경우가 존재할 수 있다. 본 절에서는 이런 상황에서 앞서 제안한 기법을 보완할 수 있는 추가적인 방법을 제안한다.

제안하는 방식은 주요 대상 파일의 암호 해시 값을 비교하는 것으로, 앞선 기법과 동일한 시점에서 검증이 이뤄진다. 대상 파일은 런타임 시 원본 DEX 및 OAT와 연관이 되는 공유라이브러리 및 실행 파일이다. 우리가 제안하는 대상 파일은 현재 우회 기법에서 대상이 된 libdvm과 libart를 포함한 총 7개로 다음과 같다.

- /system/lib[64]/libdvm.so
- /system/lib[64]/libart.so
- /system/lib[64]/libart-compiler.so
- /system/lib[64]/libcompiler_rt.so
- /system/bin/dex2oat
- /system/bin/oatdump
- /system/bin/patchoat

이 방식은 엄격할 뿐만 아니라, 용량 오버헤드도 동반한다. 256비트의 암호 해시를 사용한다고 가정할 경우, 디바이스 1000개에 대한 해시 값의 크기는 약 218KB이다. 파일 전체를 하나의 해시 값으로 계산한다 하더라도 37KB 정도 된다. 같은 디바이스이더라도 서로 다른 빌드 버전이 있으므로, 이 방식은 제한적인 상황에서만 적용 가능할 것으로 판단된다.

용량 오버헤드를 완화하는 방안으로 외부 서버를 통해 검증 값을 받아오는 방법이 있다. 패킹을 할 때, 공개키 쌍을 바이너리에 감춰두고 검증 시 빌드 버전, 공개키, 기타 정보 등을 개인키로 서명 및 암호화하여 제3의 서버로 보낸다. 서버는 이를 확인하고 포함된 공개키를 이용해 검증 값을 암호화해 전송하면 용량 오버헤드를 최소화할 수 있다. 단, 이러면 서버 유지 비용과 어플리케이션에 네트워크 권한을 무조건 포함해야 하는 단점이 발생한다.

V. 실험 및 평가

본 항에서는 제안한 기법의 성능 평가를 수행한다. 제안한 두 가지 기법 모두 구현 및 측정하였으며, 갤럭시S7에 탑재된 안드로이드 7.0 위에서 테스트하였다. Table 1.은 더미 DEX를 통한 무결성 테스트에 관한 실험 결과이며, Table 2.는 화이트 리스트를 통한 검증과 네트워크 소켓, 프로세스, 쓰레드의 이상 여부 검증에 관한 오버헤드 실험 결과이다.

더미 DEX를 통한 무결성 검증은 다음과 같이 이뤄졌다. 먼저, 해당 DEX의 클래스로더를 생성하고 그곳에 포함된 임의의 클래스 10개를 대상으로 객체를 생성하였다. 수행 시간은 Table 1.과 같이 평균적으로 DEX의 크기에 비례하였으며, 약 1메가당 1초의 추가 시간이 들어가는 것을 확인할 수 있었다.

검증의 주된 시간은 더미 DEX가 DEX2OAT 컴파일러를 통해 컴파일되는 시간으로, 클래스의 객체를 생성하고 이벤트를 모니터링하는 부분에서는 오버헤드가 거의 없었다.

화이트 리스트를 포함한 기타 무결성 검증은 다음과 같이 이뤄졌다. 암호 해시 알고리즘은 SHA-256를 사용하였으며, 앞서 언급한 파일 중 첫 번째를 제외한 6개를 상으로 하였다(첫 번째는 7.0에 포함되지 않음). 참고로 각 파일의 크기는 5.7MB, 2.3MB, 32KB, 1.8MB, 2.4MB, 0.5MB이다. 평균적으로 0.31초가 수행되었으며 대부분의 시간은 해시 값을 계산하는데 소요되었다.

최종적인 결론은 다음과 같다. 무결성을 검증하는 데 더미 덱스의 크기는 크게 중요하지 않으므로, 1MB 미만의 값을 선택하는 것이 적절하다고 판단된다. 그럴 경우, 약 2.1초의 오버헤드가 발생할 것으로 예상하며, 디바이스의 성능이 향상됨에 따라 오버헤드는 줄어들 것으로 판단된다. 또한, DexHunter와 컴파일러의 수행 흐름을 수정한 기법을 대상으로 실험하였으며, 정상적으로 탐지함을 확인하였다.

VI. 한계점

일반적으로, 역공학 보호 기법의 안전성은 암호 알고리즘과 달리 기밀성에 의존된다. 이런 이유로, 제안하는 무결성 탐지 기법이 공개되면 공격자가 쉽게 우회할 수 있게 된다. 예를 들어, 우리의 기법은 어플리케이션의 권한의 특징을 이용하는 것이므로, 공격자는 이를 우회하도록 안드로이드의 권한 체계를 바꿔서 실행할 수 있다. 특히, 안드로이드의 경우, 소스 코드가 공개되어 있어 더욱 쉽게 우회할 수 있다.

하지만, 제안하는 기법은 개선될 여지가 존재한다. 까다롭기는 하겠지만, 파일 시스템 전체를 모니터링 하도록 개선될 수 있다. 또한, 어플리케이션의 권한이 변경되었는지를 간접적으로 확인하기 위해, 외부 경로를 접근 시도하여 간접적으로 확인할 수 있다. 이처럼 한계점을 극복할 여지가 있으며, 이 부분은 추후 연구로 남겨 놓는다.

VII. 결 론

안드로이드는 소스가 공개되어 있어 공격자는 운영체제를 수정할 수 있으며, 이를 통해 보호된 어플리케이션의 원본 덱스를 쉽게 추출할 수 있다. 우리는 이를 방어하기 위해 언패킹 과정에서 실행 환경의 무결성을 검증하는 기법 두 가지를 제안하였다. 약 2초 정도의 오버헤드로 현재까지 공개된 우회 기법을 막을 수 있음을 보였다. 역공학 방어 기법의 특성상, 공개되는 순간부터 파훼 될 가능성이 커지지만, 개선의 여지가 있으며 이를 기반으로 향상된 기법이 고안될 것으로 기대한다.

References

[1] Dalvick bytecode, "https://source.android.com/devices/tech/dalvik/dalvik-bytecode", 10. Oct. 2018.

[2] DexDump, "https://github.com/plum-umd/dexdump/blob/master/dexdump/DexDump.c", 10. Oct. 2018.

[3] APKTool, "https://ibotpeaches.github.io/Apktool", 10. Oct. 2018.

[4] Dex2jar, "https://sourceforge.net/projects/dex2jar", 10. Oct. 2018.

[5] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in Proc. ICSME, pp. 388-398, Sept. 2017.

[6] A. Apvrille and R. Nigam, "Obfuscation in android malware, and how to fight back," Virus Bulletin, pp. 1-10, Jul. 2014.

[7] Execution compression, "https://en.wikipedia.org/wiki/Executable_compression", 10. Oct. 2018.

[8] Android Fragmentation, "https://opensignal.com/reports/2015/08/android-fragmentation", 10. Oct. 2018.

[9] S. Markkaveev and A. Bashan, "Unboxing Android: everything you wanted to know about Android Packers," presented at DEFCON 25, Aug. 2017.

[10] H. Shim, S. Cho, Y. Jeong, C. Lee, S. Han, and S. Cho, "A Technique for Protecting Android Applications using Executable Code Encryption and Integrity Verification," Journal of Korea Soft

ware Assessment and Valuation Society, pp. 19-26, 10(1), 2014.
〔11〕 K. Lee and J. Kim, "Study on Mechanism of Preventing Application Piracy on the Android Platform," Journal of the Korea Academia-Industrial cooperation Society, pp. 6849-6855, 15(11), 2014.
〔12〕 H. Jeong, K. Park, and J. Kim, "System for preventing forgery of application and method therefor," KR Patent KR101642267B1, 2016.
〔13〕 J. Ryu, S. Kim, H. Heo, and J. Jo, "Application Server for Verifying Integrity of Application and Controlling Method Thereof," KR Patent KR20170088858A, 2018.
〔14〕 J. Gwag, D. Lee, M. Kwon, and S. Choi, "Method for Detecting Application Repackaging in Android," KR Patent KR101498820B1, 2015.
〔15〕 Y. Park, "We Can Still Crack You! General Unpacking Method for Android Packer (no root)," presented at Black Hat ASIA, 2015.
〔16〕 Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward Extracting Hidden Code from Packed Android Applications," in Proc. ESORICS, pp. 293-311, Sept. 2015.
〔17〕 Z. Jiang, A. Zhou, and L. Liu, "CrackDex: Universal and automatic DEX extraction method," in Proc. ICEIEC, pp. 53-60, Jul. 2017.
〔18〕 inotify-tools, "https://github.com/rvoicilas/inotify-tools", 10. Oct. 2018.

〈저자소개〉

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

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