Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.27, NO.6, Dec. 2017 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2017.27.6.1271

안드로이드 덱스클래스로더 실행흐름 변경을 통한 원본 앱 노출 방지 기법

조 홍 래,[1†] 최 준 태,[1] 하 동 수,[1] 신 진 섭,[2] 오 희 국[1‡]
[1]한양대학교 컴퓨터공학과, [2]국가보안기술연구소

A Hiding Original App Method by Altering Android DexClassLoader Execution Flow

Honglae Jo,[1†] Joontae Choi,[1] Dongsoo Ha,[1] Jinseop Shin,[2] Heekuck Oh[1‡]
[1]Hanyang University Department of Computer Science and Engineering
[2]National Security Research Institute

요 약

런타임 실행 압축 기술을 이용하는 안드로이드 패커(packer) 서비스는 덱스클래스로더(DexClassLoader)를 이용하여 원본 어플리케이션으로 전환한다. 하지만 덱스클래스로더의 API 인터페이스는 로드할 덱스(Dalvik EXcutable)의 경로와 컴파일 된 파일의 경로를 입력 받으므로 원본이 파일시스템에 드러나는 문제점이 있다. 따라서 해당 API를 패커 서비스에 그대로 사용하는 것은 안전하지 않다. 본 논문에서는 이 문제를 해결하기 위해 덱스클래스로더 API의 컴파일 흐름과 로드 흐름을 변경하여 해결하였다. 이 변경된 실행 흐름으로 인해 컴파일 된 파일을 암호화하여 파일 시스템에 두거나 메모리에만 존재하도록 하고, 이후 로드할 때 복호화 또는 치환을 하여 원본 앱 전환을 가능하게 한다. 제안하는 기법을 통해 원본 파일이 파일시스템에 노출되지 않음으로써 패커의 안정성이 올라갈 것이라 예상한다.

ABSTRACT

The android packer service using runtime execution compression technology switches to the original application using DexClassLoader. However the API interface of the DexClassLoader receives the path of the loaded DEX(Dalvik EXcutable) and the path of the compiled file. So there is a problem that the original file is exposed to the file system. Therefore, it is not safe to use the API for the packer service. In this paper, we solve this problem by changing the compile and load flow of the DexClassLoader API. Due to this changed execution flow, the complied file can be encrypted and stored in the file system or only in the memory and it can be decrypted or substituted at the time of subsequent loading to enable the original application conversion. we expected that the stability of the packer will increase beacause the proposed method does not expose the original file to the file system.

Keywords: Packer, DexClassLoader, Hooking, File System, Load, Substitution

I. 서 론

안드로이드 어플리케이션은 바이트코드 기반으로 역공학에 취약하므로, 보호를 위해 패커 서비스를 이용한다. 여기서 언급하는 패커란 실행압축과 암호화를 통해 원본 덱스를 감추고, 역공학을 막기 위해

Received(09. 29. 2017), Modified(12. 06. 2017), Accepted(12. 10. 2017)

† 주저자, mme4078@gmail.com
‡ 교신저자, hkoh@hanyang.ac.kr(Corresponding author)

난독화, 런타임 보호, 안티 디버깅 등의 기법이 적용된 것을 말한다[1][2]. Fig. 1.은 패커 서비스 적용 후 변화된 APK 구조를 설명한 것이다. 원본 덱스는 암호화된 상태로 저장되고 덱스와 ELF(Executable and Linkable Format)파일이 하나씩 추가된다. 그리고 어플리케이션 시작 흐름을 추가된 덱스로 바꾸기 위해 매니페스트(manifest)가 수정된다[3].

패커 서비스 적용 후 변경된 어플리케이션 동작 메커니즘은 Fig. 2.와 같다. 먼저 어플리케이션이 구동되면 수정된 매니페스트로 인해 추가된 덱스와 ELF가 실행된다. 여기서 덱스는 바이트코드단에서 단순히 ELF 안에 존재하는 네이티브 함수를 호출하는 역할만 하고, 핵심 수행 과정은 역공학에 강건한 네이티브 함수에서 수행한다. 이 네티이브 함수에서 원본 덱스 복호화, 안티 디버깅 및 런타임 보호기법 등을 수행한다. 이후 원본 어플리케이션으로의 전환을 위해 원본 덱스가 메모리에 로드되고 원본 어플리케이션이 실행된다.

Fig. 2.의 동작 메커니즘을 보면 알 수 있듯이 원본 덱스를 로드하는 과정은 어플리케이션 실행을 위해 필수적이고, 안드로이드는 로드 API 함수로 덱스클래스로더[4]와 페스클래스로더(PathClassLoader)[5] 함수를 제공한다. 이 두 함수의 핵심 과정은 JNI(Java Native Interface)를 이용한 네이티브 코드에서 이루어지며, 이 함수의 인터페이스는 로드할 덱스 파일 경로와 AOT(Ahead-of-Time) 컴파일러를 통해 생성되는 OAT 파일의 경로를 입력으로 받는다. 즉 이 두 파일이 파일시스템에 노출되는 것을 알 수 있다.

실제로 2015년에 We can still crack you! general unpacking method for android packer(no root) 저자인 Yeongung Pack은 이 덱스클래스로더 함수를 이용해야하는 패커의 특성을 이용하여 적용된 보호 기법을 우회하지 않고도 복호화된 원본 덱스를 파일시스템에서 가져오는 발표를 하였다[6]. 발표된 취약점을 간략히 언급하면, 패커는 단순히 덱스클래스로더 함수의 인터페이스를 만족시키기 위해, 암호화된 원본 덱스를 덱스클래스로더 함수 호출 직전에 복호화 후 인터페이스를 만족하는 경로에 노출시키는 실수를 범하였다. 이 실수로 인해 공격자는 원본 덱스를 파일시스템에서 손쉽게 가져 올 수 있었다. 따라서 덱스클래스로더를 패커 서비스에 그대로 사용하는 것은 안전하지 않다.

제안하는 기법은 두 가지로, 런타임 효율성에 기반을 둔 로드 기법과 보안의 안전성에 기반을 둔 로드 기법을 제안한다. 이 두 가지 기법은 공통적으로 덱스클래스로더의 컴파일 흐름과 로드 흐름을 후킹하여 원본 덱스와 OAT 파일을 파일시스템에 노출시키

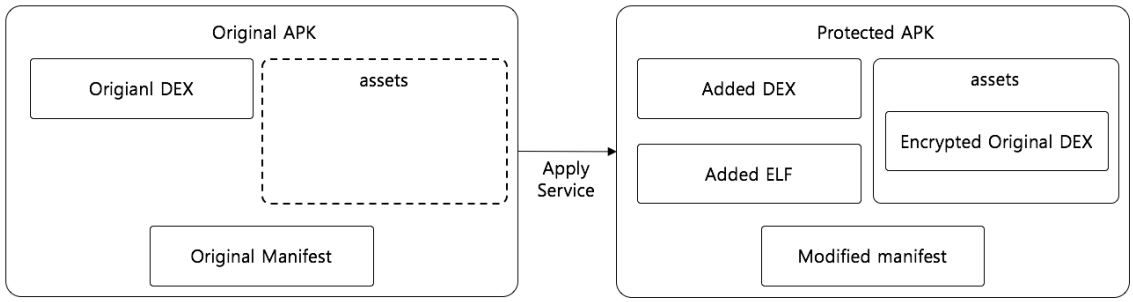

Fig. 1. Modified application operation mechanism after applying packer service

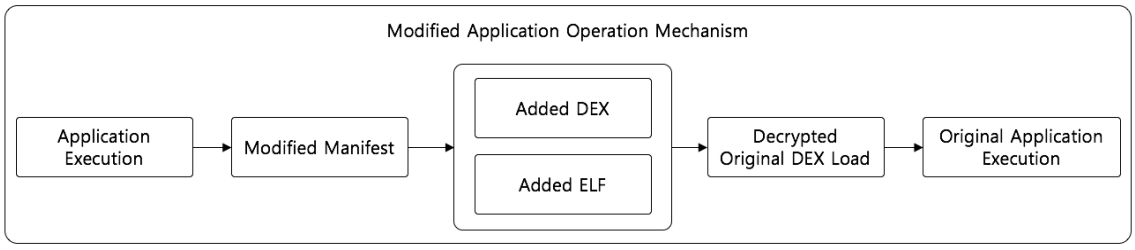

Fig. 2. Modified APK structure after applying packer service

지 않는다. 런타임 효율성에 기반을 둔 로드 기법은 OAT 파일을 암호화하여 파일시스템에 두고, 이후 로드 과정에서 복호화를 수행하여 로드한다. 안전성에 기반을 둔 로드 기법은 OAT 파일을 메모리에만 존재하도록 하고 이후 로드 과정에서 치환을 통해 원본 어플리케이션으로 전환한다. 이 두 가지 기법을 통해 원본 파일이 파일시스템에 노출되지 않으므로 패커의 안전성이 올라갈 것이라 예상한다.

이후 본 논문의 구성은 다음과 같다. 2장에서는 컴파일 흐름과 로드 흐름 후킹 지점을 정하기 위해 덱스클래스로더 동작 메커니즘에 대해 알아본다. 그리고 런타임 효율성이 좋은 로드 기법과 안전성이 높은 로드 기법에 대해 각각 3장과 4장에서 알아본다. 이후 5장에서 실험결과를 보이고 6장에서 결론을 맺으며 마무리한다.

II. 덱스클래스로더 동작 메커니즘

제안하는 기법은 후킹을 통해 덱스클래스로더의 흐름을 변경하여 원본 파일을 파일시스템에 노출시키지 않는다. 따라서 후킹 시점을 알아야 하고 이를 위해서는 덱스클래스로더 동작 메커니즘에 대한 이해가 선행되어야 한다. 이 메커니즘은 AOSP(Android Open Source Project)를 통해 분석하였고, 안드로이드 버전 6.0.1을 대상으로 하였다.

덱스클래스로더 동작 메커니즘의 핵심 과정은 네이티브코드에서 이루어지며, DexFile_openDexFileNative 함수에서 수행한다. Fig. 3.은 이 함수의 수행 과정을 도식화한 것이다. 먼저 아규먼트로 받은 OAT 파일 경로에 OAT 파일의 존재 여부를 확인하고 존재하지 않을 시, OAT 파일 생성 단계를 거친다. OAT 파일 생성 단계의 상세 과정을 보면 먼저 자식 프로세스가 생성되고, 이 프로세스는 /system/bin/dex2oat 실행파일을 통해 OAT 파일을 파일시스템에 생성한다. 이후 OAT 파일을 로드하고, OAT 파일의 최신성에 대한 검증을 수행한다.

이 네이티브 함수가 정상적으로 끝나면 OAT 파일이 메모리에 로드되고, 덱스클래스로더를 이용하여 OAT 파일 안에 존재하는 클래스를 사용할 수 있다. 패커의 관점에서 보면 원본 어플리케이션으로 전환이 가능하다는 것을 뜻하고, 제안하는 기법을 위한 후킹 시점은 Fig. 3.의 OAT 파일 생성 단계와 OAT 파일 로드 단계임을 알 수 있다.

III. 런타임 효율성에 기반을 둔 로드 기법

본 장에서는 런타임 효율성이 좋은 로드 기법에 대해 설명한다. 이 기법은 후킹을 통해 OAT 파일을 암호화된 상태로 파일시스템에 생성하고, 이후 로드 단계에서 복호화를 메모리상에서 수행함으로써 원본 어플리케이션으로 전환을 가능하게 한다. 본 논문에서 사용한 후킹 기법은 함수 프롤로그 수정을 통해 이루어진다. 자세한 설명은 하지 않지만 이와 같은 후킹을 통해 본래의 함수 흐름에 원하는 작업을 추가할 수 있다.

이후 본 장의 구성은 다음과 같다. 컴파일 과정을

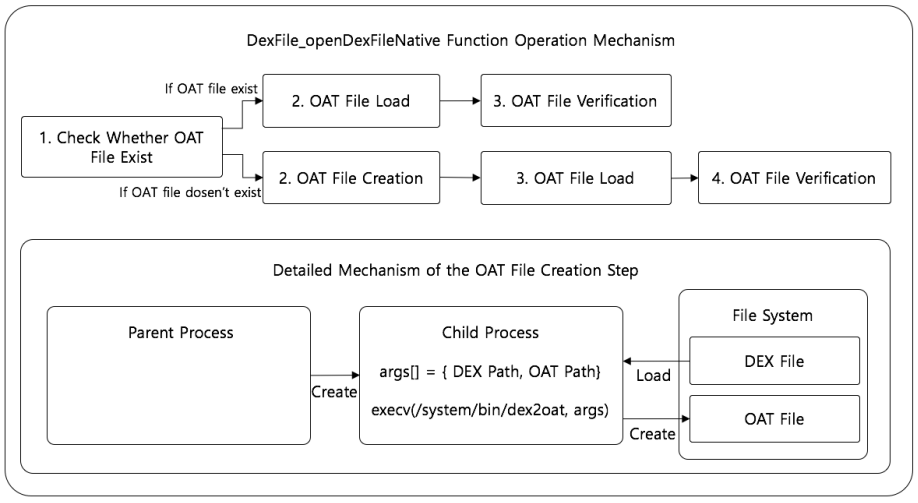

Fig. 3. Operation mechanism of the DexFile_openDexFileNative function

파악하기 위해 OAT 파일을 생성하는 /system/bin/dex2oat의 동작 메커니즘에 대해 알아본다. 그리고 컴파일 흐름과 로드 흐름을 후킹을 통해 변경함으로써, 원본 파일이 파일시스템에 노출되지 않으면서 정상적으로 로드되는 기법을 알아본다.

3.1 /system/bin/dex2oat 동작 메커니즘

/system/bin/dex2oat 프로세스는 Fig. 3.을 보면 알 수 있듯이 자식 프로세스에서 실행되고, OAT 파일을 파일시스템에 생성한다. Fig. 4.는 dex2oat 프로세스의 동작 과정을 도식화한 것이고, 동작 과정은 크게 4단계로 이루어진다. 다음은 단계별 동작 과정을 설명한 것이다.

① dex2oat 프로세스의 입력으로 들어온 아규먼트를 파싱하여 내부 필드를 설정하는 단계이다. OAT 파일을 생성하기 위해 필수적으로 들어가는 아규먼트는 덱스 파일의 경로와 OAT 파일의 경로이다.

② 0바이트 크기를 가지는 OAT 파일을 파일시스템에 생성한다. 설정된 OAT 경로에 파일의 존재 여부를 검사하고, 존재 하지 않을 시 OAT 파일을 생성한다.

③ 덱스 파일을 로드하는 단계로 먼저 로드할 파일에 대한 포맷 검사가 이루어진다. 읽기 영역에서 파일의 매직 넘버를 읽은 후 로드 대상 파일이 덱스 포맷인지 아니면 ZIP 포맷인지 결정한다. 만약 덱스 포맷이면, 덱스 크기만큼 메모리를 할당받은 후 해당 파일을 메모리에 맵핑한다.

④ AOT(Ahead-of-Time)컴파일러를 통해 나온 OAT 데이터를 2번 단계에서 생성한 OAT 파일에 쓴다.

위 동작 메커니즘을 보면 알 수 있듯이 암호화된 OAT 파일을 파일시스템에 생성하기 위해서는 AOT 컴파일러를 통해 나온 OAT 데이터를 파일에 쓰기 전에 암호화하는 과정이 필요한 것을 알 수 있다.

3.2 암복호화를 통한 덱스 로드 기법

본 절에서 설명할 기법을 도식화한 것은 Fig. 5. 이고 이후 설명은 이 그림을 중심으로 한다. 어플리케이션이 처음 구동되면 OAT 파일이 존재하지 않으므로 2단계인 OAT 파일 생성단계를 수행한다. 그러면 부모는 자식 프로세스를 생성하고, 생성된 프로세스는 dex2oat를 실행시킨다. 패킹 시 암호화된 원본 덱스를 메모리상에서 복호화하는 시점은 그림에 나타나 있듯이 2.1-2.3단계 중 선택적으로 할 수 있다. 하지만 복호화 된 원본 덱스가 메모리에 존재하는 시간이 길수록 메모리 덤프 등의 공격으로 인해 원본 덱스가 노출되는 문제점이 존재한다.

따라서 복호화 작업은 원본 덱스가 로드되기 바로 직전인 2.3단계에서 수행하는 것이 안전할 것으로 예상한다. 그리고 2.4단계에서 AOT 컴파일러를 통해 나온 OAT 데이터를 쓰기 지점에서 후킹 하여 암호화를 수행한 후 OAT 파일에 쓴다. 마지막으로 암호화된 OAT가 로드될 때 메모리상에서 복호화를 수

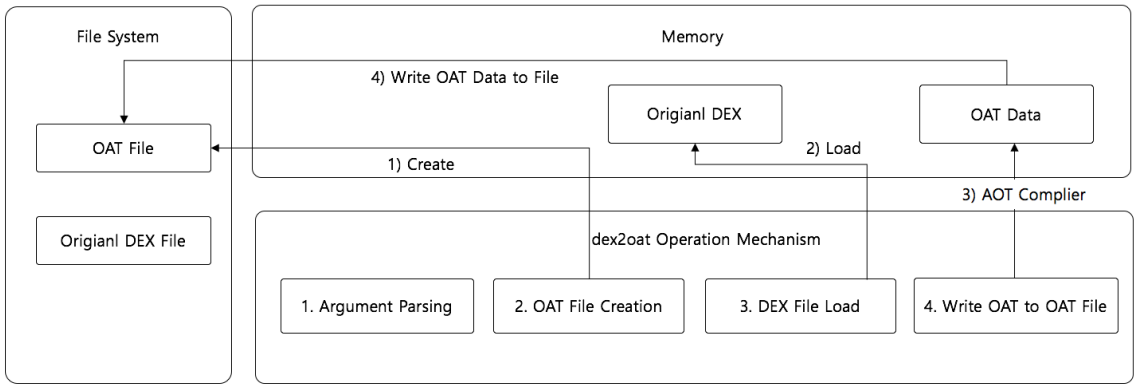

Fig. 4. /system/bin/dex2oat operation mechanism

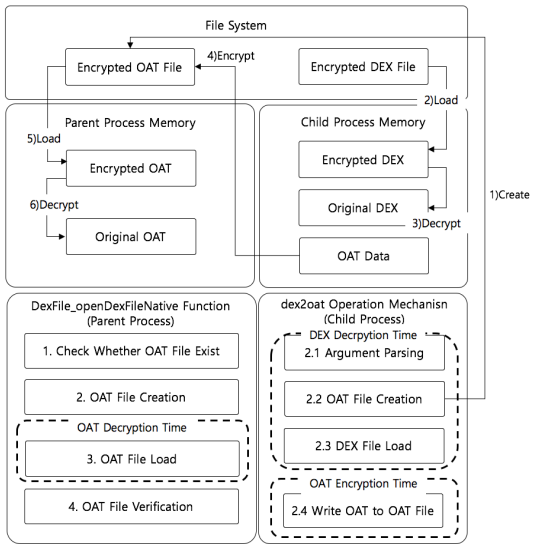

Fig. 5. Mechanism of operation of DEX technique scheme through encryption and decryption

행함으로써 원본 어플리케이션으로 전환한다. 본 논문에서는 RC4 알고리즘으로 암복호화를 수행한다. 이 알고리즘을 사용한 이유는 암호문과 복호문의 크기가 같고 복호화 연산이 가볍기 때문이다.

이 기법은 원본 파일이 아니라 암호화된 파일을 파일시스템에 노출시키므로 기존의 원본 파일이 노출되지 않는 것을 볼 수 있다. 이 기법의 장점은 어플리케이션 첫 실행 시에만 dex2oat 컴파일 과정을 거치므로 런타임 효율성이 좋다. 그 이유는 두 번째 실행부터는 암호화된 OAT 파일이 파일시스템에 존재하므로 바로 3단계인 OAT 파일을 로드하는 단계를 수행하기 때문이다.

이 기법의 단점은 원본 덱스와 OAT 파일을 보호하는 방법이 전적으로 복호화 키에 의존한다는 것이다. 비록 두 가지 파일이 암호화가 되어 있을지라도 덱스클래스로더를 만족시키는 인터페이스에 그대로 노출이 되어 있다. 즉 키가 노출되면 원본 파일이 쉽게 드러나는 문제점이 있으므로 키에 대한 안전한 관리가 필요하다. 하지만 본 논문에서 설명하는 주된 내용은 원본 파일을 보호하기 위한 전반적인 메커니즘을 설명하는 논문이므로 세부 사항인 키 관리에 대한 설명은 생략한다.

다음 장에서 설명할 안전성에 기반을 둔 로드 기법은 두 파일에 대한 안전성이 전적으로 복호화 키에만 의존하는 것이 아니다. 이 기법은 더미 파일을 이용하여 덱스클래스로더의 인터페이스를 만족시키고 이후 후킹을 통해 실행 흐름을 변경함으로써 원본 어플리케이션으로 전환한다. 즉 안정성이 키뿐만 아니라 동작 메커니즘에도 같이 포함된다.

IV. 안전성에 기반을 둔 로드 기법

본 장에서는 더미 덱스를 이용하여 덱스클래스로더의 인터페이스를 만족시키고 두 번의 치환을 통해 암호화된 파일조차도 파일시스템에 노출시키지 않는 기법에 대해 설명한다. 이후 본 장의 구성은 다음과 같다. 먼저 기법의 전반적인 개요에 대해 알아본다. 그리고 4.2절에서 원본 OAT를 메모리상에만 존재하도록 하는 기법에 대해 설명하고, 원본 어플리케이션 전환을 위해 치환하는 과정과 체크섬 검증 단계를 우회하는 방법을 4.3절에서 알아본다.

4.1 안전성에 기반을 둔 로드 기법의 개요

안전성에 기반을 둔 로드 기법은 더미 덱스를 이용하여 덱스클래스로더의 인터페이스를 만족시키므로 암호화된 원본 덱스는 어플리케이션 전환을 위해 다른 지점에 존재 해야한다. 본 논문에서는 ELF 파일 포맷의 공유라이브러리에 존재한다. 이 공유라이브러리를 로드시킴으로써 원본 덱스를 메모리에 로드한다. 기법의 전반적인 과정은 Fig. 6.에 나타나 있고 이후 설명은 이 그림을 중심으로 한다. 원본 OAT 생성 시점은 1,2,3 단계 중 어느 단계나 가능하고, 후킹을 통해 이루어진다. 이 후킹을 통해 원본 OAT를 메모리상에만 존재하도록 하는 기법은 다음 4.2절에서 설명한다. 2단계인 OAT 파일 생성 단계를 거치면 더미 덱스에 대한 더미 OAT 파일이 파일시스템에 생성되고, 이후 3단계인 OAT 파일 로드 단계에서 더미 OAT가 메모리에 로드된다. 그 시점에 메모리에 있는 원본 OAT로 치환을 하고, 검증 단계를 우회하기 위해 체크섬을 변경한다. 이 과정은 4.3절에서 자세히 설명한다.

위 기법을 이용하면 더미 덱스와 더미 OAT 파일만 파일시스템에 존재하므로, 암호화된 원본 파일조차도 파일시스템에 노출되지 않는 것을 볼 수 있다. 따라서 3장에서 설명한 복호화 키에만 의존하는 문제점을 해결하였다. 하지만 이 기법은 3장의 기법에 비해 런타임 오버헤드가 큰 단점이 있다. 실행할 때마다 원본 OAT를 생성하기 위한 컴파일 과정을 거

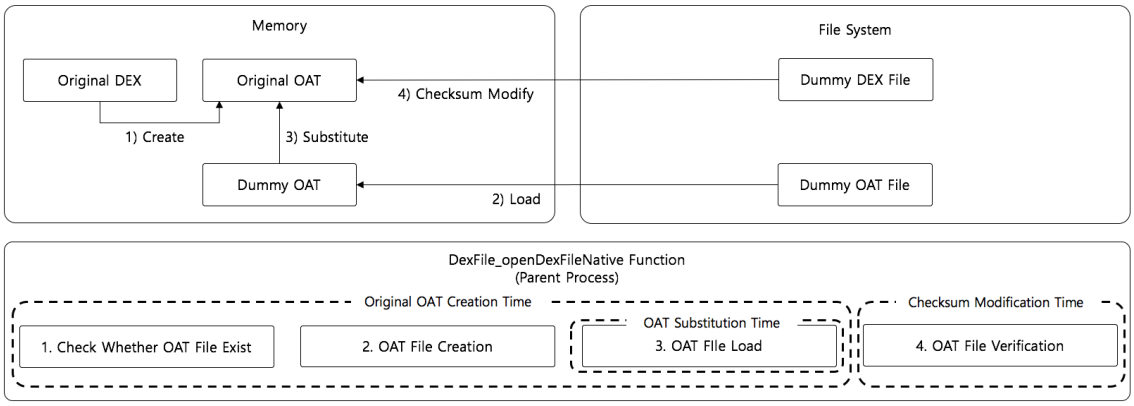

Fig. 6. Overall process of DEX load technique based on safety

치므로, 이 컴파일 시간만큼의 오버헤드가 존재한다. 그러나 이 오버헤드는 5장의 실험결과에서 확인할 수 있듯이 모바일 기기의 성능이 올라갈수록 또는 안드로이드 버전이 올라갈수록 점차 개선되므로 큰 문제가 되지는 않는다.

4.2 원본 OAT 생성 기법

원본 OAT를 생성하기 위해서는 /system/bin /dex2oat 컴파일 과정이 기존의 덱스클래스로더 흐름에 추가되어야 하고, 이 과정은 후킹을 통해 이루어진다. Fig. 7.은 원본 OAT 생성 메커니즘을 도식화 한 것이고 이후 설명은 이 그림을 중심으로 한다. 먼저 후킹을 통해 실행되는 함수는 Fig. 7.의 은닉 함수이다. 이 은닉함수는 자식 프로세스를 생성한 후 원본 덱스를 포함하는 공유라이브러리를 프리로드(preload)로 설정한다. 이후 execve 함수를 통해 dex2oat를 실행한다. 즉 원본 덱스는 프리로드로 설정된 공유라이브러리를 통해 메모리에 로드된다.

이후 dex2oat 동작 메커니즘에 따라 2단계에서 0바이트 크기를 가지는 OAT 파일이 생성되고 3단계에서는 더미 덱스 크기만큼 메모리를 할당받은 후 더미 덱스가 로드된다. 이때 할당받은 메모리 시작 주소를 원본 덱스의 시작 주소로 바꿈으로써 덱스 치환이 이루어진다. 이 과정은 2), 3)에서 확인할 수 있다. 그리고 4단계에서 AOT 컴파일러를 통해 원본

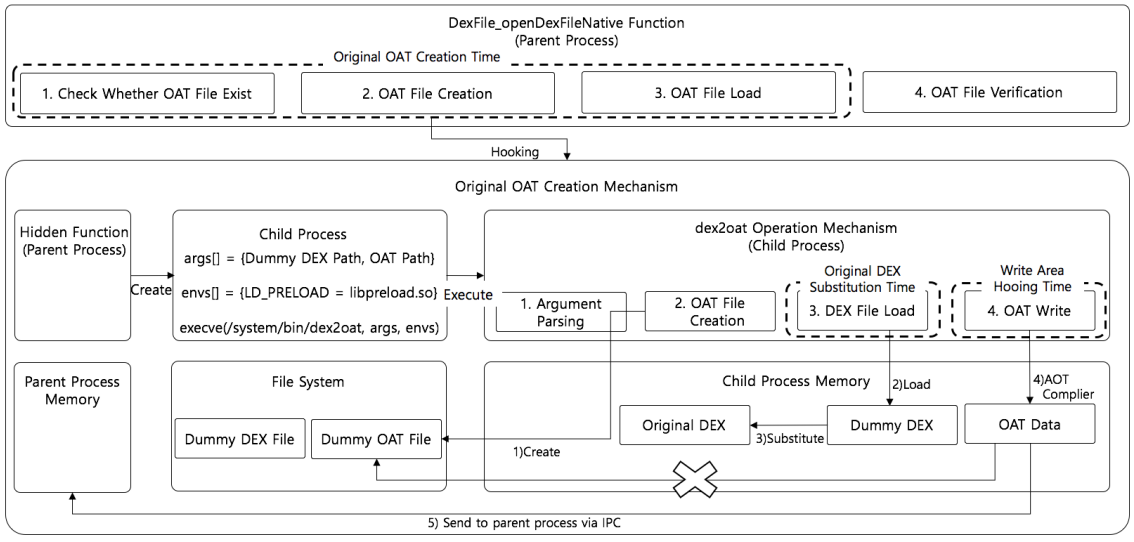

Fig. 7. Mechanism of operation of original OAT generation technique

덱스에 대한 OAT 데이터가 나오고, 이후 쓰기 지점을 후킹 하여 파일에 쓰는 것이 아니라 IPC(Inter Process Communication)를 통해 부모 프로세스로 전송한다.

위와 같은 후킹을 통해 dex2oat 동작 과정이 변형되면, OAT 파일이 파일시스템에 생성되지만 0바이트 크기를 가진다. 또한 실제 OAT는 부모 프로세스의 메모리에만 존재하는 것이 가능하다. 즉 파일시스템에는 더미 덱스와 0 바이트 크기를 가지는 더미 OAT 파일만 존재한다. 따라서 암호화된 덱스와 OAT 파일마저도 덱스클래스로더의 인터페이스를 만족하는 경로에 노출되지 않는 것을 볼 수 있다.

4.3 원본 어플리케이션 전환을 위한 로드 기법

메모리상에 존재하는 원본 OAT를 공유 라이브러리 로드 메커니즘에 따라 다른 메모리로 복사하는 작업이 필요하다. 그 이유는 ELF 파일 포맷인 원본 OAT는 공유 라이브러리 로드 메커니즘에 따라 로드된 것이 아니라 파일 전체가 로드 되어 있기 때문이다. 그리고 이 과정은 파싱을 통해 이루어진다.

원본 OAT로 치환하는 시점은 Fig. 6.의 OAT 로드 시점이고, 치환은 더미 OAT를 가리키는 포인터를 원본 OAT를 가리키도록 수정함으로써 이루어진다. 이 과정을 위해서는 더미 OAT의 soinfo 구조체 필드를 변경해야 한다. 변경 대상 필드는 총 5개로 base, size, bias, phdr, phnum이다. 변경해야 할 값은 메모리에 로드된 원본 OAT의 가상 메모리 주소를 바탕으로 설정한다.

OAT 치환 이후 원본 어플리케이션 전환을 위해서는 OAT 파일 검증 단계를 우회하는 작업이 필요하다. 안드로이드에서 수행하는 검증 메커니즘은 OAT 체크섬 필드와 덱스 체크섬 필드의 동일 여부를 통해 검증한다. 원본 OAT의 체크섬은 더미 덱스의 체크섬과 다르기 때문에 변경해야만 하고, 변경 방법은 OAT의 DEX file 구조체의 체크섬 필드[7]를 파일시스템에 존재하는 더미 덱스의 체크섬으로 변경한다. 지금까지 수행이 완료되면 이후 원본 어플리케이션으로 전환이 가능하다.

V. 실험 결과

5.1 실험 대상 모바일

본 논문에서는 제안하는 기법을 적용하기 위한 대상 모바일로 구글의 레퍼런스 폰인 넥서스5를 선정한다. 그 이유는 삼성전자[8], LG전자[9], HTC[10], 팬택[11], 모토로라[12] 등의 모바일 제조업체는 자사의 하드웨어 최적화를 위해 안드로이드 OS를 변형하지만 핵심 동작 과정은 대부분 동일하다. 실제로 각 회사의 소스 코드를 다운받아 비교를 통해 이를 확인하였고, 갤럭시 노트 4의 경우에는 기존의 기법 수정 없이 바로 실행된다. 다른 모바일의 경우에도 실험은 하지 않았지만 큰 수정 없이 가능할 것이라고 예상한다.

5.2 효율성 검증

이 절에서는 제안하는 기법 적용 시 가중되는 오버헤드에 대해 알아본다. 실험을 위한 데이터 셋은 2017.08.17을 기준으로 구글 플레이 스토어의 인기 앱 1-100 순위 중 덱스 크기가 1MB 단위로 차이가 나는 어플리케이션을 선정하였다. Table 1은 선정된 10개의 어플리케이션을 보여준다. 그리고 가중되는 오버헤드 검증을 위해 여러 제조사의 모바일을 대상으로 dex2oat 컴파일 과정을 수행하였고, Table 2와 Table 3은 사용한 모바일의 스펙과 dex2oat 컴파일 수행 시간을 정리한 것이다.

Table 3을 보면 알 수 있듯이 덱스 크기가 커질수록 수행 시간이 오래 걸린다. 실험 데이터 셋에서 덱스 크기가 제일 큰 11MB의 경우 기종마다 다르

Table 1. Selected 10 applications

	Application	DEX(MB)
1	Melon	1.9
2	Duna	3.3
3	CGV	4.1
4	U+MemberShip	5.2
5	Yin Yang	6.4
6	Foodie	7
7	Daum	8
8	Naver Papago	9.1
9	Ace Browser	10.1
10	B612	11.1

Table 2. Specification

Mobile	AP(Application Processor)	Version
Google Nexus 5	Qualcomm Snapdargon 800	4.4
Google Nexus 5X	Qualcomm Snapdargon 808	6.0
Samsung Galaxy Note 4	Exynos 5433	4.4.4
Samsung Galaxy 5	Qualcomm Snapdargon 801	4.4.2
Samsung Galaxy 6	Exynos 7420	5.0.2
Samsung Galaxy 7	Exynos 8890	6.0.1
Samsung Galaxy 8	Exynos 8895	7.0
Vega	Qualcomm Snapdargon S4 Pro	4.0.4
HTC ONE M9	Qualcomm Snapdargon 810	5.0.1
LG G3	Qualcomm Snapdargon 801	4.4

Table 3. dex2oat compile time per mobile

Manufacturer	Mobile	Performance (second)									
Google	Nexus 5	1.48	2.55	2.70	4.09	4.73	5.57	5.92	6.13	8.04	8.50
	Nexus 5X	1.40	1.79	1.93	3.06	3.74	3.95	4.37	4.39	6.14	6.61
Samsung	Galasy Note 4	1.06	1.73	1.81	2.96	3.56	4.01	4.29	4.40	6.33	6.68
	Galasy 5	1.90	2.81	2.92	4.64	5.02	5.52	5.61	6.56	7.25	7.75
	Galasy 6	1.54	1.99	2.08	3.46	3.82	3.91	4.36	4.54	6.89	6.54
	Galasy 7	1.23	1.97	1.97	3.38	3.53	3.62	4.01	4.05	6.07	6.11
	Galasy 8	1.10	1.64	1.69	2.61	3.03	3.19	3.47	3.55	5.24	5.49
LG	G3	1.21	2.36	2.49	4.17	4.54	5.27	5.72	6.47	7.26	8.56
HTC	ONE M9	2.03	2.69	2.75	4.58	5.62	5.87	10.12	10.26	15.83	16.35
Pantech	Vega R3	1.53	2.67	2.90	4.30	4.83	6.07	6.46	6.54	9.01	10.53

Table 4. dex2oat compile time per android version

Mobile	Version	Performance (second)									
Nexus 5	4.4.4	1.72	2.45	2.89	4.73	5.01	5.57	6.44	6.55	9.12	9.15
	5.1.1	1.60	2.55	2.73	4.37	4.85	5.30	6.19	6.11	8.59	9.07
	6.0.1	1.48	2.55	2.70	4.09	4.73	5.57	5.92	6.13	8.04	8.50

지만 최근에 출시된 삼성 갤럭시 8의 경우 5.49가 걸리므로, 다른 모바일과 비교해서 빠른 것을 볼 수 있다. 이를 통해 이후 출시되는 모바일의 경우에는 수행 시간이 더 줄어 들것으로 예상한다. 뿐만 아니라 어플리케이션 첫 구동 시에만 오버헤드가 걸리고, 이후 수행은 오버헤드로 인한 문제가 생기지 않는다. Table 4.는 안드로이드 버전별 dex2oat 수행 시간을 측정한 것으로 dex2oat 최적화 또는 AOT 컴파일러 최적화로 인해 작지만 줄어든 것을 확인할 수 있다. 즉 안드로이드 버전 또한 수행 시간에 영향을 주는 것을 알 수 있다. 실험 결과를 토대로 유추하면 제안하는 기법의 오버헤드 문제점은 점차 개선될 것이고 추후에는 가중되는 오버헤드가 아주 작을 것이라 예상한다.

VI. 결 론

대다수의 패커는 런타임 중간에 암호화된 원본 덱스를 복호화 하여 메모리에 로드하며, 그 과정에서 로드 API가 필연적으로 사용된다. 이 API의 인터페이스는 로드 할 덱스 파일의 경로와 컴파일 된 OAT 파일의 경로를 입력으로 받으므로, 런타임 중간에 원본 파일이 파일시스템에 노출되는 문제점이 존재한다. 본 논문에서는 이 문제점을 해결하기 위해 런타임 효율성이 좋은 로드 기법과 안전성에 기반을 둔 로드 기법을 제안한다. 이 두 기법은 후킹을 통해 덱스클래스로더의 컴파일 흐름과 로드 흐름을 변경하여 원본 파일을 파일시스템에 노출시키지 않고 로드하며, 이후 원본 어플리케이션 전환을 가능하게

한다.

런타임 효율성에 기반을 둔 로드 기법은 암호화를 통해 암호화된 OAT 파일을 덱스클래스로더의 인터페이스를 만족하는 경로에 노출시키고, 로드 과정에서 복호화 작업을 수행한다. 이 기법의 장점은 어플리케이션 첫 실행 시에만 컴파일 과정을 거치고 이후 실행 시에는 컴파일 과정을 거치지 않으므로, 런타임 효율성이 좋다는 장점이 있다. 하지만 파일에 대한 안전성이 전적으로 키에만 의존하는 단점이 있다.

안전성에 기반을 둔 로드 기법은 더미 덱스를 이용하여 덱스클래스로더의 인터페이스를 만족시키며, 두 번의 치환 과정을 통해 암호화된 원본 파일조차도 파일시스템에 존재하지 않는다. 즉 키가 노출이 되더라도 해당 기법의 동작 메커니즘을 모르면 원본 파일을 가질 수 없으므로 보안의 강도는 높다는 장점이 있다. 하지만 어플리케이션을 실행할 때마다 컴파일 과정이 포함되므로 오버헤드가 존재하는 단점이 있다. 실험 결과에서 보면 알 수 있듯이 이 오버헤드는 점진적으로 줄어들 것이고 추후에는 오버헤드가 아주 작을 것이라 예상한다.

이 두 가지 기법은 안드로이드에서 새로운 환경이 나오지 않는 한 패커의 안전성을 높이는 유용한 기법으로 기대한다.

References

〔1〕 R.Yu, “Android packers: facing the challenges,” *building solutions, in Proceeding of the 24th Virus Bulletin International conference*, Sep. 2014.

〔2〕 W Enck, D Octeau, McDaniel, S Chaudhuri, “A study of android application security,” *in Proceedings of the 20th USENIX Security Symposium,* Aug, 2011.

〔3〕 Jongsu Lim, Jeonghyun Yi, “Structural analysis of packing schemes for extracting hidden codes in mobile malware,” *EURASIP Jorunal of Wireless Communications and Networking*, Dec, 2016.

〔4〕 https://developer.android.com/reference/dalvik/system/DexClassLoader.html

〔5〕 https://developer.android.com/reference/dalvik/system/PathClassLoader.html

〔6〕 Yeongung Park, “We can still crack you! general unpacking method for android packer(no root),” *BlackHat Asia 2015,* Mar. 2015.

〔7〕 Paul Sabanal, “Hiding Behind Android Runtime(ART),” *BlackHat Asia 2015,* Mar. 2015.

〔8〕 http://opensource.samsung.com/reception.do

〔9〕 http://www.lg.com/global/support/opensource/opensource.jsp

〔10〕 http://htcdev.com/devcenter/downloads

〔11〕 http://opensource.pantech.com

〔12〕 http://sourceforge.net/motorola/

〈저자소개〉

조 홍 래 (Honglae Jo) 학생회원
2016년 2월: 국립목포해양대학교 컴퓨터공학과 학사
2016년 3월~현재: 한양대학교 컴퓨터공학과 석사과정
〈관심분야〉 정보보호, 모바일 보안, 시스템 보안

최 준 태 (Joontae Choi) 학생회원
2017년 2월: 한양대학교 컴퓨터공학과 학사
2017년 3월~현재: 한양대학교 컴퓨터공학과 석사과정
〈관심분야〉 정보보호, 모바일 보안, 시스템 보안

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

신 진 섭 (Jinseop Shin) 정회원
2012년 2월: 충남대학교 컴퓨터전공 졸업
2014년 2월: 충남대학교 컴퓨터공학과 석사
2013년 12월~현재: 한국전자통신연구원 부설연구소 연구원
〈관신분야〉 모바일 보안, 시스템 보안

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