Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.5, Oct. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.5.1209

상용 안드로이드 앱 보호 서비스 분석을 통한 강건한 앱 보호 구조 연구*

하 동 수,† 오 희 국‡
한양대학교

Study on Structure for Robust App Protection through Commercial Android App Hardening Service*

Dongsoo Ha,† Heekuck Oh‡
Hanyang University

요 약

안드로이드 앱은 바이트코드로 구성되어 있어 역공학으로부터 취약하며, 이를 보완하기 위해 앱을 강건하게 재구성해주는 보호 서비스들이 등장하였다. 암호 알고리즘과 다르게, 이런 보호 서비스의 강건함은 보호 방식을 감추는 것에 상당 부분 의존하고 있다. 그러므로 보호 서비스의 파훼 기법은 다양하더라도 보호 방식에 대한 체계적인 논의가 거의 없으며, 개발자의 직감에 따라 구현되고 있다. 정적 또는 동적분석을 방해하는 기술의 간단한 배치보다는, 강건한 보안 체인을 위한 체계적인 보호 구조에 대한 논의가 필요하다. 본 논문에서는 이를 위해, 대표 상용 안드로이드 앱 보호 서비스인 방클(bangcle)을 분석하여 보호 구조와 취약한 요소를 살펴본다. 그리고 이를 통해 강건한 구조를 위해 요구되는 사항과 보호 구조 원칙을 제안한다.

ABSTRACT

Android apps are made up of bytecode, so they are vulnerable to reverse engineering, and protection services are emerging that robustly repackage the app to compensate. Unlike cryptographic algorithms, the robustness of these protection services depends heavily on hiding the protection scheme. Therefore, there are few systematic discussions about the protection method even if destruction techniques of the protection service are various. And it is implemented according to the intuition of the developer. There is a need to discuss systematic protection schemes for robust security chains, rather than simple deployment of techniques disrupting static or dynamic analysis. In this paper, we analyze bangcle, a typical commercial Android app protection service, to examine the protection structure and vulnerable elements. We propose the requirements for robust structure and principles of protection structure.

Keywords: Android, Obfuscator, Protector, Packer, App hardening, Runtime protection

I. 서 론

안드로이드 프레임워크는 자바 API 형태로 제공되므로, 안드로이드 앱은 바이트코드 기반으로 구성된다. 이런 바이트코드 기반의 어플리케이션은 뛰어난 호환성을 가지게 되지만, 고수준 언어의 특성상

Received(06. 12. 2018), Accepted(09. 05. 2018)
* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성 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)

높은 가독성과 수정의 간편함으로 인해 역공학에 취약하다. 안드로이드 앱도 이런 특성에서 벗어나지 못한다.

2014년도 Arxan의 조사에 따르면, 안드로이드 앱마켓 상위 100개의 유료 어플리케이션 모두가 크래킹(cracking)되었고, 인기 무료 어플리케이션은 73%가 크래킹되었다고 한다[1]. 이는 iOS 앱마켓의 56%, 53%에 비해 매우 높은 수치이다. 이를 통해 알 수 있듯이 바이트코드는 역공학에 상당히 취약하다.

이러한 문제를 해결하기 위해 안드로이드 앱을 역공학으로부터 강건하게 만들어주는 서비스가 등장하였다[2-7]. 이러한 서비스는 다양한 방식으로 시작하였으나, 현재는 실행 압축(runtime packer)을 이용하는 방식으로 귀결되고 있다. 이 방식은 원본 바이트코드를 암호화하여 역공학으로부터 보호한다. 그리고 암호화된 원본은 앱 구동 시 스텁(stub) 바이트코드에 등록된 네이티브 메소드에서 복호화 및 실행된다.

이 보호 방식은 원본이 복호화되기 전 수행되는 코드의 강도에 따라 안전성이 결정되며, 코드의 강도는 복잡성에 따라 결정된다. 복잡성을 통한 낮은 가독성은 전체 코드의 분석을 저지하게끔 하여 분석에서 놓치는 지점을 만든다. 이런 지점을 많이 발생시켜야 보호 모듈을 배치할 지점이 많아지고, 결과적으로 안전성이 높아지게 된다.

이런 이유로 인해 전단부에 배치되는 코드의 기밀성이 중요하다. 구현 방식이 노출될 경우, 상대적으로 분석이 용이해지며 결과적으로 보호 강도가 내려가게 되기 때문이다. 이뿐만 아니라 경우에 따라서는 보호 구조 자체가 바로 우회될 수도 있다. 따라서 현재까지 대다수의 보호 서비스들은 기밀성에 많은 부분을 의존하고 있다.

문제는 기밀성 때문에 보호 방식에 관한 논의가 이뤄지지 않는 데 있다. 우리의 연구결과에 따르면, 상용 안드로이드 앱 보호 서비스조차 체계적인 구조를 띠고 있지 않으며, 많은 부분이 직감에 의존하여 구현되고 있는 것을 알 수 있었다. 안전성을 높이는 데 있어 요구되는 사항을 명확히 하고 올바른 구현 원칙에 대한 논의가 필요하다.

본 논문에서는 강건한 앱 구조 설계를 위한 기본 원칙에 대해 다룬다. 이를 위해 우선 상용 앱 보호 서비스를 조사 및 분석하여 그것의 보호 구조와 문제점을 파악한다. 그런 뒤 공격자 가정 사항을 세우고 이를 바탕으로 강건한 구조를 위해 요구되는 사항을 도출한다. 마지막으로 요구사항을 만족시키기 위한 보호 구조 설계 원칙을 제안하며, 그것의 타당성을 검증한다.

II. 관련 연구

앱 보호의 구조나 구현에 관해 공개적으로 논의된 연구는 우리가 아는 한 OWASP[8]를 제외하고는 없다. 그리고 OWASP에서 조차 공격 벡터에 대한 대응 방안만 있을 뿐 구조에 관한 내용은 다루지 않는다. 또한, 국내 몇 연구[9,10]에서도 실행 압축 기법을 이용한 보호 방식을 제안하였지만, 바이트코드의 암호 방식에만 초점이 맞춰져 있으며, 다른 공격 벡터에 대한 위협은 다루지 않는다. 따라서 본 장에서는 현재까지 알려진 앱 보호 서비스을 알아보고, 이들의 취약점과 보호 기법 적용의 문제점을 살펴본다.

2.1 상용 서비스의 앱 보호 방식

지금까지 공개된 안드로이드 앱 보호 서비스는 크게 난독기(obfuscator), 프로텍터(protector), 패커(packer)로 나눌 수 있다[11]. 난독기의 경우 ProGuard, DexGuard, Allatori 등의 서비스가 존재한다. ProGuard는 사용자 정의 클래스명, 변수명 등을 임의의 문자열로 바꾸는 식으로 난독화하며, DexGuard와 Allatori는 자바 리플렉션과 문자열 암호를 이용하여 난독화한다. 이와 같은 방식은 바이트코드를 수정하는 것이기 때문에 보안 강도가 낮다. 또한, 자동화 패턴이 쉽게 파악될 수 있고, 이를 통해 자동화 복원 도구 개발이 가능한 문제점이 있다.

다음으로 프로텍터는 원본을 수정하여 외부 공격으로부터 보호하는 방식이다. 정상적인 원본은 앱 구동 시 수정을 통해 복원되며, 메모리상에서만 정상적인 원본이 존재하게 된다. 메모리상의 원본은 복원되는 과정에서 구동되는 실시간 보호 모듈을 통해 보호된다. 다른 기법과 구분되는 특징으로, 프로텍터는 원본에 별도의 검증 작업을 추가하여 보호 강도를 높이기도 한다. 현재까지의 조사로는 APKProtect만이 이 방식을 사용하는 것으로 파악된다[11,12].

마지막으로 패커는 원본을 암호화하여 외부 공격으로부터 보호하는 방식이다. 원본은 앱 구동 시 복

호화되어 실행된다. 프로텍터와 유사하지만, 원본을 수정하지 않는다는 차이가 있다. 즉, 프로텍터는 수정 방식에 안전성을 두고, 패커는 암호 알고리즘에 안전성을 둔다. 현재 모바일의 경우 오버헤드와 원본 수정의 까다로움으로 인해 대다수 패커 방식을 채용하고 있다. 대표적인 서비스로 Ali, Ijiami, 360, Pangxie, Bangcle, HoseDex2Jar, Tencent, Baidu 등이 있다.

2.2 상용 서비스의 취약점

안드로이드 앱이 다른 플랫폼과 비교해 상대적으로 분석하기 쉬운 이유로 운영체제가 오프 소스인 점이 있다. 2015년 Yueqian Zhang 등의 연구에서 이 점을 이용하여, 보호된 앱을 따로 분석하지 않고도 원본을 그대로 추출하는 방법을 고안하였다[13]. 이 방식은 안드로이드 VM이 특정 클래스를 로드할 때 해당 클래스의 메모리 지점을 파라미터로 받아오는 특징을 이용한다. VM 수정을 통해, 안드로이드 두 가지 구동 모드(달빅,ART)에서, 원본 앱을 추출하는 데 성공하였다. 다만, 모든 클래스가 로드될 때까지 계속 어플리케이션을 실행시켜야하는 까다로운 점이 있다.

2017년 Slava Makkaveev 등의 연구에서도 안드로이드 라이브러리 수정을 통하여 원본 앱을 추출하는 기법을 선보였다[12]. 이 방식은 앱이 최초로 수행될 때, 수행되는 최적화(또는 컴파일) 과정의 주요 함수를 수정한 것으로, 파라미터로 주어진 메모리 지점의 원본 값을 외부로 출력하여 추출한다. 달빅 모드에서는 OpenAndReadMagic 함수를, ART 모드에서는 DexFile 생성자를 수정하는 것으로, 이 방식은 Yueqian Zhang 등의 방식보다 더 간단히 추출할 수 있음을 보여주었다. 이처럼 외부 환경을 수정하는 방식은 기존 상용 앱 보호 서비스의 대다수를 우회할 수 있는 것으로 조사된다[12].

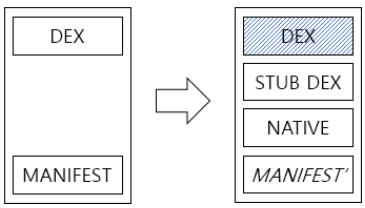

Fig. 1. Applying packing on android app

2.3 기타 문제점

이밖에도 안드로이드 앱은 난독화하기에도 어려운 문제점이 있다. 우선 바이트코드의 경우, 해당 포맷이 공개되어 있고 사람이 읽을 수 있는 형태로 출력해주는 도구가 많이 공개된 상태이다[14-17]. 따라서 손쉽게 어플리케이션의 의미를 파악할 수 있으며, 수정 또한 간단하게 이뤄질 수 있다.

네이티브코드에서도 문제가 존재한다. 90% 이상의 안드로이드 디바이스는 RISC 머신인 ARM 아키텍처를 사용한다. 이 아키텍처의 명령어는 16비트 또는 32비트의 고정 길이로, 명령어 중첩(instruction overlapping) 기법과 정크 바이트(junk byte)를 사용하는 기존 안티 디스어셈블리 기법을 적용할 수 없다[18,19].

III. 배경 지식

패커는 현재 가장 많이 사용되는 방식이며, 본 장에서는 이 방식의 보호 구조를 간략히 소개한다.

3.1 패커의 보호 구조

실행 압축 기법을 사용하여 안드로이드 어플리케이션을 보호하는 방법은 기본적으로 Fig. 1.의 구조를 따른다. 원본 바이트코드(DEX)는 암호화되며, 그 자리를 스텁 바이트코드(STUB DEX)가 대신하게 된다. 그리고 매니페스트(MANIFEST)가 수정되어 스텁 바이트코드를 가리키게 된다. 스텁 바이트코드의 핵심 구현 파트는 네이티브코드로 구현되며, 이는 JNI(Java Native Interface)를 통해 연결된다. 따라서 네이티브 메소드가 저장되어 있는 공유라이브러리 파일(NATIVE)이 추가로 삽입된다.

이렇게 재구성된 어플리케이션의 실행 흐름은 Fig. 2.와 같다. 수정된 매니페스트에 의해 스텁 바이트코드가 수행된다. 스텁 바이트코드는 공유라이브러리를 로드하고 JNI를 통해 일련의 네이티브 메소드를 호출한다. 이 과정에서 실시간 보호 모듈이 실행되게 된다. 그런 다음 네이티브코드에서 암호화된 원본 바이트코드를 복원한다. 마지막으로 복원된 바이트코드를 로드한 뒤 실행함으로써, 원본의 실행흐름을 이어가게 된다.

이 구조의 안전성은 새로 추가된 네이티브코드의 복잡성에 달려있다. 네이티브코드가 바이트코드보다

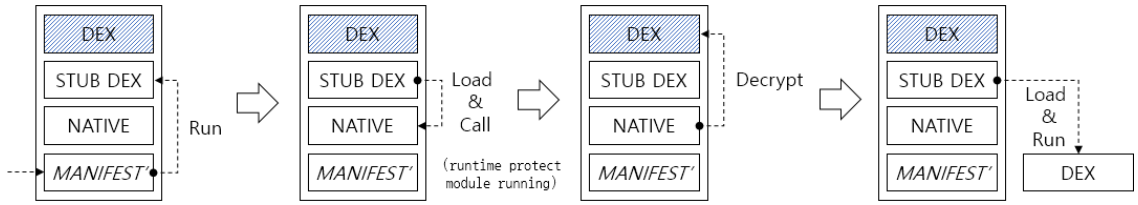

Fig. 2. Execution flow of reconstructed android app

가독성이 상대적으로 더 낮기 때문에, 스텁 바이트코드의 주요 복호화 루틴은 JNI를 통해 네이티브코드로 구현된다. 그리고 해당 루틴은 난독화나 실시간 보호 모듈 등으로 보호받게 된다. 이런 식으로 바이트코드 앱의 보안 한계를 극복한다.

3.2 보호 구조의 안전성

패커의 보호 구조는 난독화, 실시간 보호 모듈, 비밀키, 원본 이렇게 네 가지 요소로 구성되며, Fig. 3.과 같은 의존성 구조를 띤다. 우선 원본 바이트코드는 암호 알고리즘으로 암호화되어 있으므로 비밀키에 안전성이 달려있다. 또한, 실시간 보호 모듈 동작 아래에 복호화 및 구동되므로, 실시간 보호 모듈에도 의존성을 가지게 된다. 즉, 암호 알고리즘을 통해 정적으로 보호받고 실시간 보호 모듈을 통해 동적으로 보호받는다.

구조상 비밀키는 앱 자체에 포함되어야 하므로, 이를 난독화로 감춰 놓는다. 그리고 동적분석을 통한 노출을 막기 위해, 실시간 모호 모듈로 보호한다. 따라서 비밀키는 실시간 보호 모듈과 난독화에 의존성을 가진다. 비밀키도 원본과 동일하게 정적 및 동적 모두로부터 보호받는다. 실시간 보호 모듈은 원본 그대로 노출되게 되면 쉽게 우회될 수 있으므로, 이를 보호하기 위해 난독화를 적용한다. 앞선 원본 바이트코드, 비밀키와 다른 점은 난독화를 통해서만 보호받는다는 점이다.

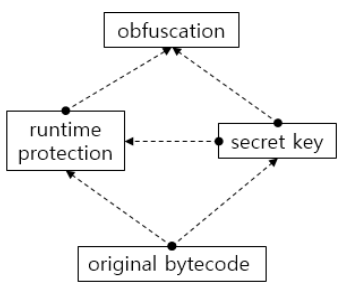

Fig. 3. Security dependancy of packer

우리의 조사에 따르면, 실행 압축을 이용한 상용 앱 보호 서비스[2-7]는 기본적으로 Fig. 3.과 같은 보호 구조를 가지며, 이 안에서 구현에 따라 가변성을 띤다. 하지만 일부 서비스에서 이 구조에 입각하여 체계적으로 설계하지 않은 것이 확인되었고, 아직도 직감에 의해 설계되는 것을 볼 수 있었다.

3.3 실시간 보호 기법

역공학은 대다수 동적분석을 통해 이뤄지기 때문에, 앱 보호 구성요소 중 실시간 보호 기법이 가장 중요하다 볼 수 있다. 이는 크게 두 가지 유형으로 분류할 수 있으며, 다음은 보편적으로 사용되는 실시간 보호 기법의 사례이다.

- 단발성 검사 유형: 환경 정보, 프로세스 상태 등과 같이 변칙성이 낮은 유형을 찾기 위한 검사
  - VM 여부 검사
  - 디버거 프로세스 검사
  - ptrace 연결 유무 검사
  - 코드 무결성 검사
  - 루팅 여부 검사
- 지속성 검사 유형: 비정규적으로 발생하는 이벤트로, 단발성 검사로 확인하기 어려운 유형을 찾기 위한 검사
  - 파일시스템 이벤트 감시
  - 프로세스의 쓰레드 상태 감시
  - 안티 JDWP(Java Debug Wire Protocol)

기존 앱 보호 서비스는 이런 기법들을 복호화 루틴에 독립적으로 분산 배치해 놓는다. 그리고 이 밖에도 메모리상에 로드된 원본 데이터를 주기적으로

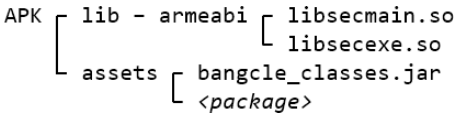

Fig. 4. Added files after applying bangcle

변경하거나, 시그니처(signature)로 사용될 수 있는 바이트 시퀀스를 지우는 형태의 방법도 존재한다.

IV. 상용 서비스의 동작 구조

본 장에서 분석할 방클은 2010년부터 현재까지 서비스되어 온 대표적인 안드로이드 앱 보호 서비스로, 지속적인 업데이트와 다양한 보호 기법이 장착된 패커 서비스이다. 이를 통해 보호 구조의 문제점과 체계적인 보호 구조의 필요성을 알아본다.

4.1 기본 구조

방클 서비스를 적용할 경우(2016년도 2월 버전 기준), Fig. 4.와 같이 네 개의 주요 파일이 안드로이드 패키지에 추가된다. 이는 두 개의 공유라이브러리 파일, 패키지명의 실행 파일, jar 파일로 구성된다.

bangcle_classes.jar은 원본 바이트코드가 압축 및 암호화된 파일이다. libsecmain.so는 네이티브 메소드가 정의되어있는 공유라이브러리 파일로, 난독화가 적용되어 있다. 〈*package*〉 파일은 실행 파일로 특정 네이티브 메소드에서 이를 실행한다. 그리고 이 파일은 난독화가 되어 있지 않다. 마지막으로 libsecexe.so는 〈*package*〉 파일에서 로드하는 공유라이브러리 파일로 실시간 보호 모듈이 들어 있다. 그리고 이 파일은 난독화되어 있다.

방클로 보호된 앱은 구동 즉시 libsecmain.so 파일을 로드한다. 그 뒤 스텁 바이트코드에 등록된 13개의 네이티브 메소드를 순차적으로 수행시키면서 원본을 복원 및 구동시킨다. 이 과정에서 총 세 개의 프로세스가 생성된다. Fig. 5.는 수행 흐름과 프로세스와 파일의 관계를 도식화한 것이다.

세 개의 프로세스는 각각의 임무를 수행한다. 첫 번째로, 바이트코드와 JNI로 연결된 프로세스는 원본을 복원하고 이를 로드시키는 임무를 수행한다. 두 번째로, 자식 프로세스는 비밀키를 복원하고 이를 부모에게 전달하는 임무를 수행한다. 마지막으로 손자 프로세스는 실시간 보호 모듈을 수행한다. 이 실시간 보호 모듈은 자식 프로세스가 부모 프로세스에게 비밀키를 전달하기 전에 생성 및 실행된다.

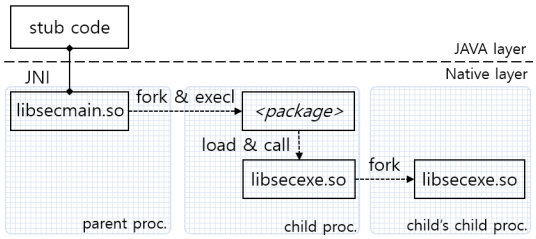

Fig. 5. Bangle's unpacking flow

4.2 정적분석 보호 구조

방클의 정적분석 보호 기법은 libsecmain.so 파일에 적용되어 있다. 첫째로, 섹션 헤더 정보가 지워져 있어 파일의 섹션 정보를 얻지 못한다. 둘째로, 파일의 코드 섹션과 데이터 섹션이 압축 및 셔플링되어 있어 복원 루틴이 수행되기 전에는 올바른 값을 보지 못하게 되어 있다. Fig. 6.의 왼쪽 그림은 해당 파일이 메모리에 로드된 직후를 나타낸다. 메모리 하단에 위치한 복원 루틴이 수행되기 전까지는 정상적인 섹션 데이터를 볼 수 없다.

복원 루틴에도 정적분석 보호 기법이 적용되어 있다. 여기에는 어셈블리어 수준의 난독화, 안티 디컴파일 등이 적용되어 있다. 또한, 동적분석을 방해하게끔 브레이크포인트를 감지하여 잘못된 수행 흐름을 만드는 기법이 적용되어 있기도 하다. 복원 루틴이 완료되면 Fig. 6.의 오른쪽 그림과 같이 정상적인 코드 및 데이터 섹션이 나타나게 된다. 특이점으로, 복원된 코드 및 데이터 섹션에는 정적분석 방해 기법이 적용되어 있지 않다.

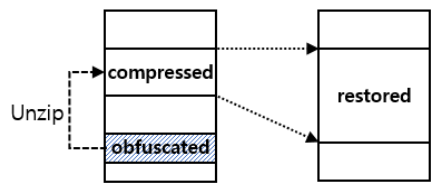

Fig. 6. Memory after loading 'libsecmain.so'

4.3 동적분석 보호 구조

방클의 동적분석 보호 코드는 libsecmain.so 파일의 복원 루틴이 완료된 다음부터 시작한다. 복원 루틴 완료 직후에 〈*package*〉 파일이 수행되며, 이

를 통해 두 개의 프로세스가 생성된다. 이 중 자식 프로세스에는 단발성으로 검사 가능한 루틴이 배치되어 있으며, 손자 프로세스에는 주기적으로 검사해야 하는 루틴이 배치되어 있다.

자식 프로세스는 〈*package*〉 파일을 수행하지만, 이 파일은 libsecexe.so를 로드하고 엔트리 함수를 호출하는 것이 전부다. 주요 코드는 libsecexe.so에 존재한다. 엔트리 함수의 수행 과정은 다음과 같다.

① 부모 프로세스에게 ptrace 연결 (실패 시 6번)
① 새로운 프로세스 생성 (포크 수행)
① 자식 프로세스에게 ptrace 연결 (실패 시 6번)
① 비밀키 복원 후 부모 프로세스에게 전달
① 종료 루틴관련 쓰레드 생성 후 블록 상태 진입
① 모든 프로세스에게 종료 시그널 전달

앞선 과정의 5단계까지 수행되고 나면 Fig. 7.과 같은 구조를 가지게 되며, 자식 프로세스가 부모와 손자 프로세스에 대해 ptrace 접근 권한을 가지게 되는 구조이다. 이를 통해 외부 프로세스가 부모와 자식 프로세스의 접근 권한을 획득하는 것으로부터 보호한다. 즉, 원본 복원 과정과 실시간 보호 작업을 보호한다. 자식 프로세스는 이 보호 구조가 완성된 이후에는 별도의 보호 작업을 수행하지 않는다.

각 프로세스는 비정상적인 행위가 발생하였을 때, 바로 종료될 수 있도록 종료 루틴을 가지는 쓰레드가 존재한다. 이 쓰레드는 부모 또는 자식 프로세스와 파이프를 통해 이상 신호를 전달받는데, 보통 상황에서는 비어있는 파이프를 읽음으로써 블록 상태에 있는다. 이상이 감지될 경우 각 프로세스는 종료용 파이프에 임의의 값을 쓰게 되며, 블록된 쓰래드가 깨어나면서 종료 루틴이 수행된다.

정리하면, 방클의 보호 과정은 libsecmain.so 파일의 생성자 함수(INIT 함수)를 통해 시작된다. 생성자 함수의 주요 역할은 코드 및 데이터 섹션의

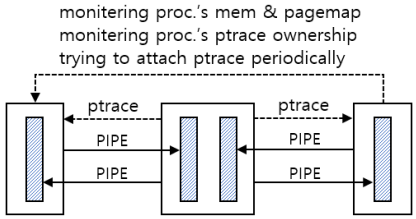

Fig. 7. Realtime protection structure

복원과 〈*package*〉 파일의 구동이다. 그리고 〈*package*〉 파일의 역할은 libsecexe.so 파일을 사용하여 Fig. 7.의 보호 구조를 생성하는 것이다. 보호 구조가 완성된 다음에는 JNI_OnLoad 함수를 통해 네이티브 메소드를 등록하며, 스텁 코드의 흐름에 따라 원본을 복호화 및 구동하게 된다. 방클은 이와 같은 방식으로 원본 앱을 보호한다.

4.4 보호 구조 취약점

방클의 보호 구조는 Fig. 8.과 같으며, 기본적으로 Fig. 3.의 구조를 따른다. 정확히는 〈*package*〉 파일을 통해 생성되는 보호 구조는 Fig. 3.과 동일하지만, libsecmain.so을 포함할 경우 복잡한 보호 구조를 띠게 된다. 여기서 주의 깊게 봐야 될 사항은 libsecmain.so에서 직접 처리해도 될 작업을 〈*package*〉 파일로 분리해서 수행하는 점이다.

〈*package*〉 파일을 둔 이유는 두 가지 이유 때문으로 추정된다. 첫째로, 보호 구조의 구현을 메인 루틴과 분리하여 구현하기 위해서이다. 이는 보호 모듈이 libsecexe.so에 있는 것으로 알 수 있다. 둘째로, 원본 복호화 루틴이 있는 부모 프로세스의 이미지를 execl을 통해 새 프로세스 이미지로 바꾸기 위한 목적으로 보인다. libsecmain.so에서 포크를 통해 바로 libsecexe.so를 로드해도 되는 것을 하지 않은 것으로 보아 알 수 있다.

보호 구조의 결함은 〈*package*〉 파일을 통해 두

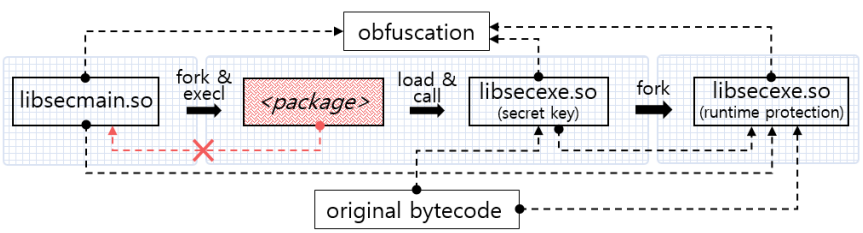

Fig. 8. Security dependency of Bangcle

번째 프로세스를 구성하는 데 있다. 〈*package*〉 파일은 난독화되어 있지 않은데, 이는 〈*package*〉 파일의 안전성은 libsecmain.so 파일에 의존된다고 판단한 것으로 보인다. 하지만, 실제 그렇지 않으며 〈*package*〉 파일의 수정을 통해 공격자는 두 번째 프로세스 초기 과정부터 나머지 과정을 쉽게 파악할 수 있다. 결과적으로 보면, 이 결함을 통해 방클의 보호 구조가 연쇄적으로 무너지게 된다.

해당 결함을 통해 가장 먼저 파훼 되는 지점은 libsecexe.so 파일의 난독화이다. 〈*package*〉 파일을 통해 엔트리 함수를 알 수 있고, 결함을 통해 정상적으로 로드된 libsecexe.so 파일을 동적분석으로 볼 수 있기 때문이다. 그리고 이를 통해 세 번째 프로세스를 차단함으로써, 여기에 걸려있는 의존성이 끊어진다. 결국은 원본 바이트코드와 자식 프로세스에 걸려있는 모든 의존성이 끊어지게 된다. 이뿐만 아니라 부모 프로세스의 이미지를 원형 그대로 확인할 수 있으므로 부모 프로세스의 난독화도 깨지게 된다.

4.5 문제점 분석

4.4절에 언급한 취약점이 발생할 수 있었던 주요 원인은 기본 보호 구조(Fig. 3.)의 복잡한 의존성 때문으로 판단된다. 이 구조는 강건한 것으로 보이지만, 의존성이 두 개 이상 얽혀있는 요소들 때문에 구현 시 고려사항이 많아진다. libsecmain.so 파일의 난독화가 〈*package*〉 파일로 전이되는 것처럼 착각한 이유도 이런 맥락이라 볼 수 있다. 그리고 보호 관점에서는 의존성이 하나라도 끊어지면 보안이 무너지므로, 여러 의존성을 가지는 것은 올바른 구조라 보기 힘들다.

두 번째 요인으로, 감추고자 하는 코드와 난독화 대상을 분리시킨 문제점이 있다. 실제 감추고자 하는 대상은 실시간 보호 모듈을 실행하는 코드이다. 하지만 난독화가 적용되는 지점은 이 코드를 복원해주는 코드이다. 따라서 공격자는 이 지점을 동적분석을 통해 이동하기만 하면 동적분석 방어 코드를 볼 수 있고, 이를 쉽게 우회할 수 있다.

세 번째 요인으로, 각 보호 모듈이 독립적으로 수행되며, 종류별로 하나씩만 배치되는 문제점이 있다. 각 모듈이 독립적으로 배치되어 있어, 수행과정을 모르더라도 해당 모듈을 차단하기만 하면 쉽게 우회할 수 있다. 그리고 각 모듈이 한 번씩만 배치되어 있어, 한번 발견한 보호 기능은 그다음부터 추가 검증하지 않아 쉽게 우회할 수 있다.

네 번째 요인으로, 무결성 검증 작업이 올바르게 수행되지 않는 문제점이 있다. libsecmain.so 파일에 코드 및 데이터 섹션이 압축되어 있어 복원 과정 자체를 무결성 검증 작업으로 두었다. 즉, 수정이 있을 경우 올바른 복원 작업이 이뤄지지 않는 것만 염두 하였다. 따라서 〈*package*〉 파일과 복원된 이후에 메모리상의 코드 수정은 따로 검증하지 않아 코드 분석을 쉽게 수행할 수 있다.

V. 가정 및 요구사항

강건한 앱 보호 구조를 설계하기 위한 첫 단계로 공격자 및 환경에 대한 가정을 세우고 그에 맞는 요구사항을 도출하기로 한다.

5.1 공격자 가정 사항

본 절에서는 외부 공격에 대한 가정을 세운다. 가정은 지금까지 알려진 공격 사례를 반영하며, 총 다섯 가지로 다음과 같다.

- 공격자는 난독화로 보호된 코드를 쉽게 파악하지 못함
- 공격자는 코드 시작 지점을 알 수 있으며, 동적 디버깅을 통해 순차적으로 분석할 수 있음
- 공격자는 코드를 수정하여 동작 과정을 변경할 수 있음 (메모리, 파일)
- 공격자는 실행 환경을 수정하여 분석할 수 있음 (에뮬레이터, 프로세스, 라이브러리)
- 공격자는 네이티브코드의 임포트(import)되는 함수를 후킹(hooking)할 수 있음
- 공격자는 하드웨어의 조작을 통해 정보를 수집하거나 수정할 수 있음

첫 번째 가정은 정적분석에 대한 것이다. 아직 네이티브코드에 대한 의미를 자동으로 추론하는 기법은 없는 것으로 알고 있다. 물론 데이터 흐름 분석(data-flow analysis), 기호 실행(symbolic execution), 오염 분석(taint analysis)과 같은 분석 기법이 있지만 네이티브코드에서는 한계가 있고, 의미 있는 결과를 얻는 것은 어렵다. 따라서 이 가정은 타당하다 볼 수 있다.

두 번째 가정은 분석 시작 지점에 관한 것이다. 운영체제는 항상 어플리케이션을 실행시킬 수 있어야 하며, 어플리케이션은 그 파일 포맷을 따라야 한다. 따라서 안드로이드의 생명주기, ELF 포맷, 파일 로드 메커니즘을 정확히 숙지만 한다면, 코드가 어디서 수행되는지 아는 것은 어렵지 않다. 그런 이유로 시작 지점을 감추는 것은 어려우며, 공격자는 그 지점부터 항상 분석한다고 가정해야 한다.

세 번째 가정은 동적분석 방법에 관한 것이다. APKTool, IDA Pro, FRIDA와 같은 도구는 바이트코드와 네이티브코드 수정을 쉽게 도와준다. 또한, 동적 디버깅 도중에도 레지스터의 값이나 메모리상의 값을 수정하면서 동작시킬 수 있다. 따라서 시점과 위치에 상관없이 항상 모든 값들이 수정될 수 있다고 가정해야 한다.

네 번째 가정은 실행 환경 수정에 관한 것으로, 어플리케이션이 동작하는 환경을 수정하는 사항이다. 안드로이드는 오픈 소스이므로, 공격자는 외부 라이브러리나 안드로이드 VM(virtual machine) 등을 분석에 유리하도록 수정할 수 있다. 이런 공격 방식은 이미 2015년도 Yueqian Zhang 등[13]과 2017년도 Slava Makkaveev 등[12]이 성공적으로 수행하였기에, 이 가정은 타당하다고 볼 수 있다.

다섯 번째 가정은 동적분석 방법 중 잘못된 외부 입력에 관한 것이다. ELF 로드 및 구동 시, 외부 라이브러리 함수가 재배치된다. 이 때 공격자는 동일 이름의 가짜 심볼을 가지는 라이브러리를 배치함으로써, 공격자의 함수로 재배치시킬 수 있다. 네 번째나 다섯 번째 가정과의 차이점은 패키지 구성물이나 실행 환경에 수정하지 않는 점이다. 즉, 일반적인 무결성 검증을 통해서는 검사하지 못한다. 아직 공개적으로 발표되지는 않았으나 충분히 공격될 수 있는 사항이므로 가정해야 한다.

여섯 번째 가정은 하드웨어를 통한 정보 조작 및 수집에 관한 것이다. 콜드 부트(cold boot) 공격[20]과 같이 하드웨어의 직접적인 조작을 통해 디램의 정보를 얻는 것은 현실성이 낮다. 하지만, 가능성은 열려 있으므로 메모리상의 정보가 그대로 노출될 수 있음을 가정해야 한다. 그리고 2016년에 발표된 Drammer 공격[21]은 하드웨어를 간접적으로 조작할 수 있다. 이는 디램의 특정 비트를 지속적으로 접근하여 플립이 되도록 하는 공격이다. 이를 통해 루트권한 획득과 같은 공격이 가능하다. 따라서 하드웨어의 조작을 통한 정보 노출의 가정은 타당하다고 볼 수 있다.

5.2 앱 보호 목표 및 요구사항

앱을 역공학으로부터 완벽히 보호하는 것은 불가능에 가깝다. 디바이스가 보호 루틴을 항상 정상 수행해야 하므로, 공격자도 그 루틴을 따라가면서 분석할 수 있기 때문이다. 따라서 공격자가 보호 루틴을 우회하지 못하고 정상 루틴을 통해서만 분석할 수 있게 만드는 것을 목표로 세워야 한다. 다음은 해당 목표를 위해 요구되는 사항을 정리한 것으로, 방클의 문제점과 공격자의 가정 사항이 반영되었다.

- 보호 의존 관계가 단순한 구조
- 보호 모듈이 서로 의존적인 구조
- 코드 크기에 분석 시간이 비례되는 구조
- 결과는 동일하나 그 과정이 가변적인 구조
- 원본 바이트코드가 완벽히 노출되지 않는 구조

첫 번째 요구사항은 의존성의 단순화를 통해 보호 구조의 결함을 피하기 위한 것이다. 단순함과 복잡함 사이에 어느 것이 더 좋은 구조인지는 쉽게 판단할 수 없다. 하지만 현재의 방식은 난독화를 통해 실시간 보호 기법을 보호하고, 실시간 보호 기법을 통해 원본을 보호하는 구조이다. 따라서 이 구조를 최대한 따르는 구조로 가는 것이, 예측하기 어려운 결함을 피할 수 있는 길이라 생각된다. 방클의 경우를 봐도, 이와 같은 구조를 따르지만, 용도에 따라 파일을 나누면서 보호 구조에 결함이 생기게 되었다. 따라서 단순하고 명확한 보호 관계가 요구된다.

두 번째 요구사항은 각 구조에 의존성을 만들어 우회를 차단하기 위한 것이다. 앞선 방클 서비스를 보면, 손자 프로세스의 경우, 부모와 자식 프로세스와는 독립적으로 돌아간다. 따라서 공격자는 손자 프로세스 생성을 차단만 하면 그 부분을 신경 쓰지 않아도 된다. 이런 상황을 막기 위해 각 보호 모듈들이 개별적으로 우회가 불가능하도록 상호 의존적인 구조가 요구된다.

세 번째 요구사항은 단순히 코드 크기의 증가를 의미하는 것이 아니다. 코드 크기가 분석의 증가와 항상 비례하는 것이 아니기 때문이다. 예를 들어, 암호 연산의 경우 연산양은 많지만 결국 마지막 결과(예로, 라운드 단위) 값만이 메모리에 저장된다. 따

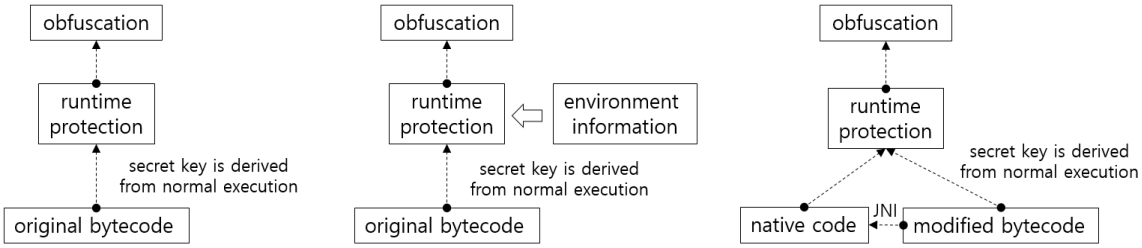

Fig. 9. The proposed security dependancy of hardening service

라서 공격자는 그 많은 연산을 지켜볼 필요가 없다. 즉, 메모리로부터의 읽기 및 저장, 함수 호출 등이 적절히 있어야 실질적 분석 비용이 추가된다. 이 요구사항은 실제 분석을 요구하는 코드의 증가를 의미한다.

네 번째 요구사항은 코드의 복잡성을 높이기 위한 것이다. 결과적으로 같은 내용을 계산하더라도 매 수행 다른 루틴으로 진행되어야, 분석을 더디게 할 수 있다. 실례로, 방클을 포함한 대다수의 보호 서비스들은 일관적인 코드 수행으로 인해, 단순한 시행착오를 통해 코드의 의미를 쉽게 파악할 수 있었다. 또한, 분석의 비용 증가를 위해 무작정 코드의 크기를 늘리기에는 오버헤드 문제가 있다. 따라서 복잡도를 높일 수 있는 요소가 요구된다.

다섯 번째 요구사항은 예상하지 못한 공격을 통해 원본이 노출되는 경우를 위한 것이다. 역공학 기술은 지속적으로 발전하기 때문에 보호 구조는 완벽할 수 없다. 따라서 공격자가 원본 바이트코드를 얻었다 하더라도 그 자체만으로는 완전한 형태가 아니거나, 정상적으로 수행할 수 없도록 할 필요가 있다.

VI. 강건한 앱 보호를 위한 구조와 원칙

본 장에서는 앱 보호 구조와 정적 및 동적분석의 올바른 대응을 위한 기본 원칙을 제안한다.

6.1 제안하는 앱 보호 구조

현재 대다수의 상용 앱 보호 서비스들은 실행 압축 기법을 이용하며, Fig. 3과 같은 보호 구조를 따른다. 본 연구에서 제안하는 보호 구조도 이를 기반으로 한다. 하지만, 강건성을 높이고 의존성 관계의 단순화를 위해 일부분 수정한다. Fig. 9는 본 연구에서 제안하는 보호 구조이다. 이 중 왼쪽은 기본형이며, 가운데는 특수 환경 정보가 있는 경우의 파생 구조이다. 그리고 오른쪽은 원본을 수정하거나 네이티브 메소드가 정의된 경우의 파생 구조이다.

제안하는 보호 구조의 핵심은 실시간 보호 모듈의 의존성을 통한 비밀키 생성이다. 실행 압축 기법을 사용할 경우 수행 코드 속에 비밀키가 포함되어야 한다. 기존 서비스들은 난독화 코드 속에 감춰 놓고 실시간 보호 모듈을 통해 보호하지만, 제안하는 구조는 실시간 보호 모듈의 정상적인 실행 결과를 통해 비밀키를 도출하는 방식이다. 이를 통해 비밀키가 코드에 직접적으로 포함되는 상황을 제거한다.

제안한 보호 구조에서 올바른 비밀키 획득은 실시간 보호 모듈 전체의 정상 동작을 의미한다. 각 보호 모듈의 의존성이 올바르게 연결되어 있다면, 공격자는 일부 모듈을 단순히 차단하거나 해당 모듈의 정확한 출력 값을 알지 못하면 올바른 비밀키를 알지 못한다. Fig. 10.은 기존 보호 모듈의 배치 예시이며, 각 모듈이 독립적인 것을 알 수 있다. 반면 Fig. 11.은 모듈 간에 의존성을 가지며 비밀키는 그것들의 최종결과물로 유도되는 것을 알 수 있다.

Fig. 11.의 두 번째 구조는 특수 환경 정보가 주어지는 경우로, 첫 번째 구조의 파생 구조이다. 실례로, 특정 디바이스에서 발급하는 토큰이나 하드웨어 지문 등을 비밀키의 입력 값으로 활용하는 경우가 있다. 제안하는 구조에서는 이 정보를 실시간 보호 모듈의 입력 값으로 사용한다.

환경 정보를 비밀키로 직접 사용할 경우 새로운

One-time inspection	Periodic inspection

if rt_check_1():
  abort()
...
if rt_check_3():
  abort()
...
if rt_check_5():
  abort()
	LOOP:
	  if rt_check_2():
	  abort()
	  if rt_check_4():
	  abort()
	  sleep(10)

Fig. 10. Example of general approach

의존성이 생기게 된다. 따라서 공격자가 환경 정보를 얻을 수 있고 그 정보가 사용되는 지점이 노출될 경우, 보호 구조는 바로 무너지게 된다. 따라서 이 정보를 실시간 모듈의 입력 값으로 사용하여 원본이 실시간 보호 모듈에만 의존되도록 만든다. Fig. 11.의 예시에서는 rt_check_1 모듈의 입력 값으로 사용할 수 있다. 이럴 경우 공격자가 환경 정보를 모르면 복원을 할 수 없으며, 알고 있다 하더라도 보호 모듈을 모두 파훼해야 한다.

Fig. 11.의 세 번째 보호 구조는 네이티브 메소드가 정의되어 있는 경우로, 첫 번째 구조의 파생 구조이다. 이 구조는 메모리 덤프나 알려지지 않은 공격을 대비하기 위해, 바이트코드로 구현된 메소드 일부를 네이티브코드로 분리시키거나 추가 검증 루틴을 삽입한 것을 의미한다. 즉, 이 보호 구조는 기존 프로텍터 방식에 해당한다.

세 번째 구조에서 주의할 점은 JNI로 연결된 네이티브코드가 실시간 보호 모듈이 아니라는 것이다. 이 코드는 Slava Makkaveev, Yueqian Zhang이 고안한 공격[12,13]과 같이 기존에 알려지지 않은 방식으로 바이트코드가 탈취되더라도 그것을 무용지물로 만들기 위해 존재한다. 주요 메소드를 분리시켜 바이트코드만으로는 전체 코드를 못 얻게 하거나, 검증 루틴을 삽입하여 검증 루틴 없이는 정상 동작이 안 되게 하는 방식을 의미한다.

One-time inspection	Periodic inspection

a = rt_check_1()
…
c = rt_check_3(a,b)
…
key = rt_check_5(c,d)
	// Results up to here
	b = rt_check_2(a)
	// Results up to here
	d = rt_check_4(c)

Fig. 11. Example of making dependancy

6.2 정적분석 대응 원칙

정적분석으로부터 보호하기 위한 기법은 Table 1. 첫 번째 열에 나열되어 있다. 이들 중 기본이 되는 작업은 컴파일된 파일 속 불필요한 정보를 제거하는 것이다. 다음은 이에 관한 원칙이다.

- **섹션 정보 제거:** 네이티브 메소드를 사용하려면, 공유라이브러리 타입 ELF 파일이 필요하다. 안드로이드는 해당 타입의 ELF 파일을 로드할 때, 프로그램 헤더 정보만 사용한다. 따라서 섹션 헤더 정보를 지움으로써, 파일을 정확히 파악할 수 없게 한다.
- **임포트(import) 줄이기:** 외부 함수 및 변수를 사용하게 될 경우, 재배치 정보가 생기게 된다. 이 정보를 통해 공격자는 사용하는 함수와 구조를 예측하거나 후킹하여 분석할 수 있다. 따라서 직접적인 사용보다는 dlopen, dlsym 함수를 통해 주소를 구하거나, '/system/lib' 하위 경로의 라이브러리 파일과 프로세스의 maps 정보를 이용하여 직접 함수 위치를 구한다. 이를 통해 불필요한 임포트 정보를 줄일 수 있다.
- **익스포트(export) 제거:** 명시적으로 익스포트하지 않아도 컴파일 및 링크의 결과물로 심볼 정보가 남을 수 있다. 공격자는 이 정보를 통해 내부 구조의 힌트를 얻을 수 있으므로, 불필요한 심볼은 지운다. 그리고 구현상 어쩔 수 없을 경우 심볼의 이름을 임의의 문자열로 치환하여 사용한다.
- **문자열 감추기:** 코드에 문자열을 사용하게 되면 문자열 테이블에 해당 값이 들어가게 된다. 공격자는 이를 통해 프로그램 구조를 파악할 수 있

Table 1. Kind of protection technique

Anti Static Analysis	Anti Dynamic Analysis	Integrity Verification
Increasing code amount	Checking debugger presence	App forgery verification
Anti-disassembly	Log output prevention	Memory permissions verification
CFG restoration prevention	Memory dump prevention	GOT table verification
Anti-decompile	Anti-emulators	External library forgery verification
Code section encryption	Software breakpoint detection	Custom Firmware verification
String encryption	Device rooting verification	
Modifying and deleting unnecessary symbol information		

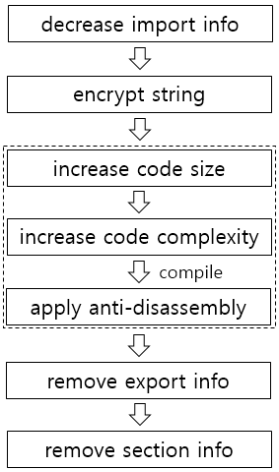

Fig. 12. Process of applying static protection

다. 문자열 테이블을 감추는 것은 어려우므로, 사용하는 문자열은 암호화하여 사용 직전 복호화하고 사용 직후 메모리에서 지우는 방식으로 구현해야 한다.

파일의 불필요한 정보를 지운다 하더라도, 코드가 감춰지는 것은 아니다. 따라서 코드를 난독화를 통해 감출 필요가 있다. 다음은 이에 관한 원칙이다.

- **난독화 적용 방법**: 난독화는 코드의 크기를 증가시키는 방향, 코드의 구조를 복잡하게 만드는 방향, 코드가 정적으로 표현되지 못하게 하는 방향으로 정리할 수 있다. 이 세 가지 종류의 난독화를 모두 적용해야 강건한 난독화 적용이라고 볼 수 있다. Fig. 12. 점선 박스는 본 논문에서 제안하는 난독화 적용 과정이다.

- **난독화 적용 범위**: 최소한 실시간 보호 모듈과 관련된 코드는 모두 난독화해야 한다. 보호 구조상 난독화는 실시간 보호 모듈이 안전하게 실행될 수 있도록 도와주는 역할이기 때문이다. 방클의 경우처럼, 초반 섹션 데이터를 복원하는 루틴에 한정적으로 적용하는 것은 위험하다.

- **코드량 증가**: 코드량 증가를 위해 삽입되는 코드는 정적분석에 강건하고 실제 수행되는 것이어야 한다. 정적분석에 강건하다는 것은 최소한 컴파일러 최적화 기법에 제거되지 않을 정도를 의미한다. 정적분석에 강건하기 위해서는 분석기법의 한계를 보면 되는데, 메모리 읽기 및 쓰기 작업이 많은 코드, 함수 호출이 많은 코드, 포인터 연산이 많은 코드 등이 분석의 오탐을 발생시킨다. 그리고 실제로 수행되지 않는 코드를 넣는 것은 효과가 확연히 낮아지게 된다. 적용되는 부분이 실시간 보호 모듈이 실행되는 부분이므로 공격자는 동적으로 실행되는 부분 위주로 보기 때문이다. 따라서 불명료 서술어(opaque predicate)를 이용하는 난독화[22]는 정적분석에는 효과적이나 동적분석을 병행한 정적분석에는 효과적이지 않다.

- **코드 복잡도 증가**: 코드의 복잡도는 무결성 검사와 실시간 보호 기법을 감출 수 있는 방향으로 높여야 한다. 무결성 검사는 읽기, 쓰기, 비교 작업이 대다수를 이루며, 실시간 보호 기법은 시스템 API 호출, 라이브러리 호출, 파일 열기 등이 대다수를 이룬다. 따라서 이와 같은 작업을 산재하여 이 작업을 찾기 어렵게 해야 하며, 그 흐름 또한 여러 갈래로 넓게 흩뜨려 놓아야 한다. 단순히 순환 복잡도(cyclomatic complexity)를 높이는 것은 의미 없다.

- **명령어 디스어셈블리 방해**: 주요 부분에만 적용하는 것이 아닌 보호와 관련된 모든 코드에 적용해야 한다. 이를 통해 공격자가 함수의 CFG를 파악하지 못하게 만들고, 동적분석으로도 지엽적인 코드만 구하게끔 만든다. 주요 부분에만 적용하는 것은 효과적으로 보이지만 적용 부분이 쉽게 드러나기 때문에 오히려 악효과를 낸다. 그리고 선형 분석(linear sweep)과 재귀 분석 방식(recursive traversal)에 대한 저지기법을 모두 적용해야 한다. 선형 분석만 저지하면 재귀 분석으로 모든 코드가 노출된다. 반대로 재귀 분석만 저지하면 IDA Pro와 같은 분석 도구는 실패 지점부터 선형 분석을 시작하는데, 고정 길이 명령어의 경우 바로 드러나며 가변 길이 명령어의 경우도 자가 복원(self-repairing) 효과로 평균 세 개의 명령어 이후부터 노출된다[18].

6.3 동적분석 대응 원칙

동적분석으로부터 보호하기 위한 기법은 Table 1. 두 번째 열에 나열되어 있다. 다음은 이에 관한 원칙이다.

- **적용 범위:** 동적분석 방해 모듈은 원본 앱이 복호화되기 전 모두 구동되어야 한다. 그리고 이 모든 모듈이 정상적으로 구동될 때는 알려진 공격기법에 한에서는 최소한 안전해야 한다. 주의할 점으로 앱의 라이프사이클에 따른 프로세스의 동작 구조나 주요 정보의 흐름에 맞게 실시간 보호 모듈을 적용해야 한다.
- **보호 모듈 다양화:** 동적분석 분석을 방해하는 알려진 모든 방법을 적용해야 하며, 중복적으로 사용한다. 동적분석 방법은 다양하며 그 중 한 가지 방법이라도 가능할 경우, 우회될 확률이 많이 높아진다. 따라서 VM 탐지, 디버거 탐지, 브레이크포인트 탐지, 메모리 덤프 방지, 로그 출력 방지, 루팅 여부 탐지 등의 각 측면 대한 기법을 모두 적용해야 한다. 또한, 각 측면에 대한 기법을 하나만 적용할 것이 아니라, 하나 이상 중복으로 적용해야 한다. 한 지점에서 특정 측면의 보호가 우회되었다 하더라도 다른 지점에 중복 검사를 통해 보완해 주어야 한다.
- **보호 모듈 의존성 증가:** 실시간 보호 모듈의 의존성을 래티스(lattice) 구조와 같이 서로 얽혀 있는 형태로 만들어야 한다. 의존성으로부터 비밀키가 유도되기 때문에 복잡성이 비밀키의 안전성과 연결되어 있다. 예로, Fig. 11.의 구조는 매우 단순한 래티스 구조를 띠고 있다. 그리고 각 모듈의 입출력은 파라미터나 리턴 값보다는 힙 메모리나 전역 변수 등으로 전달해야 한다. 그래야 정적분석을 통한 값의 흐름이 어렵기 때문이다. 또한 Fig. 11.의 rt_check와 같은 보호 모듈은 암호 알고리즘 등을 사용하여 공격자가 해당 모듈에서 요구되는 값을 유추하기 어렵게 해야 한다.
- **가변적인 보호 구조:** 보호 구조를 가변적인 형태로 만들어 동적분석을 어렵게 해야 한다. 매 수행마다 다른 루틴으로 수행되게 하여, 보호 루틴으로 인한 에러 발생의 원인을 추적하기 어렵게 만든다. 방클을 포함한 대다수의 상용 보호 서비스의 경우를 보면, 다양한 보호 기법들이 적용되어 있다. 하지만 획일적인 구조로 되어 있어, 순차적인 시행착오를 통해 원인 분석을 쉽게 할 수 있었다. 가변성을 나타낼 요소로 랜덤수가 있지만, 이 값은 공격자로부터 쉽게 변형을 당할 수 있다. 즉, 공격자가 이 값의 시작 지점을 바꾸기만 하면 가변성이 사라진다. 이 대신 프로세스 번호, 특정 지점의 스택 및 베이스 포인터 값, 모듈의 베이스 지점, 시간대 등으로 할 경우, 쉽게 바꾸지 못하며 따로 공유하지 않아도 가변성을 공유할 수 있다. 이밖에도 확률에 따라 특정 모듈의 동작 여부를 결정짓거나 오류를 내는 등의 방식도 가변성을 증가시킬 수 있다.

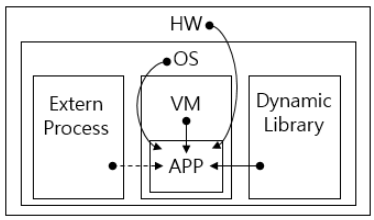

Fig. 13. Integrity threats of Android application

- **동시성을 가진 보호 구조**: 보호 구조에 동시성을 부여하여 동적분석을 어렵게 해야 한다. 올바른 동시성 프로그램은 항상 그 조건을 만족시키는 불변식(invariant)이 존재한다. 따라서 불변식의 요소를 보호 구조 의존성에 활용할 경우, 자연스럽게 보호 구조에 동시성을 부여할 수 있다. 그리고 불변식이 복잡할수록 분석하기 어렵게 된다. 동시성 분석에 대한 자동화 분석은 아직까지 걸음마 단계에 있으며, 불변식을 완전히 자동으로 찾아주는 기법은 아직 나오지 않았다. 그리고 어셈블리어 단에서의 정적 및 동적분석은 없다. 따라서 이를 보호 기법에 적용할 경우 분석 난이도 향상에 좋은 효과를 가지게 된다.
- **앱 자체의 안전성 검증:** 보호 대상 앱의 안전성을 검증해야 한다. 앱 자체의 결함으로 인해 원본 코드 노출이나 외부 DB 서버 등이 공격당할 수 있다. 예를 들어 앱의 네이티브코드 부분이 결함으로 인해 사용자의 임의 코드가 실행될 수 있는 경우, 해당 프로세스 이미지에 접근할 수 있게 된다. 따라서 앱에 외부 입력 값을 처리하는 부분이 있을 경우, 버퍼 오버플로우, SQL 인젝션 등의 취약점 여부를 검사한다.

6.4 무결성 검증 원칙

무결성 검증 작업은 보호 모듈의 한 부분으로, 앱 자체나 동작 환경의 무결성을 확인한다. 무결성 검사

가 요구되는 부분은 Fig. 13에서 실선으로 표현된 부분으로, 앱 자체를 포함하여 하드웨어, VM, 동적 라이브러리, 운영체제 총 5부분이다. 점선은 외부 프로세스가 앱을 제어하는 나타낸 것으로, 원본 앱 흐름의 무결성을 의미한다. 이는 디버거 프로세스 탐지, 브레이크포인트 탐지, 루팅 여부 탐지 등으로 차단할 수 있다. 다음은 다섯 부분에 관한 무결성 검증 원칙이다.

- **앱 무결성 검증**: 앱에 포함된 바이트코드와 네이티브코드 모두 무결성을 검증해야 한다. 패커 방식에서는 바이트코드의 무결성이 중요하지 않게 생각될 수 있으나 바이트코드가 네이티브코드 속 함수의 흐름을 결정짓기 때문에 바이트코드도 무결성을 검증해야 한다. 검증은 해시 값을 통해 직접적으로 검증할 수 있으나, 바이트코드와 네이티브코드의 상호 의존성을 만들어 서로 분리하여 분석하기 어렵게 하는 방식도 존재한다.
- **VM 무결성 검증**: 안드로이드 VM의 무결성을 검증해야 한다. 안드로이드는 두 가지 동작 모드가 존재하며, 둘 다 VM을 통해 동작한다. 따라서 공격자는 VM의 수정을 통한 공격을 시도할 수 있으므로 무결성 검증이 요구된다. VM 검증은 세 가지 세부요소로 나뉜다. 첫째로, VM 자체의 검증이다. 둘째로, VM이 사용하는 외부 라이브러리의 검증이다. 셋째로, 최적화 컴파일러(실행 파일)의 검증이다. 최적화는 두 모드에 따라 dexopt 또는 dex2oat이 첫 구동 시 실행되는데, 이를 통해서도 공격이 이뤄질 수 있기 때문에 검증해야 한다.
- **동적 라이브러리 검증**: 동적 라이브러리 재배치의 무결성을 검증해야 한다. 공격자는 보호 서비스가 사용하는 외부 라이브러리의 흐름을 변경하여 정보 추출 및 디버깅을 수행할 수 있다. 따라서 외부 함수가 재배치되거나 호출될 때에는 그 무결성을 다시 한번 검토해야 할 필요가 있다. 주의할 점으로 사용하는 외부 라이브러리가 이용하는 외부 라이브러리의 재배치도 검증이 요구된다. 최근 발표된 공격기법에서는 간접적인 재배치 과정을 통해 수행 흐름을 빼어오는 기법이 소개되었다[11]. 따라서 직접적인 것 외에도 간접적으로 이용되는 모든 재배치 과정을 검증해야 한다.
- **운영체제 무결성 검증**: 앱이 동작하는 안드로이드의 빌드 버전을 검증해야 한다. 현재 추세로 볼 때, 다음 공격 벡터는 수정된 운영체제가 될 수 있다. 이렇게 될 경우, 에뮬레이터 및 VM 검증을 통해서는 올바른 수행 환경 여부를 확인할 수 없다. 따라서 각 제조사들이 정식으로 배포한 빌드에서만 동작할 수 있도록 해야 한다. 빌드명 또는 펌웨어 서명 값 등을 통해 이를 확인하는 절차를 넣어야 한다.

VII. 결 론

기존 상용 보호 서비스의 분석을 통해 앱 보호 서비스의 개발 과정을 간접적으로 확인하였다. 이를 통해 상용 보호 서비스는 기본적으로 안전한 보호 구조로 되어 있는 것을 확인하였다. 하지만 이를 체계화된 틀 아래 구현하고 있지 않아 구조를 확장하는 과정에서 결함이 발생한 것을 확인하였다.

이뿐만 아니라 보호 구조의 약점을 분석하였다. 실시간 보호 모듈 간 독립성 때문에 쉽게 보호 구조가 파훼되는 것을 확인할 수 있었으며, 원본 암호화에 사용한 비밀키를 단순히 특정 흐름에 배치함으로써 구조가 파훼 되면 쉽게 노출됨을 확인하였다.

본 논문에서는 이를 보완하기 위해 안전한 보호 의존성 관계를 제시하였다. 이는 실시간 보호 모듈간에 의존성을 부과하는 방식으로 보호 구조를 견고하게 만들며, 이 과정을 통해 최종적으로 비밀키를 도출하게끔 만든다. 따라서 보호 모듈이 전부 파훼되지 않는 한 올바른 비밀키를 도출할 수 없게 된다. 이러한 구조는 구조 노출 여부가 구조의 안전성에 영향을 미치지 않는다.

마지막으로 이러한 구조를 구현함에 있어 고려해야 하는 사항과 원칙을 제안하였다. 이를 통해 개발자는 보호 구조 개발 시 고려해야 할 사항을 체계적으로 확인할 수 있으며, 최신 공격기법에 대응할 수 있다. 이는 원본 앱 자체의 검증부터 외부 라이브러리, VM, 운영체제까지 고려하는 것으로 현재 알려진 대다수 내용을 포함하고 있다.

References

[1] Yishay Yovel. (2014). *4 Essential Ways to Protect My Mobile*. Available: htt

ps://securityintelligence.com/how-to-protect-mobile-apps-essentials
〔2〕 Bangcle, "https://www.bangcle.com", 1. Oct. 2018.
〔3〕 Baidu, "http://apkprotect.baidu.com", 1. Oct. 2018.
〔4〕 360, "http://jiagu.360.cn", 1. Oct. 2018.
〔5〕 Ali, "http://jaq.alibaba.com", 1. Oct. 2018.
〔6〕 Tencent, "http://jiagu.qcloud.com", 1. Oct. 2018.
〔7〕 Ijiami, "http://www.ijiami.cn", 1. Oct. 2018.
〔8〕 Architectural Principles That Prevent Code Modification or Reverse Engineering, "https://www.owasp.org/index.php/Architectural_Principles_That_Prevent_Code_Modification_or_Reverse_Engineering", 10. Oct. 2018.
〔9〕 J. Kim and K. Lee, "Robust Anti Reverse Engineering Technique for Protecting Android Applications using the AES Algorithm," *Journal of KIISE*, 42 (9), pp. 1100-1108, Sept. 2015.
〔10〕 J. Kim, N. Go, and Y. Park, "A Code Concealment Method using Java Reflection and Dynamic Loading in Android," *Journal of the Korea Institute of Information Security & Cryptology*, 25 (1), pp. 17-30, Feb. 2015.
〔11〕 T. Strazzere and J. Sawyer, "Android Hacker Protection Level 0," presented at *DEF CON 22*, Aug. 2014.
〔12〕 S. Makkaveev and A. Bashan, "Unboxing Android: Everything you wanted to know about Android packers," presented at *DEF CON 25*, Aug. 2017.
〔13〕 Y. Zhang, X. Luo, and H. Yin, "The Terminator to Android Hardening Services," presented at *HITCON 11*, Aug. 2015.
〔14〕 Dalvik bytecode, "https://source.android.com/devices/tech/dalvik/dalvik-bytecode", 10. Oct. 2018.
〔15〕 APKTOOL, "https://ibotpeaches.github.io/Apktool", 10. Oct. 2018.
〔16〕 DEX2JAR, "https://github.com/pxb1988/dex2jar", 10. Oct. 2018.
〔17〕 JD-GUI, "http://jd.benow.ca", 10. Oct. 2018.
〔18〕 C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290-299, Oct. 2003.
〔19〕 I. V. Popov, S. K. Debray, and G. R. Andrews. "Binary Obfuscation Using Signals," *USENIX Security Symposium*, pp. 275-290, Aug. 2007.
〔20〕 R. Carbone, C. Bean, and M. Salois, "An In-depth Analysis of the Cold Boot Attack: Can it be Used for Sound Forensic Memory Acquisition?," *Valcartier: Defence Research and Development Canada*, Jan. 2011.
〔21〕 V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1675-1689, Oct. 2016.
〔22〕 C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," *In Proc. 25th. ACM Symposium on Principles of Programming Languages*, pp. 184-196, Jan. 1998.

〈저 자 소 개〉

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