정 보 보 호 학 회 지
제 25 권 제 2 호, 2015. 6

안드로이드 어플리케이션 역공학 보호기법

하 동 수*, 이 강 효*, 오 희 국*

요 약

현재 가장 높은 점유율을 보이고 있는 스마트 모바일 디바이스 플랫폼인 안드로이드는 바이트코드 기반의 어플리케이션을 동작시킨다. 바이트코드는 특성상 역공학에 취약하여 원본 코드의 노출이나 수정 후 재배포가 쉽게 이루어질 수 있다. 이를 보완하는 방법으로 난독화, 실행압축, 코드 분리, 기타 안티 리버싱 기법 등이 존재하지만, 이런 보호기법을 단독으로 사용하면 그 효과가 높지 않다. 이들은 각각 장단점을 가지고 있는데, 여러 기법을 조합해서 사용하면 보안성을 한층 높일 수 있다. 그렇다고 각 기법의 특징을 무시한 채 무작정 사용하게 되면 오히려 어플리케이션의 성능이 낮아지고 크기가 늘어나는 문제가 발생한다. 따라서 보호기법의 정확한 이해와 필요에 맞는 올바른 선택적 사용이 중요하다. 본 논문에서는 지금까지 사용되어 온 안드로이드 어플리케이션 역공학 보호기법의 유형과 특징에 대하여 살펴보고, 보안성을 높이기 위한 올바른 조합과 선택에 대해 고찰한다.

Ⅰ. 서 론

보편적인 디바이스로 자리 잡은 스마트 모바일 디바이스는, 이제 현대인에게 없어서는 안 될 필수품이 되었다. 안드로이드는 이 디바이스들이 사용하고 있는 플랫폼 중 가장 높은 시장 점유율을 보이고 있다[1]. 하지만 높은 점유율과는 반대로 안드로이드 어플리케이션은 역공학에 취약한 모습을 보인다[2,3].

이런 취약점으로 인해, 시장성과 더불어 각종 어플리케이션 위변조 사례가 증가하고 있다. 이는 최근 모바일 앱 결제 활성화와 더불어 더욱 증가하는 추세에 있다. 미국 모바일 앱 보안 보고서에 따르면 전 세계 상위 앱 스토어 90곳에 있는 모바일 뱅킹 앱 350,000여개 중 약 400,000개의 앱이 위변조 등을 통해 멀웨어를 포함하고 있거나 의심스러운 바이너리를 가진 것으로 나타났다[4].

또한 소스 코드 복제 문제도 심각하다. 대표적인 사례로 2014년에 국내 인기 게임 '아이러브 커피'를 그대로 복제한 사건이 있었다. 중국에서 서비스 중인 '커피러버'는 국내에서 서비스 중인 '아이러브 커피'와 많은 부분 유사하였고, 이는 디컴파일을 통한 소스 코드 복제 때문인 것으로 밝혀졌다[5].

이렇게 안드로이드 어플리케이션이 보안에 취약한 이유는 바이트코드로 이루어져 있기 때문이다. 이는 현재 온라인에 공개된 APKTool[6], DEX2JAR[7], JD-GUI[8]와 같은 도구를 이용하여 쉽게 디컴파일 및 리패키징을 할 수 있는 수준이다.

이런 문제점을 해결하기 위해 자바나 C#에서 사용하는 보호기법들[9]을 안드로이드에도 적용할 수 있다. 하지만 무분별한 보호기법의 사용은 어플리케이션의 성능을 저하시키고 코드 크기를 증가시키므로, 보호기법의 정확한 이해 아래에 적절한 사용이 요구된다.

본 논문에서는 안드로이드 어플리케이션 보호기법의 유형과 그 특징을 소개한다. 먼저 2장에서 안드로이드 어플리케이션의 구성과 구조에 대해 설명하고 3장에서 지금까지 사용되어 온 안드로이드 어플리케이션 보호기법의 유형과 그 특징을 소개한다. 그리고 4장에서 어플리케이션 유형에 따른 적절한 보호기법 선택 방법을 알아보고 5장에서 결론으로 마무리 짓는다.

Ⅱ. 안드로이드 어플리케이션

안드로이드 어플리케이션은 자바 형태의 언어로 개발되며, APK(Android Application Package) 파일 형태로 배포된다. 여기에 포함되는 코드는 바이트코드와 네이티브코드 이 두 종류이다. 안드로이드 어플리케이션

* 한양대학교 컴퓨터공학과 정보보호연구실 {hds,onyourlife,hkoh}@hanyang.ac.kr

은 바이트코드 기반으로 구성되며 필요에 따라서 빠른 연산이 요구되는 부분은 네이티브코드로 따로 작성된다.

2.1. APK 파일

APK 파일은 그림 1과 같이 크게 4가지 구성요소로 이루어진다[10]. 첫 번째 요소로 DEX 파일(.dex files)은 어플리케이션의 모든 바이트코드를 포함한다. 보통은 한 개(classes.dex)로만 구성되지만 때에 따라서는 두 개 이상인 경우도 있다. 두 번째 요소로 컴파일된 리소스(resoures.arsc)는 윈도우 레이아웃, 문자열 상수 등과 같이 컴파일될 수 있는 XML 파일들을 포함한다. 세 번째 요소로 컴파일되지 않는 리소스는 이미지, 네이티브코드 등과 같은 것으로 원본 그대로 보관된다. 마지막 요소로 매니페스트 파일(AndroidManifest.xml)은 시작 클래스, 퍼미션 등과 같은 정보를 포함한다. 이 구성요소들은 ZIP 알고리즘으로 묶여 확장자가 'apk'인 파일로 존재한다.

〔그림 1〕 APK 파일 구성요소

2.2. DEX 파일

DEX(Dalvik EXcutable format) 파일은 안드로이드 가상머신(Dalvik machine) 위에서 실행되도록 구성된 것으로, 어플리케이션 구동 시 생명 주기(life cycle)[11]에 따라 가장 처음으로 호출되는 메소드 정보를 가지고 있다. 하지만, 극히 드문 경우로 시작 메소드가 네이티브 코드에 있는 경우도 존재한다.

DEX 파일에서는 하나 이상의 클래스 정보가 저장되어 있으며, 문자열 식별자 리스트, 타입 식별자 리스트, 메소드 프로토타입 식별자 리스트, 필드 식별자 리스트, 메소드 식별자 리스트, 클래스 정의 리스트, 데이터 영역으로 구성되어 있다. 이들은 문자열 식별자 리스트를 제외하고는 다른 리스트의 인덱스를 참조하는 식으로 구성되어 있다[12]. 이렇게 구성된 DEX 파일은 클래스명, 메소드명, 필드명은 물론이고 디버그 정보에는 변수명, 파라미터명 등 코드 분석에 도움 되는 정보도 포함하고 있다.

그림 2는 DEX 파일에 포함된 하나의 클래스를 디스어셈블리(disassembly)한 예시이다. 그림을 보면 해당 클래스는 com.infosec.sample 패키지의 Main 클래스이며(1번째 줄) 부모 클래스는 android.app.Activity임을 알 수 있다(2번째 줄). 또한 Main.java가 원본 소스 코드의 이름임을 알 수 있다(3번째 줄). 그리고 생성자 메소드에서 android.app.Activity 클래스의 생성자를 호출(9번째 줄)한 뒤 종료하며 onCreate 메소드에서는 부모 클래스의 onCreate 메소드를 호출(18번째 줄)한 뒤 v1 레지스터에 상수 값을 넣는 것(20번째 줄)을 알 수 있다. 이와 같이 DEX 파일은 바이트코드의 특성상 많은 정보가 포함되어 있으며, 분석 및 조작에 용이한 구조를 띈다.

```
01 .class public Lcom/infosec/sample/Main;
02 .super Landroid/app/Activity;
03 .source "Main.java"
04
05 .method public constructor <init>()V
06     .locals 0
07     .prologue
08     .line 17
09     invoke-direct {p0}, Landroid/app/Activity;
   -><init>()V
10     return-void
11 .end method
12
13 .method public onCreate(Landroid/os/Bundle;)V
14     .locals 2
15     .param p1, "savedInstanceState"
16     .prologue
17     .line 22
18     invoke-super {p0, p1}, Landroid/app/Activit
   y;->onCreate(Landroid/os/Bundle;)V
19     .line 23
20     const v1, 0x7f03000b
21       ... (생략) ...
22 .end method
```

〔그림 2〕 디스어셈블리된 바이트코드 예시

2.3. ELF 파일

ELF(Executable and Linking Format) 파일은 리눅

스에서의 실행, 오브젝트 파일, 공유 라이브러리, 또는 코어 덤프를 할 수 있게 하는 바이너리 파일이다. 안드로이드에서는 공유 라이브러리 형태로서, 외부 라이브러리 용도로 사용된다.

이 파일은 네이티브코드를 담고 있으며, 암호 모듈이나 그래픽 엔진과 같이 많은 연산을 요구하는 어플리케이션을 위해 존재한다. 이는 확장자가 'so'이며 APK 파일 /lib/ 하위 경로에 보관되지만 어플리케이션에 따라 없는 경우도 있다.

공유 라이브러리는 실행 시간에 동적으로 로드되며, 바이트코드 단에서 JNI(Java Native Interface)를 통해 사용된다. 이것은 네이티브코드로 구성되어 있기에 상대적으로 바이트코드보다 역공학에 강한 장점을 가지고 있다.

Ⅲ. 어플리케이션 보호기법

역공학에 취약한 안드로이드 어플리케이션을 보완하기 위하여 다양한 보호기법이 사용되었으며, 보호 대상은 크게 바이트코드와 네이티브코드로 나뉜다. 네이티브코드는 바이트코드보다 상대적으로 안전하므로 보호기법의 대다수는 바이트코드 보호에 초점이 맞춰져 있다.

3.1. 바이트코드 난독화 기법

이 기법은 바이트코드의 명령어를 수정하여 코드의 이해를 어렵게 만드는 방식이다. 즉, 어플리케이션의 소스 코드를 보지 못하게 막는 것이 아닌, 소스 코드 분석의 난이도를 높이는 방식이다. 대표적인 방법으로 식별자 변환(identifier renaming), 제어 흐름 변환(control flow change), 문자열 암호화(string encryption), API 은닉(API hiding), 클래스 암호화(class encryption)가 있다. 다음은 이 보호기법들에 대한 설명이다.

3.1.1. 식별자 변환

이 기법은 클래스, 메소드, 필드 등의 이름을 의미 없는 문자열로 치환하며, 이렇게 함으로써 분석을 어렵게 한다. 바이트코드에는 분석에 도움 될 수 있는 클래스명, 메소드명, 필드명 등이 존재하며, 구조상 이를 제거할 수 없으므로 분석에 도움이 되지 않도록 변경하는 것이다. 일반적으로 시스템 API나 외부 라이브러리를 제외한 나머지의 식별자를 변경한다[13].

이 기법은 어플리케이션의 흐름과 성능에 영향을 전혀 주지 않으면서도 분석 난이도를 높일 수 있는 방법이다. 하지만 시스템 API나 외부 라이브러리의 식별자를 바꿀 수 없는 한계점을 가진다. 그림 3은 식별자 변환이 적용된 클래스의 예시이다.

```
01  private int pushServerCacheKey;
02  private int pushPolicyServerPort;
03
04  public class PushService{
05    public void setPushPort(String port){
06      int pushServerCacheKey = getKey();
07      System.out.println(pushServeCacheKey);
08    }
09
10    public void PushService() {...}
11    public int getKey() {...}
12  }
```

```
01  private int a;
02  private int b;
03
04  public class a{
05    public void a(String c){
06      int a = b();
07      System.out.println(a);
08    }
09
10    public void a() {...}
11    public int b() {...}
12  }
```

〔그림 3〕 식별자 변환 적용 전(위)과 후(아래) 예시

3.1.2. 제어 흐름 변환

이 기법은 기존 어플리케이션의 시멘틱을 건드리지 않는 한에서 명령어 흐름을 복잡한 형태로 변환한다. 이렇게 함으로써 어플리케이션을 분석하기 어렵게 하며, 소스 코드로 디컴파일하지 못하게 막아주는 역할도 한다.

분석 속도를 늦추는 변환으로 크게 두 가지가 있다[14,15,18]. 먼저 실제 시멘틱에 영향을 주지 않는 더미 코드를 코드 사이에 삽입하거나 구조를 바꾸는 유형이 있다. 이는 의미 없는 코드를 따라가게 하거나 복잡한 구조를 탐색하게 함으로써 분석 속도를 늦추게 한다. 다음으로 유사한 작업을 하는 메소드를 서로 다른 패키지나 클래스로 분리시키거나, 관련 없는 메소드를 한 곳에 모아둠으로써 그 역할을 쉽게 유추할 수 없게 만드는

유형이 있다[16-18].

디컴파일(decompile)을 못하게 하는 변환으로, 데이터 의존성이 없는 명령어들의 순서를 변경하여 컴파일되는 패턴을 지우는 방법이 있다. 또한 컴파일러의 최적화 옵션을 사용하여 부가적으로 안티 디컴파일 효과를 얻을 수도 있다.

기본적으로 이 기법은 어플리케이션의 크기가 증가하며 더미 코드 수행으로 인해 속도 저하가 발생한다. 그림 4는 제어 흐름 변환이 적용된 메소드의 예시이다.

```
01  public int CompareTo(WordCnt o){
02    int n = cnt - o.cnt;
03
04    if (n == 0)
05      n = String.Compare(word, o.word);
06
07    return(n);
08  }
```

```
01  public int CompareTo(WordCnt o) {
02    int local0;
03    int local1;
04    local0 = cnt - o.cnt;
05
06    if (local0 != 0)
07      goto i0;
08
09    goto i1;
10
11    while (true) {
12      return local1;
13      i0: local1 = local0;
14    }
15
16    i1:
17    local0 = String.Compare(this.cnt, o.cnt);
18
19    goto i0;
20  }
```

〔그림 4〕 제어 흐름 변환 적용 전(위)과 후(아래) 예시

3.1.3. 문자열 암호화

이 기법은 소스 코드에서 사용되는 문자열 상수를 어떤 연산 과정을 통해 나오도록 변환한다. 비록 그 연산 과정은 항상 같은 값을 계산하지만, 이렇게 함으로써 정적분석 과정에서 해당 문자열을 바로 알지 못하게 하여 분석 속도를 늦출 수 있다.

문자열 암호화에 사용되는 대표적인 방법은 크게 두 가지가 있다. 첫 번째로, 문자열 상수를 암호화한 뒤 그 암호문의 복호화 루틴을 삽입하는 방법이 있다[19]. 암호화에 사용된 키는 소스 코드 어딘가에 저장되어 있으므로 푸는 것이 불가능하지는 않지만 정적분석 과정에서는 문자열 상수를 확인하기 위해서 매번 키를 찾고 복호화해야 하는 번거로움이 뒤따른다.

두 번째로, 문자열 상수 생성 루틴을 삽입하는 방법이 있다. 이 루틴은 결과 값은 항상 해당하는 문자열 상수이다. 앞선 방식 대비 이 방식의 장점은 해당 루틴이 난독화를 위해 삽입된 것인지 원본 코드인지 바로 판별하기 힘들다는 점이다. 다양한 패턴의 문자열 생성 루틴을 사용할 경우, 암호화하는 방식보다 더 많은 시간을 들이게 할 수 있다.

이 기법의 적용 대상은 일반적으로 URI(Uniform Resource Identifier) 문자열이다. 안드로이드의 어플리케이션은 여러 개의 액티비티(Activity)[20]로 구성되어 있으며, 이들 간의 통신은 인텐트(Intent)[21]를 통해 이뤄진다. 대상 액티비티의 주소는 URI 형태로 인텐트에 담겨지는데, 이 주소들이 문자열 상수이며 핵심 분석 요소에 해당한다.

이 기법은 제어 흐름 변환과 마찬가지로 어플리케이션의 크기가 증가하며 속도 저하가 발생한다. 그림 5는 이 기법이 적용된 예시이다.

```
01  String pn = "tel:010-0000-0000";
02  String action = Intent.ACTION_DIAL;
03
04  intent = new Intent(action, Uri.parse(pn));
05  startActivity(intent);
```

```
01  String e1 = "asSdfgh23ASdfWg34gdfgGa";
02  String e2 = "hgf34GfdfSDv2G5223fDSbd";
03  String d1 = decryption(e1,key);
04  String d2 = decryption(e2,key);
05
06  intent = new Intent(d1, Uri.parse(d2));
07  startActivity(intent);
```

〔그림 5〕 문자열 암호화 적용 전(위)과 후(아래) 예시

3.1.4. API 은닉

이 기법은 소스 코드 속 API 호출을 리플렉션(reflection)[22,23]을 통해 호출되도록 변환한다. 이렇게 함으로써 정적분석 과정에서 API 호출을 쉽게 알지 못하도록 하여 분석 속도를 늦출 수 있다.

여기서 리플렉션이란 컴파일 시간이 아닌 실행 시간에 동적으로 클래스의 구조와 행위를 검사 및 수정하는 프로그래밍 기법을 말한다. 이를 통해 실행 시간에 동적

으로 클래스 정보를 얻을 수 있으며, 클래스를 로딩할 수도 있다. 그리고 이렇게 얻은 클래스 정보를 바탕으로 메소드를 호출할 수 있다. 여기서 정보를 얻고자 하는 클래스명, 로딩하고자 하는 클래스 데이터, 호출하고자 하는 메소드명 등은 문자열이나 바이트 스트림 타입으로 파라미터를 통해 전달된다.

API 은닉은 이런 리플렉션 특징을 이용하여 소스 코드에 명시적으로 드러나는 API의 사용을 가린다. 하지만 파라미터로 전달되는 문자열 상수나 바이트 스트림 때문에 리플렉션만으로는 제대로 가릴 수 없다. 이런 이유로 API 은닉은 일반적으로 문자열 암호화와 같이 사용되어 파라미터로 전달되는 값을 바로 알 수 없도록 한다. 그림 6은 이를 보여준다.

API 호출을 감추는 근본적인 이유는 호출되는 메소드를 감춤으로써 분석 속도를 늦추는 것에 있지만, 보통은 어플리케이션의 호출 그래프(call graph)를 쉽게 얻지 못하게 함에 있다. 호출 그래프는 어플리케이션의 구조를 파악하기 위해서 중요한 요소이다. 그렇기 때문에 일반적으로 호출 그래프를 자동으로 그려주는 정적 분석 도구를 활용하는데, 리플렉션을 사용할 경우 정적 분석 도구는 이를 그려주지 못한다.

설령 파라미터로 넘겨지는 문자열 값이 암호화가 되어 있지 않더라도 대게의 정적 분석 도구는 이를 그려주지 못한다. 또한 스트링 분석(string analysis)을 하여 문자열 값을 찾아주는 도구가 있다하더라도 암호화를 한다면 현재의 기술로는 문자열 값을 제대로 알아내지 못한다.

```
01  Hello obj = new com.sample.Hello();
02  obj.print();
```

```
01  String cnm = "com.sample.Hello";
02  String mnm = "print";
03
04  Class hello = Class.forName(cnm,null);
05  Object obj = hello.newInstance();
06  Method mtd = hello.getMethod(mnm,null);
07  mtd.invoke(obj,null);
```

```
01  String enc_cls_nm = "awnDR34GADRGsdd3";
02  String enc_mtd_nm = "ASdcfNzx42gFS4vj";
03  String cnm = decryption(enc_cls_nm,key);
04  String mnm = decryption(enc_mtd_nm,key);
05
06  Class hello = Class.forName(cnm,null);
07  Object obj = hello.newInstance();
08  Method mtd = hello.getMethod(mnm,null);
09  mtd.invoke(obj,null);
```

〔그림 6〕 API 은닉 적용 전(상), 후(중) 예시와 문자열 암호화와 같이 사용된 예시(하)

이 기법은 앞선 기법들과 마찬가지로 어플리케이션의 크기가 증가하고 속도 저하가 발생한다. 그리고 안드로이드 어플리케이션은 객체지향 언어의 특성상 많은 메소드 호출로 구성된다. 따라서 해당 기법을 모든 메소드 호출에 적용하지 않고 중요한 호출 지점이나 시스템 API 또는 외부 라이브러리와 같이 문자열 암호화를 적용할 수 없는 지점에 사용한다.

3.1.5. 클래스 암호화

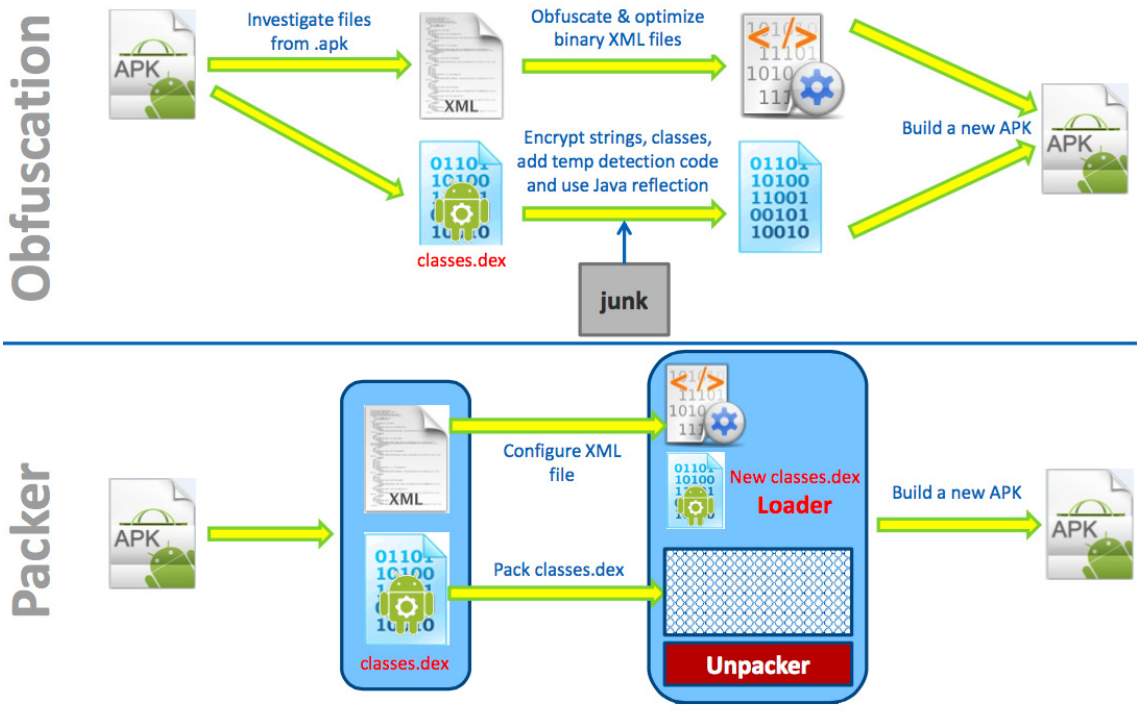
이 기법은 암호화된 클래스 파일을 실행 시간에 복호화하고 리플렉션을 통해 사용하도록 변환한다. 이 기법은 모든 클래스를 암호화할 수 없지만 중요 클래스를 선별하여 암호화할 수 있다. 이렇게 함으로써 정적 분석을 통해 클래스 정보를 바로 볼 수 없게 만들고 분석 속도를 늦추게 할 수 있다.

자바에서는 클래스 파일이 클래스 당 하나씩 존재하지만 안드로이드에서는 모든 클래스 파일이 하나의 DEX 파일에 기록되어 있다. 그러므로 자바에서 사용하는 클래스 암호화 방식을 그대로 차용할 수 없다. 대신 이와 유사하게 추가적인 DEX 파일을 만들어 거기에 보호할 클래스를 따로 담고 암호화하는 식으로 보호한다.

DEX 파일 암호화는 일반적으로 두 가지 방식이 존재한다. 첫 번째로 DEX 파일의 일부만 암호화하는 방식이다. 이 방식은 DEX 파일 헤더의 특정 파트만 암호화하여 로딩할 때에 그 부분만 복호화하여 사용한다. 이

```
01  String dexPath = "/sdcard/sample.dex";
02  String decPath = "/sdcard/dec.dex";
03  String odexPath = "/sdcard/odex.dex";
04
05  // dexPath 경로의 DEX 파일을 복호화하여
06  // decPath 경로에 저장
07
08  DexClassLoader dex = new DexClassLoader(
09    decPath, odexPath, null,
10    getClassLoader());
11
12  // decPath 경로의 복호화된 DEX 파일 삭제
13
14  Class<?> hello =
15    dex.loadClass("com.sample.Hello");
16
17  Object obj = hello.newInstance();
18  Method mtd = hello.getMethod("print",null);
19  mtd.invoke(obj,null);
```

〔그림 7〕 암호화된 클래스 파일 로딩 예시



〔그림 8〕 난독화와 실행압축 기술을 APK에 적용하는 과정〔24〕

방식은 속도가 빠른 대신 모든 정보를 보호하지는 못한다. 두 번째 방식은 DEX 파일 전체를 암호화하는 방식이다. 이 방식은 속도가 느린 대신 모든 정보를 보호할 수 있다.

이 기법은 암호화된 DEX 파일이 복호화될 경우 한 번에 많은 정보가 노출되는 단점이 있지만, 암호화된 DEX 파일을 APK 파일 밖으로 분리시킬 수 있는 장점이 있다. 즉, 실행 시간에 서버나 기타 외부로부터 암호화된 DEX 파일을 받아와 동적 로딩을 할 수 있다. 이렇게 함으로써 APK 파일만 가지고는 해당 어플리케이션의 모든 정보를 알 수 없게 할 수 있다. 그림 7은 클래스 암호화의 사용 예시를 보여준다.

3.2. 실행압축 기술을 이용한 보호기법

실행압축(runtime packer)이란 실행 시간에 압축되어 있는 실행 코드를 풀어 메모리에 로드한 후 실행시키는 기술이다[25]. 본래 해당 기술은 실행 파일의 크기를 줄이기 위한 목적으로 사용되었지만, 최근에는 실행 코드의 보호와 악성코드들이 자신의 코드를 숨기기 위한 목적으로 많이 사용되고 있다.

이 기법은 난독화 기법과는 보호 방향에서 큰 차이점이 있다. 난독화 기법은 실행 코드 노출에는 신경을 쓰지 않으며, 코드의 분석을 어렵게 하는데 초점을 맞춘다. 하지만 실행압축 기법은 실행 코드의 노출 방지에 초점을 맞춘다. 그렇기 때문에 APK 파일에 난독화 기능을 추가하는 것과 실행압축 기술을 적용하는 과정은 차이가 있으며, 또한 그 구조에서도 차이가 존재한다. 그림 8은 이를 보여주고 있다.

이 기법은 DEX 파일을 보호하는 용도로 사용된다. 그렇게 때문에 DEX 파일을 압축 알고리즘이 아닌 암호 알고리즘을 사용하여 보호한다. 또한 성능 문제 등의 이유로 비대칭키 암호 알고리즘보다는 대칭키 암호 알고리즘이 선호되는 편이며 AES와 같은 블록 암호 알고리즘 보다는 RC4와 같은 스트림 암호 알고리즘이 선호되는 경향이 있다.

실행압축 기술은 다양한 방식으로 사용되고 있지만 대표적인 방법은 그림 8과 같다. 원본 DEX 파일은 암호화된 상태로 APK 파일 속에 보관되며 새로운 DEX 파일이 추가된다. 그리고 매니페스트 파일은 새로 추가된 DEX 파일 속 클래스를 시작 지점으로 가리킨다. 여기서 새로 추가된 DEX 파일 속 코드는 암호화된 원본

DEX 파일을 복호화하고 메모리에 로드시키는 작업을 수행한다.

바이트코드는 역공학이 쉬우므로 암호화된 DEX 파일을 복호화하는 핵심 코드는 네이티브코드로 보관된다. 그림 8에서는 'Unpacker'가 이것을 의미하며, 공유 라이브러리 파일을 의미한다. 결과적으로, 안드로이드 실행압축 기술의 안전성은 네이티브코드의 복잡성에 달려있다. 그러므로 대부분의 경우 네이티브코드에 난독화를 적용해 놓는다. 또한 동적 분석을 막기 위해 안티 디버깅이나 안티 에뮬레이션 같은 기술이 함께 사용된다.

이 기법은 어플리케이션 시작 부분의 암호화된 DEX 파일 로딩을 제외하고는 성능 오버헤드가 없으며, 새로 추가되는 DEX 파일을 제외하고는 코드 크기도 증가하지 않다. 하지만 복호화 루틴이 파훼되거나 우회되는 취약점이 있다면 원본 DEX 파일이 바로 노출되는 단점이 있다. 실제로 일부 실행압축 도구에서 언패킹 과정 중 복호화된 DEX 파일이 메모리에 로드되기 전에 파일로 노출되는 취약점이 발견되었으며, 이를 이용하여 루팅(rooting) 없이 복호화된 파일을 구하는 방법이 공개되었다[26].

3.3. 코드 분리를 통한 보호기법

이 기법은 핵심 코드를 APK 파일 밖으로 분리하여 보관하는 방식으로 공유 라이브러리나 분리된 DEX 파일을 보호할 때 사용된다. 일반적으로 공유 라이브러리를 보호하기 위해 많이 사용되며 이를 APK 파일이 아닌 서버에 두고, 실행 시간에 불러와서 사용한다. 이렇게 함으로써 APK 파일을 열어본다 하더라도 공유 라이브러리를 얻을 수 없게 한다.

기본적으로 이 방식은 네트워크 패킷 모니터링을 통해 가로채는 방식을 막을 수 없다. 그러므로 실제 도구에서는 암호 프로토콜을 사용하여 이를 해결한다.

쉬운 예로, 어플리케이션이 사전에 서버의 공개키를 가지고 있다고 할 경우, 대칭키로 사용할 값을 랜덤으로 생성하고 서버의 공개키로 암호화를 한다. 그 다음 그 값을 서버에 보내면 서버는 자신의 개인키로 대칭키를 구한 다음, 코드를 대칭키로 암호화하여 보낸다. 최종적으로 어플리케이션은 대칭키를 사용하여 코드를 복원한 다음 로드하여 사용한다. 이러한 암호 프로토콜 과정에서 내부 변수 값을 보지 못하도록 안티 디버깅이나 안티 에뮬레이션 기술이 같이 사용된다.

3.4. 기타 보호기법

앞선 보호기법들은 암호 알고리즘이 아니며 성능과 크기 문제 때문에 이런 기법들을 무한정 적용할 수 없다. 따라서 충분한 분석 시간이 주어진다면 필히 원본 코드가 노출되며 동적 분석을 통하면 그 분석 시간마저 많이 단축된다.

특히 문자열 암호화, API 은닉, 클래스 암호화와 같은 난독화 기법들은 동적 분석이 허용될 경우 거의 의미가 없어지게 된다. 또한 실행압축 기술도 동적 분석이 허용될 경우 분석이 많이 쉬워진다. 이러한 문제를 해결하기 위해 다양한 안티 리버싱 기법들이 사용된다.

3.4.1. 안티 디버깅(anti-bugging)

안티 디버깅[27,28]은 디버깅을 방해하여 분석을 어렵게 한다. 만약 어떤 어플리케이션이 디버깅을 당하고 있다면 해당 디버거 프로그램을 종료시키거나 에러를 발생시키는 방법 등 다양한 방법을 사용하여 분석을 방해한다. 디버깅 탐지 방법으로 API 기반 탐지, 프로세스 및 스레드 블록 탐지, 하드웨어 및 레지스터 기반 탐지, 예외 기반 탐지, 타이밍 기반 탐지 기법이 존재한다.

3.4.2. 안티 에뮬레이션(anti-emulation)

안티 에뮬레이션[29]은 에뮬레이션을 방해하여 분석을 어렵게 한다. 만약 어떤 어플리케이션이 에뮬레이션되고 있다면 즉, 가상 머신 위에서 동작되고 있다면 프로그램을 종료시키거나 에러를 발생시키는 방법 등 다양한 방법을 사용하여 분석을 방해한다. 가상 머신을 통해서 어플리케이션의 동작 방식을 분석할 수 있기 때문이다.

3.4.3. 안티 디스어셈블리(anti-disassembly)

안티 디스어셈블리[30]는 바이트코드의 디스어셈블리 과정을 잘못되게 만든다. 안드로이드에서는 일반적으로 프로그램 흐름에 영향을 주지 않는 정크바이트(junkbyte)를 명령어 사이에 넣음으로써 디스어셈블리

를 잘못되게 만든다. 그림 9,10,11은 안티 디스어셈블리에 관한 것이다.

그림 9는 정수 6을 리턴하는 메소드를 수정한 예시이다. 수정사항으로 주소 0x3BE에 goto 명령어를 넣고 뒤이어 정크바이트 2 바이트를 삽입한다. 여기서 삽입된 코드는 기존 코드의 시멘틱을 건드리지 않는다. 이 방식은 재귀탐색(recursive traversal) 알고리즘을 사용하는 디스어셈블러에서는 정상적으로 디스어셈블리한다. 하지만 선형탐색(linear sweep) 알고리즘을 사용하는 것에서는 그렇지 않다. 그림 10이 그 예를 보여 준다. 그림 10은 그림 8에서 삽입된 goto 명령어(unconditional branch)를 if 명령어(conditional branch)로 바꾼 것이다. 이렇게 할 경우 선형탐색은 물론이고 재귀탐색 알고리즘을 사용하는 디스어셈블러에서도 잘못된 디스어셈블리 결과를 보이게 할 수 있다.

```
0003bc | 1250
0003be | 2900 0400
0003c2 | 0001
0003c4 | 0000
0003c6 | d800 000
0003ca | 0f00
-------+-----------------------
0000:  | cont/4 v0, #int 5
0001:  | goto/16 0005
0003:  | <Junkbytes>
0004:  | <Junkbytes>
0005:  | add-int/lit8 v0, v0
0007:  | return v0
```

〔그림 9〕 정크바이트를 올바르게 찾은 디스어셈블리 결과

```
0003bc | 1250
0003be | 2900 0400
0003c2 | 0001 0000 d800 0001
0003ca | 0f00
-------+-----------------------
0000:  | const/4 v0, #int 5
0001:  | goto/16 0005
0003:  | packed-switch-data
0007:  | return v0
```

〔그림 10〕 정크바이트로 인해 dexdump〔31〕(선형탐색 알고리즘)가 잘못된 디스어셈블리한 결과

Ⅴ. 보호기법 선택

지금까지 알아본 기법들을 정리하면 세 종류로 분류할 수 있다. 첫 번째로 소스 코드 분석 난이도를 높여주는 난독화 기법이 있다. 두 번째로 소스 코드를 숨기는 실행압축과 코드 분리 기법이 있다. 마지막으로 앞선 보호기법들의 단점을 보완해주는 안티 리버싱 기법이 있다.

```
0003bc | 1250
0003be | 3c00 0400
0003c2 | 0001 0000 d800 0001
0003ca | 0f00
-------+-----------------------
0000:  | const/4 v0, #int 5
0001:  | if-gtz v0, 0005
0003:  | packed-switch-data
0007:  | return v0
```

〔그림 11〕 정크바이트로 인해 대다수의 디스어셈블러가 잘못된 디스어셈블리한 결과

이 기법들을 모두 적용할 경우, 역공학에 대해서는 강인한 면모를 보일지 모르겠으나, 어플리케이션의 성능 저하와 코드 증가 때문에 최선의 선택이 아닐 수 있다. 따라서 어플리케이션 성격에 맞는 적절한 보호기법의 선택이 중요한다.

예를 들어 빠른 반응이 중요시 되는 게임 어플리케이션의 경우, 문자열 암호화나 API 은닉과 같은 난독화를 사용하는 것은 올바르지 않다. 이런 경우 식별자 변환과 실행압축 기술의 사용이 성능하락을 최소화시키며 역공학을 막아줄 수 있는 선택지가 될 것이다. 또한 온라인 게임의 경우 내부 변수 값이 중요하므로 안티 디버깅이나 안티 에뮬레이션의 사용이 필요할 것이다.

〔표 1〕 어플리케이션 중요 요소에 따른 보호기법 만족 여부 (○:높음, △:보통, X:낮음)

	반응속도 감소	성능 감소	크기 증가	보호 강도
식별자 변환	X	X	X	△
제어흐름 변환	△	○	○	△
문자열 암호화	△	△	△	△
API 은닉	○	○	△	△
클래스 암호화	△	X	X	○
실행 압축	△	X	X	○
코드 분리	○	X	X	○

다른 예로, 교육관련 어플리케이션의 경우 게임과 같이 빠른 반응이 요구되지 않으므로 난독화 기술을 사용해 내부 소스 코드 분석을 어렵게 하는 것이 적절한 선택지가 될 수 있다. 또한 이러한 경우에는 빠른 응답이 필요하지 않으므로 중요 콘텐츠가 담긴 코드를 코드 분리 기법을 사용하여 안전하게 보호할 수 있을 것이다.

이렇게 어플리케이션 성격에 따른 적절한 조합은 어플리케이션의 요구사항을 만족시키면서 소스 코드를 역공학으로부터 지킬 수 있게 해준다. 표 1은 어플리케이션이 요구하는 요소에 대한 보호기법의 만족 여부를 정리한 것이다.

Ⅵ. 결 론

본 논문에서는 안드로이드 어플리케이션에 사용되는 다양한 보호기법 유형에 대하여 살펴보았다. 소개된 기법들은 하나의 기법만으로는 완전한 보호 역할을 하지 못하며, 이 보호기법을 모두 적용하는 것 또한 오버헤드 발생과 크기 증가로 인해 적절하지 못하다. 따라서 대상 어플리케이션 특징과 용도에 맞게끔 기법들을 조합하여 적용하는 것 최선이다. 본 논문이 적절한 보호기법 선택에 도움이 될 것으로 기대한다.

참 고 문 헌

[1] Strategy Analytics, http://www.strategy analytics.com

[2] 장준혁, 한승환, 조유근, 최우진, 홍지만, “안드로이드 환경의 보안 위협과 보호 기법 연구 동향,” *보안공학연구논문지*, 11(1), pp.01-12, 2014년 2월.

[3] W.Zhou, et al., “Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces,” *ACM CODASPY 2012,* Feb. 2012.

[4] Staying safe when using mobile banking and payment applications, http://www. bobsguide.com/guide/news/2015/Jun/1/staying-safe-when-using-mobile-banking-and-payment-applications.html

[5] 모든 앱 개발자들은 자신의 앱을 보호받을 권리가 있다, http://www.ddaily.co.kr/news/articl e.html?no=123585

[6] APKTool, http://ibotpeaches.github.io/Apktool/

[7] DEX2JAR, http://sourceforge.net/projects/dex2jar/files/

[8] JD-GUI, http://jd.benow.ca/

[9] 김영기, 윤희용, “자바 클래스 파일 암호화를 이용한 자바 역컴파일 방지기법,” *한국정보과학회 한국컴퓨터종합학술대회 논문집*, 36(1), pp. 571-574, 2009년 6월.

[10] Building and Running Overview, http://developer.android.com/tools/building/index.html

[11] Activity, http://developer.android.com/reference/android/app/Activity.html

[12] Dalvik Executable format, https://source.android.com/devices/tech/dalvik/dex-format.html

[13] 김지윤, 홍수화, 고남현, 이우승, 박용수, “자바 자동 식별자 리네이밍 기법 및 보호 기법,” *한국통신학회 논문지*, 40(4), pp. 709-719, 2015년 4월.

[14] T.W. Hou, H.Y. Chen, and M.H. Tsai, “Three Control Flow Obfuscation Methods for Java Software,” *IEE Proceedings-Software*, Vol. 153, Issue 2, pp. 80-86, Apr. 2006.

[15] 노진욱, 조병민, 오현수, 장혜영, 정민규, 이승원, 박용수, 우제학, 조성제, “C++ 언어를 위한 Control Flow Obfuscator 구현 및 평가,” *한국정보과학회 한국컴퓨터종합학술대회 논문집*, 33(1), pp. 295-297, 2006년 6월.

[16] 최석우, 박희완, 한태숙, “메소드 분산을 통한 자바 프로그램 난독화 기법,” *한국정보과학회 2003년도 가을 학술발표논문집*, 30(2), pp. 238-240, 2013년 10월.

[17] 이병용, 최용수, “Obfuscation 기술의 현황 및 분석과 향후 개발 방향,” *보안공학연구논문지*, 5(3), pp. 219-228, 2008년 6월.

[18] 김정일, 이은주, “제어 흐름 난독화를 효과적으로 수행하기 위한 전략,” *한국컴퓨터정보학회 논문지*, 16(6), pp. 41-50, 2011년 6월.

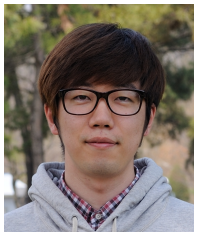
[19] Y. Piao, J.H. Jung, and J.H. Yi, “Server based code obfuscation scheme for APK tamper detection,” *Security and Communication Networks*, Mar. 2014.

[20] Activities, http://developer.android.com/guide/components/activities.html

[21] Common Intents, http://developer.andro id.com/guide/components/intents-common.html
[22] 김지윤, 고남현, 박용수, "안드로이드 환경에서 자바 리플렉션과 동적 로딩을 이용한 코드 은닉법," *정보보호학회논문지*, 25(1), pp. 17-30, 2015년 2월.
[23] Invoking methods, https://docs.oracle. com/javase/tutorial/reflect/member/methodInvocation.html
[24] Android packer: facing the challenges, building solutions, https://www.virus btn.com/conference/vb2014/abstracts/Yu.xml
[25] Guo, Fanglu, Peter Ferrie, and Tzi-Cker Chiueh. "A study of the packer problem and its solutions," *Recent Advances in Intrusion Detection, Springer Berlin Heidelberg*, 2008.
[26] Yeongung Park, "We Can Still Crack You! General Unpacking Method for Android Packer (no root)," *Blackhat Asia 2015*, Mar. 2015.
[27] 박진우, 박용수, "분석 환경에 따른 안티 디버깅 루틴 자동 탐지 기법," *인터넷정보학회논문지*, 15(6), pp. 47-54, 2014년 12월.
[28] Anti-debugging and Anti-VM techniques and anti-emulation, http://resources.infosecinstitute.com/anti-debugging-and-anti-vm-techniques-and-anti-emulation/
[29] M.G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating Emulation-Resistant Malware," *Proceedings of the 1st ACM workshop on Virtual machine security(VMSec '09)*, pp. 11-22, 2009.
[30] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *In CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290-299. 2003.
[31] Dex Dump 1.1.0, http://dex-dump.soft112.com/

〈저자소개〉

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



이 강 효 (Kanghyo Lee)
학생회원
2014년 2월 : 한양대학교 컴퓨터공학과 학사
2014년 3월~현재 : 한양대학교 컴퓨터공학과 석사 과정
관심분야 : 정보보호, 클라우드 보안, 모바일 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