Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.25, NO.5, Oct. 2015 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2015.25.5.1201

APK에 적용된 난독화 기법 역난독화 방안 연구 및 자동화 분석 도구 구현*

이 세 영,† 박 진 형, 박 문 찬, 석 재 혁, 이 동 훈‡
고려대학교 정보보호대학원

A Study on Deobfuscation Method of Android and Implementation of Automatic Analysis Tool*

Se Young Lee,† Jin Hyung Park, Moon Chan Park,
Jae Hyuk Suk, Dong Hoon Lee‡
Graduate School for Information Security, Korea University

요 약

안드로이드 환경에서 악의적인 역공학으로부터 APK(Android application PacKage)를 보호하기 위해 다양한 난독화 도구가 이용되고 있다. 그러나 이런 난독화 도구는 악의적인 공격자에 의해 악용될 수 있으며, 실제로 많은 공격자들이 안티 바이러스 등에 의한 탐지를 우회하기 위해 악성 APK를 난독화하고 있다. 난독화된 악성 APK는 역난독화가 되지 않으면 그 기능성을 분석하는 것이 어렵기 때문에, 난독화된 악성 APK에 대응하기 위해서는 역난독화 방안이 반드시 요구된다. 본 논문에서는 상용 난독화 도구로 난독화된 APK를 분석하고, 적용된 난독화 기법을 정적으로 식별하고 역난독화할 수 있는 방안을 제안한다. 또한 이를 기반으로 난독화 옵션 식별 및 역난독화가 가능한 자동화된 도구를 구현하여 검증한 결과를 제시한다.

ABSTRACT

Obfuscation tools can be used to protect android applications from reverse-engineering in android environment. However, obfuscation tools can also be misused to protect malicious applications. In order to evade detection of anti-virus, malware authors often apply obfuscation techniques to malicious applications. It is difficult to analyze the functionality of obfuscated malicious applications until it is deobfuscated. Therefore, a study on deobfuscation is certainly required to address the obfuscated malicious applications. In this paper, we analyze APKs which are obfuscated by commercial obfuscation tools and propose the deobfuscation method that can statically identify obfuscation options and deobfuscate it. Finally, we implement automatic identification and deobfuscation tool, then show the results of evaluation.

Keywords: android, deobfuscation, obfuscation, DexProtector

I. 서 론

오늘날 스마트폰 시장이 꾸준히 성장하고 이를 이용하여 금융거래, 회사업무 등 일상의 중요 업무 처리가 가능해짐에 따라 스마트폰을 대상으로 하는 악성코드가 점차 증가하고 있다. 특히 안드로이드 기반 스마트폰은 악성코드의 주요 공격 대상으로써 Kaspersky Lab에서 발표한 보고서에 따르면 안드로이드를 대상으로 하는 악성코드는 전체 모바일 대상 악성코드 중 98.05%를 차지하고 있다[1].

접수일(2015년 9월 2일), 게재확정일(2015년 9월 20일)
* 본 연구는 한국연구재단 운영체제 안전성 연구과제(NRF-2014M3C4A7030649)의 일환으로 수행하였습니다.
† 주저자, seyoung0131@korea.ac.kr
‡ 교신저자, donghlee@korea.ac.kr(Corresponding author)

이처럼 안드로이드가 스마트폰 악성코드의 주요 공격 대상이 되는 이유는 스마트폰 플랫폼 시장에서 안드로이드가 높은 점유율을 차지하고 있고, 기존 APK의 리패키징을 통한 악성 APK의 제작과 배포가 용이하기 때문이다. 실제로 안드로이드는 전체 스마트폰 플랫폼 시장의 약 80%를 점유[2]하고 있기 때문에 악성코드에 의한 파급 효과가 타 플랫폼보다 훨씬 크다고 할 수 있다. 또한 안드로이드 어플리케이션은 역공학과 리패키징 절차가 잘 알려져 있기 때문에 악성코드 제작자는 잘 알려진 어플리케이션에 공격 코드를 삽입한 악성코드를 비교적 쉽게 제작할 수 있고, 이를 개방형 마켓에 등록함으로써 악성코드의 배포도 용이하게 할 수 있다. Yajin Zhou 등의 연구[3]에서는 대다수의 악성 APK 중 86%가 리패키징된 APK임을 보였으며, 또한 Pulse Secure에 따르면 악성 APK 중 99.9%는 서드 파티(third party) 마켓에서 발견되었다[4].

악성 APK는 ProGuard[5], DexGuard[6], DexProtector[7] 등의 알려진 안드로이드 어플리케이션 보호 도구로 난독화되어 더욱 문제가 되고 있다. 난독화란 프로그램의 기능성은 유지하면서 역공학은 어렵게 하기 위해 내부 구조를 변환하는 기법으로써[8], 악성 APK에 난독화가 적용되면 안티 바이러스 등을 우회할 수 있기 때문에 그 피해가 더욱 커질 수 있다. 난독화된 APK는 역난독화가 되지 않으면 그 기능성을 분석하는 것이 어렵다. 그러므로 악성 APK에 적용된 난독화 기법을 분석하고 역난독화하는 연구가 점차 중요해지고 있다.

본 논문의 분석 대상은 안드로이드 기반 상용 난독화 도구인 DexProtector이다. DexProtector에서는 문자열 암호화, 실행코드 암호화, 리소스 암호화의 난독화 옵션을 제공하며 각 옵션을 독립적으로 선택하여 적용하거나 여러 옵션을 동시에 적용 할 수 있다. 따라서 같은 악성 APK를 DexProtector로 난독화하더라도 적용한 난독화 옵션의 종류에 따라 다양한 형태의 변종 악성 APK가 나올 수 있다. 이러한 APK에 대응하기 위해서는 적용된 난독화 옵션이 식별이 선행되어야 하며, 이후 각 난독화 옵션 별로 역난독화할 수 있는 방안에 대한 연구가 반드시 필요하다.

본 논문에서는 DexProtector에서 제공하는 각 난독화 옵션의 특징 및 동작 방식을 파악하기 위해 DexProtector로 난독화한 APK를 분석한 결과를 설명한다. 이를 기반으로 DexProtector로 난독화된 APK에 적용된 난독화 옵션을 식별할 수 있는 방안과 식별한 난독화 옵션에 대해 역난독화를 수행할 수 있는 방안을 제안한다. 마지막으로 자동화된 역난독화 도구를 구현하여 DexProtector로 난독화된 APK를 입력받았을 때, 적용된 난독화 옵션을 정적으로 식별하고 이를 역난독화한 형태로 출력할 수 있도록 하였다.

본 논문의 기여도는 다음과 같다.

- DexProtector가 제공하는 각 난독화 옵션의 고유한 특징 및 코드 패턴을 추출하여 이를 난독화 옵션 식별에 활용할 수 있는 방안을 제안하였다.
- DexProtector의 각 난독화 옵션별 기능 및 동작 방식을 분석하고, 이를 역난독화할 수 있는 방안을 제안하였다.
- 이를 구현한 자동화된 정적 분석 도구는 DexProtoector로 난독화된 APK를 입력받으면 이를 역난독화한 APK 파일 형태로 출력해준다. 이는 악성 APK 분석가에게 난독화 루틴이 아닌 실제 악성 행위에 관한 루틴만 집중하여 분석할 수 있는 환경을 마련하는데 기여할 수 있다.
- 분석 도구를 정적으로 구현하였기 때문에 동적 분석 도구가 가지는 한계점을 극복할 수 있고, 환경의 제약 없이 본 도구를 활용할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 APK에 대한 분석 방법에 대한 배경 지식 및 APK의 역난독화와 관련된 연구를 정리한다. 3장에서는 APK에 적용된 난독화 옵션을 식별하고 이를 역난독화하기 위한 분석 과정 및 결과를 서술한다. 4장에서는 분석 결과를 이용하여 자동화 식별 및 역난독화 도구를 구현하고 5장에서는 자동화 식별 및 역난독화 도구의 검증 결과를 제시한다. 마지막으로 6장에서는 결론을 맺는다.

II. 배경 지식 및 관련 연구

이번 장에서는 APK 분석 방법에 대한 배경지식과 APK의 역난독화 관련 연구를 정리한다.

2.1 APK 분석 방법

안드로이드 APK를 분석하는 방법은 APK를 실

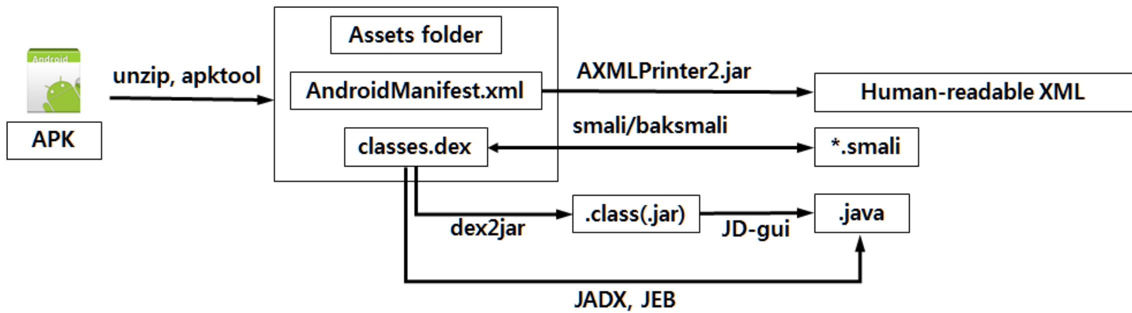

Fig. 1. Flow of General Static Analysis

행하지 않고 APK 자체를 분석하여 내부 함수 사이의 관계 및 구조를 파악하는 정적 분석 방법과 APK를 실행하면서 발생하는 행위를 분석하는 동적 분석 방법으로 분류할 수 있다[9].

정적 분석은 APKtool[10], Smali/Baksmali[11] 등의 도구를 활용하여 DEX(Dalvik EXecutable format) 코드와 대응되는 smali 코드를 변환하여 분석하거나 dex2jar[12], JD-GUI[13], Jadx[14] 등의 도구로 DEX 코드를 자바 소스 코드로 디컴파일 하여 분석하는 방법이다. 일반적인 정적 분석의 절차는 Fig.1.과 같다. 정적 분석 방법은 APK의 실행이 요구되지 않기 때문에 동적 분석 방법에 비하여 빠른 분석을 할 수 있으며 모든 영역에 대한 분석이 가능하다[15]. 그러나 난독화 혹은 패킹 기법이 적용된 APK에 대한 분석이 어렵다는 한계점이 있다.

동적 분석은 안드로이드 디버깅 도구를 이용하거나 Sandbox, 가상 머신 등의 분석 환경에서 APK를 실행시켜 실행 루틴을 분석하는 방법이다. 이러한 동적 분석은 실제 실행되는 루틴의 분석이 가능하다는 장점이 있지만 실행되지 않는 루틴은 분석하기 어렵다는 한계점이 있다. 또한 디버깅 도구 및 분석 환경을 탐지·우회하는 기법이 적용된 APK는 동적 분석을 수행할 수 없으며[16] 특정 이벤트가 발생했을 때 실행되도록 구현된 코드에 대해서도 분석을 수행할 수 없다[17]. 그러므로 이러한 APK의 분석을 위해서는 정적인 분석이 요구된다.

2.2 관련 연구

Hannes Schulz 등의 연구[18]는 안드로이드 상용 난독화 도구인 DexGuard에서 제공하는 문자열 암호화 옵션에 대한 역난독화 방법을 제안하였다. Hannes Schulz 등은 암호화된 문자열을 복호화하는 복호화 메서드와 입력 파라미터를 정적으로 추출하고 이를 호출하는 도구를 구현하여 복호화된 문자열을 추출하였다. 이후 암호화된 문자열을 추출한 복호화 문자열로 치환하고 복호화 메서드를 지움으로써 역난독화를 수행하였다. 그러나 복호화 메서드를 찾을 수 없도록 클래스 암호화 옵션까지 다중 적용된 경우에는 역난독화를 수행할 수 없다는 한계점이 있다.

Yuxue Piao 등의 연구[19] 역시 DexGuard를 분석 대상으로 하였으며 문자열 암호화와 클래스 암호화 옵션을 분석하였다. 문자열 암호화 옵션의 경우 복호화 메서드가 호출될 때 복호화된 문자열을 로그로 출력하고, 이 로그가 원본 문자열과 동일함을 보임으로써 원본 문자열을 추출할 수 있음을 보였다. 클래스 암호화 옵션의 경우 DEX 코드 내에 복호화 메서드, 키, IV(Initial Vector)를 추출하고 복호화 메서드 호출을 통해 복호화된 클래스 파일을 추출하였다. 그러나 Yuxue Piao 등은 문자열 암호화 옵션의 경우 역난독화를 위해 동적으로 로그를 출력하는 방법을 활용하였다. 이는 동적 분석 환경을 탐지·우회하는 APK인 경우 문자열의 역난독화를 수행할 수 없으며, 실행되지 않는 루틴의 문자열은 추출할 수 없다. 악성 APK 내 공격자 IP 주소나 호출되는 악성 API 명 등의 문자열이 역난독화 되지 않으면 해당 APK의 악성 여부를 판단하지 못할 수 있다.

[18]과 [19]는 복호화 메서드와 키와 IV의 입력 파라미터만을 사용하여 역난독화를 수행할 수 있었다. 그러나 본 논문의 역난독화 대상 프로그램인 DexProtector는 암호화된 정보를 복호화하기 위해 APK의 시그니처, 클래스·메서드 명 등 외부의 정보를 활용해 키와 IV를 생성하고 복호화를 수행한다. 따라서 DexProtector의 분석을 위해서는 복호화 메서드 내부 루틴에 대한 추가적인 분석이 요구된다.

또한 Hannes Schulz 등과 Yuxue Piao 등은 단일 난독화 옵션이 적용된 경우에 대한 역난독화 연구를 수행하였다. 일반적으로 난독화 옵션을 적용할 때 하나의 난독화 옵션 적용이 아닌 여러 난독화 옵션을 다중 적용하는 경우가 많다. 따라서 다중으로 난독화 옵션이 적용된 경우의 적용된 난독화 옵션 식별 및 역난독화 연구가 요구된다. 본 논문에서는 리소스 암호화 옵션을 포함하여 다중으로 난독화 옵션이 적용된 APK에 대해서도 난독화 옵션 식별 및 역난독화가 가능함을 보인다.

III. DexProtector 난독화 옵션 식별 및 역난독화

이번 장에서는 본 논문에서 분석 대상인 DexProtector의 난독화 옵션에 대하여 설명하고 APK에 적용된 난독화 옵션을 정적으로 식별하고 역난독화하기 위해 수행한 분석 과정을 서술한다. 모든 분석 과정은 APK의 DEX 파일을 smali 코드로 추출 및 변환하여 수행하였다. 변환된 각 smali 코드는 하나의 클래스를 나타낸다. 디버깅 도구 및 동적 분석 환경을 탐지·우회하는 기법이 적용된 APK는 동적으로 역난독화하기 어렵기 때문에 정적인 역난독화 방안이 요구된다. 따라서 본 논문의 분석 결과를 활용하면 동적 분석 환경을 탐지·우회하는 APK에 대해서도 정적으로 역난독화할 수 있다.

3.1 DexProtector 난독화 옵션

본 논문의 분석 대상은 상용 난독화 도구인 DexProtector v.5.0.3이며, 도구가 제공하는 옵션 중에서 세 가지 난독화 옵션(문자열/클래스/리소스 암호화)을 대상으로 연구를 수행하였다. 각 난독화 옵션은 단일 또는 다중으로 적용 가능하며 Table 1.은 3개의 난독화 옵션과 각각의 기능을 나타낸 표이다.

3.2 난독화 옵션 식별을 위한 분석 과정

다중으로 난독화 옵션이 적용된 APK를 분석하기 위해서는 적용된 난독화 옵션의 식별이 선행되어야 한다. 이번 절에서는 난독화 옵션이 적용된 APK에 대하여 난독화 옵션을 식별하기 위한 분석 과정을 서술한다. 분석을 통하여 알아낸 각 난독화 옵션의 식별 요소는 정적으로 추출 가능하며 적용된 난독화 옵션을 식별하기 위해 활용할 수 있다.

3.2.1 문자열 암호화

문자열 암호화 옵션은 원본 APK의 내부에 선언된 모든 문자열을 암호화한다. Fig.2.는 원본 APK의 문자열과 문자열 암호화 옵션이 적용된 APK의 암호화된 문자열을 나타낸 것이다. 암호화가 적용된 문자열은 원본 문자열인 “Hello”와는 달리 알아볼 수 없는 형태인 바이트 배열이며 문자열이 사용되는 시점에 동적으로 복호화된다.

문자열 암호화 적용 시 암호화된 문자열 아래에는 새로운 명령어 루틴(*invoke-static*, *move-result-object*)이 추가된다(본 논문에서는 새롭게 추가된 명령어 루틴을 복호화 루틴이라 한다). *invoke-static*은 특정 클래스의 메서드를 호출하는 명령어이고 *move-result-object*는 *invoke-static*의 결과를 객체에 저장하는 명령어이다. 암호화된 문자열은 DexProtector에서 생성한 복호화 클래스의 복호화 메서드를 통해 복호화된다.

복호화 클래스는 문자열 암호화 옵션 적용 시 새롭게 추가되는 클래스이며, 이 클래스를 분석할 수 있는 정형화된 방법을 찾기 위해 문자열 암호화 옵션이 적용된 APK를 다수 생성하여 복호화 클래스를 비교 분석하였다. 비교 분석 결과로써 복호화 클래스 내부에는 Fig.3.과 같이 필드 1개와 메서드 4개가 항상 포함되어 있음을 확인하였다. 필드와 메서드의

Table 1. Obfuscation options of Dexprotector

Option	Description
String Encryption	String Encryption converts the constant string value to meaningless data. When using it, the data is decrypted dynamically.
Class Encryption	Class Encryption protects class files. Classes protected by Class Encryption are encrypted and moved from classes.dex(common storage of all classes in the APK).
Resource Encryption	Resource Encryption protects resources in assets directory of APK from duplication and modification. The encrypted resource files are not readable.

식별자는 문자열 암호화 옵션 적용 시마다 다르지만 키워드와 메서드의 입·출력 자료형은 항상 동일하다. 이를 문자열 암호화의 코드 패턴이라 정의한다.

따라서 문자열 암호화 옵션 적용 시 추가되는 Fig.2.의 복호화 루틴과 문자열 암호화의 코드 패턴을 이용하여 문자열 암호화 옵션을 식별할 수 있다. 이를 문자열 암호화 옵션을 식별하기 위한 시그니처로 정의한다.

```
const-string v0, "Hello"
invoke-virtual {v2, v0},
   Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
```

(A)

```
const-string v0, "₩u8cf2₩u0cc8₩u4c4c₩u2287₩u5e24 "
invoke-static {v0},
   Decryption Class;->Decryption Method(Ljava/lang/String;)Ljava/lang/String;
move-result-object v0
invoke-virtual {v2, v0},
   Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
```

(B)

Fig. 2. (A) before String Encryption, (B) after String Encryption

3.2.2 클래스 암호화

클래스 암호화 옵션은 원본 APK의 DEX 파일을 암호화하여 역공학 시에 원본 클래스 파일의 추출을 어렵게 한다. Fig.4.는 원본 APK의 파일 구조와 클래스 암호화 옵션이 적용된 APK의 파일 구조를 나타낸다.

Fig.4.와 같이 클래스 암호화 옵션이 적용되면

```
# static fields
.field private static transient name_1:[Ljava/lang/Object;

# direct methods
.method private static final method_name_1()void
.method private static final method_name_2(int, int)int
.method private static final method_name_3(byte[], int)int
.method static final method_name_4(Ljava/lang/String;)Ljava/lang/String;
```

Fig. 3. The code pattern of String Encryption

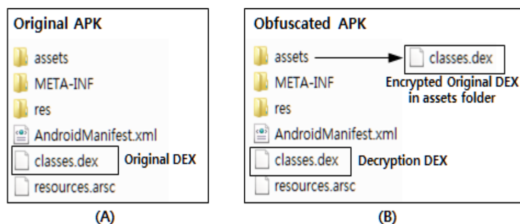

Fig. 4. (A) before Class Encryption, (B) after Class Encryption

assets 폴더 내부에 DEX 파일이 새롭게 추가되며, 이 DEX 파일의 매직 넘버는 일반적인 DEX 파일의 매직 넘버가 아닌 임의의 값으로 변형된다. 이를 통해 assets 내부의 DEX 파일은 암호화 되어 있음을 예상할 수 있으며 실제 분석을 통해 이 DEX 파일이 암호화된 원본 DEX 파일임을 확인하였다. 즉, 클래스 암호화 옵션을 적용하면 원본 APK의 DEX 파일이 암호화 되어 assets 폴더 내부에 저장되고, 이를 동적으로 복호화하기 위한 새로운 DEX 파일이 추가된다(새롭게 추가된 DEX 파일을 복호화 DEX 파일이라고 칭한다).

클래스 암호화 옵션이 적용된 APK를 다수 생성하

```
# static fields
.field private static transient name_1:[Ljava/lang/Object;
.field private static name_2:Z

# direct methods
.method public constructor <init>()V
.method private static final method_name_1()void
.method private static final method_name_2(int, int)int
.method private static final method_name_3(byte[], int)int
.method static final method_name_4(Ljava/lang/String;)Ljava/lang/String;
.method private method_name_5()V

#virtual method
.method protected method_name_6(Landroid/content/Context;)V
```

Fig. 5. The code pattern of Class Encryption

```
<manifest package="xray.ruocco" ... >
        abbreviate
   <application ... >
        abbreviate
   </application> </manifest>                before Class Encryption
```

```
<manifest package="xray.ruocco" ... >
        abbreviate
   <application android:name=".Applicaion" ... >
        abbreviate
   </application> </manifest>                after Class Encryption
```

(case 1)

```
<manifest package="fr.nghs.android.dictionnaires" ... >
        abbreviate
   <application android:name="fr.nghs.android.dictionnaires.DApp"... >
        abbreviate
   </application> </manifest>                before Class Encryption
```

```
<manifest package="fr.nghs.android.dictionnaires" ... >
        abbreviate
   <application android:name=".ProtectedDApp" ... >
        abbreviate
   </application> </manifest>                after Class Encryption
```

(case 2)

Fig. 6. (case 1) : Addition of "android:name" attribute in 〈application〉 element, (case 2) : Change of "android:name" attribute in 〈application〉 element

고 각각의 복호화 DEX 파일의 클래스를 비교하였다. 비교 결과, 복호화 DEX 파일은 하나의 클래스만을 가지고 있으며 이 클래스는 문자열 암호화의 코드 패턴과는 다른 코드 패턴을 가지고 있다. Fig.5.는 클래스 암호화의 코드 패턴을 나타내는 것으로 문자열 암호화의 코드 패턴에 추가적으로 1개의 필드와 3개의 메서드가 추가된 패턴이다.

Fig.6.은 원본과 난독화된 APK의 XML 파일 내 변화를 나타낸 그림이다. 원본 APK와 비교하여 클래스 암호화가 적용된 APK는 XML 파일의 〈application〉 요소에 "android:name" 속성이 추가되거나 수정된다. 이 속성 값에는 항상 ".Application" 또는 ".Protected" 문자열이 포함되어 있으며 복호화 DEX 파일로부터 변환된 하나의 클래스를 가리킨다.

따라서 클래스 암호화 옵션은 assets 폴더 내의 DEX 파일 존재 여부와 매직 넘버, 클래스 암호화의 코드 패턴, XML 파일의 "android:name" 속성을 확인하여 식별 가능하다. 이러한 특징을 클래스 암호화 옵션의 시그니처로 정의한다.

3.2.3 리소스 암호화

리소스 암호화 옵션은 APK에서 사용하는 assets 폴더 내의 리소스를 암호화하여 도용·변조되는 것을 방지한다. 이 옵션이 적용된 APK의 리소스 파일은 이름은 변형되지 않지만 그 내용은 암호화되어 정적으로는 읽을 수 없고, 호출 시에 동적으로 복호화되어 사용된다.

암호화된 리소스의 바이너리를 확인해 보면 Fig. 7.과 같이 암호화된 리소스 파일의 매직 넘버는 확장자에 해당하는 매직 넘버가 아닌 임의의 값으로 변형되며, 모든 암호화된 리소스들은 동일한 첫 4바이트의 값을 가진다. 예를 들어, Fig.8.과 같이 암호화된 APK 내 리소스 파일들은 확장자가 모두 다르더라도 첫 4바이트는 모두 동일하다.

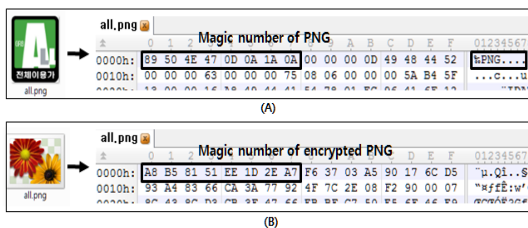

Fig. 7. (A) before Resource Encryption, (B) after Resource Encryption

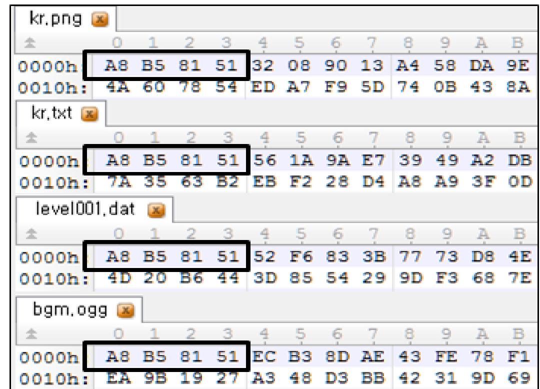

Fig. 8. The encrypted resource files have the same header, but the extension of the files is different

원본 APK와 리소스 암호화가 적용된 APK의 리소스 호출 코드 부분을 비교하면 Fig.9.와 같이 리소스 암호화 옵션 적용 시 새로운 명령어 루틴(*invoke-static*, *move-result-object*)이 추가된다. 이는 문자열 암호화 옵션과 유사한 특징이지만 리소스 암호화 옵션의 경우 이 루틴이 AssetManager 클래스의 "open" 메서드 호출 뒤에 추가된다. AssetManager 클래스는 assets 폴더 내의 파일과 관련된 클래스이다. 암호화된 리소스

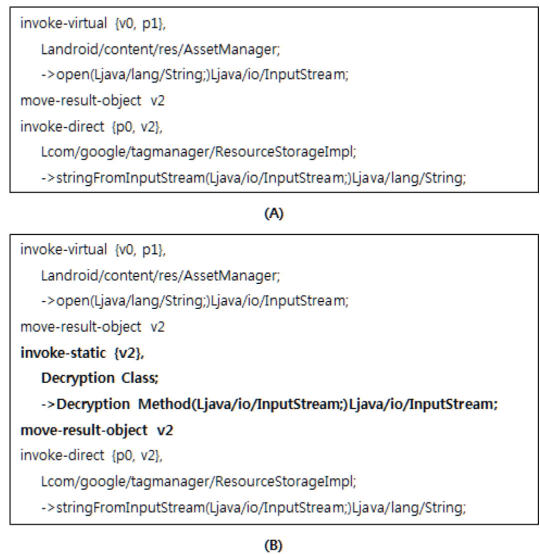

Fig. 9. (A) before Resource Encryption, (B) after Resource Encryption

는 invoke-static 명령어를 통해 InputStream 자료형으로 복호화 되어 사용된다.

그리고 클래스 암호화 옵션과 동일하게 리소스 암호화 옵션 역시 XML 파일의 〈application〉 요소에 “android:name” 속성이 추가되거나 수정된다. 이 속성 값 역시 “.Application” 또는 “.Protected” 문자열이 포함되어 있다.

따라서 리소스 암호화 옵션을 식별하기 위해 assets 폴더 내 리소스 파일의 바이너리와 Fig.9.와 같이 난독화 시 추가되는 복호화 루틴, XML 파일의 “android:name” 속성을 활용할 수 있다. 이를 리소스 암호화 옵션을 식별하기 위한 시그니처로 정의한다.

3.2.4 분석 결과

Table 2.는 각 난독화 옵션의 시그니처를 정리한 것이며 이를 통해 APK에 적용된 각 난독화 옵션을 정적으로 식별할 수 있다.

3.3 난독화 옵션 분석 과정

본 논문에서 역난독화하고자 하는 난독화 옵션은 DexProtector에서 제공하는 문자열/클래스/리소스 암호화 옵션이다. 역난독화를 위해서 우선적으로 복호화 과정에 대한 분석을 수행하였으며, 이를 통해 복호화 과정에서 사용되는 변수와 메서드의 호출 관계를 파악하였다. 이번 절에서는 각각의 난독화 옵션을 역난독화하기 위한 분석 과정을 제시한다.

3.3.1 문자열 암호화

Fig. 2.의 (B)와 같이 복호화 루틴은 우선 변수에 암호화된 문자열을 저장하고 이를 인자로 하여 복호화 메서드를 호출한다. 복호화 메서드의 결과를 다시 변수에 저장한 이후에는 원본 루틴과 동일하게 동작한다.

3.2절에서 수행한 분석 과정을 통해 복호화 클래스 내부의 동일한 영역 및 차이가 있는 영역을 확인할 수 있었으며, 이 중 동일한 영역은 AES 복호화 루틴임을 확인하였다. 그리고 차이가 있는 영역은 키와 IV를 동적으로 생성하기 위한 값이며 문자열 암호화 적용 시마다 임의의 값을 가진다.

복호화 클래스 내에서 키와 IV는 Fig.10.과 같은 방식으로 생성된다. 먼저 키는 키 참조 배열(ref_key)을 인덱스로 하여 두 개의 치환 테이블(S_Table_1, S_Table_2)을 순차적으로 참조해 생성된다. 복호화 대상 문자열을 포함하는 클래스·메서드 명으로 생성된 해시 코드(hashCode)와 IV 참조 배열(ref_iv)을 XOR하여 IV가 생성된다. 해시 코드를 생성하기 위하여 필요한 클래스·메서드 명은 두 개의 메서드(getClassName, getMethodName)를 호출하여 구한다. 이 호출되는 두 메서드 명의 문자열은 치환 테이블(S_Table_1)에 의해 저장되어 있으므로, 이를 복원하는 과정이 선행된다.

따라서 복호화 대상 문자열을 포함하는 클래스·메서드 명, 그리고 복호화 클래스 내의 키·IV 참조 배열과 치환 테이블을 파싱함으로써 암호화된 문자열을 정적으로 복호화할 수 있다. Fig.11.은 문자열 암호화 옵션의 복호화 과정을 도식화한 그림이다.

DexProtector는 복호화 과정에서 복호화 클래스 외부의 정보(복호화 대상 문자열을 포함하는 클래스·메서드 명)를 이용하기 때문에 [18]과 같이 복호화 클래스만을 추출하는 방법으로는 역난독화를 수행할 수 없다.

Table 2. Signatures of obfuscation options

Option	Signatures
String Encryption	• Decryption routine of String Encryption • Code pattern of String Encryption
Class Encryption	• Magic number of existent DEX file in assets folder • Code pattern of Class Encryption • String value of <application android:name> in AndroidManifest.xml
Resource Encryption	• Magic number of resources in assets • Decryption routine of Resource Encryption • String value of <application android:name> in AndroidManifest.xml

```
// KEY generation
byte[ ] KEY = new byte[16];
Key[0] = S_Table_1[ S_Table_2[ ref_key[0] ] ] ;
Key[1] = S_Table_1[ S_Table_2[ ref_key[1] ] ] ;
                abbreviate
Key[15] = S_Table_1[ S_Table_2[ ref_key[15] ] ] ;

// IV generation
Temp = ( getClassName( )||getMethodName( ) ).hashCode( ) ;

int[ ] IV = new int[4];
IV[0] = Temp ^ ref_iv[0] ;
IV[1] = Temp ^ ref_iv[1] ;
                abbreviate
IV[3] = Temp ^ ref_iv[3] ;
```

Fig. 10. The pseudocode represents the generating process of the KEY and the IV in String Decryption class

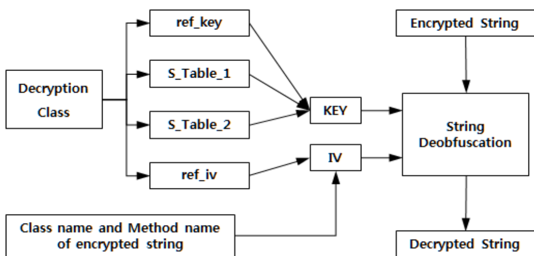

Fig. 11. Flow of String Decryption processing

3.3.2 클래스 암호화

클래스 암호화 옵션이 적용된 APK는 assets 폴더 내부에 원본 DEX 파일이 암호화 되어 있으므로 이 DEX 파일이 복호화되기 전까지는 분석하기 어렵다. 따라서 클래스 암호화 옵션 적용 시 추가되는 복호화 DEX 파일의 분석이 먼저 수행되어야 한다.

복호화 DEX 파일은 하나의 클래스만을 가지고 있으며 이 클래스는 Fig.12.의 (A)와 같이 문자열 암호화가 적용되어 있기 때문에 문자열 복호화가 요구된다. 이에 따라 문자열 암호화 옵션의 복호화 과정을 선행하였다.

```
    const-string v30, "₩u1982₩uf678₩u7251₩ub94d₩u0
₩udfb6₩u6429₩u8e2f₩ube10₩ub64f₩u4e80₩u8630₩u
₩ue682₩u8af6₩u1404₩uff9a₩udde2₩uad25₩u35ac₩u
₩u650e₩u81db₩ubbee₩u62af₩u4d33₩ua77e₩u3862₩
```
(A)

```
    const-string v30, "PK
 classes.dex
```
Magic Number of ZIP file

(B)

Fig. 12. (A) before String Decryption, (B) after String Decryption

Fig.12.의 (B)와 같이 복호화된 문자열 중 하나의 문자열은 앞부분이 “PK”로 시작함을 볼 수 있다. 이는 ZIP 파일의 매직넘버를 나타내는 것이므로 해당 문자열을 ZIP 파일로 변환 후 압축을 푼 결과 새로운 DEX 파일을 추출할 수 있었고, 이 안에 암호화된 DEX 파일을 복호화하는 클래스가 존재함을 확인할 수 있었다.

복호화 클래스는 APK의 인증서의 해시 코드 값과 내부의 키 참조 배열(ref_key)을 통해 키를 생성하고 assets 폴더 내의 암호화된 원본 DEX 파일을 읽어 복호화한다. 문자열 암호화 옵션에서 AES를 사용한 것과는 달리 클래스 암호화 옵션에서는 자체적인 복호화 알고리즘을 사용한다.

따라서 assets 폴더 내의 암호화된 DEX 파일과 APK의 인증서 정보, 그리고 추출한 복호화 클래스 내의 키 참조 배열을 파싱하여 암호화된 DEX 파일을 복호화하는 것이 가능하다. 클래스 암호화 옵션 역시 복호화 클래스 외부의 정보(APK의 인증서)를 사용하기 때문에 복호화 클래스의 추출만으로는 역난독화를 수행할 수 없다. Fig.13.은 클래스 암호화 옵션의 복호화 과정을 도식화한 그림이다.

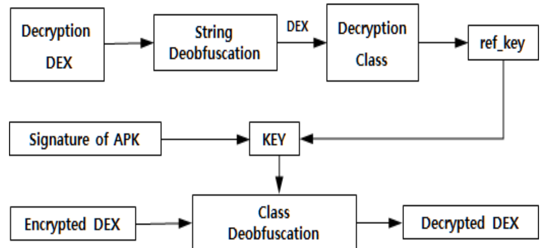

Fig. 13. Flow of Class Decryption processing

3.3.3 리소스 암호화

리소스 암호화 옵션이 적용된 APK는 assets 폴더 내의 암호화된 리소스를 호출 시에 동적으로 복호화하여 사용한다. 리소스 복호화 과정을 분석하기 위해서는 Fig.8.과 같이 복호화 루틴에서 호출하는 복호화 클래스의 분석이 요구된다. 그러나 DEX 파일로부터 변환된 smali 코드들에는 이 복호화 클래스가 존재하지 않았기 때문에 바로 복호화 클래스의 분석을 수행할 수 없었다. 따라서 APK의 실행흐름에

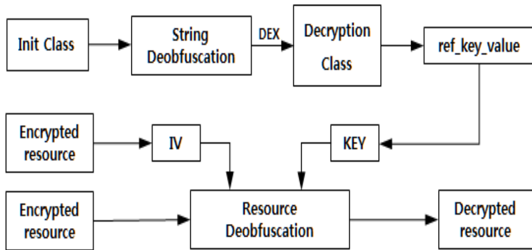

Fig. 14. Flow of Resource Decryption processing

대한 분석을 먼저 수행하였다.

Fig.14.는 리소스 암호화 역난독화 과정을 도식화한 그림이다. 리소스 암호화 옵션 적용 시 XML 파일 〈application〉 요소의 “android:name” 속성 값이 가리키는 클래스가 추가·변경된다(이를 초기 실행 클래스라 칭한다). 이 클래스에는 문자열 암호화가 적용되어 있으며, APK는 실행 시 이 초기 실행 클래스를 가장 먼저 실행한다. 따라서 클래스 암호화 옵션과 동일한 과정으로 문자열을 복호화하여 새로운 DEX 파일을 추출하였으며, 이 DEX 파일에 복호화 클래스가 존재함을 확인하였다.

복호화 클래스 분석을 통해 DexProtector는 암호화된 리소스를 복호화하기 위해 AES 복호화 루틴을 사용하며 복호화 시에 사용되는 키와 IV는 Fig. 15.와 같은 방식으로 생성한다. 우선 암호화된 리소스의 복호화 시에 리소스의 첫 4바이트와 복호화 클래스 내부에 저장된 4바이트가 값이 동일한지 검증한다. 만약 동일하지 않다면 해당 리소스의 복호화 과정을 중지한다. 검증이 통과될 경우 암호화된 리소스의 첫 16바이트를 IV로 사용한다. 그리고 복호화 클래스 내부의 키 참조 값(ref_key_value)으로부터 키를 생성한다. 이 키와 IV를 이용해 암호화된 리소스를 복호화한다. 따라서 키 참조 값과 IV를 파싱하여 암호화된 리소스를 복호화할 수 있다.

```
// KEY generation
long  Long_1 = ref_key_value_1 ;
long  Long_2 = ref_key_value_2 ;

byte[ ]  ref_key = new byte[16] ;
for( int i=0; i<16; i++ ){
   ref_key[ i ] = (i <= 7 ? Long_1 : Long_2) >> ( ( 7 - (i % 8) ) * 8 ) ;
}
int[ ] KEY = new int[4];
KEY[0] = ref_key[3] || ref_key[2] || ref_key[1] || ref_key[0] ;
KEY[1] = ref_key[7] || ref_key[6] || ref_key[5] || ref_key[4] ;
                abbreviate
KEY[4] = ref_key[15] || ref_key[14] || ref_key[13] || ref_key[12] ;

// IV generation
if ( Check( res_byte[0], res_byte[1], ..., res_byte[3] ) ){
   int[ ] IV = new int[4];
   IV[0] = res_byte[0] || res_byte[1] || res_byte[2] || res_byte[3] ;
   IV[1] = res_byte[4] || res_byte[5] || res_byte[6] || res_byte[7] ;
                abbreviate
   IV[3] = res_byte[12] || res_byte[13] || res_byte[14] || res_byte[15] ;
}
```

Fig. 15. The pseudocode represents the generating process of the KEY and the IV in Resource Decryption class

```
# static fields
.field public static final str_1:Ljava/lang/String;  = "Hello"
.field private static final str_2:Ljava/lang/String;  = "Hello_2"

#virtual methods
.method protected onCreate(Landroid/os/Bundle)V
    .locals 2
    .param p1, "savedInstanceState"  # Landroid/os/Bundle;
    const/4 v0, 0x0
    .local v0, "i": I
    const/4 v1, 0x1
    .local v0, "j": I
                abbreviate
.end method
```

(A)

```
# static fields
.field public static final str_1:Ljava/lang/String;
.field private static final str_2:Ljava/lang/String;

#direct methods
.method static constructor <clinit>()V
    .locals 1
    const-string v0, "Hello"
    sput-object v0,
       Lcom/example/hello/TEST;->str_1:Ljava/lang/String;
    const-string v0, "Hello_2"
    sput-object v0,
       Lcom/example/hello/TEST;->str_2:Ljava/lang/String;
    return-void
.end method

#virtual methods
.method protected onCreate(Landroid/os/Bundle)V
    .locals 2
    const/4 v0, 0x0
    const/4 v1, 0x1
                abbreviate
.end method
```

(B)

Fig. 16. (A) before pre-processing, (B) after pre-processing

3.4 기타

DexProtector의 분석 과정에서 난독화 옵션 적용 시 공통적으로 수행되는 전처리 단계와 다중 난독

화 옵션 적용 순서를 파악할 수 있었다. DexProtector의 전처리 단계에서는 Fig.16.과 같이 클래스 내의 변수 및 메서드 인자 관련 정보(키워드, 식별자 등)를 삭제하고, static final String의 초깃값은 클래스 초기화 메서드(clinit)에서 초기화되도록 수정한다. 이와 같이 전처리 단계에서 변환되는 정보는 되돌릴 수 없는 정보이거나 역난독화할 필요가 없는 정보이기 때문에 역난독화 도구 구현에서는 다루지 않는다. 난독화 옵션 다중 적용 시에는 문자열, 리소스, 클래스 암호화 순으로 옵션이 적용된다.

IV. 자동화 식별 및 역난독화 도구 구현

이번 장에서는 3장의 분석 결과를 바탕으로 DexProtector로 난독화된 APK에 대해서 적용된 난독화 옵션을 식별하고 이를 역난독화하기 위한 도구를 구현한다.

자동화 옵션 식별 및 역난독화 도구의 구성도는 Fig.17.과 같으며 Fig.18.과 같이 하나의 통합 프로그램으로 구현되어 있다. 이 도구는 초기 단계(initial phase), 식별 및 역난독화 단계(identification and deobfuscation phase), 최종 단계(final phase)로 구성되어 있다.

초기 단계에서는 입력된 APK로부터 필요한 파일을 추출한다. 식별 및 역난독화 단계에서는 각 난독화 옵션을 식별하고 역난독화한다. 그리고 최종 단계에서는 식별된 난독화 옵션과 역난독화된 APK가 출력된다. 본 도구는 APK 파일을 정적으로 역난독화하는 도구이기 때문에 동적 분석 환경을 탐지·우회하는 APK에 대해서도 역난독화를 수행 가능하다는 장점이 있다.

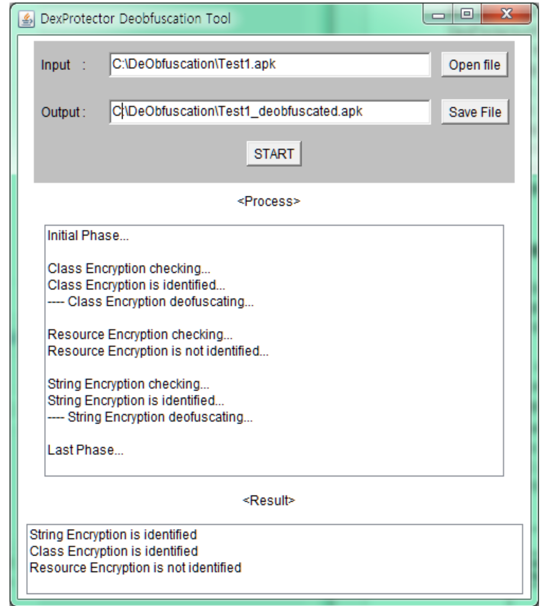

Fig. 18. Automatic Obfuscation Identification and Deobfuscation Tool

4.1 초기 단계

초기 단계는 난독화 옵션의 식별 및 역난독화를 위해 우선적으로 수행되는 단계이다. 입력된 APK로부터 XML 파일, DEX 파일, assets 폴더를 추출한다. 또한 DEX 파일로부터 복호화 루틴 및 복호화 클래스 정보들을 추출하기 위해 이 DEX 파일을 smali 코드로 변환한다.

4.2 난독화 옵션 식별 및 역난독화 단계

난독화 옵션 식별 및 역난독화 단계는 클래스/리소스/문자열 모듈 3개로 구성되어 있다. 각 모듈은

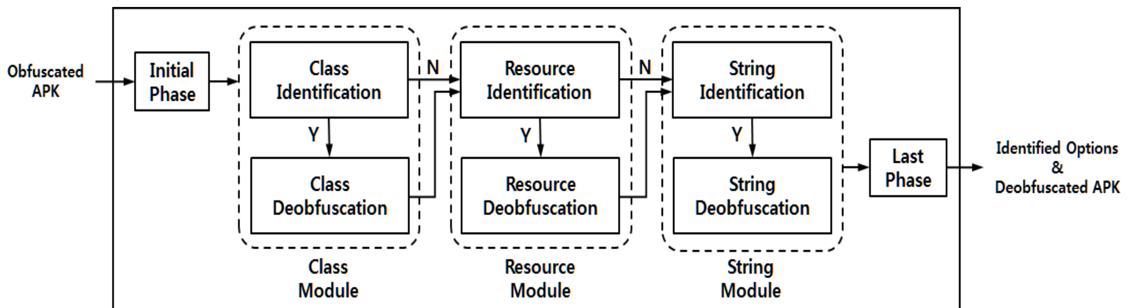

Fig. 17. Flow of Automatic Obfuscation Identification and Deobfuscation Tool

초기 단계에서 추출된 정보를 활용하여 각 난독화 옵션의 적용 여부를 정적으로 식별하고 역난독화한다.

DexProtector 난독화 옵션을 다중으로 적용할 경우에는 원본 APK에 문자열, 리소스, 클래스 암호화 옵션이 순차적으로 적용된다. 따라서 Fig.17.과 같이 이를 역순으로 식별 및 역난독화를 수행한다. 즉, 클래스 암호화 옵션의 식별 및 역난독화가 선행되고 이후에 리소스 암호화, 문자열 암호화 옵션 순으로 식별 및 역난독화가 수행된다. 만약 각 모듈에서 해당 난독화 옵션이 식별되지 않은 경우에는 다음의 모듈로 진행된다.

4.2.1 클래스 모듈

클래스 암호화 옵션의 식별을 위해 다음과 같은 시그니처를 활용한다.

- assets 폴더 내에 DEX 파일 존재 여부와 매직 넘버
- 클래스 암호화 옵션의 코드 패턴
- XML 〈application〉 요소 "android:name" 속성 값에 특정 문자열(".Application" 또는 ".Protected") 포함 여부

클래스 암호화 옵션이 식별된 경우 클래스 역난독화를 진행한다. 먼저 초기 단계에서 추출한 smali 코드의 문자열을 복호화한다. 이 중 "PK"로 시작하는 문자열을 새로운 DEX 파일로 변환하고, 이로부터 복호화 클래스를 추출한다. 다음으로 복호화 클래스 내부의 키 참조 배열과 APK 인증서의 해시 코드 값을 통해 키를 생성한 후 assets 폴더 내의 원본 DEX 파일을 복호화한다. 마지막으로 복호화 DEX 파일을 복호화된 원본 DEX 파일로 교체하고 assets 폴더 내의 DEX 파일을 삭제한다. Fig.19.는 암호화된 원본 DEX 파일과 이를 역난독화한 후의 DEX 파일의 헤더를 비교한 결과이며 복호화된 DEX 파일이 올바른 매직 넘버를 가지고 있음을 확인할 수 있다.

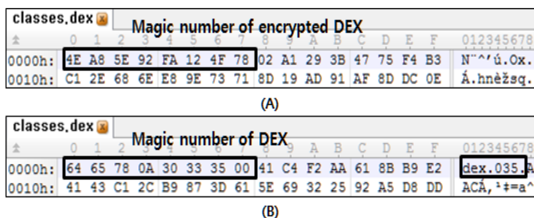

Fig. 19. (A) before Class Deobfuscation, (B) after Class Deobfuscation

4.2.2 리소스 모듈

리소스 암호화 옵션의 식별을 위해 다음과 같은 시그니처를 활용한다.

- assets 폴더 내 각 리소스의 매직 넘버
- 리소스 복호화 루틴
- 클래스 암호화 옵션 미적용 시 XML 파일의 〈application〉 요소 "android:name" 속성 값에 특정 문자열(".Application" 또는 ".Protected") 포함 여부

리소스 암호화 옵션이 식별된 경우 리소스 역난독화를 진행한다. 역난독화는 클래스 암호화 옵션이 식별된 경우와 식별되지 않은 경우 두 가지로 나뉘어 진행된다. 클래스 암호화 옵션이 식별된 경우에는 이의 역난독화 과정에서 암호화된 리소스를 복호화하기 위한 복호화 클래스도 함께 추출되기 때문에, 별도의 추출 과정이 요구되지 않는다. 그러나 클래스 암호화 옵션이 식별되지 않은 경우, XML 파일 〈application〉 요소의 "android:name" 속성이 가리키는 클래스의 문자열을 복호화하여 이 중 "PK"로 시작하는 문자열을 새로운 DEX 파일로 변환하고, 이로부터 복호화 클래스를 추출한다.

복호화 클래스 내부에서 키 참조 값을 파싱하여 키를 생성하고 IV인 암호화된 리소스의 첫 16바이트를 추출한다. 키와 IV를 이용하여 assets 폴더 내의 암호화된 리소스를 모두 복호화한다. 마지막으로 암호화된 리소스 파일들을 복호화된 원본 리소스 파일들로 교체하고 복호화 루틴과 복호화 클래스를 삭제한다. 이러한 일련의 과정의 통해 Fig.7.의 (B)와 같이 암호화되었던 리소스가 (A)와 같이 원본 리소스로 복원됨을 확인하였다.

4.2.3 문자열 모듈

문자열 암호화 옵션의 식별을 위해 다음과 같은 시그니처를 활용한다.

- 문자열 복호화 루틴

```
const-string v7, "₩u287d₩u6c78₩u124e₩u5c5c₩uf55c₩uff5c₩uaec2₩uf149"
invoke-static {v7},
   Decryption Class;->Decryption Method(Ljava/lang/String;)Ljava/lang/String;
move-result-object  v7
                 abbreviate
const-string v8, "₩ub003₩u21a6₩u2697₩u5aae₩u673b₩u95f2"
invoke-static {v8},
   Decryption Class;->Decryption Method(Ljava/lang/String;)Ljava/lang/String;
move-result-object  v8
                 abbreviate
```

(A)

```
const-string v7, "password"
                 abbreviate
const-string v8, "server"
                 abbreviate
```

(B)

Fig. 20. (A) before String Deobfuscation, (B) after String Deobfuscation

- 복호화 클래스 내 문자열 암호화의 코드 패턴

문자열 암호화 옵션이 식별된 경우 문자열 역난독화를 진행한다. 복호화 클래스 내에서 키·IV 참조 배열과 두 개의 치환 테이블을 파싱하고, 암호화된 문자열을 포함하는 클래스·메서드 명을 이용해 해시 코드 값을 생성한다. 이로부터 키와 IV를 생성하여 암호화된 문자열을 복호화한다.

마지막으로 복호화 루틴과 복호화 클래스를 삭제하고 암호화된 문자열을 복호화된 원본 문자열로 교체한다. Fig.20.은 암호화된 문자열과 역난독화 후의 문자열을 비교한 결과이다.

4.3 최종 단계

최종 단계에서는 식별된 난독화 옵션과 역난독화된 APK를 출력한다. 먼저 Fig.6.의 after Encryption과 같이 클래스/리소스 난독화 옵션 적용 시에 변형된 XML 파일의 <application> 요소 "android:name" 속성 값을 before Encryption과 같이 원본으로 수정한다. 이는 난독화 옵션 적용으로 인해 변형된 실행흐름을 원본의 실행흐름으로 복원시키는 과정이다. 마지막으로 식별된 난독화 옵션과 역난독화된 APK를 리패키징하여 출력한다.

V. 식별 및 역난독화 도구 검증

이번 장에서는 4장에서 구현한 자동화 난독화 옵션 식별 및 역난독화 도구의 검증 결과를 제시한다. 자동화 도구의 검증을 위해 Google Play Store의 5개 카테고리(교육, 금융, 사진, 소셜, 게임)에서 각각 10개씩 50개의 정상 APK를 다운받았고 웹페이지 및 스팸 문자 등으로부터 악성 APK 50개를 수집하였다. APK에 따라 DexProtector로 난독화가 적용되지 않는 경우가 있기 때문에 정상적으로 난독화를 적용할 수 있는 APK만 선택하여 수집하였다. 또한 리소스 암호화 옵션이 적용될 수 있도록 APK의 assets 폴더 내에 리소스 파일이 존재하는 APK를 선택하여 수집하였다.

본 논문에서 구현한 자동화 식별 및 역난독화 도구의 검증 과정은 Fig.21.과 같다. 먼저 수집한 APK에 대해 DexProtector의 난독화 옵션(문자열/클래스/리소스 암호화)을 단일 및 다중으로 적용시켜 총 700개의 난독화된 APK를 생성한다. 다음으로 난독화된 APK를 구현한 식별 및 역난독화 도구의 입력으로 하여 식별된 난독화 옵션과 역난독화된 APK를 출력한다. 마지막으로 자동화 도구에 의해 식별된 옵션이 실제 적용된 난독화 옵션과 일치하는지 비교하고, 역난독화된 APK와 원본 APK에 대해 다음과 같은 항목을 비교하여 역난독화 여부를 검증한다.

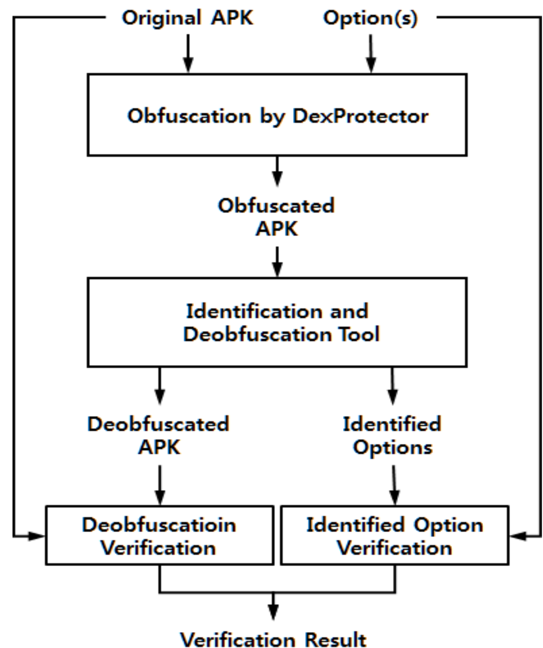

Fig. 21. The flow of verification processing

Table 3. The result of verification (S: String Encryption, C: Class Encryption, R: Resource Encryption, #: The number of obfuscated APK, %: Result)

		S	C	R	S/C	S/R	C/R	S/C/R	Total
Identification	#	100	100	100	100	100	100	100	700
	%	100	100	100	100	100	100	100	**100%**
De-Obfuscation	#	100	100	100	100	100	100	100	700
	%	100	100	100	100	100	100	100	**100%**

- 문자열 암호화 옵션: 두 개의 APK에서 모든 문자열을 추출하여 동일한지 비교
- 클래스 암호화 옵션: 두 개의 APK에서 XML 파일과 모든 클래스를 추출하여 동일한지 비교
- 리소스 암호화 옵션: 두 개의 APK에서 XML 파일과 assets 폴더 내의 모든 리소스를 추출하여 동일한지 비교

Table 3.은 검증 결과로써 본 논문에서 구현한 도구는 다운로드한 APK에 적용된 난독화 옵션을 성공적으로 식별 및 역난독화하였다. DexProtector에서 제공하는 난독화 옵션은 암호화 기반이기 때문에 복호화 절차를 파악하고 키·IV를 올바르게 생성한다면 원본과 복호화된 데이터(문자열/클래스/리소스)가 일치하게 된다.

즉, 식별 및 역난독화 도구의 검증 결과는 본 연구에서 수행했던 분석이 각 난독화 옵션의 복호화 절차를 정확히 파악하였고 복호화 과정에서 사용되는 정보(키·IV 참조 값, 치환 테이블, APK 인증서, 클래스·메서드 명 등)로부터 키·IV를 적절히 생성하였음을 보여준다.

Fig.22.와 같이 모든 원본과 역난독화된 APK에서 추출한 클래스는 완벽하게 일치하지는 않는다. 이는 3.4에서 언급하였듯이 전처리 단계에서 변환되는 정보는 되돌릴 수 없는 정보이거나 역난독화할 필요가 없는 정보이기 때문이다. 그러나 원본과 역난독화된 APK의 기능성과 복호화된 결과물(클래스 암호화의 경우 클래스 내부)은 동일하므로 모든 난독화 옵션에 대해 올바르게 역난독화를 수행하였다고 볼 수 있다.

```
public class InstructionsActivity extends Activity {
    public static final String EXTRA_DONT_SHOW = "idonteverwannaseeyouagain";
    private Switch dontShow;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        overridePendingTransition(2130968577, 2130968578);
        getWindow().requestFeature(1);
```

(A)

```
public class InstructionsActivity extends Activity {
    public static final String EXTRA_DONT_SHOW;
    private Switch dontShow;

    static {
        EXTRA_DONT_SHOW = "idonteverwannaseeyouagain";
    }

    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        overridePendingTransition(2130968577, 2130968578);
        getWindow().requestFeature(1);
```

(B)

Fig. 22. (A) original source code, (B) deobfuscated source code

VI. 결 론

본 논문에서는 상용 안드로이드 난독화 도구인 DexProtector로 난독화된 APK를 분석하여, 적용된 난독화 옵션을 정적으로 식별하고 역난독화하는 도구를 구현 및 검증하였다. 구현한 도구를 검증하기 위해 총 700개의 단일 및 다중 옵션으로 난독화된 APK를 생성하였으며 자동화 도구는 적용된 난독화 옵션의 식별과 이의 역난독화를 성공적으로 수행하였다.

따라서 본 논문의 연구 결과를 활용하여 DexProtector로 난독화된 악성 및 도용·변조된 APK의 탐지를 위해 역난독화를 먼저 수행함으로써 APK에 대한 분석 시간을 줄일 수 있을 것으로 기대된다. 또한 자동화 분석 도구는 정적으로 APK에 적용된 난독화 옵션의 식별과 역난독화를 수행하므로 동적 분석 환경을 우회·탐지하는 APK에 대해서도 역난독화가 가능하다.

References

〔1〕 KASPERSKEY LAB, “MOBILE CYBER THREATS,”http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf, Oct, 2014,

〔2〕 IDC, “Worldwide Quarterly Mobile Phone

Tracker,"http://www.idc.com/getdoc.jsp?containerId=prUS25860315, Aug, 2015.

〔3〕 Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," Security and Privacy(SP) 2012 IEEE Symposium on, pp. 95- 109, May, 2012

〔4〕 Pulse Secure, "MOBILE THREAT REPORT 2015," 2015, https://www.pulsesecure.net/lp/mobile-threat-report-2014/

〔5〕 ProGuard, http://proguard.sourceforge.net/

〔6〕 DexGuard, https://www.guardsquare.com/dexguard

〔7〕 DexProtector, https://dexprotector.com/

〔8〕 C. Collberg, C. Thomborson and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, 1997.

〔9〕 SBA Research Technical report, "Android application obfuscation," Jun, 2013.

〔10〕 APKtool, http://ibotpeaches.github.io/Apktool/

〔11〕 Smali/Baksmali, https://github.com/JesusFreke/smali

〔12〕 dex2jar, https://github.com/pxb1988/dex2jar

〔13〕 JD-GUI, http://jd.benow.ca/

〔14〕 Jadx, https://github.com/skylot/jadx

〔15〕 M. Chandramohan, H. B. K. Tan, "Detection of mobile malware in the wild," Computer, vol. 45, no. 9, pp. 65 - 71, Sep. 2012.

〔16〕 T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," Proceedings of the 9th ACM symposium on Information, computer and communications security, pp. 447-458, Jun. 2014.

〔17〕 Dong woo Kim, Jin Kwak and Jae cheol Ryou, "DWroidDump: Executable Code Extraction from Android Applications for Malware Analysis," International Journal of Distributed Sensor Networks, vol. 20, no. 8, pp. 1-9, Feb. 2015.

〔18〕 H. Schulz, D. Titze, J. Schutte, T. Kittel and C. Eckert, "Automated De-Obfuscation of Android Bytecode," Department of Computer Science, The University of Munchen, Germany, Jul. 2014.

〔19〕 Y. Piao, Jin hyuk Jung and Jeong hyun Yi, "Server based code obfuscation scheme for APK tamper detection," Security and Communication Networks, Mar. 2014.

〈저자소개〉

이 세 영 (Se Young Lee) 학생회원
2014년 2월: 서울시립대학교 수학과 졸업
2014년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 모바일 앱 분석, 소프트웨어 난독화

박 진 형 (Jin Hyung Park) 학생회원
2010년 2월: 건국대학교 컴퓨터공학과 졸업
2012년 2월: 고려대학교 정보보호대학원 석사 졸업
2012년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 암호프로토콜, 스마트폰 보안, 암호 알고리즘, 데이터마이닝

박 문 찬 (Moon Chan Park) 학생회원
2013년 2월: 서울시립대학교 수학과 졸업
2015년 2월: 고려대학교 정보보호대학원 석사 졸업
2015년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 분석, 소프트웨어 난독화

석 재 혁 (Jae Hyuk Suk) 학생회원
2012년 2월: 서울시립대학교 전자전기컴퓨터공학부 졸업
2014년 2월: 고려대학교 정보보호대학원 석사 졸업
2014년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 분석, 소프트웨어 난독화

이 동 훈 (Dong Hoon Lee) 종신회원
1983년 8월: 고려대학교 경제학사 졸업
1987년 12월: Oklahoma University 전산학과 석사 졸업
1992년 5월: Oklahoma University 전산학과 박사 졸업
1993년 3월~1997년 2월: 고려대학교 전산학과 조교수
1997년 3월~2001년 2월: 고려대학교 전산학과 부교수
2001년 3월~현재: 고려대학교 정보보호대학원 교수
〈관심분야〉 암호프로토콜, 암호이론, USN이론, 키 교환, 익명성 연구, PET 기술