KIPS Tr. Comp. and Comm. Sys.
Vol.4, No.1 pp.23~30 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2015.4.1.23

An Android API Obfuscation Tool using Reflection and String Encryption

Joohyuk Lee† · Heewan Park††

ABSTRACT

Reflection is a feature of the Java programming language that can examine and manipulate components of program itself. If you use the reflection, you can get an obfuscation effect of Java source because it converts sources into complicated structures. However, when using it, strings of components name of program are exposed. Therefore, it cannot prevent static analysis. In this paper, we presents a method and a tool of interfere with static analysis using reflection. And in this case, exposed strings are encoded using Vigenère cipher. Experimental results show that this tool is effective in increasing the overall complexity of the source code. Also the tool provides two types decryption method based on server and local. It can be selected based on the importance of the API because it affects the execution speed of the application.

Keywords : Reflection, Android, Static Analysis, Obfuscation, Encryption

리플렉션과 문자열 암호화를 이용한 안드로이드 API 난독화 도구

이 주 혁† · 박 희 완††

요 약

자바 리플렉션은 프로그램 구성 요소들을 조사하여 호출 및 조작할 수 있는 자바 언어의 기능이다. 이를 이용하면 보다 많은 호출 단계를 거치는 구조로 변형되기에 난독화 효과를 얻을 수 있다. 그러나 이를 이용할 때, 프로그램 자체의 구성 요소 이름이 문자열 형태로 노출된다. 본 논문에서는 안드로이드 애플리케이션에서 리플렉션을 적용하여 난독화하고, 이때 노출되는 문자열들을 비즈네르 암호화 알고리즘으로 은닉하여 정적분석을 방해하는 기법 및 도구를 제시한다. 실험 결과 소스 코드의 전체적인 복잡도를 증가시키는 데 효과가 있었다. 또한 서버와 로컬 기반의 두 가지 복호화 방법을 제공하는데, 이는 애플리케이션의 실행속도에 영향을 미치기 때문에 API의 중요도에 따라 선택할 수 있다.

키워드 : 리플렉션, 안드로이드, 정적 분석, 난독화, 암호화

1. 서 론

최근 스마트폰 시장에서 안드로이드 플랫폼의 점유율이 급격히 증가하고 있다. 2014년 1분기 출하량을 기준으로 세계 스마트폰 시장 점유율에서 안드로이드는 81.1%[1]를 차지하고 있고, 2013년 7월 이용 가능한 애플리케이션 수는 약 100만개[2]이다. 그런데 안드로이드 애플리케이션은 보안에 취약하다는 문제를 갖고 있다. 모바일 단말기에서 확장자가 apk인 파일 형태로 쉽게 추출할 수 있으며, PC에서 압축해제 프로그램으로 쉽게 내용을 확인할 수 있다. 또한 자바 언어로 작성된 후 컴파일하여 달빅(Dalvik) 코드로 변환되고 가상 머신(Virtual Machine)에서 실행되기 때문에 디컴파일이 쉽게 되는 특성을 갖는다. 이는 단말기에서 추출한 apk 파일의 압축을 해제하고 classes.dex 파일을 얻은 후 디컴파일하면 쉽게 자바 소스 파일로 복원하여 살펴볼 수 있다. 이는 애플리케이션에 악의적인 코드를 추가하고, 새로 서명하여 리패키징 후 배포하는 것도 가능하다.

자바 리플렉션(Reflection)은 자바 프로그램을 실행해서 해당 프로그램을 조사하거나 내부 구성 요소들을 살펴볼 수 있는 자바 언어의 기능이다[3]. 이를 이용하면 각 변수들은 변환 단계를 필요로 하는 레퍼런스 타입으로 변환되기 때문에 난독화 효과를 얻을 수 있다. Fig. 1은 리플렉션을 이용하여 *System.out.println("Hello World")*를 호출하는 예제이다.

Fig. 1과 같이 println() 메소드를 호출하기 위해 여러 과정을 거치게 되어 난독화 효과를 얻을 수 있으나 2번째 줄에 해당 메소드가 포함된 클래스 이름 문자열이, 5번째 줄에서는 해당 메소드의 이름 문자열이 노출되어 있기 때문에 어떤 클래스 구성 요소를 사용했는지 쉽게 파악할 수 있다.

※ 이 연구는 2012년도 한라대학교 교내연구비 지원에 의한 것임.
† 비 회 원 : 한라대학교 정보통신방송공학부 학사과정
†† 정 회 원 : 한라대학교 정보통신방송공학부 조교수
Manuscript Received : July 09, 2014
First Revision : November 18, 2014
Accepted : December 12, 2014
* Corresponding Author : Heewan Park(heewanpark@halla.ac.kr)

```
try {
Class<?> cls = Class.forName("java.io.PrintStream");

Class<?> arg = String.class;
Method m = cls.getDeclaredMethod("println", arg);

PrintStream printStream = new PrintStream(System.out);
m.invoke(printStream, "Hello World");
} catch (Exception e) {}
```

Fig. 1. Reflection example of *System.out.println("Hello World")*

본 논문에서는 안드로이드에서 악용될 수 있는 위험 API군[4]에 리플렉션을 적용하고, 이때 노출되는 문자열들을 비즈네르(Vigenère) 암호화[5]를 하여 은닉하는 기법과 구현을 제안한다. 2절에서는 관련연구로 기존의 리플렉션이 사용된 연구와 정적 분석 방해 기법에 대해 살펴본다. 3절에서는 본 논문에서 제안하는 정적 분석을 방해하는 소스 변환 기법과 도구를 설명하고, 4절에서는 위험 API군을 기반으로 표본 애플리케이션을 제작하여 본 논문에서 제안하는 기법을 적용 전후로 디컴파일 후 정적분석을 시행한다. 시행방법은 두 가지로 자바 디컴파일러인 jd-gui[6]로 직접 변형된 구조를 확인하고, 자바 언어의 정적 분석 도구인 Codepro Analytix[7]를 활용하여 수치적인 소스 복잡도를 알아본다. 마지막으로 5절에서는 결론과 향후 연구에 대해 논의한다.

2. 관련 연구

2.1 리플렉션의 활용

자바 리플렉션에 대해 클래스와 그 구성요소들을 조사 및 감지한 연구[8]로는 상황에 적합한 오버로딩을 유도한 툴킷 jContext에 관하여 소개된 바 있으며, 분산시스템의 실행환경에서 오게 되는 여러 가지 동적 변화들에 대하여 영향을 받지 않고 품질이 유지될 수 있는 복제관리 시스템에 대하여 리플렉션의 동적인 기능을 이용하여 모듈을 구성하고 시스템을 조작하여 환경적응력을 갖추는 데 사용한 연구[9]가 있었다.

자바 리플렉션을 안드로이드 환경에 적용한 연구로는 1,179개의 앱을 조사하여 리플렉션의 사용 빈도를 조사하여 주로 어떠한 라이브러리에서 사용되었는지 밝힌 바 있다[10]. 최근 리플렉션을 통해서 안드로이드 API에 대한 난독화 도구가 제안되었지만[11], 실행 성능 평가가 이루어지지 않았으며 리플렉션에 의해서 문자열이 노출되는 문제점에 대한 대책이 제시되지 못하였다. 본 논문은 기존 리플렉션 난독화 연구를 확장하여 리플렉션에 의한 실행 성능 저하를 평가하고, 메소드 이름이 문자열 형태로 노출되는 문제점에 대한 해결 방법으로 암호화 기법을 추가한 리플렉션 난독화를 제안한다.

2.2 정적 분석 방해 기법

정적 분석이란 프로그램을 실행하지 않고 소스 자체를 분석하는 것이다. 디컴파일(decompile)이나 디스어셈블(disassemble)이 이에 해당한다. 정적분석을 방해하는 데는 크게 난독화와 암호화로 나누어진다.

난독화는 프로그램의 구조를 복잡하게 만드는 것이다. 이에 대한 첫 번째 방법으로는 자바에서 클래스와 메소드, 변수명들을 변경하는 방법이다. 이는 자바 및 안드로이드에서 Proguard[12]란 도구를 사용하여 적용할 수 있다. 두 번째 방법으로는 프로그램의 자료구조를 변경할 것이다. 데이터의 형태나 위치를 변경하거나 메소드에서 처리 분기를 늘려 가중치를 높이는 방법이다[13, 14].

암호화는 소스 전체가 아닌 특정 부분을 암호화 알고리즘을 사용하여 변환하는 기법이다. 이 경우 역공학으로 원본 소스를 얻었다 할지라도 분석이 어려운 장점이 있다. 구현 방법으로는 애플리케이션의 클래스 파일 중 일부를 분리하여 서버에 두고 애플리케이션이 요청할 때마다 실행하도록 하는 방법이 있다. 이 경우 네트워크의 지연이나 연결로 정상적인 실행이 되지 않을 수 있는 단점이 있다. 또 하나의 방법으로는 dex파일 자체를 암호화하는 것이다. 이 경우 완전한 형태의 앱이 존재하기 때문에 역공학적으로 안전하지 못하다[15].

3. 본 문

3.1 위험 API군을 적용한 표본 앱 제작

실험을 위해 위험 API군과 이를 모두 적용하여 작성된 애플리케이션은 Fig. 2와 같다.

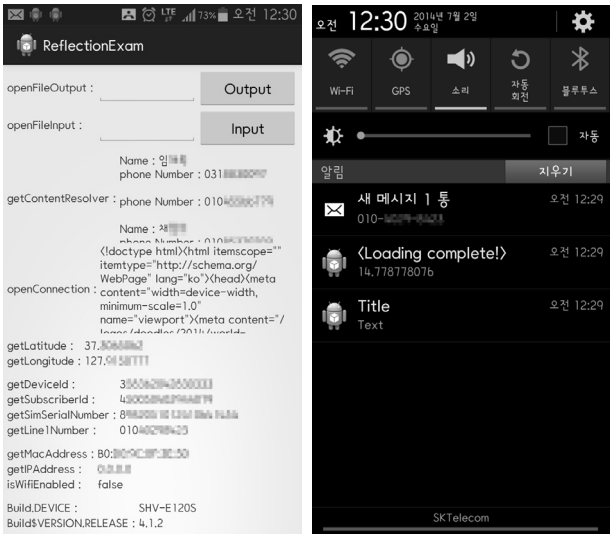

Fig. 2. Sample app based on dangerous API group

이 애플리케이션은 "onCreate" 메소드에서 위험 API군이 모두 실행되도록 작성되었다. 또한 모든 API가 호출된 시점까지의 성능 측정 결과를 보여주기 위해서 "Notification"을 이용하여 시간을 출력하였다.

Table 1. Dangerous API List of Android for applying Reflection A

Method name	Class name
openFileOutput	android.content. ContextWrapper
openFileInput	
isWifiEnabled	android.net.WifiManager
getIpAddress	
getMacAddress	
getDeviceId	android.telepony. TelephonyManager
getSubscriberId	
getSimSerialNumber	
getLine1Number	
getLatitude	android.location.Location
getLongitude	
openConnection	java.net.URL
getContentResolver	android.content. ContextWrapper
sendTextMessage	android.telephonyManager. SmsManager
setLatestEventInfo	android.app. NotificationManager
Field name	**Class name**
MODEL	android.os.Build
RELEASE	android.os.Build$VERSION

Table 2. Dangerous API List of Android for applying Reflection B

※ String in parentheses is the return type.

Method name	Parameter exists and return type is not void.	Parameter does not exists and return type is not void.	Parameter exists and return type is void.
openFileOutput	(OutputStream)	-	-
openFileInput	(InputStream)	-	-
isWifiEnabled	-	(boolean)	-
getIpAddress	-	(Integer)	-
getMacAddress	-	(String)	-
getDeviceId	-		-
getSubscriberId	-		-
getSimSerialNumber	-		-
getLine1Number	-		-
getLatitude	-	(Double)	-
getLongitude	-		-
openConnection	-	(HttpURL Connection)	-
getContentResolver	-	(Content Resolver)	-
sendTextMessage	-	-	(void)
setLatestEventInfo	-	-	(void)

Field name	Return value
MODEL	String
RELEASE	

Table 3. Method that applies Reflection based on parameter and return value

Parameter exists and return type is not void.

```
FileOutputStream writer = null;

Class <?> cls = Class.forName("andorid.content.ContextWrapper");

String methodName = "openFileOutput";

Class<?>[] paramTypes = null;
for(Method m : cls.getDeclaredMethods()) {
  if(m.getName().eqauls(methodName)) {
     paramTypes = m.getParameterTypes();
  }
}

Method m = cls.getDeclaredMethod(methodName, paramTypes);
writer = new OutputStreamWriter(
((FileOutputStream) m.invoke(new ContextWrapper (getApplication
Context()),
new Object[]{"exam.txt", MODE_PRIVATE})));
```

Parameter does not exists and return type is not void.

```
Class<?> cls = Class.forName("android.net.wifi.wifiManager");
Method m = cls.getDeclaredMethod("isWifiEnabled");

Boolean b = (Boolean) m.invoke(wifiMgr);
```

Parameter exists and return type is void.

```
Class<?> cls = Class.forName("android.telephony.SmsManager");

String methodName = "sendTextMessage";

Class<?> paramTypes = null;
for(Method m : cls.getDeclaredMethods()) {
  if(m.getName().eqauls(methodName)) {
    paramTypes = m.getParameterTypes();
  }
}

Method m = cls.getDeclaredMethod(methodName, paramTypes);
m.invoke(sms, new Object[]{"821012345678", null, "TEST", null,
null});
```

3.2 위험 API군에 리플렉션 적용을 위한 호출 구조 분석

메소드(Method)에 리플렉션을 적용하기 위해서는 API의 소속 클래스 이름, API 이름, 인수형(Argument type), 해당 메소드를 호출하는 객체, 인수와 같은 형의 실질적으로 받는 값 5가지가 필요하다. 해당 메소드를 호출하는 객체란, "((TextView) findViewById(R.id.tv)).setText()"를 예를 들어 setText()를 호출하려고 할 때 setText() 앞에 있는 호출 객체를 뜻한다. 이외 정적 메소드(Static method)의 경우 null이 들어간다. 필드에 적용하기 위해서는 API의 소속 클래스 이름, API 이름 두 가지가 필요하다. 이를 바탕으로 위험 API를 분류한 것이 Table 1, Table 2이다. Table 1은 메소드의 이름과 소속 클래스 이름

을 구분하였다. Table 2는 매개변수와 반환값에 의한 분류이다.

Table 2와 같이 메소드의 경우 매개변수 유무와 반환값의 종류에 따라 분류하였고, 이에 공통적인 특성으로 3가지 패턴이 나오는 것을 파악하였다. 그리고 또 하나의 공통점으로 모두 오버로드되지 않아 각 메소드의 매개변수 종류가 하나씩만 있는 점도 발견하였다. 그리고 메소드에 리플렉션을 적용하기 위해서는 “getDeclaredMethod” 메소드를 사용해야 하는데, 이 메소드의 두 번째 매개변수는 Class<?>형의 메소드의 매개변수형을 요구한다. 이때, 매개변수와 형의 개수와 종류는 하나 이상일 수 있고 또한 오버로딩된 경우 다양한 조합의 형태가 있을 수 있다. 위 메소드들에서는 오버로딩이 된 경우가 없었으므로 이를 배제한 채 매개변수와 형의 개수와 종류를 파악해서 3가지 유형으로 호출하는 구조의 예시는 Table 3과 같다.

이외 위험API군에서 필드의 경우 리플렉션을 적용할 때 메소드보다 요구 조건이 간단하고, 반환값이 모두 String이라는 공통점이 있었다. 필드의 경우 필드 이름 문자열로 “getDeclaredField” 메소드를 호출한 다음 변수에 저장하고 해당 변수를 “get” 메소드로 쉽게 호출 할 수 있다.

그러나 리플렉션을 사용하여 복잡도를 높인다고 해도 API 이름과 API의 소속 클래스 이름의 문자열이 그대로 노출되는 단점이 있다. 이를 보완하기 위해 비즈네르 암호를 이용해 문자열을 암호화한다.

3.3 비즈네르 암호화

비즈네르 암호(Vigenère cipher)란 1586년 프랑스 외교관 비즈네르(Blaise de Vigenère)에 의해 발표된 다중 대체 암호이다. 시저 암호(Caesar Cipher)를 기반으로 평문의 길이만큼 암호키를 계속 더해 여러 가지 문자에 대응하도록 통계적 정보를 없앤 것이 특징이다.

Table 4는 리플렉션을 이용할 때 노출되는 API 문자열들을 기반으로 정의하여 작성된 암호표이다.

암호표의 문자는 소문자 26개, 대문자 26개, 숫자 10개, 기타 문자 2개로 총 64개의 문자열로 이루어져 있다.

예를 들어 만약 평문이 “android.os.Build$VERSION”이고, 암호키가 “abc”인 경우 평문 첫 번째 글자 a와 암호키 첫 번째 글자 a가 대응하는 문자는 a, 평문 두 번째 글자 n과 암호키 두 번째 글자 b에 대응하는 문자는 o, 이런 방법으로 문자를 계속 대응하여 찾게 되면 암호문은 “aofrpkdqsDujndaXESUIPP”가 된다.

이를 평문 M, 암호키 K, 암호문 C라고 할 때, 수식적으로 나타내면 다음과 같다.

$$C_i = (M_i+K_i) \bmod 64$$
$$M_i = (C_i-K_i) \bmod 64 \ (\text{If } M_i<0 \text{ Then } M_i=M_i+64) \quad (1)$$

Table 4. Customized Vigenère table

	a	b	c	…	z	A	B	C	…	Z	0	1	2	…	9	.	$
a	a	b	c	…	z	A	B	C	…	Z	0	1	2	…	9	.	$
b	b	c	d	…	A	B	C	D	…	0	1	2	3	…	.	$	a
c	c	d	f	…	B	C	D	E	…	1	2	3	4	…	$	a	b
⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮
z	z	A	B	…	Y	Z	0	1	…	8	9	.	$	…	w	x	y
A	A	B	C	…	Z	0	1	2	…	9	.	$	a	…	x	y	z
B	B	C	D	…	0	1	2	3	…	.	$	a	b	…	y	z	A
C	C	D	F	…	1	2	3	4	…	$	a	b	c	…	z	A	B
⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮
Z	Z	0	1	…	8	9	.	$	…	w	x	y	z	…	W	X	Y
0	0	1	2	…	9	.	$	a	…	x	y	z	A	…	X	Y	Z
1	1	2	3	…	.	$	a	b	…	y	z	A	B	…	Y	Z	0
2	2	3	4	…	$	a	b	c	…	z	A	B	C	…	Z	0	1
⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮		⋮	⋮	⋮
9	9	.	$	…	w	x	y	z	…	V	W	Y	Z	…	6	7	8
.	.	$	a	…	x	y	z	A	…	W	Y	Z	0	…	7	8	9
$	$	a	b	…	y	z	A	B	…	Y	Z	0	1	…	8	9	.

그러나 비밀키를 짧은 키를 사용하거나 길더라도 통계적 확률이 있는 완성된 문장을 사용하면 카지스키(Kasiski)와 프리드만(Friedman) 공격법에 의해 해독할 수 있다[17]. 그렇기 때문에 본 연구에서는 쉽게 해독이 되는 것을 방지하기 위해 암호키의 길이를 “가장 긴 API가 속한 클래스의 이름” 평문 길이만큼, 암호키의 모든 문자는 중복되지 않고 모두 독립적인 구성하였다.

3.4 제안한 암호화 기법의 성능 검증

Table 5는 대칭 키 암호화 알고리즘을 직접 구현하고, 실행 속도를 비교한 결과이다. 비즈네르 암호 알고리즘의 경우 3.3에서 제안한 방법으로 직접 구현하였고, 나머지 알고리즘들은 “Apache Commons Codec”[16] 라이브러리를 사용하여 구현하였다. 실험 환경은 Table 8과 같다. 그리고 평문은 “android.os.Build$VERSION”, 암호키는 1회만 구하여 고정되어 있다는 전제하에 각 알고리즘을 10^7회 실행하여 평균을 측정하였다.

Table 5. Comparison of encryption average times for various common symmetric encryption algorithms(second)

Vigenère	AES (128bit)	AES (256bit)	DES	Tripple DES
0.68	1.86	2.97	3.27	3.85

이는 본 연구의 실험 및 평가에서 PC에 비해 성능의 제한이 있는 모바일 환경이라는 특수성과 서버를 이용한 복호화 방법을 이용할 시 상당히 중요한 요소가 된다. 측정 결과 제안한 암호화 기법이 AES128에 비해 약 2.75배, DES에 비해 약 4.83배 빠른 성능을 보였다. 그래서 본 연구에서는 제안한 암호화 알고리즘을 선택하였다.

3.5 리플렉션 소스 변환 도구의 구현

위 기법들을 기반으로 구현된 변환도구는 Fig. 3과 같은 절차를 갖는다.

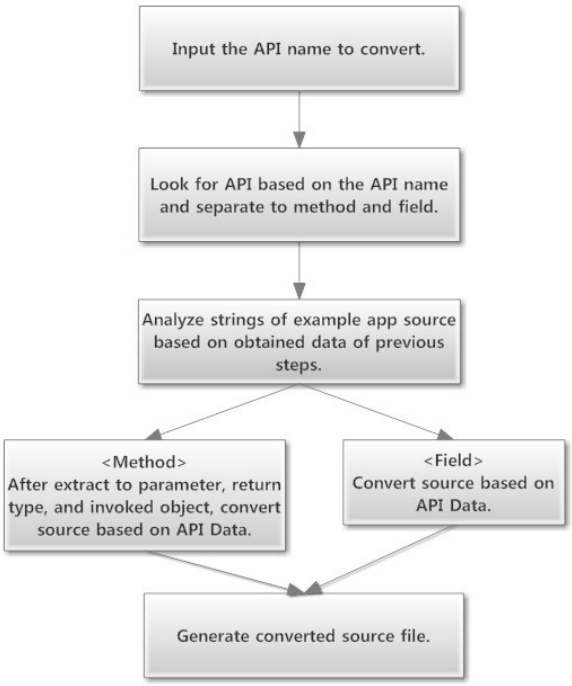

Fig. 3. Operating procedures of conversion tool

첫 번째로 변환하려는 API 이름을 입력받고, 이를 기반으로 찾아서 크기 2의 Object형 배열에 API정보를 추가한다. 메소드의 경우 매개변수 개수와 반환값 종류에 따라 3개의 유형을 판별하여 “1, 2, 3”, 필드의 경우 “-1”이 Object형 배열의 첫 번째 인덱스(Index)에 추가된다. 그리고 두 번째 인덱스에는 Method형 변수와 Field형 변수가 들어가게 된다. 그 다음과정으로는 3.1에서 작성한 위험API군을 적용한 애플리케이션의 소스를 “BufferedReader”와 “BufferedWriter” 메소드를 이용하여 줄 단위로 문자열을 분석한다. 메소드의 경우 매개변수(Parameter)과 반환형(Return Type), 호출 객체(Receiver)를 추출 후 Table 2에서 정의한 각 세 유형 중에 적합한 형태로 리플렉션을 적용하여 변환한다. 이때, 원본 소스 상에서 메소드의 반환값이 전역변수로 사용되는 경우 추가적인 예외처리가 되도록 하였다. 필드의 경우 반환형과 호출 객체를 추출 후 리플렉션을 적용하여 변환한다. 호출 객체란 각 메소드와 필드를 가지고 있는 객체를 의미한다. 또한 리플렉션 적용과정에서 메소드와 필드의 소속 클래스와 API의 이름 문자열이 그대로 노출되어 어떤 API를 사용했는지 유추할 수 있는데, 이 문자열을 3.3에서 정의한 방법으로 암호화한다.

최종적으로 구현된 변환도구는 Fig. 4와 같다.

```
sendTextMessage
setLatestEventInfo
---------------------------------
---------- Field List ----------
Build.MODEL
Build.VERSION.RELEASE
---------------------------------
* Method 이름을 입력하고 Enter를 누르면 그 Method를 변환.
* 그냥 Enter를 누르면 전체 변환.

>>

Key : 4A1bP3wHxM60qOEaFdV7GmsL8Jc$ugzijn

 60 :          String model = Build.MODEL;

[변환 결과]
String model = null;
try {
Class<?> cls = Class.forName(decoder.decode("4N4s3$zFL44pKWPd"));
Field f = cls.getField(decoder.decode("EcsFo"));
model = (String) f.get(f);
} catch (Exception e) {}
```

Fig. 4. Conversion tool

3.6 복호화 방법의 선택

구현된 변환 도구는 간단한 변수 조작으로 두 가지 문자열 복호화 방법을 선택할 수 있다. 기본 선택은 서버 암호화 방법으로, 이는 암호키와 복호화 방법을 숨기기 위해 애플리케이션 소스 자체에 복호화 메소드를 삭제하고 서버 측의 문자열 해독 프로그램으로 암호화된 문자열을 전송하면 해독 문자열을 클라이언트(스마트폰) 측에 전송하는 방법이다. Fig. 5은 구현된 문자열 해독 서버 프로그램이다.

```
Problems  @ Javadoc  Declaration  Console  Progress

Key : .Lg3$U7klxur5nfYpEy86iJKZqoeRs0bGX
Hyuk-Desktop/221.[redacted]:8888
Operating...
SHV-E120S is connected.


Receive from SHV-E120S> .Yjin2.iyBNppvk6noGbcUJXZwsv
Send to SHV-E120S> android.net.wifi.WifiManager

Receive from SHV-E120S> g32$e2zxlyFv8
Send to SHV-E120S> isWifiEnabled
```

Fig. 5. Server program for string decryption

그러나 네트워크의 지연, 로컬 암호화에 비해 보다 많은 호출 구조로 성능이 지연될 수 있다. 그렇기 때문에 로컬 암호화를 지원하게 되는데, 이는 복호화 메소드를 애플리케이션 자체에 삽입하여 문자열을 해독하는 방식이다. 이는 암호화 알고리즘이 그대로 노출되는 단점이 있다.

4. 실험 및 평가

전체적인 실험 및 평가 방법은 Fig. 6과 같다.

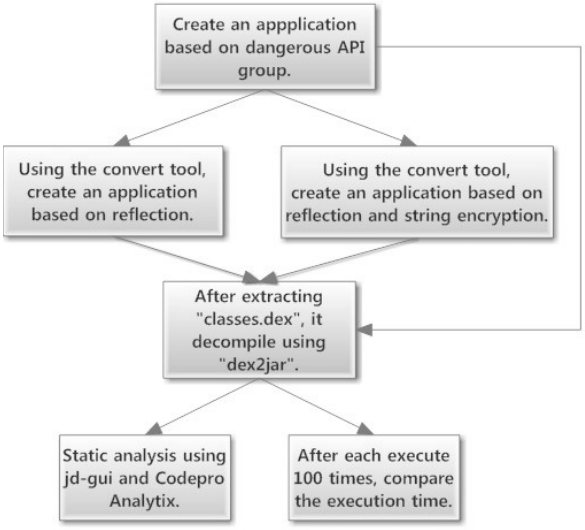

Fig. 6. Experiments and Evaluation

두 가지 실험 평가를 하게 된다. 첫 번째는 애플리케이션에서 'classes.dex'를 추출 후 디컴파일하여 정적 분석하는 방법이다. 위험 API군을 기반으로 한 애플리케이션과 이를 기반으로 리플렉션, 리플렉션 및 문자열을 적용한 3개의 애플리케이션을 단말기에서 추출 후 classes.dex 파일을 추출하고, 이를 dex2jar[17]로 컴파일하고 jd-gui를 이용하여 변환된 문자열들을 직접 분석한다. 그리고 정량적인 측정을 위해 정적 분석 도구인 Codepro Analytix를 이용하여 수치적인 코드 복잡도를 측정한다. 두 번째는 변환 도구의 성능이 검증이 잘 되었다고 할지라도 스마트폰이라는 환경 특성상 애플리케이션의 실행 속도에 크게 영향을 미치면 실 활용에 무리가 있다. 그렇기 때문에 각각의 3개의 소스를 삽입한 애플리케이션을 100회씩 실행하여 실행속도를 대조한다.

4.1 정적 분석 및 정량적인 복잡도 측정

Table. 6은 dex2jar로 디컴파일 후 jd-gui로 정적 분석한 결과이다.

1) 매개변수가 있고, 반환값이 void가 아닌 경우
2) 매개변수가 없고, 반환값이 void가 아닌 경우
3) 매개변수가 있고, 반환값이 void인 경우

암호화를 추가적으로 적용한 결과에서는 짧은 단락의 스레드 종료 메소드가 삽입되어 메소드 평균 라인수와 메소드 내 블록 계층 깊이 수치가 떨어진 결과가 보였지만, 원본에 비해 전체적인 복잡도가 올라간 것을 확인할 수 있었다. 리플렉션을 적용하였을 시 구성요소들이 Object형 배열이나 Class형 변수들로 이루어졌을 뿐만 아니라 세 가지 유형 중 일부는 while문이 추가되어 전체적인 복잡도와 전체적인 소스량도 약 1.86배 증가했음을 확인할 수 있었다. 또한 리플렉션을 적용했을 때 노출되는 문자열들은 암호화가 잘 적용되어 은닉된 것이 확인 되었다. 또한 각 유형별로 군을 묶어 변환 후 실행해본 결과 모두 오차 없는 동작이 되어 변환이 잘 되어진 것을 확인하였다.

Table 7은 정적분석 도구인 Codepro Analytix를 이용하여 수치적인 분석을 한 결과이다.

Table 6. Analysis results of the API call using jd-gui

Original source	1	jd-gui judged "Error".
	2	(Omission) localWifiManager.isWifiEnabled());
	3	SmsManager.getDefault().sendTextMessage("821012345678", null, "TEST", null, null);
Converted source using reflection	1	jd-gui judged "Error".
	2	Method localMethod3 = Class.forName("android.net.wifi.WifiInfo") .getDeclaredMethod("isWifiEnabled", new Class[0]); (Omission) ((Boolean)localMethod3 .invoke(localWifiManager, new Object[0]));
	3	while (true) { (Omission) Class localClass = Class.forName("android.telephony.SmsManager"); (Omission) Method localMethod1 = localClass.getDeclaredMethod("sendTextMessage", (Class[])localObject); Object[] arrayOfObject = new Object[5]; arrayOfObject[0] = "821012345678"; arrayOfObject[2] = "TEST"; (Omission)
Converted source using reflection and encryption	1	jd-gui judged "Error".
	2	Method localMethod3 = Class.forName(this.decoder.decode(".Yjin2.iyBNppvk6noGbcUJXZwsv")); .getDeclaredMethod(this.decoder.decode("g32$e2zxlyFv8"), new Class[0]); (Omission) ((Boolean)localMethod3.invoke(localWifiManager, new Object[0]));
	3	while (true) { (Omission) Class localClass = Class.forName(".Yjin2.iEBFviut$NCeimUJXZwsv"); (Omission) Method localMethod1 = localClass.getDeclaredMethod("qPt6SYsDXBMJ5tj", (Class[])localObject); Object[] arrayOfObject = new Object[5]; arrayOfObject[0] = "821012345678"; arrayOfObject[2] = "TEST"; (Omission)

Table 7. Analysis results of the API call using Codepro Analytix (lines)

	Original source	Converted to using reflection	Converted to using reflection and encryption
Number of Lines	452	770	839
Average Lines of Code Per Method	13.16	26.36	23.57
Average Block Depth	1.46	1.54	1.43
Average Cyclomatic Complexity	2.16	1.72	1.85

4.2 실행시간 측정

실험 방법을 도식화하면 Fig. 7과 같다.

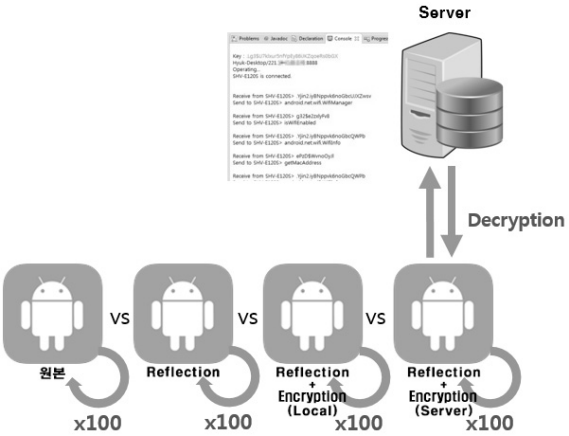

Fig. 7. Diagram of the experimental method

각각의 표본 애플리케이션을 100회씩 실행 후 onCreate(), onResume()에서 System.nanoTime()을 호출하고 그 시간의 차를 측정하여 “Notification”으로 확인하였다. 서버를 이용하여 문자열을 해독하는 경우 별도의 해독 프로그램을 작성하여 실험하였다. 실험 환경은 Table 8과 같다.

Table 8. Experimental environment

Android Phone	Specification	CPU :Qualcomm Snapdragon S3 APQ8060 SoC 1.2GHz RAM : 1GB Platform : Android 4.1.2
	Network [18]	Download : 24.32Mbps Upload : 21.04Mbps Delay : 35.62ms
PC	Specification	CPU : Intel i3-3220 @ 3.30GHz, RAM : 4GB OS : Windows 7 SP1 64bit
	Network [18]	Download : 91.8Mbps Upload : 93.1Mbps Delay : 5.0ms

Fig. 8와 Table 9는 원본 소스코드와 리플렉션을 적용한 소스코드, 리플렉션과 로컬 암호화를 적용한 소스코드, 리플렉션과 서버를 통한 암호화를 적용한 소스코드를 사용한 애플리케이션을 100회씩 실행하고 얻은 결과이다.

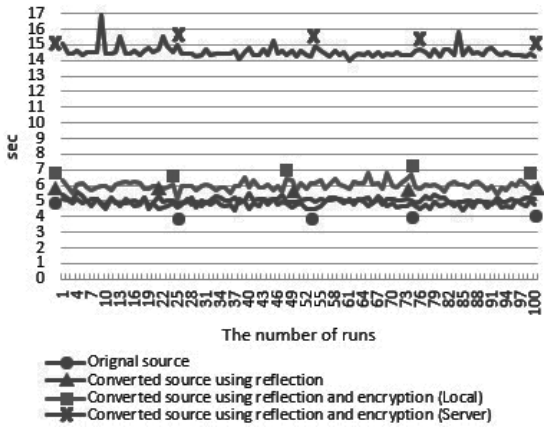

Fig. 8. Experimental results

Table 9. The results of time measurements(second)

	Minimum	Maximum	Average
Original source	4.39	5.27	4.81
Converted source using reflection	4.54	5.59	5.04
Converted source using reflection and encryption (Local)	5.10	6.75	5.98
Converted source using reflection and encryption (Server)	13.99	16.89	14.55

실행시간 측정 결과로는 리플렉션만 적용한 애플리케이션은 원본보다 0.23초 실행시간이 증가하였는데 비율로 계산하면 4.8%의 실행시간 속도가 감소하는 결과를 보였다. 리플렉션에 문자열 암호화를 추가로 적용한 경우 로컬에서 복호화할 경우에 평균 1.17초 증가하였고, 서버를 통한 복호화의 경우에 평균 9.74초 더 소요되는 결과를 보였다. 비율로 계산하면 원본보다 각각 24.3%와 202.5% 실행시간이 증가한 것인데, 이는 onCreate()메소드에서 실험 대상으로 사용된 모든 API들이 동시에 실행되도록 작성된 것이 원인이다.

4.3 난독화 성능 평가

서버를 이용한 문자열 복호화는 네트워크 지연으로 인해 실행 속도에 영향을 줄 수 있다. 안드로이드는 이벤트 기반의 플랫폼이기 때문에, 실제 환경에서는 어떤 특정 이벤트 실행에 따라 지엽적인 한 부분이 실행되므로 성능 저하가 크지 않을 것으로 기대한다. 그리고 일반적인 안드로이드 애플리케이션의 경우 처음 구동될 때 초기화면을 띄운 후 네트워크 접속 가능 여부를 판단하거나 데이터베이스를 로딩 하는 등 초기화 과정을 거치기 때문에 이 과정에서 백그라운드 작업으로 모든 문자열에 대한 복호화를 일시에 수행하는 방식을 선택하면 애플리케이션이 실행될 때 발생하는

오버헤드를 최소화할 수 있을 것으로 기대한다. 특별히 빠른 실행 속도가 중요한 애플리케이션의 경우에는 난독화하고자 하는 최소한의 API를 선별하고, 서버를 통한 문자열 암호화보다는 로컬 암호화를 선택하는 것이 좋다.

5. 결론 및 향후 연구

최근 안드로이드 플랫폼의 시장 점유율이 계속 증가하고 있는 추세이다. 그러나 안드로이드는 단말기에서 애플리케이션 패키지 파일을 추출할 수 있고, 쉽게 디컴파일이 되기 때문에 보안에 취약하다.

본 논문에서는 안드로이드 애플리케이션의 보안 취약성을 보완하기 위해서 리플렉션을 이용하여 복잡도를 높이고, 이때 노출되는 문자열들을 암호화하여 정적분석을 방해하는 기법을 제안하였고 이를 수행하는 도구를 구현하였다.

실험을 통해 제안된 기법이 API 이름을 알 수 없는 형태로 변경하여 API 호출에 대한 정적분석을 방해하는 데 효과적인 것을 확인하였다. 리플렉션만을 이용하는 경우 원본보다 실행 속도가 평균 4.8% 증가하였고, 리플렉션과 로컬 문자열 암호화를 수행한 경우 실행속도가 평균 24.3% 증가하였다. 서버를 통한 문자열 암호화는 네트워크 접속 오버헤드로 인해서 실행속도가 평균 202.5% 증가하는 문제가 발생했다. 이 문제를 해결하기 위해서는 애플리케이션이 처음 구동될 때 초기화면을 띄운 후, 백그라운드 작업으로 모든 문자열에 대한 복호화를 동시에 수행하는 방식을 선택하면 오버헤드를 최소화할 수 있을 것으로 기대한다.

향후 연구 과제로 메소드 오버로딩을 판별하여 리플렉션 적용 대상을 넓히는 방법과, 리플렉션과 복호화에 대한 성능 저하를 최소화시킬 수 있는 연구가 병행해야 할 것이다.

References

[1] Smartphone OS Market Share, Q1 2014, http://www.idc.com/prodserv/smartphone-os-market-share.jsp/.

[2] Google Play Hits 1 Million Apps, http://mashable.com/2013/07/24/google-play-1-million/.

[3] Using Java Reflection, http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html/.

[4] W. T. Sim, J. M. Kim, J. C. Ryou, and B. N. Noh, "Android Application Analysis Method for Malicious Activity Detection," Journal of KIISC, Vol.21, No.1, pp.213-219, 2011.

[5] Vigenère cipher, http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher/.

[6] jd-gui, https://code.google.com/p/innlab/.

[7] Codepro Analytix, https://developers.google.com/java-dev-tools/codepro/doc/.

[8] S. Y. Cho and T. Y. Kim, "Dynamically Reconfigurable Object Replication Management System Using Reflection approach in Java," Journal of KIISE, Vol.26, No.1(A), pp.373-375, 1999.

[9] J. M. Choi, "A Toolkit for Context-aware Systems using Java Reflection and Method Overloading," Journal of KIISE, Vol.39, No.1, pp.12-23, 2012.

[10] W. C. Lee and K. K. Lee, "Java Reflections in Action: A Case Study of Android Apps," Proceedings of KIISE, Vol.39, No.2(A), pp.277-279, 2012.

[11] J. H. Lee and H. W. Park, "Design and Implementation of An Auto-Conversion Tool for Android API Obfuscation Based on Java Reflection", Proceedings of KIPS, Vol.21, No.1, pp.487-490, 2014.

[12] Proguard, http://developer.android.com/tools/help/proguard.html.

[13] H. S. Ahn, "A study on the Java Decompilation-Preventive Method by Obfuscating Algorithm," Journal of KIPS, Vol.14, No.1, pp.1458-1458, 2007.

[14] E. M. Kim and K. S. Han, "A Study on the Code Obfuscation Techniques for Java Source Code," Journal of KIISE, Vol.35, No.2(A), pp.307-308, 2008.

[15] C. H. Lee, Y. U. Park, J. H. Lim, H. G. Kim, C. H. Lee, S. J. Cho and J. S. Yang, "Access Control Mechanism Preventing Application Piracy of the Android Platform," Proceedings of KIISE, Vol.18, No.10, pp.692-700, 2012.

[16] Apache Commons Codec, http://commons.apache.org/proper/commons-codec/.

[17] dex2jar, http://code.google.com/p/dex2jar/.

[18] Benchbee, http://www.benchbee.co.kr/.

이 주 혁

e-mail : joohyukxlee@gmail.com

2009년~현 재 한라대학교 정보통신방송공학부 학사과정

관심분야 : 프로그래밍 언어, 모바일 플랫폼, 사물 인터넷, 프로그램 난독화, 역공학

박 희 완

e-mail : heewanpark@halla.ac.kr

1997년 동국대학교 컴퓨터공학과(학사)

1999년 KAIST 전산학과(공학석사)

2010년 KAIST 전산학과(공학박사)

2004년~2007년 삼성전자 무선사업부 책임연구원

2010년~2011년 ETRI 부설연구소 선임연구원

2011년~현 재 한라대학교 정보통신방송공학부 조교수

관심분야 : 프로그램 난독화, 역공학, 악성코드 분석, 소프트웨어 워터마킹, 정적 및 동적 분석