자바 리플렉션 기반의 안드로이드 API 난독화를 위한 자동 변환 도구의 설계 및 구현

이주혁, 박희완
한라대학교 정보통신방송공학부
e-mail:joohyukxlee@gmail.com, heewanpark@halla.ac.kr

Design and Implementation of An Auto-Conversion Tool for Android API Obfuscation Based on Java Reflection.

Joo-Hyuk Lee, Heewan Park
School of Information Communication & Broadcasting Engineering,
Halla University

요 약

리플렉션은 자바 프로그램을 실행하여 객체 내부의 모든 요소를 조사하거나 호출 혹은 조작할 수 있는 자바 언어의 한 기능이다. 한 클래스 내부의 메소드에 리플렉션을 적용하여 호출하게 되면 String형의 메소드 이름으로 간접 호출하기에 정적 분석 도구의 API 호출 탐지를 방해하게 되어 분석 결과의 정확도를 떨어뜨릴 수 있고, 또한 일반적인 호출보다 복잡한 절차를 거치게 되어 소스 자체의 난독화 효과를 갖게 된다. 또한 디컴파일러의 역공학 분석을 어렵게 만드는 장점도 있다. 이 특성을 이용한다면 안드로이드 환경에서 특정 API를 은닉하여 개인정보를 누출하도록 악용하거나 디컴파일러 이용을 방지하는 데 활용될 수 있다. 본 연구에서는 안드로이드 환경에서 직접 설계한 도구와 표본 앱을 이용하여 API 메소드에 리플렉션을 적용하고, 원본 소스와 리플렉션 후 디컴파일된 소스를 비교하여 API 호출이 리플렉션을 통해서 은닉 가능함을 보여준다.

1. 서론

자바 리플렉션(Reflection)[1]은 자바 언어가 제공하는 기능 중 하나이다. 이를 이용하면 클래스 이름으로부터 동적으로 객체 내부의 모든 요소를 조사하거나 조작할 수 있다.

(그림 1)은 리플렉션을 이용하여 "*System.out.println()*"을 호출하여 "Hello World!"를 출력하는 예제이다.

```
1  import java.io.PrintStream;
2  import java.lang.reflect.Method;
3
4  class Example {
5    public static void main(String[] args) {
6      try {
7        Class<?> cls = Class.forName("java.io.PrintStream");
8
9        Class<?>[] parTypes = null;
10       for(Method m : cls.getDeclaredMethods()) {
11         if(m.getName().equals("println"));
12           parTypes = m.getParameterTypes();
13       }
14
15       PrintStream printStream = new PrintStream(System.out);
16       Method m = cls.getDeclaredMethod("println", parTypes);
17       m.invoke(printStream, new Object[]{"Hello World!"});
18     } catch (Exception e) {
19     }
20   }
21 }
```

(그림 1) 리플렉션을 이용하여 "Hello World"를 출력하는 예제

위 소스에서 주목할 부분은 7, 11, 16번째 줄에서의 메소드들이 파라미터 값을 클래스 혹은 메소드의 이름으로 String 타입을 사용하고 있다는 것이다. 이는 클래스 혹은 메소드 이름으로 API 호출 여부를 검사하는 정적 분석 기법을 방해하는 요소로 작용할 수 있다. 만약 이 특성을 악용해 안드로이드 환경에서 사용자 개인 정보를 외부로 유출하는 API 메소드 호출을 은닉하는 목적으로 사용된다면 기존의 정적 분석 도구들은 이러한 악성 행위를 탐지하기 어렵게 된다.

또한, 리플렉션을 사용하는 경우에는 일반적인 메소드 호출문도 파라미터의 존재여부, 리턴 값의 종류, 해당 메소드를 가지고 있는 객체를 조사한 후 호출하게 되어 보다 많은 절차를 거치게 되어 복잡한 구조를 갖게 되어 난독화 효과를 갖게 된다. 또한 이는 디컴파일시에도 그대로 적용되어 직접적인 메소드 호출문을 찾을 수 없을뿐더러 더욱 더 복잡한 구조를 갖게 되어 분석하기 어렵게 되거나 간혹 디컴파일이 실패하는 경우가 있어 디컴파일이 쉽게 되는 안드로이드 앱에 있어서 분석을 방해하는 데 이용될 수 있다. 이와 같이 어떻게 이용하는가에 따라 두 가지 대립되는 측면으로 활용될 수 있다.

2. 관련 연구

자바 리플렉션에 대해 클래스와 그 구성요소들을 조사 및 감지한 연구는 다음과 같다.

먼저 상황에 적합한 오버로딩을 유도한 툴킷 jContext에 관하여 소개된 바 있다[2].

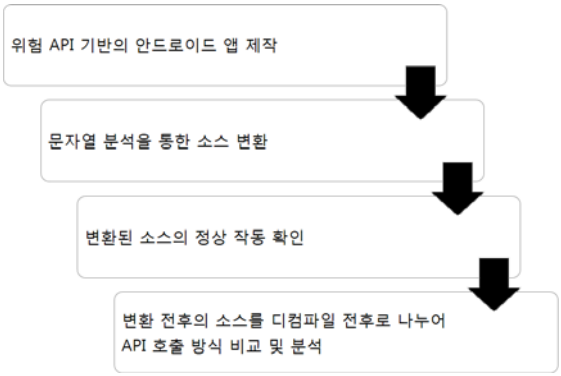
그리고 분산시스템의 실행환경에서 오게 되는 여러 가지 동적 변화들에 대하여 영향을 받지 않고 품질이 유지될 수 있는 복제관리 시스템에 대하여 리플렉션의 동적인 기능을 이용하여 모듈을 구성하고 시스템을 조작하여 환경적응력을 갖추는데 사용되기도 하였다[3].

자바 리플렉션을 안드로이드 환경에 적용한 연구로는 1,179개의 앱을 조사하여 리플렉션의 사용 빈도를 조사하여 주로 어떠한 라이브러리에 사용되었는지 연구된 바 있다[4].

그러나 위 연구들은 자바 언어에만 국한되거나 안드로이드에 실질적으로 적용하여 연구된 바가 없었다. 본 논문에서는 안드로이드 기반 환경에 자바 리플렉션을 규정된 위험 API[5]에 직접적으로 적용하여 정적 분석을 방해하는 연구를 진행한다.

3. 리플렉션을 이용한 API 정적 분석 무력화 기법

API 분석 방법의 전체 절차는 아래 (그림 2)와 같다.



(그림 2) 리플렉션을 적용한 안드로이드 API 분석 방법

먼저 위험 API군[5]을 참고하여 실험에 사용할 안드로이드 앱을 제작한다.

두 번째로 표본 앱의 소스를 행 단위로 읽어 문자열 분석(Parsing)을 하며 해당 메소드가 있는 부분을 리플렉션을 적용한 소스로 변환한다. 그런데 서론에서 언급했듯이 소스에 리플렉션을 적용하기 위해서는 파라미터 존재 여부와 리턴 타입의 종류, 그리고 해당 메소드를 갖고 있는 객체가 필요하다. 해당 메소드를 갖고 있는 객체는 대체로 변수로 선언되어 있기 때문에 소스 상에서 찾을 수 있고, 파라미터 값이나 리턴 값의 경우 해당 메소드를 조사하면 구할 수 있다. 그래서 실험에 앞서 안드로이드 위험 API군[5]을 참고하여 실험에 이용할 API들을 파라미터 존재 여부와 리턴 값이 "void" 여부에 대한 유형별로 <표1>와 같이 정리하였다.

<표1> 위험 API (괄호안의 문자열은 리턴 값)

API 이름 설명	파라미터가 있고, 리턴 값이 void가 아닌 경우	파라미터가 없고, 리턴 값이 void가 아닌 경우	파라미터가 있고, 리턴 값이 void인 경우
openFileOutput 파일 열고 쓰기	(OutputStream)		
openFileInput 파일을 열고 읽기	(InputStream)		
setWifiEnabled Wi-Fi 작동 설정	(boolean)		
getIpAddress IP주소 가져오기		(Integer)	
getMacAddress MAC주소 가져오기		(String)	
getDeviceId IMEI값 가져오기			
getSubscriberId IMSI값 가져오기			
getSimSerialNumber USIM번호 가져오기			
getLine1Number 전화번호 가져오기			
getLatitude 위도 가져오기		(Double)	
getLongitude 경도 가져오기			
openConnection 인터넷 연결 요청		(HttpURL Connection)	
getContentResolver 저장된 주소록 열람		(Content Resolver)	
sendTextMessage SMS 전송			(void)
setLatestEventInfo Notification 쓰기			(void)

위험API라고 지칭된 메소드들은 위와 같은 3가지 패턴으로 구분이 되었고, 이를 바탕으로 각 패턴에 맞게 변환한다.

세 번째로, 실험에 쓰일 표본 앱 중에서 선택된 API에 대해서 리플렉션을 적용시켜 변환된 소스로 교체한 후, 실행시켜서 변환 전의 소스 작동과 동일하게 정상 작동하는지 확인한다.

마지막으로 변환 전후의 원본 소스 및 dex2jar[6]와 Java Decompiler[7]를 사용하여 디컴파일된 두 소스를 비교하여 직접적인 API 호출이 있는 지 검증한다.

4. 리플렉션을 적용하는 자동 변환 도구의 설계 및 구현

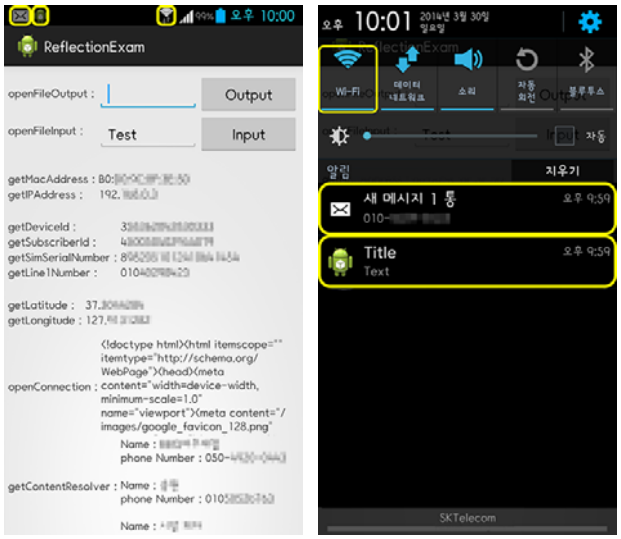
먼저 메소드를 탐색하려면 "*Class.forName()*"의 파라미터에 들어갈 전체 경로의 패키지 이름이 필요하기에 <표1>를 참고하여 각 패키지 이름들을 String 배열로 미리 만들어 놓는다. 이 후 그 String 배열로 Class<?>형 변수를 만들고, 각 변수가 갖는 메소드 이름과 입력받은 메소드 이름을 비교하여 파라미터와 리턴 값을 조사하여 <표1>에서 정리한 3가지 패턴 중 한 패턴값과 메소드를 반환받는다. 그 다음, 행 단위로 문자열을 읽어 반환받은 해당

메소드의 이름이 있는 지 분석하고 파라미터 존재 여부와 리턴 값의 void 여부, 그리고 해당 메소드가 있는 객체를 추출한다. 마지막으로 3가지 패턴 중 반환받은 패턴 값에 맞는 리플렉션 소스로 변환한다. 이 때 각 메소드를 사용하기 위해 각 패키지의 import를 필요로 하게 되는 데, 원본 소스를 처음 열고 읽을 때 첫 행에서 문자열 "package"가 있는 부분을 분석 후 "*concat()*" 메소드를 이용하여 import를 필요로 하는 모든 package 문자열을 삽입한다. 설계된 도구의 구현은 아래 (그림 3)과 같다.

(그림 3) 구현된 리플렉션 자동 변환 도구

4. 실험 및 평가

첫 번째 확인 단계로, 리플렉션 적용 전후로 표본 앱의 실행 결과 비교이다. 결과는 동일했으며 (그림 4)와 같다.



(그림 4) 리플렉션이 전체 적용된
위험 API를 기반으로 만들어진 안드로이드 앱

위 어플리케이션은 실행되자마자 <표1>에서 제시된 전체 API들이 동작하도록 구성되었다.

두 번째 확인 단계로서, 리플렉션 적용 전후의 원본 소스 및 디컴파일된 두 소스를 비교하여 API 호출방식을 비교한다.

먼저 디컴파일 이전의 리플렉션 적용 전후 소스 비교로 결과는 (그림 5)와 같다.

파라미터가 있고, 리턴값이 void가 아닌 경우

•[변환 전]
```
wifiMgr.setWifiEnabled(true);
```
•[변환 후]
```
<중략>
Class<?>[] parTypes = null;
  for(Method m : cls.getDeclaredMethods()) {
    if(m.getName().equals(methodName))
      parTypes = m.getParameterTypes();
  }
  Method m = cls.getdeclaredmethod(methodName, parTypes);
  m.invoke(wifiMgr, new Object[]{true});
<중략>
```

파라미터가 없고, 리턴값이 void가 아닌 경우

•[변환 전]
```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
```
•[변환 후]
```
HttpURLConnection conn = null;
<중략>
  Method m = cls.getDeclaredMethod("openConnection");
  conn = (HttpURLConnection) ((URLConnection)m.invoke(url));
<중략>
```

파라미터가 있고, 리턴값이 void인 경우

•[변환 전]
```
sms.sendTextMessage("8210", null, "TEST", null, null);
```
•[변환 후]
```
<중략>
  String methodName = "sendTextMessage";
  Class<?>[] parTypes = null;
  for(Method m : cls.getDeclaredMethods()) {
    if(m.getName().equals(methodName))
      parTypes = m.getParameterTypes();
  }
  Method m = cls.getDeclaredMethod(methodName, paramTypes);
  m.invoke(sms, new Object[]{"8210", null, "TEST", null, null});
<중략>
```

(그림 5) 디컴파일 이전의 리플렉션 적용 전후 소스 비교

중략된 부분은 try~catch문과 "*getDeclaringClass() .getName()*"를 이용하여 해당 메소드가 속한 클래스의 이름을 얻고 "*Class.forName()*"을 이용하여 Class<?>형 변수를 얻는 부분이다.

공통적으로 메소드 이름이 "String"형의 파라미터로 이용되어 간접호출 되는 것을 알 수 있다. 그리고 각 API에 따라 사용 방법에 따라 몇 가지 예외처리가 필요했다. 위에서부터 순서대로 패턴 1, 2, 3이라 칭할 때, 1번 패턴에서 "setWifiEnabled"의 경우 리턴 값으로 boolean을 갖는

2번 패턴이었으나 단순히 Wi-Fi를 작동시키는 데 사용하였으므로 1번으로 변경하여 구조를 바꾸었다. 또한 2번 패턴 중에서 만약 해당 메소드를 가지고 있는 객체 변수가 다른 행의 전역변수로 사용될 때 설계도구의 작동 원리상 행 단위로 소스를 읽기에 한계가 있어 따로 전역변수와 "= null" 문자열을 시작 부분에 삽입하고, try~catch안에 있는 동명의 변수 앞에 형(Type)을 제거하는 추가적인 처리를 하였다. (이 예제에서는 openConnection, getContentResolver이 해당한다.)

마지막으로 디컴파일 이후의 리플렉션 적용 전후 소스 비교이다. 3장에서 말한 바와 같이 dex2jar와 Java Decompiler를 사용하여 분석하였다. 각 버전은 0.0.9.15, 0.3.5에서 실험되었고, 결과는 (그림 6)과 같다.

파라미터가 있고, 리턴값이 void가 아닌 경우

•[변환 전]
```
localWifiManager.setWifiEnabled(true);
```
•[변환 후]
```
<중략>
Method localMethod3 =
  localClass.getDeclaredMethod("setWifiEnabled", (Class[])localObject);
Object[] arrayOfObject = new Object[1];
arrayOfObject[0] = Boolean.valueOf(true);
localMethod3.invoke(localWifiManager, arrayOfObject);
<중략>
```

파라미터가 없고, 리턴값이 void가 아닌 경우

•[변환 전]
```
HttpURLConnection localHttpURLConnection =
  (HttpURLConnection)new URL("http://www.google.com")
  .openConnection();
```
•[변환 후]
```
Error. 검출 불가.
```

파라미터가 있고, 리턴값이 void인 경우

•[변환 전]
```
SmsManager.getDefault().sendTextMessage("8210[illegible]", null,
"TEST", null, null);
```
•[변환 후]
```
<중략>
SmsManager.getDefault().sendTextMessage("8210[illegible]", null,
  "TEST", null, null);
Method localMethod1 =
  localClass.getDeclaredMethod("sendTextMessage", (Class[])localObject);
Object[] arrayOfObject = new Object[5];
arrayOfObject[0] = "8210[illegible]";
arrayOfObject[2] = "TEST";
localMethod1.invoke(localSmsManager, arrayOfObject);
<중략>
```

(그림 6) 디컴파일 이후의 리플렉션 적용 전후 소스 비교

결과를 보면 리플렉션을 적용하지 않은 소스에서는 직접적인 API 호출문이 보이지만, 적용한 소스에는 보이지 않을뿐더러 몇몇 API에서는 Java Decompiler에서 "Error"로 판단하여 해당 부분이 전체 주석되어 분석이 불가능한 경우도 있었다. (이 예제에서는 openFileOutput, openFileInput, openConnection이 해당한다.)

5. 결론

본 논문은 자바 언어에서 클래스의 모든 요소들을 간접적으로 호출하거나 조작하는 기능 중 하나인 리플렉션을 안드로이드 환경에 적용하여 API 정적 분석을 회피할 수 있을 것이라는 가능성에 착안하였다.

실험에서는 안드로이드 환경에서 위험 API라고 알려진 군들을 리플렉션을 적용하여 앱을 제작하여 리플렉션을 적용한 전후 소스의 디컴파일 전후 결과를 비교하였다. 4장에서 살펴본 바와 같이 API 호출 방식이 API 이름의 문자열과 Object형 배열의 파라미터 등으로 이루어져 보다 복잡해지고 간접적으로 바뀌었고, 디컴파일 후에도 똑같이 적용되었으며 간혹 디컴파일러가 인식할 수 없어 기존의 정적 분석을 통한 API 탐지 방법으로부터 회피할 수 있음을 확인하였다.

그러나 사용 목적상 다른 패턴을 사용하는 경우, 혹은 설계 도구의 원리상 행 단위로 문자열을 읽어 분석 후 변환하기 때문에 다른 행에서 해당 메소드를 전역 변수에 대입하여 사용하는 경우에는 부가적인 예외처리가 필요하다. 또한 범위를 국한하여 일부 API에만 적용한 것이기에 보다 범용적이고 정확한 분석에 대한 연구 및 보완이 필요하다.

참고문헌

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

[2] 조성연외 1명, 자바의 리플렉션을 이용한 동적 재구성을 지원하는 객체 복제 관리 시스템의 설계, 한국정보과학회 학술발표논문집, 26권, 1호(A), pp.373-375, 1999.

[3] 최종명, 자바 리플렉션과 메소드 오버로딩을 활용한 상황인지 툴킷, 정보과학회논문지 : 소프트웨어 및 응용 39호, 1호, pp.12-23, 2012.

[4] 이원찬외 1명, 자바 리플렉션 쓰임에 대한 연구, 한국정보과학회 학술발표논문집, 39권, 2호(A), pp.277-279, 2012.

[5] 심원태 외 3명, 안드로이드 앱 악성행위 탐지를 위한 분석 기법 연구, 정보보호학회논문지, 21권, 1호, pp.213-219, 2011.

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

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