논문 13-38B-08-07

한국통신학회논문지 '13-08 Vol.38B No.08
http://dx.doi.org/10.7840/kics.2013.38B.8.654

프로가드 난독화 도구 구조 및 기능 분석

Yuxue Piao◆, 정 진 혁*, 이 정 현°

Structural and Functional Analyses of ProGuard Obfuscation Tool

Yuxue Piao◆, Jin-hyuk Jung*, Jeong Hyun Yi°

요 약

안드로이드 앱은 자바 언어의 특성과 셀프사인 정책에 따라 역컴파일을 통한 위변조된 앱의 생성이 쉬운 구조적 특성이 있다. 이러한 구조적 취약점을 금융 앱과 같이 보안에 민감한 앱에 적용되면 매우 치명적일 수 있다. 따라서 앱 위변조 방지 기술 중 하나로써 난독화(obfuscation)을 적극 도입하여 활용하고 있는데, 현재 안드로이드 마켓에 등록된 많은 앱들은 프로가드(ProGuard) 난독화 도구를 사용하고 있다. 프로가드는 자바 클래스 파일을 난독화하여 안드로이드 앱의 역공학을 어렵게 만든다. 하지만 프로가드 난독화 도구는 여러 가지 난독화 기법 중 앱 속의 식별자를 변환하는 식별자변환 기법만 적용되어 있기 때문에, 역공학 시 프로그램 로직을 쉽게 파악할 수 있게 된다. 따라서 본 논문에서는 프로가드 소스코드 분석을 통한 프로가드의 난독화 기법을 상세 분석하고, 이를 통해 현재 프로가드 난독화 기술의 한계점을 파악한 후, 향후 안드로이드 난독화 기술의 개선 방향을 제시한다.

Key Words : Obfuscation, ProGuard, Android Security

ABSTRACT

Android applications can be easily decompiled owing to their structural characteristics, in which applications are developed using Java and are self-signed so that applications modified in this way can be repackaged. It will be crucial that this inherent vulnerability may be used to an increasing number of Android-based financial service applications, including banking applications. Thus, code obfuscation techniques are used as one of solutions to protect applications against their forgery. Currently, many of applications distributed on Android market are using ProGuard as an obfuscation tool. However, ProGuard takes care of only the renaming obfuscation, and using this method, the original opcodes remain unchanged. In this paper, we thoroughly analyze obfuscation mechanisms applied in ProGuard, investigate its limitations, and give some direction about its improvement.

Ⅰ. 서 론

개방적인 안드로이드 플랫폼 정책은 안드로이드 플랫폼 대중화를 단기간에 이끌어 냈지만, 보안관점에서 여러 가지로 취약한 특성을 갖고 있다. 특히 위변조된 앱[1]의 마켓 등록이 가능하여 금융 앱, 혹은 모바일 지갑[2]과 같은 보안에 민감한 앱에 위협이 된다. 따라서 앱의 데이터나 알고리즘의 유출을 방지하는 기술은 반드시 필요하다. 난독화(obfuscation)[3]는 프로그램 변환의 일종으로 코드를 읽기 어렵게 함으로써 역공학(reverse engineering)공격을 방해하는 기술이다. 현재 안드로이드 마켓에 등록된 많은 앱들은 프로가

※ 본 연구는 미래창조과학부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(NIPA-2013-H0301-13-1003)
◆ 주저자 : 숭실대학교 대학원 컴퓨터학과, qianbin127@ssu.ac.kr, 정회원
° 교신저자 : 숭실대학교 컴퓨터학부, jhyi@ssu.ac.kr, 종신회원
* 숭실대학교 대학원 컴퓨터학과, nemojjh@ssu.ac.kr, 학생회원
논문번호 : KICS2013-04-190, 접수일자 : 2013년 4월 25일, 최종논문접수일자 : 2013년 7월 10일

드(ProGuard)[4] 난독화 도구를 적용한 후 배포되고 있다.

따라서 본 논문에서는 프로가드에 적용된 난독화 기술의 상세한 기법을 분석한다. 분석 결과를 통해 프로가드의 난독화 기법을 평가하고 현재 프로가드 난독화 기술의 한계점과 앞으로 안드로이드 난독화 기술의 개선 방향을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 난독화 기술 동향과 자바 가상머신에서 사용되는 기본 개념들을 살펴본다. 3장에서는 프로가드의 소스코드 아키텍처와 클래스 자료 구조들을 분석하고, 4장에서는 프로가드 기능 분석을 통하여 난독화 알고리즘의 구조를 설명한다. 5장에서는 보안 측면에서 프로가드의 난독화 기법을 평가한다. 6장에서는 개선 방향을 제시하고 결론을 맺는다.

Ⅱ. 관련연구

안드로이드 앱 생성 과정을 보면 자바 소스코드를 컴파일하여 자바 바이트코드로 생성한 후 다시 컴파일 하여 달빅 바이트코드를 생성한다. 프로가드는 이와 같은 코드 변환단계 중 컴파일되어 생성된 자바 바이트코드에 적용되고 난독화가 적용된 클래스 파일이 달빅 코드로 변환된다. 따라서 프로가드소스 코드를 분석하기 위해서는 클래스 파일 포맷과 자바 가상 머신에서 사용되는 개념에 대한 이해가 필요하다. 본 장에서는 이를 이해하기 위한 개념들과 기존 난독화 기술 동향 및 프로가드 외에 현재 사용되고 있는 난독화 도구들을 비교 분석한다.

2.1. 클래스 파일 포맷

클래스 파일 포맷[5]의 정의에 따르면 클래스 파일은 그림 1과 같이 크게 헤더(Header), 상수 풀(Constant Pool), 접근 플래그(Access Flag), 자식 클래스(This Class), 부모 클래스(Super Class), 인터페이스(interfaces), 필드(Fields), 메소드(Methods), 속성(Attributes)들로 구성된다.

헤더는 “0xCAFEBABE”라는 16진 숫자로 시작하는 매직넘버와 실행될 가상머신의 버전 넘버가 저장되어 있다. 매직넘버는 자바 가상머신이 클래스 파일을 식별하기 위한 것이다. 상수 풀에는 필드명, 메소드명, 클래스명, 디스크립터, 속성, 상수 등을 문자열이나 정수를 통해 부동한 자료 구조로써 이 영역을 구성하고 있다. 전체적인 파일 구조상 상수 풀에는 실질적인 데이터가 나열되어 저장되며 다른 영역에는 상

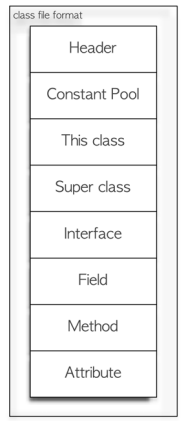


Fig. 1. Class file format

수 풀에 저장된 데이터의 인덱스를 참조하여 정의한다. 접근 플래그는 private, public, protected 등의 접근 제어자와 관련된 자식 클래스의 접근권한을 정의하고 있다. 자식 클래스영역에는 자식 클래스의 디스크립터 인덱스를 저장하고 있으며 부모 클래스(Super Class) 영역은 자식 클래스가 직접 상속받게 되는 클래스의 디스크립터 인덱스를 담고 있다. 인터페이스는 클래스가 상속받은 인터페이스들의 정보를 담고 있다. 필드 영역에는 클래스에서 정의된 정적(static)필드 혹은 일반 필드의 정보를 담고 있고 메소드 영역에는 클래스에서 정의된 정적 메소드와 일반 메소드의 정보를 가지고 있다. 또한, 여기에는 코드 속성도 포함되어 메소드의 구체적인 행위를 결정하게 된다. 제일 마지막에 있는 속성 영역에는 디버깅을 위한 정보나 소스코드 파일의 매핑을 위한 부가 정보 등을 포함하고 있다.

2.2. 디스크립터(descriptor)

디스크립터는 클래스, 필드 혹은 메소드의 타입을 표현하기 위한 문자열이다.

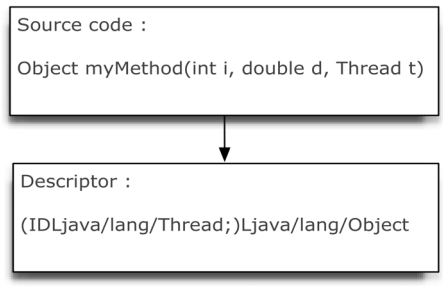


Fig. 2. Source code and descriptor

그림 2는 자바 소스코드가 컴파일된 후 디스크립터가 생성되는 과정을 설명한다. myMethod라는 메소드

는 반환타입이 Object이고 3개의 파라미터 int 타입 i, double 타입 d, Thread 타입 t를 가지고 있다. 이 메소드가 자바 컴파일러에 의하여 컴파일 된 후 클래스 파일에는 int가 I로, double이 D로, Thread가 Ljava/lang/Thread로 변환되고 소괄호로 묶인다. 또한 반환타입은 소괄호 다음으로 Ljava/lang/Object로 변환되어 조합된다. 이런 문자열이 바로 JVM(Java Virtual Machine)이 식별 할 수 있는 디스크립터이다. 디스크립터는 자바 프로젝트에서 매개 객체의 구체적인 타입을 클래스에서 식별하게 하는 식별자와 같은 역할을 한다. 프로가드도 이런 특성을 이용해서 매개 클래스 파일 포맷의 디스크립터를 변경하는 것을 통하여 식별자변환 난독화 기법을 구현한다.

2.3. 자바 난독화 기술 동향

자바에서 주로 적용되고 있는 난독화 기술로는 식별자변환(renaming), 제어 흐름(control flow), 문자열 암호화(string encryption), API 은닉(API hiding)과 클래스 암호화(class encryption)를 들 수 있다. 식별자 변환은 클래스, 필드 혹은 메소드의 이름을 의미 없는 이름으로 대체하여 역컴파일된 소스코드 분석을 어렵게 하는 기법이다. 흐름 제어는 클래스 파일 속의 코드영역의 명령어 위치를 바꾸거나, 불투명 술부를 삽입 혹은 쓰레기 명령어들을 삽입하여 역컴파일 시 잘못된 결과를 초래하거나 제어흐름을 분석하기 어렵게 하는 기법이다. 문자열 암호화는 말 그대로 사용되는 문자열을 일련의 암호화를 통해 암호화된 문자열로 대체하고 복호화 메소드를 클래스 파일 속에 추가한 후 실행시 암호화된 문자열을 복호화하도록 한다. 이는 클래스 파일의 정적 분석으로부터 민감한 문자열을 숨겨줄 수 있다. API 은닉은 민감한 라이브러리 사용 혹은 메소드 호출을 감추는 기법이고 클래스 암호화는 특정한 클래스 파일 자체를 암호화를 해서 저장하고 사용 시 복호화 과정을 거쳐 프로그램이 동적으로 복호화된 코드를 실행토록 하는 기법이다.

Table 1. Feature comparison of Android obfuscators

	Pro Guard	DashOPro	Allatori	Dex Guard
Renaming	Yes	Yes	Yes	Yes
Control Flow	No	Yes	Yes	Yes
String Encryption	No	Yes	Yes	Yes
API Hiding	No	No	No	Yes
Class Encryption	No	No	No	Yes

표 1은 앞서 살펴본 5가지 난독화 기법들을 분류 기준으로 현재 사용되고 있는 도구들을 비교분석한 것이다. DashOPro[6], Allatori[7], DexGuard[8] 등은 상업용 난독화 도구로 프로가드에 비하면 식별자변환 난독화 기법뿐만 아니라 다른 기능들도 지원하고 있다.

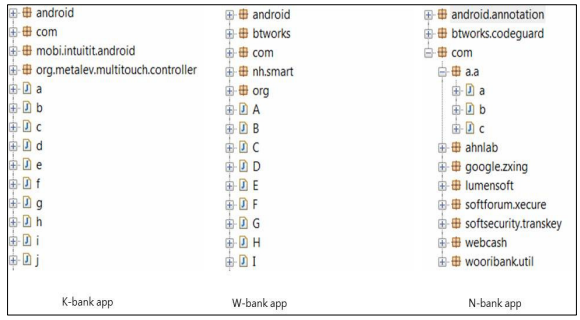


Fig. 3. Only identifier renaming is applied to K,W, and N banking apps

K, W, N 은행에서 제공하는 뱅킹 앱들을 역컴파일해 본 결과 그림 3과 같이 식별자변환 난독화만 적용이 되어 있음을 확인할 수 있다.

Ⅲ. 프로가드 난독화 모듈 정적 분석

본 장에서는 프로가드에 적용된 식별자변환 기법을 파악하기 위해 프로가드(버전 4.7) 소스코드를 상세 분석한다. 프로가드 소스코드 패키지는 기능별로 3가지 유형으로 분류된다. 첫째, 클래스 파일 포맷을 표현하는 엔티티(Entity) 클래스들로 이루어진 패키지이고 둘째, 엔티티 클래스들을 방문하는 클래스들로 이루어진 방문자(Visitor) 패키지 집합이다. 셋째, 클래스 파일 포맷에서 바이트코드를 실제로 조작하는 행위를 하는 액션(Action) 클래스들의 패키지이다.

3.1. 프로가드 엔티티 클래스

그림 4는 엔티티 클래스들로 이루어진 패키지에 대한 설명이다. proguard.classfile.constant 패키지에 들어 있는 클래스들은 클래스 파일 포맷의 상수 풀 영역에 저장된 각 상수 자료 구조들을 포함하고 있다.

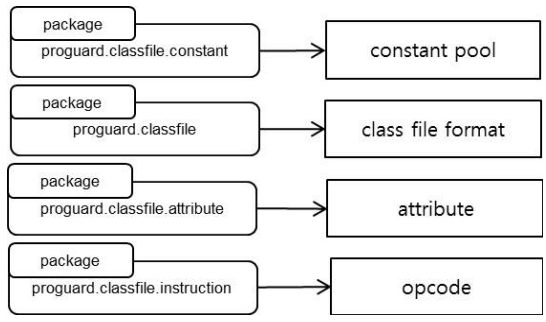


Fig. 4. Packages for entity classes

proguard.classfile.attribute 패키지속의 클래스들은 클래스 파일 포맷의 속성 영역을 의미하고, proguard.class.instruction 패키지 속의 클래스들은 자바 가상 머신이 식별할 수 있는 연산 명령어들을 포함하고 있다. 마지막으로 이런 엔티티 클래스들을 구조적으로 조합하는 클래스들은 proguard.classfile 패키지 속에 들어 있다.

3.1.1. 클래스 파일 포맷 엔티티 클래스 집합

프로가드는 proguard.classfile 패키지 속의 클래스를 통해 클래스 파일을 객체로 구조화한다.

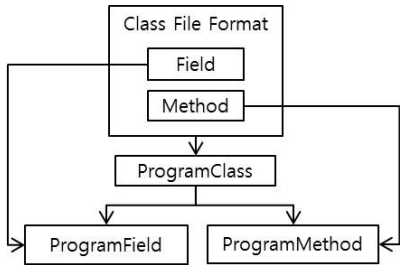


Fig. 5. Process for loading class file as entity classes

그림 5는 객체 구조와 클래스 파일을 읽는 과정을 보여준다. 프로가드는 클래스 파일을 ProgramClass와 매핑시켜 ProgramField 엔티티 클래스로 필드를 읽어들이고 ProgramMethod 엔티티 클래스로 메소드를 읽어 들인다.

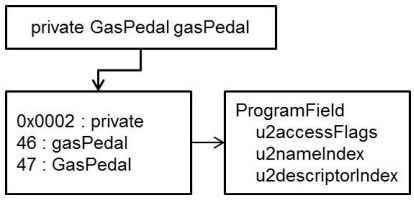


Fig. 6. Process for mapping Bytecode field and ProgramField class

그림 6은 이와 같은 로딩 과정의 예로 private 권한을 가지고 타입이 GasPedal인 gasPedal 필드를 컴파일한 후 생성된 바이트코드 값들이 프로가드의 ProgramField 클래스의 필드와 매핑되는 과정을 보여준다. private 접근자는 0x0002로 표현되고 이 값은 ProgramField의 u2accessFlags 필드에 저장된다. gasPedal의 디스크립터는 “Ljovial/slotCar/GasPedal;”로써 실제 문자열은 상수 풀 Constant_Utf8 자료 구조에 저장이 되고 상수 풀에 있는 인덱스 값 47이 u2descriptorIndex에 저장된다. 필드의 이름 gasPedal은 상수 풀에 Constant_Utf8 자료 구조로 저장되고 저장된 인덱스 값 46이 u2nameIndex에 매핑된다. ProgramMethod는 ProgramField와 같은 구조를 가지는데 다만 메소드이기 때문에 실제 행위를 하는 연산자를 추가로 가지고 있다. 이 연산자는 ProgramMethod의 attributes 필드에 저장된다.

3.1.2. 상수풀 엔티티 클래스 집합

다음은 proguard.classfile.constant 패키지 속의 중요한 클래스들을 설명한다. 클래스는 클래스 파일 포맷의 상수 풀에서 정의한 상수 자료 구조를 표현하며 다음 표 2는 상수 자료 구조를 나타내는 클래스들과 대응되는 클래스 파일포맷의 상수 풀에 대한 설명이다.

아래 클래스 중 Utf8Constant 클래스에는 문자열 형식의 데이터가 저장되며 이 중 디스크립터 문자열이 해당 상수 타입으로 저장된다. 이는 역공학시 공격자가 클래스명, 메소드명, 필드명 등의 정보를 활용하기 때문에 프로가드는 이런 데이터들의 변경을 통하여 난독화를 수행하고 있다.

Table 2. Data structure of constant pool

ProGuard Constant Classes	Description
ClassConstant	represent a class or interface
FieldrefConstant	represent fields
InterfaceMethodrefConstant	represent interface methods
MethodrefConstant	represent methods
NameAndTypeConstant	represent a field or method, without indicating which class or interface type it belongs to
DoubleConstamt	represent 8-byte numeric constants
FloatConstant	represent 4-byte numeric constants
LongConstant	represent 8-byte numeric constants
IntegerConstant	represent 4-byte numeric constatns
StringConstant	represent constant objects of the type String
Utf8Constant	represent constant string values

3.1.3. 연산코드 엔티티 클래스 집합

자바 가상 머신은 201개의 연산자를 사용하고 있다. 프로가드는 각 연산자에 대응한 엔티티 클래스를 사용하지 않고 연산자를 기능별로 분류하여 각 기능별로 상응한 엔티티 클래스를 정의한다. 표 3은 연산자 클래스들을 정리한 것이다. 프로가드는 이런 연산자들을 검색하면서 리플랙션에서 Class.forName 메소

드로 동적으로 초기화하는 클래스들도 난독화를 할 수 있다. 동적으로 초기화하거나 호출되는 클래스, 필드, 메소드들은 클래스 파일 포맷에서 상세한 정보가 없다. 다만 Constant_Utf8 에 디스크립터만 저장이 되어 있기에 연산자를 통하여 동적 할당을 처리한다.

Table 3. Package for instruction classes

Class Name	Description
SimpleInstruction	represent instructions that using no constant pool
ConstantInstruction	represent instructions that using constant pool
VariableInstruction	represent instructions that using local stack
BranchInstruction	represent conditional jump instructions
TableSwitch Instruction	represent short path switch instructions
LookUpSwitch Instruction	represent long path switch insructions

3.2. 프로가드의 방문자 패키지와 클래스

방문자 클래스는 각 필드가 복잡하게 연결된 클래스 파일을 엔티티 클래스를 이용해 자료 접근과 탐색을 수월하게 하는 클래스 집합으로 방문자 패턴(Visitor Pattern)[9]을 사용한다.

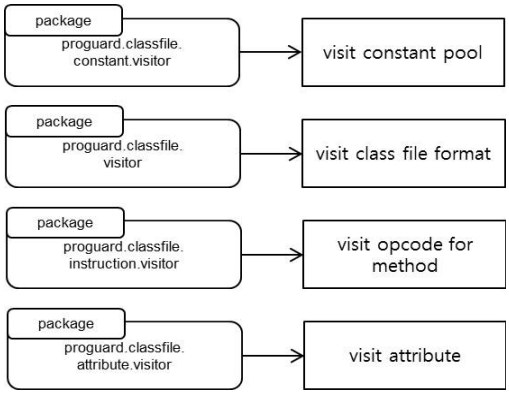


Fig. 7. Roles of visitor classes

그림 7은 프로가드에서 엔티티 클래스들을 방문하는 방문자 클래스들의 패키지와 기능들을 보여주고 있다. 그림에서 표현한것처럼 상수풀, 자바연산코드, 속성과 클래스 파일 포맷 별로 방문자 클래스 패키지를 형성한다.

3.3. 프로가드의 액션 클래스

액션(action)클래스들이 실질적인 난독화 알고리즘이 구현된 클래스이다. 모듈 별로 구분하면 난독화 클래스, 최적화 클래스, 축소화 클래스들로 분류할 수 있다. 이 클래스들은 방문자 클래스들을 통하여 지정된 클래스 파일 포맷의 데이터를 접근하고 바이트코드 조작을 한다. 실질적으로는 클래스 파일 포맷에서 명시한 클래스, 필드, 메소드, 상수, 속성들의 자료 구조를 생성, 수정, 삭제하는 것이다. 표 4는 액션 클래스들을 기능별로 분류한 패키지들을 나타낸다.

Table 4. Packages for logic classes

Package Name	Description
proguard.classfile.editor	Edit class file format
proguard.classfile.io	Read and write the class file
proguard.obfuscate	Obfuscation
proguard.optimize	Optimization
proguard.shrink	Shrinking

Ⅳ. 프로가드 난독화 모듈 동적 분석

프로가드는 읽기(ReadInput), 초기화(Initialize), 축소화(Shrink), 최적화(Optimize), 난독화(Obfuscate), 사전검증(Preverify)과 쓰기(WriteOutput) 모듈로 구성되어 있는데, 본 논문에서는 난독화와 직접 관련된 모듈들을 중심으로 분석하고자 한다.

4.1. 난독화 모듈

난독화 모듈은 읽기 모듈과 초기화 모듈에 의해 생성된 엔티티 객체들을 통해 식별자변환 기법을 적용하는 모듈이다. 식별자변환은 패키지, 클래스, 필드, 메소드의 이름을 대상으로 하며 변환된 이름들은 알파벳으로 구성된 무의미한 이름으로 바꾸어 준다. 또한, 지역 변수의 정보 속성과 디버그 속성들을 제거하기 때문에 역컴파일 했을 때, 코드 분석을 좀 더 어렵게 할 수 있다. 식별자변환 절차 과정을 7단계로 나누어 보면 다음과 같다.

1) 엔티티 클래스들의 visitorInfo 필드를 null로 초기화한다.
2) 상속관계를 이룬 클래스들의 메소드들을 연결한다.
3) 난독화를 보류할 클래스, 필드, 메소드, 속성들을 검사한다.
4) 제거 가능한 속성들을 제거한다.
5) 클래스, 필드, 메소드의 이름을 생성한다.
6) 변경한 이름을 실제 클래스, 필드, 메소드에 적용한다.
7) 상수 풀에서 사용되지 않는 상수 자료 구조들을

제거한다.

4.1.1. 상속관계 메소드 연결

초기화 과정에서 프로가드가 읽어 들인 클래스들의 상속관계를 연결한다. 이는 난독화가 적용되는 클래스와 의존성이 있는 메소드나 필드를 처리하기 위한 작업이다. 이렇게 연결이 된 후 사용자가 지정해 주거나 시스템적으로 난독화를 거치지 말아야 할 메소드가 하나라도 이 연결과 관계되면 해당 연결의 모든 대상은 난독화를 거치지 않는다.

4.1.2. 난독화 보류대상 설정

사용자가 지정한 필드, 메소드, 속성과 라이브러리 클래스나 API와 같은 외부 라이브러리 등을 보류 대상으로 설정한다. 이와 같이 난독화 보류 대상을 설정하는 이유는 해당 부분을 난독화하면 프로그램이 정상적으로 실행되지 않는 문제가 발생하기 때문이다. API와 같이 호출되는 외부 라이브러리 호출 함수명을 난독화하면 호출에 필요한 디스크립터 문자열이 변경되어 API 링킹이 정상적으로 수행되지 않는다. 이는 가상머신에서 동작하는 자바 특성상 디스크립터 문자열로 링킹이 되어야만 하기 때문인데 외부 라이브러리에 정의된 API 역시 동일한 이름으로 변경하지 않는 한 API를 난독화할 수는 없다.

4.1.3. 속성 제거

클래스 파일에서 속성은 디버깅 정보나 코드 라인 번호 등 클래스의 부가적인 정보를 가지고 있다. 역컴파일러는 이런 속성들을 이용하여 소스코드를 복원하는데 참조한다. 기본적으로는 프로가드는 Code_attribute 속성과 ConstantValue_attribute 속성을 제외한 모든 속성을 제거한다. Code_atrribute 속성에는 자바 가상 머신에서 실제로 실행될 연산코드가 있고 ConstantValue_attribute에는 정적 필드의 실제 값을 저장하고 있다. 이외의 모든 속성은 디버깅 같은 다른 보조기능을 위한 자료 구조로써 삭제해도 무방하다.

4.1.4. 식별자 생성

ClassObfuscator 클래스와 MemebrObfuscator 클래스는 로딩된 클래스 파일의 객체들을 검색하면서 새로운 이름을 부여한다. 사용자가 난독화할 문자열 사전을 지정하지 않으면 프로가드는 내부에서 간단한 난독화 알고리즘을 이용하여 a, b, c, d 순으로 새로운 이름들을 생성하고 객체들에서 visitorInfo 영역의 값이 null이면 이 영역에 새로운 이름을 추가한다. 다만 visitorInfo 영역의 값이 원소의 이름으로 채워지면 새로운 이름을 부여하지 않는다. 부여한 이름은 난독화 후 새로운 클래스들을 생성할 때 사용하게 될 이름이다. 이는 실제로 상수 풀에 새로운 이름의 Constant_utf8 자료 구조를 추가하는 것과 같다. 그림 8는 원본 클래스의 상수 풀에 “a” 라는 값이 Utf8_constant자료 구조가 추가되는 과정을 보여주고 있다.

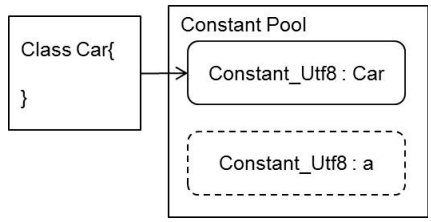


Fig. 8. Step 1: adding a renamed identifiers into constant pool

4.1.5. 식별자 변환 문자열 적용

ClassRenamer 클래스는 실제 클래스 파일들을 별도의 이름으로 변경한다. 해당 루틴 실행시 visitorInfo에 있는 문자열과 주체 이름의 동일여부를 검사하고, 다를 경우 visitorInfo에 있는 이름으로 원래 주체의 이름을 대체한다. 실제는 상수 풀에 새로운 Constant_Utf8 자료 구조를 추가하고 Constant_Utf8 속에 새로운 이름의 바이너리 값으로 채운다. 그 다음 Constant_Class의 인덱스 값을 새로 추가한 Constant_Utf8의 인덱스로 수정한다. 이러한 과정에서 상수 풀에는 사용되지 않는 자료 구조들이 생성된다. 이런 자료 구조들은 다음 단계에서 제거하게 된다. 그림 9은 새로운 이름을 담고 있는 Constant_Utf8의 인덱스값으로 원본 클래스의 이름이 되도록 변경하는 과정을 보여준다.

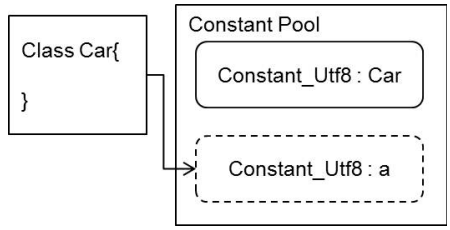


Fig. 9. Step 2 : adjusting the index to the structure with the renamed identifiers

4.1.6. 미사용 상수 구조 제거

새로운 이름을 적용한 후 상수 풀에 새로운 Constant_Utf8 자료 구조가 추가된다. 그러면 원래 사용했던 Constant_Utf8은 사용하지 않게 된다.

ConstantPoolShrinker 루틴은 이러한 필요 없는 상수 자료 구조를 찾아서 삭제하고 상수 풀을 재배치해준다.

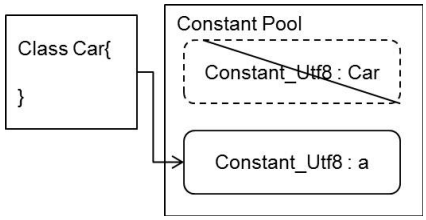


Fig. 10. Step 3 : Removing the unused constants

그림 10은 원본의 상수 풀에 있던 Car 문자열의 자료 구조를 제거하는 과정이다. 이런 과정을 거친 후에 새로운 클래스 실행 파일을 생성하게 된다.

V. 프로가드 난독화 평가

프로가드는 클래스명, 필드명, 메소드명 등을 식별자 변환 기법을 통해 난독화한다. 이 방법은 클래스명 등으로부터 추출할 수 있는 정보를 무의미하게 하여 코드 이해를 어렵게 하는 것이다. 또한, 프로가드에서 제공하는 사용자 정의 템플릿을 활용하면 별도의 알고리즘을 추가할 필요 없이 여러 가지 종류의 식별자 변환 기법을 적용할 수 있는 것이 장점이 있다. 그림 11은 프로가드 적용 전과 후를 jd[10] 역컴파일러로 역컴파일하여 비교한 것이다. 대부분의 식별자가 변환되어 식별자명으로부터 클래스나 필드의 역할을 유추하기 어려운 것을 확인할 수 있다.

Fig. 11. Comparison before and after applying Proguard obfuscator

```
public Car(TrackPosition pos, Color drawColor,
{
  this.pos_ = pos;
  this.drawColor_ = drawColor;
  this.eraseColor_ = eraseColor;
  this.gasPedal_ = gasPedal;
  int[] xs = new int[4];
  int[] ys = new int[4];
  this.poly_ = new Polygon(xs, ys, 4);
}
```

original method

```
public a(h paramh, Color paramColor1, Color paramColor2,
{
  this.a = paramh;
  this.b = paramColor1;
  this.c = paramColor2;
  this.e = paramb;
  int[] arrayOfInt1 = new int[4];
  int[] arrayOfInt2 = new int[4];
  this.d = new Polygon(arrayOfInt1, arrayOfInt2, 4);
}
```

obfuscated method

Fig. 12. The program logic is easily identifiable even if identifiers are randomized

하지만, 그림 12에서 보는 것과 같이 난독화 전과 후의 역컴파일 결과를 비교해보면, 메소드의 로직은 원상태로 보존되어 있는 것을 확인할 수 있다. 따라서 공격자는 식별자명을 파악하는데 약간의 어려움이 있지만, 여전히 실행코드의 제어흐름은 쉽게 파악할 수 있기 때문에 역공학이 비교적 쉽다.

VI. 개선 방향

프로가드는 식별자 변환을 통하여 문자열에 의한 정보 노출을 줄일 수 있다. 하지만 메소드의 제어 흐름이 그대로 보전되고 API 호출 정보가 남아 있어 역공학을 통해 여전히 코드 분석이 쉽게 이루어진다. 따라서 연산 자원이 상대적으로 부족한 안드로이드 모바일 기기 특성을 고려하면서 프로가드 난독화 수준을 개선하기 위해서는 제어 흐름 난독화를 추가적으로 적용할 필요가 있다. 제어 흐름 난독화는 다양한 형태로 제공 가능하다. 주로 사용되는 기법으로는 메소드 속의 연산코드 순서를 동일 기능을 하는 다른 연산코드들로 대체하거나 순서를 재배치하여 역컴파일러가 해석하지 못하게 하는 것이다. 다른 방법은 메소드 속에 쓰레기 연산코드를 추가하여 역컴파일된 코드가 길어지게 만드는 것이다. 또 다른 방법은 중요한 배열 또는 정수 값을 원래 값 대신 모듈로(modulo) 연산 통해 다른 값으로 대체하여 정적 분석 시에 메소드 파악을 어렵게 할 수 있다.

```
Class hello = Class.forName(
    "edu.example.Hello"
);
Object object = hello.newInstance();

Method method = hello.getMethod(
    "say",null
);

method.invoke(object, null);
```

Before Descriptor Encryption

```
String x = A(a,b,c);
Class hello = Class.forName(x);
Object object = hello.newInstance();

String y = A(e,o,f);
Method method = hello.getMethod(
    y, null
);

method.invoke(object, null);
```

After Descriptor Encryption

Fig. 13. Example of String Encryption and API Hiding

이와 같은 제어 흐름 난독화의 예로, 그림 13은 Reflection API와 문자열 암호화를 이용해 API 호출 정보를 난독화한 결과를 보여주고 있다. Reflection API를 통해 메소드를 호출하고 사용되는 문자열을 암호화하여 실행 시점에 메소드 A에 의해 문자열이 복호화되어 정상적으로 동작하게 된다. 이 경우 정적인 역공학 분석 방법으로는 어떤 메소드가 호출되는지 알아내기가 어렵게 된다.

또한, 중요한 클래스는 별도로 추출하여 암호화할 수 있다. 암호화된 클래스는 별도 파일 형태로 저장하거나 다른 클래스 속의 바이트 배열로 저장할 수 있다. 그러므로, 제어흐름 난독화와 클래스 암호화 기법들을 추가적으로 구현하여 기존 프로가드와 통합하면 보다 개선된 안드로이드 바이트코드 난독화 도구를 확보할 수 있을 것이다.

Ⅶ. 결 론

본 논문에서는 안드로이드 SDK에 기본 탑재된 프로가드 난독화 도구의 주요 모듈의 소스코드를 분석한 결과를 제시하였다. 현재 많은 앱이 사용하고 있는 프로가드 도구의 난독화 수준을 평가함으로써 프로가드의 난독화 도구의 장단점을 파악하였다. 특히, 뱅킹 앱과 같은 민감한 정보를 내포하고 있는 앱들에 프로가드 난독화 도구를 적용할 경우, 앱의 위변조가 여전히 쉽게 가능하다는 점을 알 수 있었다. 따라서 프로가드의 난독화 수준을 개선하기 방안으로 기존 식별자변환 기법 외에 코드 노출을 줄여주는 코드 분리 기법이나 코드 암호화 등에 대한 개선방안을 제시하였다.

References

[1] J. H. Jung, J. Y. Kim, H. C. Lee, and J. H. Yi, “Repackaging attack on android banking applications and its countermeasures,” *J. Wireless Personal Communications(WPC).*, [Online], Available: http://link.springer.com/content/pdf/10.1007%2Fs11277-013-1258-x.pdf, June. 2013.

[2] G. I. Ma, H. C. Lee, and J. H. Yi, “A secure short-range wireless connectivity scheme for mobile wallet services,” *J. KIISE : Inform. Networking*, vol. 38, no. 5, pp. 394-404, Oct. 2011.

[3] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” *Technical Report No. 148*, Univ. Auckland, New Zealand, 1997.

[4] Eric Lafortune, *ProGuard*, Retrieved May., 2013, from http://proguard.sourceforge.net/.

[5] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, “The Java virtual machine specification: Java SE 7 Edition,” *Oracle America*, Feb. 2013.

[6] PreEmptive Solutions, *DashOPro*, Retrieved May., 2013, from http://www.preemptive.com.

[7] Allatori, *Java obfuscator*, Retrieved May., 2013, from http://www.allatori.com.

[8] Saikoa, *DexGuard*, Retrieved May., 2013, from http://www.saikoa.com/.

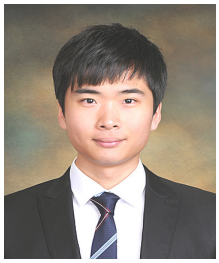
[9] Wikipedia, *Visitor Pattern*, Retrieved May., 2013, from http://en.wikipedia.org/wiki/Visitor_pattern.

[10] *Java Decompiler*, Retrieved May., 2013, from http://java.decompiler.free.fr/.

Yuxue Piao

2009년 6월 JiLin University China 학사
2011년~현재 숭실대학교 컴퓨터학과 석사과정
<관심분야> 모바일 시스템보안, 모바일 서비스보안

정 진 혁 (Jin-hyuk Jung)



2011년 2월 숭실대학교 컴퓨터학과 학사
2013년 2월 숭실대학교 컴퓨터학과 석사
2013년~현재 숭실대학교 컴퓨터학과 박사과정
<관심분야> 모바일 시스템보안, 모바일 서비스보안

이 정 현 (Jeong Hyun Yi)

1993년 2월 숭실대학교 컴퓨터학과 학사
1995년 2월 숭실대학교 컴퓨터학과 석사
2005년 8월 University of California at Irvine, Computer Science 박사
1995년~2001년 한국전자통신연구원 연구원
2000년~2001년 미국 표준기술연구원(NIST) 객원연구원
2005년~2008년 삼성종합기술원 수석연구원
2008년~현재 숭실대학교 컴퓨터학부 조교수
<관심분야> 모바일보안, 컴퓨터보안, 네트워크보안