Journal of the Korea Convergence Society
Vol. 10. No. 1, pp. 61-69, 2019

ISSN 2233-4890
https://doi.org/10.15207/JKCS.2019.10.1.061

안드로이드 저작권 보호를 위한 메소드 생성 기반 워터마킹 기법의 설계 및 구현

박희완
한라대학교 정보통신소프트웨어학과 조교수

Design and Implementation of Method Generation based Watermarking Technique for Android Copyright Protection

Heewan Park
Assistant Professor, Department of Information Communication and Software, Halla University

요 약 스마트폰이 널리 보급되고 수많은 애플리케이션들이 개발됨에 따라서 소프트웨어 저작권 관련하여 사회적인 이슈가 발생하고 있다. 소프트웨어 워터마킹은 디지털 워터마킹 기술을 소프트웨어에 적용한 것으로서 소프트웨어 원저작권자를 판별하는데 사용될 수 있는 기술이다. 안드로이드 환경에서 앱을 개발하기 위해서 일반적으로 자바 언어를 사용한다. 자바는 객체지향 언어로서 메소드 오버로딩과 오버라이딩을 지원한다. 본 논문에서는 메소드 생성 기반 워터마킹 기법을 제안하고 구현하였다. 워터마크로 인한 오버헤드를 평가한 결과, 실행 파일 크기의 증가와 실행 속도의 저하가 크지 않다는 것을 확인하였다. 본 논문에서 제안하는 워터마킹 기법을 사용하면 불법 복제가 의심되거나 실제로 프로그램이 도용되었을 때 저작권 정보를 확인할 수 있으며 불법 복제 시도를 사전에 예방하는 효과도 있을 것으로 기대한다.

주제어 : 소프트웨어 도용, 저작권보호, 소프트웨어 워터마크, 메소드 오버로딩, 메소드 오버라이딩

Abstract As smartphones become widespread, numerous applications are developed and social issues related to software copyright are emerging. Software watermarking is digital watermarking technology applied to software and is a technology that can be used to recognize copyright owners. Generally, Java language is used to develop applications on the Android environment. The Java is an object-oriented language that supports method overloading and overriding. In this paper, we propose and implement a method generation based watermarking technique. As a result of evaluating the overhead due to the watermark, it was confirmed that the increase of the executable file size and the decrease of the execution speed are not large. Using the watermarking technique proposed in this paper, it is expected that copyright information can be verified when illegal copying is suspected or actual program is stolen, and piracy attempts will be prevented in advance.

Key Words : Software piracy, Copyright protection, Software watermark, Method overloading, Method overriding

1. 서론

스마트폰이 가지고 있는 편리함과 다양한 기능 때문에 남녀노소를 가리지 않고 보편적으로 사용하는 기기가 되었으며, 보급률은 해가 갈수록 급격히 증가하여 2018년 기준으로 전 세계 성인 중 3명중 2명이 스마트폰을 사용하고 있다[1,2].

스마트폰 사용자가 증가할수록 스마트폰 앱 개발자들

This work was supported by 2016 Research Grant of Halla University.
이 연구는 2016년도 한라대학교 교내연구비 지원에 의한 것임.
*Corresponding Author : Heewan Park (heewanpark@halla.ac.kr)
Received October 29, 2018 Revised November 22, 2018
Accepted January 20, 2019 Published January 28, 2019

은 사용자들이 원하는 앱을 개발하여 수익을 확대하고자 한다. 그런데 앱을 정상적인 절차를 통해서 구매하지 않고 불법 복제하여 사용하는 경우가 발생하고 있으며 실제로 불법 복제된 앱으로 인한 피해는 연간 40억 달러에 달한다[3,4].

Fig. 1. Kim-gisa and T-map copyright dispute[5]

Fig. 1은 우리나라에서 발생한 T맵과 김기사 앱의 저작권 분쟁 사례를 보여준다. T맵이 저작권을 식별하기 위해 전자지도 DB에 디지털 워터마크를 삽입했고 ‘성웅교’라는 존재하지 않는 지명을 워터마크로 사용했는데 김기사 측에서 사용하는 전자지도 DB에도 ‘성웅교’가 발견되어 저작권 분쟁이 발생하였다[5,6].

Fig. 2. Cymera and Analog Filter copyright dispute[7]

Fig. 2.에서 살펴볼 수 있듯이 카메라 필터 앱에서도 저작권 분쟁이 발생하였다. 아날로그 부타페스트 앱 필터의 개발자인 오디너리 팩토리 대표는 sk커뮤니케이션즈의 싸이메라에서 아날로그 필터를 그대로 베껴 썼다고 주장하고 있다[7,8].

디지털 워터마킹은 주로 콘텐츠 저작권을 입증하기 위한 용도로 사용된다. 이와 유사하게 소프트웨어의 저작권 보호를 위해서 소프트웨어 워터마킹 기법이 제안되었다. 소프트웨어 워터마킹이란 소프트웨어에 저작권자의 정보를 포함한 워터마크를 삽입하고 추출하는 기술이다. 워터마크를 이용해 개발자는 프로그램에 대한 저작권을 증명할 수 있고 무분별한 도용 및 복제로 인한 피해를 예방할 수 있다[9-13].

본 논문에서는 저작권 침해에 대한 대처 방안으로 메소드 생성 기반 자바 클래스 워터마킹 기법을 제안한다. 워터마킹 기법을 사용하면 프로그램의 원저작자를 확인할 수 있기 때문에 저작권 보호에 활용할 수 있으며 코드 도용 및 불법복제를 예방할 수 있다. 본 논문의 구성은 다음과 같다. 2장에서 관련 연구를 살펴보고, 3장에서 본 논문에서 제안하는 메소드 생성 기반 워터마킹 기법에 대해서 소개하며, 4장에서 시스템 구현 및 평가를 하고, 5장에서 결론과 향후 연구 과제에 대해서 논의한다.

2. 관련 연구

소프트웨어 워터마크는 워터마크 추출 방법에 따라서 정적 소프트웨어 워터마크와 동적 소프트웨어 워터마크로 나눌 수 있다.

정적 소프트웨어 워터마크는 파일 안에 정적인 형태로 저장되며 프로그램 실행 상태와 관련 없이 언제든지 삽입되어 있는 워터마크를 추출 가능하다. 국내의 기존 연구에서는 지적 재산권 보호를 위해 저수준 언어 기반 정적 워터마킹[9]과 함수 호출 규약에 기반한 소프트웨어 워터마킹 기법[10]등에서 정적 소프트웨어 워터마크를 삽입하는 방식을 제안하였고, 국외 연구로는 이미지를 프로그램의 정적 영역에 삽입하는 방식이 제안되었다[11].

동적 소프트웨어 워터마크는 프로그램이 반드시 실행되고 있을 때 워터마크를 추출할 수 있다는 특징이 있다. 프로그램 실행 중에 키 입력이나 마우스 동작에 의해서 숨겨진 워터마크가 나타나기도 하고[12] 프로그램에서 사용된 자료구조의 특징을 이용하거나 힙 또는 스택 데이터 영역에 삽입되는 방식으로도 구현가능하다[13].

한국저작권 위원회가 제공하는 정적 워터마킹 시스템으로는 Codejam이 있다[14]. Codejam은 소스코드에 원본 소스 코드에 워터마킹에 사용될 변수를 추가하는 방식으로 동작한다.

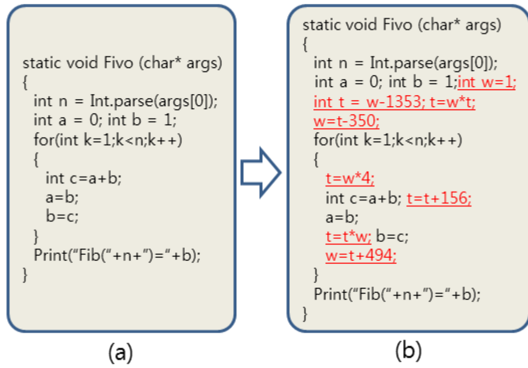


Fig. 3. (a) Original source and (b) Modified source with copyright information

Fig. 3은 Codejam 시스템의 동작에 대한 예시 코드이다. 원본 소스 (a)에는 없었던 w 변수와 t 변수가 변형된 소스 (b)에 추가되었고, 이 변수들에 대한 연산에 사용된 상수인 1, 1353, 350, 4, 156, 494은 워터마킹에 사용될 수 있다. 그러나 Codejam 시스템은 두 가지 문제점을 가지고 있다. 첫째, 코드 최적화에 취약하다. 변수 w와 t는 실제로 프로그램 동작과 상관없는 변수들이기 때문에 데이터의존성 분석을 통해서 제거 가능하므로 워터마크가 삭제될 수 있다. 둘째, 워터마크를 삽입하기 위해서 추가된 소스가 실제로 실행되기 때문에 워터마크의 양에 따라서 실행 성능이 저하될 수밖에 없다.

본 논문에서는 기존 워터마크 시스템의 단점을 보완한 메소드 생성 기반 워터마킹 기법을 제안한다. 메소드 생성과 관련된 기존 연구[15]가 제시되었으나, 본 논문과의 차별화된 내용은 다음과 같이 요약할 수 있다. 첫째, 기존 연구는 메소드 생성에 대한 기본 개념만을 제시했으나 본 논문에서는 아이디어를 구체화하고 시스템을 구현하였다. 둘째, 기존 논문에서 제시하지 않았던 실행 성능을 측정하고 워터마킹에 의한 오버헤드를 평가하였다.

3. 메소드 생성 기반 워터마크

본 논문에서는 제안하는 워터마크의 핵심 개념은 은닉성과 실행 성능으로 요약할 수 있다. 워터마크의 은닉성을 높이기 위해서 메소드 오버로딩(overloading)과 오버라이딩(overriding)을 사용한다[16]. 메소드 오버로딩과 오버라이딩을 사용하면 기존 메소드와 이름이 동일한 메소드를 복제하기 때문에 원본 메소드와 워터마크를 위해서 생성된 가짜 메소드를 구별하기 어렵기 때문에 은닉성을 높일 수 있다.

그리고 워터마크로 인한 실행 성능 저하를 막기 위해서 불분명 술어(opaque predicate)을 사용한다[17]. 불분명 술어란 참인지 거짓인지 구분이 어려운 조건식을 의미한다. 불분명 술어를 사용하면 워터마크 정보가 포함된 가짜 메소드가 데드코드(dead code)로 인식되어 삭제되는 것을 예방할 수 있으며 실제로 가짜 메소드는 실행되지 않기 때문에 실행 성능 저하를 막을 수 있다.

3.1 메소드 오버로딩 생성기(Method overloading generator)

Fig. 4.는 메소드 오버로딩의 대표적인 예제이다. 즉, 동일한 이름의 메소드를 다수 생성하되, 메소드 호출이 컴파일 타임에 정적 바인딩될 수 있도록 매개변수의 개수나 타입이 서로 달라야만 한다.

```
public void m(int a) { …… }
public void m(int a, int b) { …… }
public void m(float a) { …… }
public void m(double a) { …… }
```

Fig. 4. Example of the method overloading

메소드 오버로딩 기법을 활용하여 임의로 워터마크를 삽입하기 위해서는 오버로딩 메소드를 다수 생성할 필요가 있다. 오버로딩 메소드를 생성하기 위해서는 첫째, 오버로딩하고자 하는 대상 메소드를 선택해야하고, 둘째, 오버로딩 대상 메소드의 인자를 분석해야하고, 셋째 오버로딩 규칙에 적합한 메소드를 생성하여야 한다.

본 논문에서는 오버로딩 대상 메소드를 쉽게 선택할 수 있도록 오버로딩 대상 메소드의 시작 지점을 알리는 '// OVERLOADING {' 와 종료 지점을 알리는 '// OVERLOADING }' 두 개의 선택자를 사용하였다. 즉, 오버로딩 생성기는 원본 소스 코드를 읽어서 전체 소스 코드를 파싱할 필요가 없이 선택된 영역에 대해서만 메소드 매개변수를 분석하여 메소드를 생성하기 때문에 메소드 생성 과정이 단순해지는 장점이 있다.

```
class MakeOverloading {
  // OVERLOADING {
  public void m(int a) { …… }
  // OVERLOADING }
  public void n(int a) { …… }
}
```

Fig. 5. Example before method overloading

```
class MakeOverloading {
   public void m(int a) { …… }
   public void m(int a, int a1) { …… }
   public void m(int a, int a1, int a2) { …… }
   public void m(int a, int a1, int a2, int a3) { ……}
   public void n(int a) { …… }
}
```

Fig. 6. Example after method overloading

오버로딩 영역 선택자를 사용하여 오버로딩 전과 후는 각각 Fig. 5, Fig. 6과 같다. Fig. 5에서 메소드 m은 오버로딩 대상 메소드로 선택되었고, 그 결과 Fig. 6과 같이 m 메소드가 오버로딩된 형태로 3개의 메소드가 생성된 것을 확인할 수 있다. 예시에서는 오버로딩을 통해서 생성하고자 하는 메소드 숫자는 3개로 지정하였으나 원하는 개수만큼 변경이 가능하다.

3.2 메소드 오버라이딩 생성기(Method overriding generator)

Fig. 7.은 메소드 오버라이딩의 대표적인 예제이다. 즉, 클래스 상속 관계에서 슈퍼클래스의 메소드와 동일한 이름의 메소드를 서브클래스에서 생성하고, 메소드 호출이 런타임에 동적 바인딩될 수 있도록 매개변수의 개수나 타입까지 모두 일치해야만 한다.

```
class SuperClass {
   public void m() { …… }
}
class SubClass extends SuperClass {
   public void m() { …… }
}
```

Fig. 7. Example of the method overriding

오버라이딩 메소드를 생성하기 위해서는 첫째, 오버라이딩하고자 하는 대상 메소드를 선택해야하고, 둘째, 오버로딩 대상 클래스를 분석해야하고, 셋째 오버라이딩 규칙에 적합한 메소드를 생성하여야 한다

본 논문에서는 오버로딩과 유사하게 오버라이딩 대상 메소드를 쉽게 선택할 수 있도록 '// OVERRIDING {' 와 '// OVERRIDING }' 선택자를 사용하였다.

```
class MakeOverriding {
   // OVERRIDING {
   public void m() { …… }
   // OVERRIDING }
}
```

Fig. 8. Example before method overriding

```
class MakeOverriding {
   public void m() { …… }
}
class MakeOverriding1 extends MakeOverriding {
   public void m() { …… }
}
class MakeOverriding2 extends MakeOverriding {
   public void m() { …… }
}
class MakeOverriding3 extends MakeOverriding {
   public void m() { …… }
}
```

Fig. 9. Example after method overriding

오버라이딩 영역 선택자를 사용하여 오버로딩 전과 후는 각각 Fig. 8, Fig. 9과 같다. 클래스 상속 관계를 생성하여 서브클래스와 오버라이딩 메소드를 3개를 생성한 결과를 확인할 수 있다. 오버라이딩을 통해서 생성하고자 하는 서브클래스와 메소드 개수는 임의로 변경 가능하다.

3.3 불분명 술어(Opaque predicates)

오버로딩 생성기와 오버라이딩 생성기에 의해서 생성된 메소드에는 원본에서 존재하지 않았던 메소드이고 실제로 실행되지 않는 가짜 메소드이다. 따라서 가짜 메소드에 의해서 실행 성능이 저하되지 않으며 원하는 워터마크를 임의로 삽입할 수 있는 장소로 사용된다. 그러나 소스 최적화 과정에서 실제로 호출되지 않는 메소드는 데드코드(Dead Code)로 인식하여 삭제될 수 있다. 따라서 생성된 가짜 메소드가 삭제되지 않도록 하기 위해서 불분명 술어를 사용한다[17].

Table 1. Example of opaque predicates

Opaque predicates	Result for any integer x, y
if (x*x >= 0)	Always true
if ((2*x+1) % 2 == 0)	Always false
if (x*(x+1) % 2 == 0)	Always true
if (x*(x+1)*(x+2) % 3 == 0)	Always true
if ((x*x+1) % 7 == 0)	Always false
if ((x*x+x+7) % 81 == 0)	Always false
if ((4*x*x + 4) % 19 == 0)	Always false
if (x*x*(x+1)(x+1) % 4 == 0)	Always true

Table 1은 불분명 술어의 예시이다[18,19]. 불분명 술어는 정적 분석에 의해서 참, 거짓을 판단하기 어려운 표

현식 의미하며 워터마킹에 사용되는 코드를 숨기는 목적으로 사용 가능하다. 불분명 술어를 사용하면 실제로 실행 여부를 판단하기 어렵기 때문에 정적 분석에 의한 최적화에도 워터마크가 사라지지 않고 남을 수 있다.

3.3.1 불분명 술어를 적용한 오버로딩 생성기

오버로딩 또는 오버라이딩 메소드가 생성되었다면 불분명 술어를 추가하는 과정이 필요하다. 불분명 술어 집합으로부터 임의로 집합을 추출하여 결과가 참인 술어를 원본 메소드 호출에 사용하고, 결과가 거짓인 술어에 생성된 메소드 호출을 연결시킨다. 참과 거짓은 NOT 연산을 이용해서 반대로 바꿀 수 있기 때문에 불분명 술어 집합의 모든 데이터를 활용할 수 있다.

```
void overloadingWithOpaquePredicate {
  // OVERLOADING OPAQUE {
  s.m(10);
  // OVERLOADING OPAQUE }
}
```

Fig. 10. Method overloading example before adding opaque predicate

```
void overloadingWithOpaquePredicate {
  int x = (int)(Math.random()*99);
  if ( x*x >= 0 ) // Always true
    s.m(10); // Original mehod call
  if ( (x*x+x+7) % 81 == 0 ) // Always false
    s.m(10, 7264); // Generated method call
  if ( (2*x+1) % 2 == 0 ) // Always false
    s.m(10, 51352, 10696); // Generated method call
  if ( (2*x+1) % 2 == 0 ) // Always false
    s.m(10, 54637, 38215, 5724); // Generated method call
}
```

Fig. 11. Method overloading example after adding opaque predicates

Fig. 10과 Fig. 11은 각각 메소드 오버로딩에 불분명 술어를 적용하기 전과 후의 예제이다. Fig. 10에서 불분명 술어를 적용하고자 하는 영역을 선택하기 위해서 시작 위치에 선택자 '// OVERLOADING OPAQUE {'를 추가하였고, 종료 위치에 '// OVERLOADING OPAQUE }'를 추가하였다. Fig. 11에서의 4번째 라인의 소스가 원본이며 그 이외의 소스는 불분명 술어 생성기에 의해서 자동으로 생성된 소스이다. 즉, 원본 소스는 항상 실행되며, 5번째 라인부터 이어지는 3개의 메소드 호출문은 불분명 술어의 결과가 항상 false이므로 실행되지 않는다. 그러나 의도적으로 삽입된 불필요한 메소드 호출이 정적 분석에 의해서 삭제되지 못한다. 그 이유는 결과를 알기 어려운 불분명 술어를 사용했기 때문이다. 따라서 실제로 실행되는 첫 번째 m 메소드를 제외한 나머지 m 메소드에 워터마크 정보를 삽입할 수 있다.

3.3.2 불분명 술어를 적용한 오버라이딩 생성기

오버라이딩 메소드에 불분명 술어를 적용하는 방법은 오버로딩에서 사용한 방법과 유사하다. 그러나 메소드 오버라이딩은 클래스 상속 관계가 필요하기 때문에 불분명 술어를 추가할 때 클래스 생성하는 부분에 적용해야 한다.

```
void overloadingWithOpaquePredicate {
  // OVERRIDING OPAQUE {
  SuperClass s = new SuperClass();
  s.m(10);
  // OVERRIDDING OPAQUE }
}
```

Fig. 12. Method overriding example before adding opaque predicate

```
void overloadingWithOpaquePredicate {
  SuperClass s = null;
  int x = (int)(Math.random()*99);
  if ( x*x*(x+1)*(x+1) % 4 == 0 ) // Always true
    s = new SuperClass(); // Original class
  if ( (2*x+1) % 2 == 0 ) // Always false
    s = new SuperClass1(); // Generated class
  if ( (x*x+x+7) % 81 == 0 ) // Always false
    s = new SuperClass2(); // Generated class
  s.m(10);
}
```

Fig. 13. Method overriding example after adding opaque predicates

Fig. 12와 Fig. 13은 각각 메소드 오버라이딩에 불분명 술어를 적용하기 전과 후의 예제이다. Fig. 12에서는 '// OVERRIDING OPAQUE' 선택자를 이용하여 원하는 영역을 지정하였다. Fig. 13에서의 5번째 라인의 소스가 원본이며 그 이외의 소스는 불분명 술어 생성기에 의해서 자동으로 생성된 소스이다. 따라서 실제로 사용되는 SuperClass의 메소드 m을 제외한 SuperClass1과 SuperClass2에 정의된 메소드 m에 워터마크 정보를 삽입할 수 있다.

4. 시스템 구현 및 평가

4.1 시스템 구현

메소드 생성 기반 워터마킹 시스템의 전체 시스템 구조는 Fig. 14와 같다. 시스템 동작은 크게 두 단계로 진행된다. 1단계에서는 메소드 오버로딩과 오버라이딩 생성 모듈이 동작한다. 오버로딩 생성 모듈은 원본 메소드 매개변수를 분석하여 오버로딩 메소드를 생성하고, 오버라이딩 생성 모듈은 클래스 상속 관계를 분석하여 오버라이딩 클래스와 메소드를 생성한다. 2단계에서는 1단계의 결과물에 불분명 술어를 이용한 메소드 호출문을 생성한다. 불분명 술어 리스트로부터 추출된 불분명 술어를 오버로딩 또는 오버라이딩 메소드 호출문을 생성할 때 사용한다.

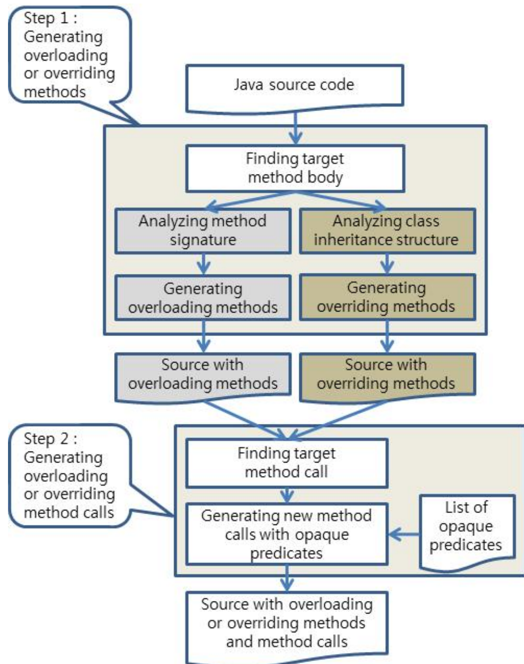


Fig. 14. Structure of watermarking system

메소드 생성 기반 워터마킹 시스템 구현을 위해서 Java 언어를 사용하였고 Eclipse 4.2.2 도구를 사용하여 Windows 7 환경에서 개발되었다.

4.2 실험 및 평가

워터마킹 시스템을 평가하기 위한 예제로서 안드로이드 SDK에 포함된 샘플 프로젝트 중에서 'JetBoy'를 사용하였다[20]. 성능 측정을 위해서 사용된 안드로이드 스마트폰은 LG X400, 플랫폼 버전은 Oreo(8.1.0)이다.

Table 2. Changes of source and apk file size according to method overloading numbers

Overloading method numbers	File size(bytes)		Ratio
Original	source	56,997	-
	apk	1,559,729	
50	source	88,232	1.548
	apk	1,566,209	1.004
100	source	146,539	2.571
	apk	1,577,293	1.011
150	source	240,590	4.221
	apk	1,594,021	1.022
200	source	369,355	6.480
	apk	1,617,257	1.037

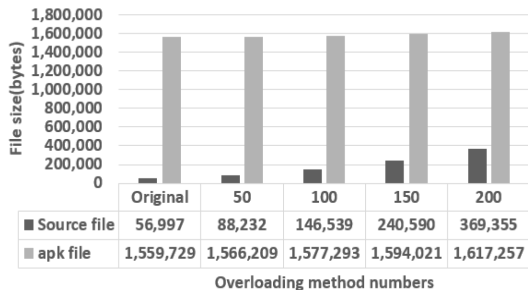


Fig. 15. Changes of source and apk file size according to method overloading numbers

Table 2와 Fig. 15는 오버로딩 메소드 생성에 의해서 증가된 자바 소스 파일과 릴리즈 모드로 빌드된 apk 파일의 크기이다. 워터마킹 시스템에 의해서 생성된 오버로딩 메소드 개수가 증가할수록 소스 파일의 크기와 apk 파일의 크기가 증가한다. 오버로딩의 특성상 메소드 생성 개수가 늘어날수록 메소드 매개변수 숫자가 늘어나기 때문에 소스 파일의 크기가 급격히 증가하는 문제가 발생한다. 그러나 릴리즈 모드로 빌드된 apk 파일은 원본과 비교했을 때 크지 않았다. 예를 들어, 200개의 오버로딩 메소드가 생성된 경우에도 소스 파일의 크기는 6배 이상 증가하였으나 apk파일은 1.037배에 머물렀다.

Table 3. Changes of source and apk file size according to method overriding numbers

Overriding method numbers	File size(bytes)		Ratio
Original	source	56,997	–
	apk	1,559,729	
50	source	71,488	1.254
	apk	1,562,777	1.002
100	source	84,994	1.491
	apk	1,565,237	1.004
150	source	98,694	1.732
	apk	1,567,533	1.005
200	source	112,392	1.972
	apk	1,569,689	1.006

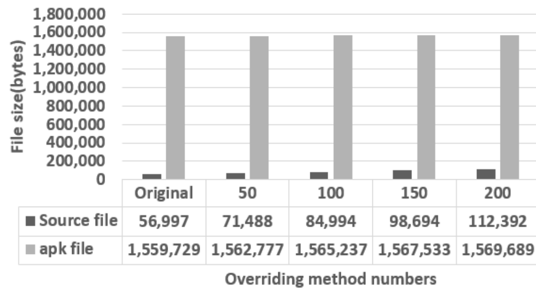


Fig. 16. Changes of source and apk file size according to method overriding numbers

Table 3과 Fig. 16은 오버라이딩 메소드 생성에 의한 파일 사이즈 측정 결과이다. 오버라이딩 메소드 개수가 증가할수록 소스 파일의 크기와 apk 파일의 크기가 증가했으나 그 비율은 원본과 비교했을 때 크지 않았다. 예를 들어, 200개의 오버라이딩 메소드가 생성된 경우에도 소스 파일의 크기는 1.972배였으며 apk파일은 1.006배에 머물렀다.

이 실험을 통해서 오버로딩에 의한 메소드 생성보다 오버라이딩에 의한 메소드 생성이 파일 크기 오버헤드가 적다는 것을 파악할 수 있다. 오버라이딩의 경우에는 클래스 상속 관계만 추가되고 매개변수는 변화가 없기 때문에 오버라이딩에 의해서 메소드가 생성될 때 원본 메소드보다 복잡해질 이유가 없다. 그러나 오버로딩에 의해서 생성되는 메소드들은 반드시 매개변수의 형식이나 개수가 모두 달라야 하므로 매개변수의 개수가 지속적으로 늘어나기 때문에 소스 파일 크기가 오버라이딩에 비해서 크게 증가하였다. 따라서 소스 파일이나 apk 파일의 크기 증가에 민감한 상황이라면 오버라이딩 메소드 생성 기법을 적용하는 것이 바람직할 것이다.

Table 4. Runtime overheads according to opaque predicate numbers

Opaque predicate numbers	times(nano sec.)
50	13
100	25
150	38
200	49

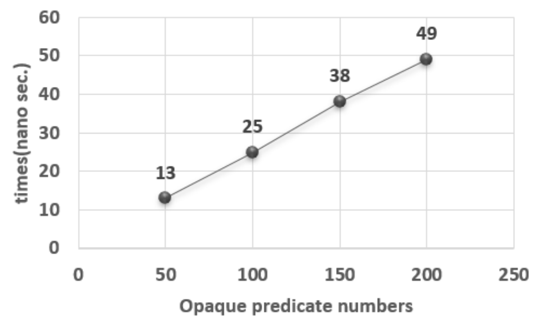


Fig. 17. Runtime overheads according to opaque predicate numbers

Table 4와 Fig. 17은 불분명 술어의 개수에 의한 실행 시간 오버헤드를 측정한 결과이다. 오버로딩과 오버라이딩에 의해서 메소드가 추가되더라도 불분명 술어의 조건식의 결과가 거짓이므로 실제로 실행되지 않는다. 따라서 실행시간 오버헤드는 오버로딩이나 오버라이딩 메소드의 실행 시간이 아니며 단지 불분명 술어의 조건식 실행 시간이 추가될 뿐이다.

시간 측정을 위해서 currentTimeMillis() 메소드를 사용하였는데 이 메소드는 밀리초 단위로만 측정 가능하므로 1,000,000회 반복 실행한 후 나노초 단위로 환산하였다. 그 결과 불분명 술어의 개수에 비례하여 실행 시간이 증가하였는데 200개의 불분명 술어가 추가되더라도 증가되는 시간은 49나노초에 불과했기 때문에 실행 성능 저하는 무시할 수 있을 수준이라고 판단된다.

5. 결론 및 향후 과제

본 논문에서는 코드 도용 방지를 위해서 메소드 생성 기법을 활용한 워터마킹을 제안하였고 시스템을 구현하였다. 워터마크로 인한 오버헤드를 평가한 결과, 실행 파일 크기의 증가와 실행 속도의 저하가 크지 않다는 것을 확인하였다. 본 논문에서 제안하는 메소드 생성 기반 워터마킹 기법의 특징은 다음과 같다. 첫째, 저작자 정보가 포함된 워터마크가 여러 개의 메소드에 다수 복제되어 포함될 수 있기 때문에 저작권자의 워터마크를 모두 찾아서 삭제하는 것이 쉽지 않다. 둘째, 오버로딩 또는 오버라이딩된 메소드가 원본 코드가 유사하게 구성되었기 때문에 원본을 확인하는데 시간이 오래 걸린다. 셋째, 참 또는 거짓을 판별하기 어려운 불분명 술어를 사용했기 때문에 자동 분석 도구에 의해서 워터마크를 무력화하는 것은 예방할 수 있다. 이러한 이유로 본 논문에서 제안하는 워터마킹 기법이 저작권 정보를 안전하게 유지할 수 있을 것으로 기대한다.

향후 연구 과제는 다음과 같다. 현재 개발된 워터마킹 시스템은 워터마크 삽입을 위한 공간을 확보해주는 역할을 하기 때문에 워터마크의 삽입과 추출 과정은 개발자가 직접 수행해야한다. 따라서 워터마크의 삽입과 추출도 자동화하는 작업이 필요하다. 추가로 워터마킹을 시스템을 초보자도 쉽게 사용할 수 있도록 편의를 제공하는 사용자 인터페이스의 개발이 필요하다. 워터마킹의 난이도를 설정할 경우에 그에 따르는 오버헤드 정보를 미리 제공한다면 사용자가 적정 수준의 워터마킹을 선택하는데 도움이 될 것이다.

REFERENCES

[1] Asia economy. (2017). *Share of adults who own smartphones by country in 2018*. http://www.asiae.co.kr/news/view.htm?idxno=2017101707023014904.

[2] Money week. (2016). *Smartphone holding rate in Korea*. http://www.moneyweek.co.kr/news/mwView.php?no=2016022408518055090.

[3] Joins economy. (2017). *Piracy of smartphone application*. https://news.joins.com/article/21799946.

[4] Keyeslabs. (2018). *A Global Piracy Heat Map*. http://keyeslabs.com/joomla/projects/auto-app-licensing/152-a-global-piracy-heat-map.

[5] Kinews. (2015). *Kim Gisa and T-map copyright dispute*. http://www.ittoday.co.kr/news/articleView.html?idxno=66263.

[6] CNB news. (2015). *The watermark of T-map*. http://www.cnbnews.com/news/article.html?no=309110.

[7] Financial news. (2016). *Cymera and Analog Filter copyright dispute*. http://www.fnnews.com/news/201605091205212639.

[8] Huffington post. (2016). *SK Communications 'Cyamera' and 'Analog Filter' copyright dispute*. https://www.huffingtonpost.kr/2016/05/11/story_n_9903264.html.

[9] D. Kim, H. Jang & S. Cho. (2009). Low-level Language based Static Watermarking for Intellectual Property Protection of Mobile Software, *Proc. of KIISE, 36(1-D)*, 55-60.

[10] C. Jun, J. Jung, B. Kim, J. Jang, Y. Cho & J. Hong. (2011). A new approach to software watermarking based on calling convention, *Proc. of KIISE, 38(2-C)*, 156-159.

[11] S. A. Moskowitz & M. Cooperman. (1998). Method for stega-cipher protection of computer code, *United States Patent 5745569A*.

[12] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn & M. Stepp. (2004). Dynamic Path-Based Software Watermaking, *Proc. of ACM SIGPLAN Conf. on PLDI*. 107-118.

[13] Y. Wang, D. Gong, B. Lu, F. Xiang & F. Liu.. (2018). Exception Handling-Based Dynamic Software Watermarking. *Proc. of IEEE Access 6*, 8882-8889.

[14] Korea copyright commission. (2010). *Codejam service*. http://www.ddaily.co.kr/news/article.html?no=72525.

[15] H. Park. (2018). Java Class Watermarking Technique based on Object-Oriented Characteristics, *Proc. of 12th KIISE and KBS Symp*, 132-135.

[16] K. Arnold, J. Gosling & D. Holmes. (2005). *The Java Programming Language, Fourth Edition*. Addison Wesley Professional.

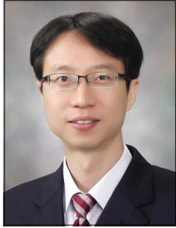
[17] C. Collberg, C. Thomborson & Douglas Low. (1998). Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *Proc. of 25th ACM IGPLAN-SIGACT Symp. on POPL*, 184-196.

[18] J. Ming, D. Xu, L. Wang & D. Wu. (2015). LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. *Proc. of 22nd ACM SIGSAC Conf. on CCS*, 757-768.

[19] G. Myles & C. Collberg. (2006). Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electron Commerce Research, 6(2-2)*, 155-171.

[20] Android Open Source Sample Project. (2014). *Jet Boy*, https://github.com/Miserlou/Android-SDK-Samples/tree/master/JetBoy.

박 희 완(Park, Heewan) [정회원]



- 1997년 2월 : 동국대학교 컴퓨터공학과(공학사)
- 1999년 2월 : KAIST 전산학과(공학석사)
- 2010년 1월 : KAIST 전산학과(공학박사)
- 2004년 3월~2007년 2월 : 삼성전자 무선사업부 책임연구원
- 2010년 2월~2011년 8월 : ETRI 부설연구소 선임연구원
- 2011년 9월~현재 : 한라대학교 정보통신소프트웨어학과 조교수
- 관심분야 : 소프트웨어 난독화, 정적 및 동적 분석
- E-Mail : heewanpark@halla.ac.kr