Journal of The Korea Society of Computer and Information
Vol. 18, No. 10, October 2013
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2013.18.10.129

대입문 병합을 이용한 효율적인 자바 난독화 기법

이 경 호*, 박 희 완*

An Effective Java Obfuscation Technique Using Assignment Statements Merging

Kyong-Ho Lee*, Hee-Wan Park*

요 약

자바 바이트코드는 타겟 머신이 아닌 자바 가상머신 상에서 동작된다. 이러한 자바 바이트코드는 바이너리 코드보다 고수준 표현을 사용하고 있어서 대부분의 자바 바이트 코드는 다시 자바 소스 파일로 바꿀 수도 있다. 난독화란 기본적으로 코드를 이해하기 어렵게 만드는 기술을 의미한다. 자바 코드의 보호를 위해서는 난독화가 사용될 수 있다. 그러나 프로그램을 복잡하게 만드는 난독화 기법은 대부분 프로그램의 크기를 증가시키고 실행 속도 또한 느려지게 만드는 원인이 된다.

본 논문에서는 대입문 병합 기법을 이용한 효율적인 자바 난독화 기법을 새롭게 제안한다. 대입문 병합 기법이 적용되면 대입문에 부작용(side effects)이 추가되어 대입문을 이해하기 어렵게 된다. 추가적으로 바이트코드의 크기가 줄어드는 장점도 있다.

▸ Keywords : 자바 바이트코드, 코드 난독화, 부작용, 대입문 병합, 소프트웨어 보호

Abstract

Java bytecodes are executed not on target machine but on the Java virtual machines. Since this bytecodes use a higher level representation than binary code, it is possible to decompile most bytecodes back to Java source. Obfuscation is the technique of obscuring code and it makes program difficult to understand. However, most of the obfuscation techniques make the code size and the performance of obfuscated program bigger and slower than original program.

In this paper, we proposed an effective Java obfuscation techniques using assignment statements merging that make the source program difficult to understand. The basic approach is to merge assignments statements to append side effects of statement. An additional benefit is that

•제1저자 : 이경호 •교신저자 : 박희완
•투고일 : 2013. 9. 3, 심사일 : 2013. 9. 25, 게재확정일 : 2013. 10. 5.
* 한라대학교 정보통신방송공학부(School of Information & Communication Engineering, Halla University)

the size of the bytecode is reduced.

▸ Keywords : Java bytecode, Code obfuscation, Side effects, Assignment Statements merging, Software Protection

I. 서 론

자바 프로그램은 일반적으로 클래스 파일 형태로 배포되고, 이 클래스 파일은 하드웨어 독립적인 자바 가상 기계(Java virtual machine)에서 실행되어야 하기 때문에 원본 자바 소스 프로그램의 심볼릭 정보를 그대로 유지하고 있다. 따라서 자바 클래스 파일은 기계어로 번역되는 다른 언어들과 비교하여 역공학 기술을 적용하기 쉽다. 단순한 역컴파일러를 시작으로 수많은 자바 클래스 분석 도구들이 개발되어왔고, 이러한 도구를 이용하면 자바 클래스 파일로부터 원본 소스와 거의 동일한 소스 코드를 얻어낼 수 있다[1-7].

그 결과로 자바 프로그램들은 악의적인 역공학자들에 의해서 크게는 프로그램의 전체, 작게는 프로그램에 사용된 자료구조나 알고리즘이 도용당할 수 있는 위험이 있다. 이 문제에 대처하기 위해서 난독화라는 기법이 제안되었다[1,2]. 난독화란 프로그램의 기능을 그대로 유지한 채 소스코드를 해석하기 어려운 형태로 변환하는 방법이다. 난독화 기법을 적용하면 프로그램이 역공학 도구들에 의해서 쉽게 분석되는 것을 방지할 수 있다.

일반적인 난독화 기법은 원본 프로그램에 부가적인 코드를 추가하여 복잡도를 증가시키는 방법이다. 프로그램의 복잡도가 증가할수록 프로그램에 대한 이해도가 떨어지기 때문에 역공학에 의한 프로그램 분석이 어려워진다. 그러나 난독화 작업으로 인해서 추가되는 부가적인 코드 때문에 프로그램 사이즈가 증가하고 실행 속도가 떨어지는 단점이 생긴다. 이 단점은 프로그램 사이즈와 실행 성능이 중요하게 고려되어야 하는 모바일 환경에서 큰 문제점이 된다[7].

프로그램의 이해도를 떨어뜨리고 분석을 어렵게 만드는 작용을 하는 측면에서 프로그램의 부작용(side-effect)은 난독화의 기능과 유사하다. 프로그램의 부작용이란 명령문을 수행할 때 프로그램 실행 상태에 여러 가지 변화가 생기는 것을 의미한다. 예를 들면, 한 명령문에 의해서 여러 가지 변수값이 동시에 바뀌는 경우도 부작용의 예이다. 소프트웨어 공학적인 측면에서는 가급적 부작용이 없는 프로그램을 디자인하기 위해서 노력하고, 이미 프로그램에 존재하는 부작용을 없애는 프로그램 변환 기법(side-effect removal transformation)도 연구되었다. 부작용을 인위적으로 없애는 변환을 하면 프로그램의 이해도가 높아지는 장점이 있지만 프로그램의 사이즈가 커지는 단점이 있다.[8,9]

난독화의 목적은 역공학에 의한 분석을 어렵게 하기 위한 방법으로 프로그램의 복잡도를 높이는 것에 있다. 즉, 난독화에 프로그램의 부작용을 이용한다면 프로그램 분석을 어렵게 만들면서 프로그램의 사이즈도 줄일 수 있는 부가적인 효과를 얻을 수 있다.

본 논문에서는 부작용에 관한 연구를 난독화 분야에 적용시킨 새로운 난독화 기법을 제안한다. 이 난독화 기법은, 두 개 이상의 대입문들을 병합하는 과정에서 생기는 부작용을 이용한 난독화 기법이다. 즉, 원본 프로그램에서 부작용으로 변환할 수 있는 형태의 프로그래밍 패턴들을 발견하고, 이것들을 부작용으로 변환시켜서 난독화를 시도하는 기법이다. 인위적으로 부작용을 만들어내면 프로그램의 이해도를 떨어뜨리는 난독화의 장점과 함께, 부작용에 의한 함축된 표현으로 인해서 프로그램의 사이즈를 줄일 수 있는 효과도 얻을 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대한 소개로 난독화 기법과 프로그램 부작용, 컴파일러 최적화와 관련된 기존의 연구들을 정리한다. 3장에서는 본 논문에서 새롭게 제안하는 대입문 병합을 이용한 난독화 기법에 대해서 알아보고, 4장에서는 실험 및 평가를 한다. 마지막 5장에서는 결론을 맺고 보완해야 할 점과 향후 연구 과제에 대해서 논의한다.

II. 관련 연구

본 논문에 대한 관련연구로서 프로그램 난독화 기법의 분류와 문제점, 평가기준을 알아본다. 그리고 프로그램의 부작용과 컴파일러 최적화에 연관된 대입문 병합에 대해서 알아본다.

2.1 프로그램 난독화 기법의 개요

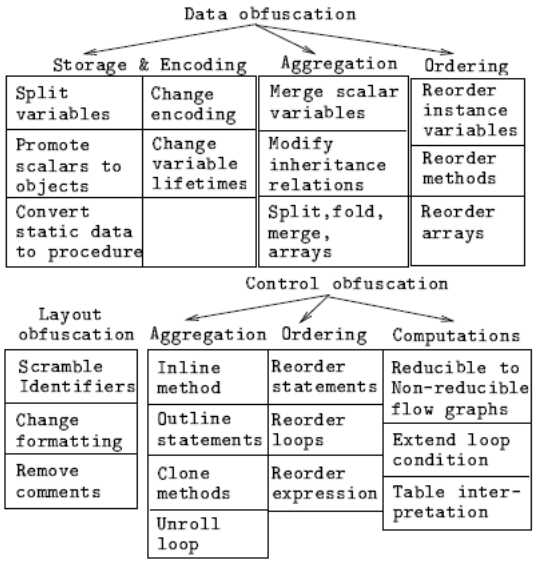


그림 1. 프로그램 난독화 기법의 분류
Fig. 1. Classification of program obfuscations

그림 1은 난독화 기법의 분류에 대한 내용이다[4]. 난독화 기법은 변환 대상에 따라서 크게 네 가지로 구분된다. 가장 단순한 변환 대상은 소스 코드의 포맷이나 변수의 이름 등을 대상으로 하는 레이아웃 난독화(layout obfuscation)이다. 그리고 프로그램에서 사용된 자료구조가 대상이 되는 데이터 난독화(data obfuscation) 기법이 있고, 프로그램의 제어 구조를 바꾸는 제어 난독화(control obfuscation) 기법 등이 있다.

난독화 기법들은 간단한 레이아웃 변환 기법[5]을 제외하면 원본 프로그램에 부가적인 난독화 코드를 삽입한다. 따라서 복잡한 변환을 반복해서 적용할수록 프로그램의 사이즈가 커지고 실행속도가 저하된다. 이것은 난독화 기법을 적용할 때 프로그램을 보호하는 대가로 감당해야 할 부분이다.

난독화 기법의 평가는 세 가지 측면에서 이루어진다. 첫째, 프로그램이 얼마나 복잡(potency)하게 변환되었는가에 대한 평가와, 둘째, 난독화 해석 도구로부터 얼마나 잘 견딜 수 있는지에 대한 강인도(resilience) 평가와, 셋째, 프로그램 사이즈의 증가와 실행에서 생기는 속도 저하와 같은 오버헤드에 대한 비용(cost)의 평가이다[4].

2.2 대입문 병합과 프로그램의 부작용

프로그램에서 부작용(side-effect)이란 표현식들의 평가에 의해서 상태 변화가 일어나는 것이라고 정의된다[8,9]. 대입문은 변수에 값이 정의되는 명령문이기 때문에, 적어도 하나의 부작용이 발생한다. 따라서 프로그램을 이루는 모든 대입문이 단 하나만의 부작용만을 가질 때 이 프로그램을 부작용 자유(side-effect free)프로그램이라고 한다[8,9].

부작용 자유 프로그램은 가독성(readability)이 높아서 사람이 이해하기 쉽고, 또한 도구에 의한 분석이 용이하기 때문에 프로그램 병렬화, 슬라이싱과 같은 각종 변환 기법을 쉽게 적용할 수 있다[10,11]. 따라서 부작용에 대한 분석 및 부작용을 없애기 위한 변환 기법에 대한 연구가 있었다[8,9].

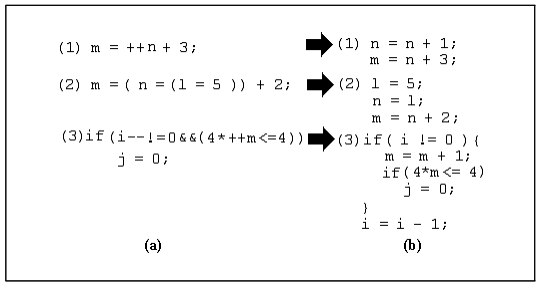


그림 2. (a) 부작용(side-effect)이 포함되어 있는 코드, (b) 부작용이 없도록 변환된 코드
Fig. 2. (a) Original code with side-effect, (b) Transformed code without side-effect

그림 2의 (a)와 (b)는 각각 부작용이 포함된 대입문과, 부작용이 없도록 변환된 대입문을 보여준다. 그림의 예제를 통해서 알 수 있듯이 부작용이 포함된 대입문은 한 대입문에서 두 가지 이상의 값이 평가(evaluation)되기 때문에 함축된 표현식 형태로 나타나고, 따라서 대입문의 의미를 쉽게 파악하거나 이해하기 어렵다.

(b)와 같이 부작용이 없도록 변환된 프로그램은 한 대입문에서 한 변수의 값만 평가되기 때문에 각각의 대입문을 이해하기 쉽다. 그러나 프로그램의 사이즈가 증가하게 된다는 단점이 있다.

일반적인 프로그램들은 최대한 부작용을 많이 포함한 형태도 아니고, 부작용 자유 형태도 아닌 중간적인 형태를 유지한다. (a)의 (1)에서 보여주는 '++' 또는 '--'와 같은 증감 연산자와 대입문의 결합으로 나타나는 부작용은 관습화된 프로그래밍 패턴이기 때문에 프로그램에서 빈번히 나타나지만, (2)에서 보여주는 두 대입문이 한 대입문으로 연결된 형태와 (3)에서 보여주는 'if' 명령문의 특성을 이용한 부작용은 프로그램의 부분적인 가독성을 떨어뜨리고 전체적인 이해도를 저하시키기 때문에 특수한 경우 외에는 사용하지 않는다[8,9,12].

본 논문에서 새롭게 제안하는 대입문 병합을 이용한 난독화 기법이란, 두 개 이상의 대입문들을 병합하는 과정을 통해서 생기는 부작용을 이용한 난독화 변환 기법이다. 즉, 원본 프로그램에서 부작용으로 변환할 수 있는 형태의 프로그래밍 패턴들을 발견하고, 이것들을 부작용으로 변환시켜서 난독화를 시도하는 기법이다. 인위적으로 부작용을 만들어내면 프로그램의 이해도를 떨어뜨리는 난독화의 장점과 함께, 부작용의 함축된 표현으로 인해서 프로그램의 사이즈를 줄일 수 있는 효과도 얻을 수 있다.

2.3 대입문 병합과 컴파일러 최적화

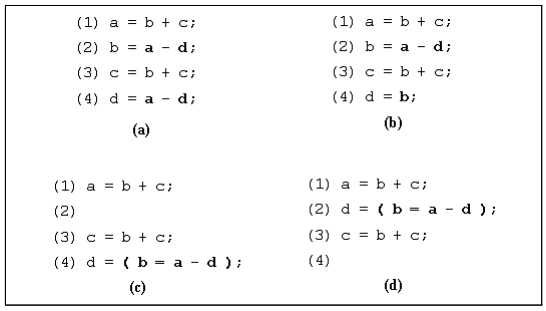


그림 3. (a) 원본 코드, (b) 공통 부분식 제거로 변환된 코드, (c),(d) 두 대입문을 병합한 코드
Fig. 3. (a) Original code (b) Transformed code by common sub-expression elimination, (c),(d) Transformed code by merging two assignment statements

그림 3에 포함되어 있는 예제 (a)와 (b)는 컴파일러 최적화 단계에서 공통 부분식 제거 기법(common sub-expression elimination)을 설명하는 예제 프로그램이다[13]. 예제 (a)의 2번과 4번 대입문은 'a-d'라는 공통 부분식을 갖고 있기 때문에 4번 대입문은 예제 (b)의 4번 대입문과 같이 'b'로 변경될 수 있다. 예제 (a)의 1번과 3번 대입문도 'b+c'라는 공통 부분식을 갖고 있지만 두 대입문의 사이에 위치한 2번 대입문에서 공통 부분식에 포함되는 변수인 'b'의 값이 재 정의되기 때문에 단순하게 공통 부분식 제거 기법을 적용할 수 없다. 공통 부분식 제거 기법을 적용할 때는 반드시 데이터 종속성(data dependency)을 고려해야만 한다.

공통 부분식이 제거된 예제 (b)에서 한 단계 더 최적화를 할 수 있는 여지가 있다. 한 대입문 안에서 두 개 이상의 대입문이 포함된 형태가 가능하기 때문에 2번과 4번 대입문을 병합(assignment merging)하여 연속된 두 개의 대입문을 포함하는 하나의 대입문으로 만들 수 있다. 이 결과로 예제 (c)와 예제 (d)가 만들어진다. 여기에서 주의해야 할 점은 병합될 수 있는 두 개의 대입문 사이의 데이터 종속성을 엄밀하게 검증해야만 한다는 것이다. 병합 가능한 대입문들을 선택하고 병합하는 알고리즘에 대한 내용은 3장에서 상세하게 다룬다.

III. 대입문 병합을 이용한 난독화 기법

3.1 대입문 병합을 이용한 난독화 기법의 개요

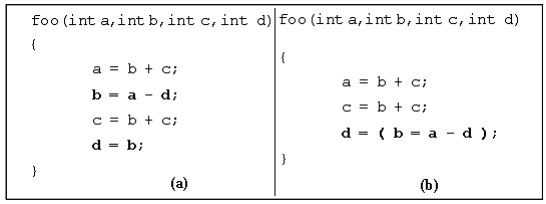


그림 4. (a) 공통 부분식 제거 변환을 한 프로그램, (b) 두 개의 대입문을 병합하여 변환한 프로그램
Fig. 4. (a) Transformed program by common sub-expression elimination, (b) Transformed program by merging two assignment statements

그림 4의 예제 (a)는 컴파일러 최적화 단계에서 사용되는 공통 부분식 제거 변환을 거친 프로그램이고, 예제 (b)는 대

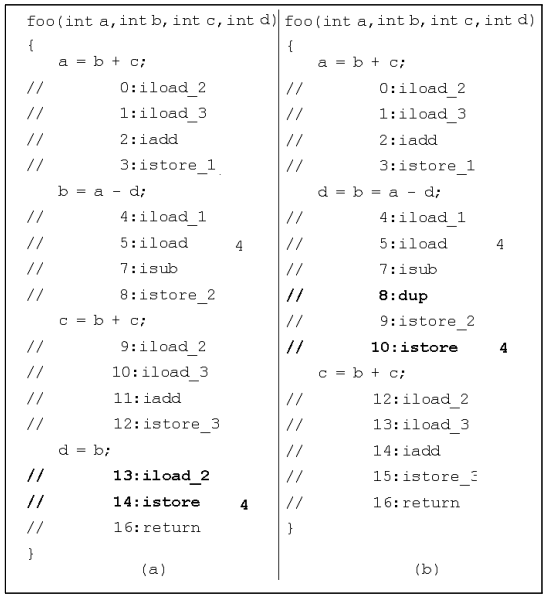


그림 5. (a) 지역 변수에 대한 공통 표현식 제거 최적화 결과, (b) 지역변수에 대하여 부작용을 이용하여 대입문을 병합한 결과
Fig. 5. (a) Transformed result by common sub-expression elimination of local variables, (b) Transformed result by merging assignment statements using side-effect of local variables

입문을 병합하여 하나의 대입문으로 변환한 프로그램이다. 예제 (a)는 하나의 대입문에 많아야 하나의 값이 결정되는 형태이기 때문에 부작용 자유 프로그램이고, (b)는 'b'와 'd'의 값이 하나의 대입문에서 함께 결정되기 때문에 부작용 자유 프로그램이 아니다.

그림 5는 그림 4의 예제를 자바 컴파일러로 컴파일한 결과로 생성된 자바 바이트코드를 보여준다. 예제(a)에서는 변수 'd'의 값을 결정하기 위해서 'b' 값을 'load'해서 'd' 값에 'store'하는 방식을 사용했다. 그러나 예제(b)에서는 'b'값에 저장될 값을 스택에서 복제하는 'dup' JVM대입문을 활용하여 'load'를 사용하지 않고 'd'에 복제된 값을 다시 'store'하는 방식을 사용했다. (a)와 (b)의 두 메소드는 모두 16바이트를 차지하고 있기 때문에 바이트코드를 줄이는 효과는 얻지 못했지만 메모리에서 스택에 값을 저장하는 "iload"명령보다 스택의 값을 복제하는 'dup' 대입문이 빠르게 수행될 수 있다는 장점이 있다[13,14].

```
void foo1(...)                          void foo2(...)
{  ...                                  {  ...
   b = a - d;                              d = b = a - d;
// 10:getstatic #4 <Field int a>        // 10:getstatic #4 <Field int a>
// 13:getstatic #5 <Field int d>        // 13:getstatic #5 <Field int d>
// 16:isub                              // 16:isub
// 17:putstatic #2 <Field int b>        // 17:dup
   ...                                  // 18:putstatic #2 <Field int b>
   d = b;                               // 21:putstatic #5 <Field int d>
// 30:getstatic #2 <Field int b>           ...
// 33:putstatic #5 <Field int d>        // 34:return
// 36:return                            }
}                 (a)                                    (b)

void foo3(...)                          void foo4(...)
{  ...                                  {  ...
   b = a - d;                              d = b = a - d;
// 13:aload_0                           // 13:aload_0
// 14:aload_0                           // 14:aload_0
// 15:getfield #4 <Field int a>         // 15:aload_0
// 18:aload_0                           // 16:getfield #4 <Field int a>
// 19:getfield #5 <Field int d>         // 19:aload_0
// 22:isub                              // 20:getfield #5 <Field int d>
// 23:putfield #2 <Field int b>         // 23:isub
   ...                                  // 24:dup_x1
   d = b;                               // 25:putfield #2 <Field int b>
// 39:aload_0                           // 28:putfield #5 <Field int d>
// 40:aload_0                              ...
// 41:getfield #2 <Field int b>         // 44:return
// 44:putfield #5 <Field int d>         }
// 47:return
}                 (c)                                    (d)
```

그림 6. (a),(b) 정적 변수에 대하여 대입문 병합 전과 후의 바이트코드, (c),(d) 인스턴스 변수에 대한 대입문 병합 전과 후의 바이트코드

Fig. 6. (a),(b) Bytecodes before and after merging assignment statement, (c),(d) Bytecodes before and after merging assignment statements of instance variables

그림 6은 그림 5의 예제를 정적 변수(static variable)와 인스턴스 변수(instance variable)에 대해서 적용한 결과이다. (a)와(b)는 정적 변수로 바꾼 예제이고, (c)와 (d)는 인스턴스 변수로 바꾼 예제이다.

정적 변수를 이용하는 경우에는 변수값을 읽거나 저장할 때 각각 'getstatic' 과 'putstatic' JVM 대입문을 사용해야 한다. 따라서 예제(a)에서 사용된 'getstatic' 대입문이 예제(b)에서는 'dup'로 바뀐 것을 확인할 수 있다. JVM에서 'getstatic'을 이용할 때 3byte를 사용하지만 'dup'는 1byte를 사용하기 때문에 2byte를 절약하는 장점이 있다.

예제 (c)와 (d)는 인스턴스 변수를 사용한 예제이다. 인스턴스 변수의 값을 읽어오거나 저장할 때는 먼저 객체의 참조를 'aload'를 이용하여 스택에 저장하고 그 후에 'getfield'와 'putfield' 명령문을 이용해야 한다. 그런데 (d)와 같이 대입문 병합을 이용하는 경우에는 'dup_x1' 명령어로 대치할 수 있기 때문에 (c)와 비교하면 3byte를 절약할 수 있는 장점이 있다.

자바는 객체지향 언어로서 기본적으로 클래스를 설계하고, 여러 가지 클래스로부터 객체를 생성해서 객체의 메소드 연산을 수행하는 방식으로 동작하다. 따라서 객체로부터 생성된 인스턴스 변수의 사용이 많다는 것을 고려하면 인스턴스 변수에 대한 부작용을 이용한 대입문 병합 기법으로 인해서 얻게 되는 자바 바이트코드 축소 효과가 큰 장점으로 활용될 수 있을 것으로 예상할 수 있다.

3.2 대입문과 대입문의 결합

```
S ::= E;                          E ::= PreOp I
    | if ( E ) S                      | I PostOp
    | if ( E ) S1 else S2             | E1 BinOp E2
    | while( E ) S                    | I = E
    | do S while( E )                 | I
    | for( E1; E2; E3 ) S             | N
    | E = Method( E )                 | I[E]
    | return E                        | E1 ? E2 : E3
    | S1 S2                           | null

    PostOp   ::= ++ | --
    PreOp    ::= PostOp | !
    BinOp    ::= LogicOp | ArithOp
    ArithOp  ::= + | - | * | / | %
    LogicOp  :: && | ||
```

그림 7. 자바 언어의 축약된 문법

Fig. 7. Reduced grammar of Java language

그림 7은 자바 언어의 축약된 문법이다. 본 논문에서 제시한 자바를 위한 축약된 문법은 C언어 문법에 대한 기존 연구[5,6]를 참고로 하여 자바 언어에 맞도록 새롭게 디자인한 것이다. 알고리즘의 설명을 용이하기 위해서 축약된 형태를 기반으로 설명하지만, 이론을 실제로 적용할 때는 자바의 전체 문법을 기반으로 한다.

$$
\begin{aligned}
&DEF[\![++I]\!] = \{I\}\\
&DEF[\![--I]\!] = \{I\}\\
&DEF[\![I++]\!] = \{I\}\\
&DEF[\![I--]\!] = \{I\}\\
&DEF[\![!E]\!] = DEF[\![E]\!]\\
&DEF[\![E_1 \ AOp \ E_2]\!] = DEF[\![E_1]\!] \cup DEF[\![E_2]\!]\\
&DEF[\![E_1 \ \&\& \ E_2]\!] = DEF[\![E_1]\!] \cup DEF[\![E_2]\!]\\
&DEF[\![E_1 \ || \ E_2]\!] = DEF[\![E_1]\!] \cup DEF[\![E_2]\!]\\
&DEF[\![I = E]\!] = \{I\} \cup DEF[\![E]\!]\\
&DEF[\![I]\!] = \{\}\\
&DEF[\![N]\!] = \{\}\\
&DEF[\![I[E]]\!] = DEF[\![E]\!]\\
&DEF[\![E_1 \ ? \ E_2 \ : \ E_3]\!] = DEF[\![E_1]\!] \cup DEF[\![E_2]\!] \cup DEF[\![E_3]\!]
\end{aligned}
$$

그림 8. 표현식에 의해서 정의되는 변수들
Fig. 8. Defined variables of expressions

$$
\begin{aligned}
&REF[\![++I]\!] = \{I\}\\
&REF[\![--I]\!] = \{I\}\\
&REF[\![I++]\!] = \{I\}\\
&REF[\![I--]\!] = \{I\}\\
&REF[\![!E]\!] = REF[\![E]\!]\\
&REF[\![E_1 \ AOp \ E_2]\!] = REF[\![E_1]\!] \cup REF[\![E_2]\!]\\
&REF[\![E_1 \ \&\& \ E_2]\!] = REF[\![E_1]\!] \cup REF[\![E_2]\!]\\
&REF[\![E_1 \ || \ E_2]\!] = REF[\![E_1]\!] \cup REF[\![E_2]\!]\\
&REF[\![I = E]\!] = REF[\![E]\!]\\
&REF[\![I]\!] = \{I\}\\
&REF[\![N]\!] = \{\}\\
&REF[\![I[E]]\!] = REF[\![E]\!] \cup \{I\}\\
&REF[\![E_1 \ ? \ E_2 \ : \ E_3]\!] = REF[\![E_1]\!] \cup REF[\![E_2]\!] \cup REF[\![E_3]\!]
\end{aligned}
$$

그림 9. 표현식에 의해서 참조되는 변수들
Fig. 9. Referenced variables of expressions

그림 8과 그림 9는 각각 자바 표현식에 의해서 정의되는 변수들과 참조되는 변수들을 나타낸다. 대입문과 대입문을 병합하는 과정에서 이 정보들을 사용하여 규칙의 적용 여부를 판단한다.

본 논문에서는 DEF와 REF를 사용해서 자바 프로그램에 적용할 수 있는 규칙을 새롭게 디자인하였다. 이러한 규칙들은 자바 프로그램을 안전하게 난독화하는데 사용된다.

$$
\frac{i_1 \neq i_2,\ i_1 \notin DEF(e_2),\ i_2 \notin DEF(e_1),\ i_2 \notin REF(e_1),\ i_1 \notin REF(e_2)}{[\![i_1 = e_1;\ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_2;\ i_1 = e_1;]\!]}
$$

그림 10. 대입문 교환 규칙
Fig. 10. Assignment statement exchanging rule

그림 10은 대입문 교환 규칙을 나타낸다. 대입문 교환 규칙이란 인접한 두 대입문의 위치를 서로 교환하는 것을 의미한다. 원본 프로그램의 의미를 변화시키지 않고 안전하게 두 대입문이 서로 교환되기 위해서는 먼저 대입문 사이에 종속성을 고려해야 한다. 두 대입문이 서로 다른 변수의 값을 정의하고, 각각의 대입문에서 정의되거나 참조하는 변수의 집합이 다른 대입문에 의해서 참조되거나 정의되지 않을 경우에 두 대입문의 위치를 교환할 수 있다.

$$
\frac{i_1 \in REF(e_2),\ e_3 = SUB[e_2,\ i_1,\ (i_1 = e_1)]}{[\![i_1 = e_1;\ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3;]\!]}
$$

※SUB(e1, x, e2): e1에서 첫 번째로 나타난 x를 e2로 대치시킴

그림 11. 대입문 병합(Merging) 규칙
Fig. 11. Assignment statement merging rule

그림 11은 대입문 병합 규칙을 나타낸다. 두 개의 대입문이 하나의 대입문으로 병합되기 위해서는 앞서는 대입문에서 정의되는 변수를 뒤따르는 대입문에서 참조해야 한다. 두 대입문은 'SUB'연산을 이용해서 병합할 수 있는데, 여기서 SUB[e1, x, e2]는 'e1'에 나타난 첫 번째 'x'를 'e2'로 대치하라는 의미가 있다. 'e1'표현식에서 나타나는 모든 'x'가 아니고 첫 번째 'x'만을 대치해야하는 이유는 그림 12의 예제를 이용해서 설명할 수 있다.

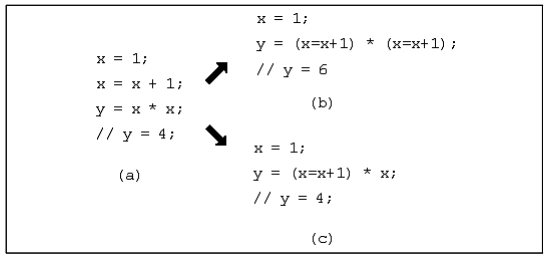


그림 12. (a) 예제 코드 (b) 잘못된 병합의 결과, (c) 올바른 병합의 결과
Fig. 12. (a) sample code, (b) incorrect result of merging, (c) correct result of merging

자바 컴파일러는 표현식을 이루는 항(term)들을 가장 왼쪽부터 하나씩 평가하기 때문에 첫 번째로 대치된 곳 이후로는 대입문에 의해서 정의된 값을 이용한다. 따라서 그림11의 (b)와 (c)의 곱하기 연산에서 두 번째 피연산자 'x'는 1이 증가된 값을 사용하기 때문에 (c)의 형태로 병합되어야만 원본 프로그램인 (a)와 같은 결과를 얻을 수 있다.

대입문 교환규칙과 병합 규칙이 적용되는 간략한 과정은 다음과 같다. 먼저, 병합하고자 하는 두 개의 대입문을 선택하고, 두 대입문을 서로 인접하게 배치하기 위해서 대입문 교환 규칙을 차례로 적용한다. 그리고 인접하게 배치된 두 대입

문에 대해서 대입문 병합 규칙을 적용하여 두 개의 대입문을 하나의 대입문으로 병합한다.

```
algorithm  Simplex_Assignment_Merging
 input Basic Block of Original Program
 output Basic Block of Assignment Merged Program
 terms
       B : Basic Block
       L : Line Number
       S : Statement
       i : Variable
       e : Expression
begin Assignment_Merging
 1.foreach assignment statement S_i
 2. { S_i| S_i ∈B_i, L_i : i_i = e_i;}
 3. foreach assignment statement S_j
 4.  { S_j| S_j ∈B_i, L_j : i_j = e_j;, L_i < L_j, i_i ∈REF(e_j)}
 5.  while L_i + 1 ≠ L_j do
 6.   select next assignment statement S_k
 7.   {S_k| S_k ∈B_i, L_k : i_k = e_k;,L_k = L_i + 1}
 8.   if  DEF(S_i)⊄REF(S_k) and DEF(S_k)⊄REF(S_i)then
 9.    Apply_AssignmentExchangingRule(S_i , S_k)
10.   else break while-loop;
11.   endif
12.  endwhile
13.  if L_i + 1 = L_j  then
14.   Apply_AssignmentMergingRule(S_i , S_j)
15.  endif
16. endfor
17.endfor
```

그림 13. 대입문의 단방향(Simplex) 병합 알고리즘
Fig. 13. Simplex merging algorithm of assignment statements

그림 13은 병합하고자 하는 두 개의 대입문이 있을 경우에 앞의 대입문(선행 대입문)을 뒤의 대입문(후행 대입문)에 인접한 위치로 이동시켜서 병합하는 단방향 병합 알고리즘을 나타낸다. 알고리즘을 간략히 설명하면 다음과 같다. 먼저 1번째 줄과 3번째 줄에서 병합하고자 하는 두 대입문을 선택한다. 그리고 5번째 줄부터 12번째 줄까지의 반복문을 통해서 앞쪽에 선행 대입문이 후행 대입문과 인접할 때까지 대입문 교환을 반복한다. 대입문 교환의 결과로 두 대입문이 인접하게 되었다면 14번째 줄의 대입문 병합 규칙을 통해서 두 대입문이 병합된다. 만일, 데이터 종속성의 제약 때문에 대입문의 교환이 가능하지 않을 경우에는 처음에 선택했던 두 대입문이 서로 인접하지 못한 상황에서 반복문을 종료하기 때문에 14번째 줄의 대입문 병합 규칙은 적용되지 못한다.

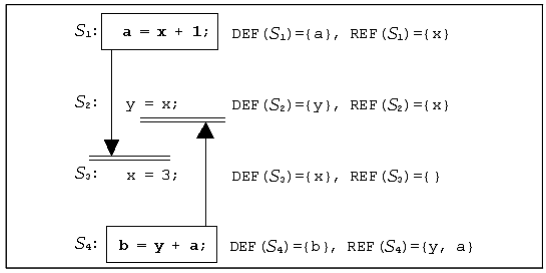


그림 14. 단방향 병합 알고리즘의 문제점
Fig. 14. Problems of simplex merging algorithm

그림 14는 단방향 대입문 병합의 문제점을 나타낸다. 대입문 'S1'과 대입문 'S4'를 병합하고자 할 경우에 'S1'에 대입문 교환 규칙을 적용하면서 'S4'와 인접하게 이동하여야 하는데 'S1'에서 참조되는 변수 집합인 REF(S1)에 속한 변수 'x'가 'S3' 대입문의 DEF(S3)에 포함된다. 따라서 데이터 종속성을 유지하면서 'S1'을 'S4'와 인접하게 이동시킬 수 없고 그 결과로 'S4'와의 병합이 불가능하다.

그런데 'S1' 대신에 'S4'가 이동한다면 데이터 종속성을 유지하면서 'S2'와 'S3' 사이까지 이동할 수 있다. 즉, 'S4'가 이동할 수 있는 위치가 'S1'과 인접하지는 않지만 'S1'과 'S4'가 서로를 향하여 이동하고, 두 대입문이 사이에 서로 공유할 수 있는 위치가 있다면 두 대입문은 병합될 수 있다.

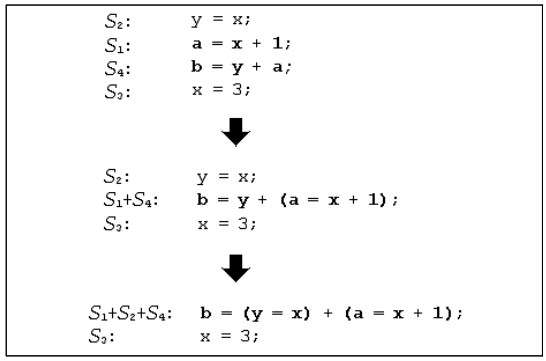


그림 15. 양방향 대입문 병합을 적용한 결과
Fig. 15. Result of applying duplex assignment statement merging

그림 15는 양방향 대입문 병합을 적용하여 'S1'과 'S4'가 'S2'와 'S3' 사이에서 병합된 결과를 보여준다. 먼저 'S1'과 'S2'가 서로 교환되었다. 그리고 'S3'와 'S4'가 교환되어서 두 대입문 'S1'과 'S4'는 인접하게 되었다. 이제 대입문 병합 규칙을 적용하여 'S1'과 'S4'가 병합된다. 이어서 'S2'도 함께 병합하는 것도 가능하다.

```
algorithm Duplex_Assignment_Merging
 input Basic Block of Original Program
 output Basic Block of Assignment Merged Program
 terms B : Basic Block, L : Line Number
       S : Statement, i : Variable, e : Expression
begin Assignment_Merging
 1.foreach assignment statement S1
 2.{ S1| S1 ∈B1, L1 : i1 = e1;}
 3. foreach assignment statement Sj
 4. { Sj| Sj ∈Bi, Lj : ij = ej;, Li < Lj, ii ∈REF(ej)}
 5. while1 Li + 1 ≠ Lj do
 6. select assignment statement Sdown
 7. { Sdown| Sdown ∈Bi, Ldown : idown = edown;,Ldown = Li + 1}
 8.  if  DEF(Si) ⊄REF(Sdown) and DEF(Sdown) ⊄REF(Si) then
 9.   Apply_AssignmentExchangingRule(Si , Sdown)
10.  else break while1-loop;
11.  endif
12. endwhile1
13. while2 Li + 1 ≠ Lj do
14. select assignment statement Sup
15. { Sup| Sup ∈Bi, Lup : iup = eup;,Lup = Lj - 1}
16.  if DEF(Sj) ⊄REF(Sup) and DEF(Sup) ⊄REF(Sj) then
17.   Apply_AssignmentExchangingRule(Sj , Sup)
18.  else break while2-loop;
19.  endif
20. endwhile2
21. if Li + 1 = Lj  then
22.  Apply_AssignmentMergingRule(Si , Sj)
23.  endif
24. endfor
25.endfor
end Assignment_Merging
```

그림 16. 대입문의 양방향(Duplex) 병합 알고리즘
Fig. 16. Duplex merging algorithm of assignment statements

그림 16은 대입문의 양방향 병합 알고리즘을 정리한 것이다. 단방향 병합 알고리즘과의 차이점은 두 개의 반복문을 이용하여 선행 대입문의 교환 규칙 적용 이후에 후행 대입문의 교환 규칙도 적용된다는 점이다. 후행 대입문의 교환 규칙은 선행 대입문의 교환 규칙이 진행된 지점까지만 이루어진다. 즉, 두 대입문이 서로 인접하면 반복문을 종료하고 두 대입문을 병합한다. 만일 선행 대입문의 교환 규칙만으로 후행 대입문과 인접하게 된다면 후행 대입문의 교환 규칙은 적용하지 않고 곧바로 대입문 병합 규칙을 적용한다.

3.3 대입문과 기타 명령문과의 병합

1) 배열 인덱스, 메소드 인자와의 병합

```
x = a + b;
y = c[ x ];        ➡  y = c[ x = a + b ];
                   (a)

x = a + b;
y = foo( x );      ➡  y = foo( x = a + b );
                   (b)
```

그림 17. (a) 대입문과 배열 인덱스의 병합, (b) 대입문과 메소드 인자의 병합
Fig. 17. Merging between assignment statement and array index, (b) Merging between assignment statement and method argument

그림 17과 같이 대입문은 배열을 사용하거나 메소드를 호출을 포함할 수 있다. 이런 경우에도 대입문은 배열 인덱스나 메소드 인자와 병합될 수 있다. 병합에 필요한 대입문 교환 규칙과 병합 규칙은 대입문과 대입문에서 사용되었던 규칙들과 동일하다.

2) if, for, return 명령문과의 병합

```
x = a + b;
    +
if( x >0) {...};   ➡  if ((x = a + b) > 0)
                   (a) {...};

   x = a + b;
       +
for(i= x; i<10;i++)  ➡  for( i=(x=a+b); i<10; i++)
                     (b)

x = a + b;
    +
 return x;         ➡  return (x = a + b);
                   (c)
```

그림 18. (a) 대입문과 if 명령문의 병합, (b) 대입문과 for 명령문의 병합, (c) 대입문과 return 명령문의 병합
Fig. 18. (a) Merging between assignment and if statement, (b) Merging between assignment and for statement, (c) Merging between assignment and return statement

그림 18과 같이 대입문은 if, for, return 명령문들과 병합될 수 있다. 두 명령문이 병합되기 위해서는 대입문 교환 법칙에 의해서 두 명령문이 서로 인접하게 이동시키고 병합 규칙을 적용해야 한다. 여기서 주의할 점은 병합 알고리즘을 적용할 때 양방향(Duplex)알고리즘을 사용하지 못하고 단방향(Simplex)알고리즘을 사용해야 한다는 점이다. 그 이유는 if 명령문과 for 명령문이 이동하기 위해서는 if 와 for에 포함되어 있는 블록도 함께 이동해야 하고 블록에 대한 데이터 종속성 검사가 필요하기 때문이다. 향후 if 명령문과 for 명령문에 대한 향상된 병합 알고리즘을 제안하면 이러한 명령문들에

$$\frac{i_1 \in \mathrm{REF}(e_2),\ e_3 = \mathrm{SUB}[e_2,\ i_1,\ (i_1 = e_1)]}{[\![\ i_1 = e_1;\ \mathtt{if}(\ e_2\ )\ S;\ ]\!] \Rightarrow [\![\ \mathtt{if}(\ e_3\ )\ S;\ ]\!]} \quad \text{(a)}$$

$$\frac{i_1 \in \mathrm{REF}(e_2),\ e_3 = \mathrm{SUB}[e_2,\ i_1,\ (i_1 = e_1)]}{[\![\ i_1 = e_1;\ \mathtt{for}(e_2;e_3;e_4)\ S;]\!] \Rightarrow [\![\ \mathtt{for}(e_3;e_3;e_4)\ S;]\!]} \quad \text{(b)}$$

$$\frac{i_1 \in \mathrm{REF}(e_2),\ e_3 = \mathrm{SUB}[e_2,\ i_1,\ (i_1 = e_1)]}{[\![\ i_1 = e_1;\ \mathtt{return}(\ e_2\ );\ ]\!] \Rightarrow [\![\ \mathtt{return}(e_3);\ ]\!]} \quad \text{(c)}$$

그림 19. (a) 대입문과 if 명령문의 병합 규칙, (b) 대입문과 for 명령문의 병합 규칙, (c) 대입문과 return 명령문의 병합 규칙
Fig. 19. (a) Merging rule between assignment and if statement, (b) Merging rule between assignment and for statement, (c) Merging rule between assignment and return statement

대해서도 양방향 병합이 가능할 것이다. 또한 return 명령문은 프로그램의 흐름을 무조건 분기로 바꾸기 때문에 다른 대입문과 교환하여 이동하는 것이 불가능하다.

그림 19는 그림 18의 예제를 통해서 설명한 대입문과 if 명령문의 병합 규칙, 대입문과 for 명령문의 병합 규칙, 대입문과 return 명령문의 병합 규칙을 나타낸다.

IV. 실험 및 평가

표 1. SPEC JVM 벤치마크 프로그램을 이용한 대입문 병합 실험 결과 : 병합 개수
Table 1. Statements merging experiment result using SPEC JVM benchmark programs : merging numbers

SPEC JVM 벤치마크 프로그램	대입문 병합 개수						
	대입문	배열	메소드	if	for	return	합계
200_check	14	0	2	38	1	2	57
201_compress	13	2	2	6	0	0	23
202_jess	6	8	1	1	0	0	16
205_raytrace	11	4	43	25	1	0	84
209_db	5	2	5	11	1	0	24
213_javac	21	2	131	57	7	5	223
222_mpegaudio	191	9	18	11	1	3	233
228_jack	274	0	27	13	2	0	316
999_check	5	1	3	1	0	0	10
평균	60.0	3.1	25.8	18.1	1.4	1.1	109.6

표 2. SPEC JVM 벤치마크 프로그램을 이용한 대입문 병합 실험 결과 : 클래스 크기
Table 2. Statements merging experiment result using SPEC JVM benchmark programs : class size

SPEC JVM 벤치마크 프로그램	원본 클래스 사이즈 (byte)	변환된 클래스 사이즈 (byte)	축소된 크기 (byte)
200_check	29,856	29,837	19
201_compress	13,800	13,761	39
202_jess	8,907	8,907	0
205_raytrace	48,521	48,441	80
209_db	9,949	9,940	9
213_javac	486,399	486,223	176
222_mpegaudio	112,374	112,220	154
228_jack	105,967	105,715	252
999_check	3,841	3,838	3
평균	91,068	90,987	81.3

대입문 병합을 이용한 난독화 기법에 대한 실험으로 SPEC JVM 벤치마크 프로그램을 이용하였다. 각각의 벤치마크 프로그램에 6가지 범주의 대입문 병합 기법을 적용하여 병합된 대입문의 개수를 세고, 난독화 변환 전과 후의 클래스 크기를 비교하였다. 단방향과 양방향 병합 알고리즘 중에서 분석이 단순한 대입문의 병합에는 양방향 알고리즘을 사용하고, 분석이 복잡한 if 명령문과 for 명령문에 대해서는 단방향 알고리즘을 사용하여 실험하였다.

표 1에서 볼 수 있듯이, 대입문 병합의 6가지 대상 중에서 대입문, 메소드 인자, if 명령문과의 병합이 대부분의 변환을 차지하고, 배열 인덱스와의 결합, for 명령문과의 결합, return 명령문과의 결합은 평균 5개 미만으로 병합된 결과를 얻었다. 또한, 프로그램의 특성에 따라서 병합되는 대상이 큰 차이를 보였다. 예를 들어 javac의 경우에는 메소드와의 병합 비율이 높지만, mpegaudio와 jack의 경우에는 대입문과의 병합 비율이 높은 것을 확인할 수 있다. 병합된 대입문의 개수는 평균 109.6개인데 축소된 클래스 사이즈의 평균은 81.3byte이다. 병합된 개수와 비교하여 클래스 사이즈 축소가 작은 이유는 지역 변수의 병합인 경우에 'load'가 'dup'로 바뀌는 역할만을 하고 클래스 사이즈가 축소되지는 않기 때문이다. 그러나 'dup'는 '일반적으로 'load'보다 빠르게 동작하기 때문에 실행속도 향상이라는 부가적인 이익을 얻을 수 있다.

본 논문에서 제안한 대입문 병합 난독화 기법을 난독화 평가 기준에 의해서 평가를 해보면 다음과 같다. 첫째, 복잡도(Potency) 측면에서 살펴보면 프로그램의 크기가 증가하지 않았지만 대입문이 다른 명령문과 병합되어 명령문의 개수가 줄어들었기 때문에 원본 프로그램과 비교해보면 프로그램의 밀도가 증가한 것으로 해석할 수 있다. 따라서 동일한 사이즈의 프로그램으로 생각해보면 복잡도는 증가되었다고 평가할 수 있다. 둘째, 난독화 해석 도구로부터 얼마나 잘 견딜 수 있는지에 대한 강인도(resilience)를 평가해보면 부작용 제거 변환 기법에 의해서 부작용 자유 프로그램으로 변환될 수 있기 때문에 강인도는 높지 않다고 평가할 수 있다. 그러나 부작용 자유 프로그램으로 변환된 경우에는 프로그램의 사이즈가 증가하고, 실행 속도도 저하되는 것을 감수해야한다. 셋째, 난독화 변환에 의해서 생기는 오버헤드에 대한 비용(cost)의 평가해보면 프로그램 사이즈가 감소하고 실행속도도 향상되기 때문에 난독화에 의한 비용은 매우 낮다고 평가할 수 있다.

V. 결론 및 향후 연구 과제

자바는 분석 도구들에 의해서 분석이 용이하다는 문제점을 해결하기 위해서 다양한 난독화 기법이 제안되었다. 난독화를 사용하면 프로그램이 역공학 도구들에 의해서 쉽게 분석되는 것을 방지할 수 있다. 그러나 일반적인 난독화를 적용하면 프로그램의 사이즈가 증가하고 실행 속도가 떨어지는 단점이 생

긴다. 이 단점은 프로그램 사이즈와 실행 성능이 중요하게 고려되어야 하는 모바일 환경에서 문제가 된다.

본 논문에서는 두 개 이상의 대입문들을 병합하는 과정에서 생기는 부작용을 이용한 난독화 변환 기법을 새롭게 제안하였다. 대입문의 병합을 위해서 대입문 교환 규칙과 대입문 병합 규칙을 제시하였고, 두 가지 규칙을 이용한 대입문 병합 알고리즘을 제시하였다.

대입문 병합을 이용한 난독화 기법을 적용하면 프로그램의 이해도를 떨어뜨리는 난독화의 효과를 얻을 수 있고, 병합을 통한 배정문의 함축된 표현으로 인해서 프로그램의 사이즈가 줄어드는 효과도 얻을 수 있다. 그러나 대입문 병합 난독화 기법을 통한 사이즈 감소 비율이 크지 않고, 코드 사이즈가 감소되더라도 실행 속도가 느려질 수 있다. 따라서 대입문 병합을 통한 난독화 전과 후의 코드 사이즈와 실행 성능에 대한 분석 및 추가적인 연구가 필요하다.

본 논문의 연구를 바탕으로 앞으로 해결해야 할 문제점을 세 가지로 요약하였다. 첫째, 대입문과 대입문의 병합에 있어서 참조형 변수의 대입문의 병합을 고려해야 한다. 참조형 변수에 동일한 null 값을 배정하더라도 변수의 타입이 다른 경우에는 두 대입문을 병합하는 것은 컴파일 에러를 발생시킨다. 따라서 대입문 병합 규칙을 적용할 때 타입에 대한 정보도 유지하여 같은 타입에 대해서만 병합 규칙을 적용해야 한다.

둘째, 현재는 기본 블록을 단위로 대입문의 양방향 병합 규칙을 적용하였고, if와 for 명령문과 같이 프로그램 블록을 포함하는 대입문에 대해서는 단방향 병합 알고리즘을 적용하였다. 따라서 if와 for 명령문이 포함하는 블록에서 사용되는 모든 변수들에 대해서 데이터 종속성을 조사하고, 대입문 교환 규칙에 어긋나지 않는다면 블록이 이동할 수 있는 양방향 병합 규칙과 알고리즘을 제안해야 한다.

셋째, 대입문의 병합이 아닌 그 밖의 명령문들의 병합에 대해서도 고려할 수 있다. 특히 두 개 이상의 if 명령문은 각각의 포함하고 있는 표현식을 AND(&&) 또는 OR(||) 연산자로 연결하여 단락(short-circuit)을 이루는 한 명령문으로 표현할 수 있다. 단락을 포함하는 if 명령문은 한 명령문에 다수의 표현식을 포함할 수 있기 때문에 부작용이 높아지고 가독성이 떨어진다. 또한, AND 연산의 왼쪽 표현식의 결과가 false일 경우에 오른쪽 표현식을 평가하지 않은 상태로 false를 반환하고, OR 연산의 경우에는 왼쪽 표현식의 결과가 true인 경우에 오른쪽 표현식을 평가하지 않고 true를 반환하는 특징이 있다. 따라서 if 명령문의 병합은 프로그램을 난독화하는 효과적인 수단으로 사용할 수 있다.

참고문헌

[1] Christian Collberg, Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," IEEE Transactions on Software Engineering, vol.28, no.8, pp.735-746, 2002.

[2] Christian Collberg, Clark Thomborson, Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," Proc. of the Principles of Programming Languages, POPL98, pp.184-196, Jan. 1998.

[3] Christian Collberg, Clark Thomborson, Douglas Low, "Breaking Abstractions and Unstructuring Data Structures," Proc. of the International Conference on Computer Languages, ICCL98, pp.28-38. 1998.

[4] Christian Collberg, Clark Thomborson, Douglas Low, "A Taxonomy of Obfuscating Transformation," Technical Report #148, Department of Computer Science, The University of Auckland, 1997.

[5] Jien-Tasi Chan, Wuu Yang, "Advanced Obfuscations on Java Bytecode," Journal of Systems and Software, vol.71, no.1-2, pp.1-10, Apr. 2004.

[6] E. Kim, K. Han, "A Study on the Code Obfuscation Technique for Java Source Code," Proc. of the 35th KIISE Fall Conference, vol.35, no.2(A), pp. 307-308, Oct. 2008.

[7] P. Yuxue, J. Jung, J. Lee, "The Technological Trend of the Mobile Obfuscation," Information & Communications Magazine, vol.29, no.8, pp.65-71, Jul. 2012.

[8] Mark Harman, Lin Hu, Xingyuan Zhang Malcolm Munro., "Side-Effect Removal Transformation," 9th IEEE International Workshop on Program Comprehension (IWPC 2001). Toronto, Canada, pp. 309-319, May. 2001.

[9] Mark Harman, Lin Hu, Rob Hierons, Xingyuan Zhang, Malcolm Munro, Jose Javier Dolado,

Mari Carmen Otero and Joachim Wegener, "A Post-Placement Side-Effect Removal Algorithm," 18th IEEE International Conference on Software Maintenance (ICSM 2002), Montreal, Canada. pp.2-11, Oct. 2002.

[10] Lars R. Clausen, "A Java Bytecode Optimizer Using Side-effect Analysis," Concurrency and Computation : Practice and Experience, vol.9, no.11, pp.1031-1045, 1997.

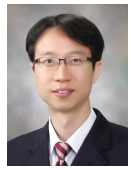
[11] Chrislain Razafimahefa, "A Study of Side-effect Analyses for Java," MS Thesis, School of Computer Science, McGill University, 1999.

[12] JJ. Dolado, M.Harman, M.C. Otero, and L.Hu, "An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension," IEEE Transactions on Software Engineering, vol.29, no.7, pp.665-670, Jul. 2003.

저 자 소 개

이 경 호

1991: 한국방송통신대학교 이학사.
1994: 한국과학기술원 공학석사.
2008: 단국대학교 공학박사
현 재: 한라대학교
정보통신방송공학부 교수
관심분야: 게임, 패턴인식, HCI,
디지털 신호처리,
컴퓨터 기술 응용
Email : khlee@halla.ac.kr



박 희 완

1997: 동국대학교 컴퓨터공학과 학사.
1999: 한국과학기술원 전산학과 석사.
2010: 한국과학기술원 전산학과 박사.
현 재: 한라대학교
정보통신방송공학부 교수.
관심분야: 프로그램 난독화, 역공학,
악성코드 분석,
소프트웨어 워터마킹,
정적 및 동적 분석 등
Email : heewanpark@halla.ac.kr