17-3-R02

비트 연산을 이용한 향상된 연산 난독화 기법

(Advanced Operation Obfuscating Techniques using Bit-Operation)

박 희 완, 최 석 우, 한 태 숙
한국과학기술원 전자전산학과 전산학전공
{hwpark, swchoi, han}@pllab.kaist.ac.kr

요 약

자바 프로그램은 클래스 파일에 소스 코드의 심볼릭 정보를 그대로 유지하고 있어서 클래스 파일로부터 원본과 거의 동일한 소스 코드를 얻어낼 수 있다. 이 문제에 대처하기 위해서 프로그램의 기능을 그대로 유지하고, 소스 코드를 해석하기 어려운 형태로 변환하는 난독화 방법이 제안되었다. 지금까지 연구된 난독화 기법들은 레이아웃을 위주로 변환하기 때문에 프로그램에 포함된 알고리즘 구조 자체를 변경시키지는 못한다. 따라서 중요한 데이터나 알고리즘을 보호하려고 할 때 난독화 도구를 적용하는 것은 무의미했다. 본 논문에서는 관련연구로서 선형함수를 사용하여 데이터와 그 연산을 보호하는 난독화 기법을 소개하고, 선형 함수 난독화 기법의 단점인 실행 속도 저하와 클래스 크기 증가에 대한 단점을 보완한 비트 연산자를 사용한 연산 난독화 기법을 새롭게 제안했다. 비트 연산자를 이용해서 연산을 난독화하면 프로그램의 실행 성능을 유지하면서 데이터와 연산들에 대해서 복잡한 난독화를 수행할 수 있다. 또한 후처리 과정으로서 연산의 최적화 과정을 제안하여 클래스 크기 증가와 프로그램 실행 속도 저하의 단점을 보완할 수 있게 하였다.

1. 서론*

1.1 연구 배경

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[1,2,3,4].

그 결과로 자바 프로그램들은 악의적인 역공학자들에 의해서 크게는 프로그램의 일부분이나, 작게는 프로그램에 사용된 자료 구조나 알고리즘이 도용당할 수 있는 위험이 있다. 이 문제에 대처하기 위해서 난독화라는 기법이 제안되었다[1]. 난독화란 프로그램의 기능을 그대로 유지한 채 소스코드를 해석하기 어려운 형태로 변환하는 방법이다. 난독화를 사용하면 프로그

* 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음.

램이 역공학 도구들에 의해서 쉽게 분석되는 것을 방지할 수 있다.

지금까지 개발된 난독화 도구들의 주된 관심은 소스코드의 포맷이나 변수의 이름 등을 대상으로 하는 레이아웃(layout) 변환에 있었다. 레이아웃 변환은 의미있는 변수들의 이름을 무의미하게 바꾸는 변환을 통해서 프로그램 이해를 어렵게 만들 수 있다. 하지만 프로그램에 포함된 알고리즘 구조 자체를 변경시키지는 못한다.

1.2. 문제 제기 및 해결책

(그림 1)은 간단한 계산식이 포함된 자바 예제 프로그램이다.

```
class Test {
    static int bonus = 12, salary = 52;
    public static void main(String params[])
    {
        for(int year = 1; year <=50; year++)
            bonus = (bonus * (salary - year - 1))/year;
    }
}
```

(그림 1) 원본 프로그램

이 프로그램을 javac로 컴파일 한 후에 jad 역컴파일러[5]를 사용하면 (그림 2)의 결과를 얻을 수 있다.

```
class Test {
    static int bonus = 12;
    static int salary = 52;
    public static void main(String args[])
    {
        for(int i = 1; i <= 50; i++)
            bonus = (bonus * ((salary - i) + 1)) / i;
    }
}
```

(그림 2) 원본 프로그램을 jad로 역컴파일한 결과

(그림 1,2)를 비교해 보면 javac에 의해서 컴파일 될 때 main 메소드 안에 포함되어 있는 지역변수인 'year'의 이름이 삭제되어 역컴파일러에 의해서 'i'라는 이름으로 새롭게 결정된 것과 main 메소드의 인자가 'params'에서 'args'로 바뀐 것을 제외하면 원본 소스 프로그램과 동일한 결과를 얻을 수 있다는 것을 알 수 있다.

역공학자의 분석을 어렵게 하기 위한 전처리로서 난독화 도구를 사용할 수 있다. RetroGuard[6]라는 난독화 도구를 이용하여 원본 프로그램의 난독화 과정을 거친 후 역컴파일러를 사용하였다. 그 결과로 (그림 3)과 같은 소스 코드를 얻을 수 있다.

```
class Test {
    static int a = 12;
    static int b = 52;
    public static void main(String args[])
    {
        for(int i = 1; i <= 50; i++)
            a = (a * ((b - i) + 1)) / i;
    }
}
```

(그림 3) RetroGuard 난독화 과정을 거친 후 역컴파일한 결과

(그림 3)의 소스 프로그램은 난독화 도구 RetroGuard에 의해서 변수이름이 난독화되었다. 난독화 과정을 거치면 클래스 이름과 메소드 이름, 변수 이름등이 모두 난독화되어야 하지만 메인 클래스인 Test와 main 메소드 이름은 프로그램 실행에 영향을 미치기 때문에 난독화 대상에서 제외되었다. 결과적으로 클래스 필드인 'bonus'와 'salary'가 각각 'a', 'b'라는 이름으로 변환되었는데 이것만으로는 for 문장을 실행하면서 수행되는 중요한 연산식은 변화없이 노출되고있기 때문에 적절한 난독화가 이루어졌다고 보기 어렵다.

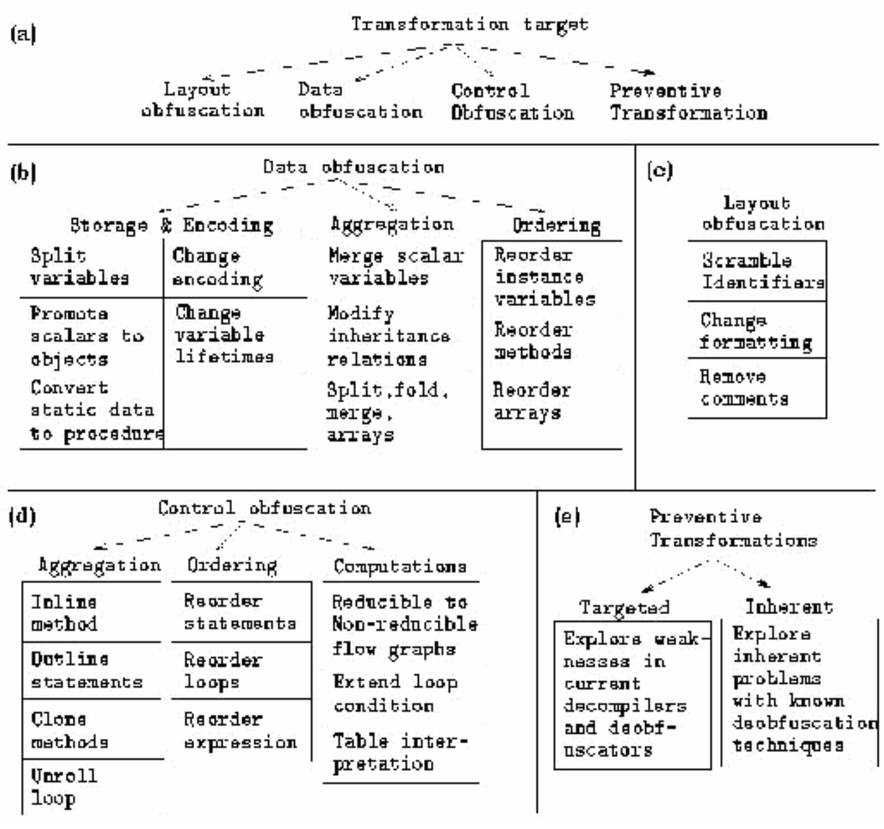
지금까지 개발된 난독화 도구들은 주로 레이아웃 난독화에 집중되었기 때문에 데이터나 연산에 대한 난독화가 이루어지지 못하였다. 따라서 중요한 상수 데이터나 알고리즘을 보호하려고 할 때 난독화를 적용하는 것은 무의미했다. 따라서 본 논문에서는 관련연구로서 선형 함수를 사용하여 데이터와 그 연산을 보호하는 난독화 기법을 소개하고, 선형 함수 난독화 기법의 단점인 실행 속도 저하와 클래스 크기 증가

에 대한 단점을 보완한 비트 연산자를 사용한 연산 난독화 기법을 새롭게 제안한다. 비트 연산자를 사용해서 연산 난독화를 하면 일반적인 레이아웃 난독화에 의해서 변환되지 않는 데이터와 연산들에 대해서 복잡한 난독화를 수행할 수 있다. 또한 후처리 과정으로서 연산의 최적화 과정을 제안하여 클래스 크기 증가와 프로그램 실행 속도 저하의 단점을 보완할 수 있게 하였다.

1.3. 논문 구성

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대한 소개로 난독화에 대한 기존의 연구들을 간단히 정리한다. 3장에서는 선형함수를 이용한 상수와 그 연산에 대한 난독화에 대해서 알아보고 이 방법이 가지는 단점을 분석한다. 4장에서는 비트 연산자를 이용한 연산 난독화 기법을 소개한다. 5장에서는 연산 난독화 변환후에 이루어질 수 있는 최적화 작업에 대해서 알아보고, 마지막 6장에서는 결론을 맺고 비트 연산을 사용한 난독화 기법이 보완해야 할 점들과 향후 연구 과제에 대해서 생각해본다.

2. 관련연구



(그림 4) 난독화 기법의 분류

(그림 4)[1]는 난독화 기법의 분류에 대한 내용이다. 그림 2의 항목(a)에서와 같이 난독화 기법은 변환 대상에 따라서 크게 네 가지로 구분된다. 가장 단순한 변환 대상은 항목(a)에서 설명하는 소스 코드의 포맷이나 변수의 이름 등을 대상으로 하는 레이아웃(layout)이다. 그리고 항목(b)에서 설명되는 프로그램에서 사용된 자료구조가 대상이 되는 자료(data) 혼란이 있고 항목(c)에서 설명한 프로그램의 제어구조를 바꾸는 제어(control) 난독화 기법이 있다. 항목(e)에서 설명하는 Preventive변환기법은 특정한 역컴파일러나 자동화된 난독화 변환 해석기(deobfuscator)에 의한 공격을 무력화하기 위한 변환을 의미한다.

항목(b)의 자료 혼란 변환은 크게 세 가지로 다시 분류할 수 있다. 첫째, 프로그램에서 쓰인 변수나 객체에 해당하는 저장 장소의 형태를 변경시키거나 코딩을 바꾸는 방법인 저장 장소(storage)와 인코딩(encoding) 혼란 기법이 있고, 둘째, 변수나 객체의 상속관계, 배열등을 통합하거나 분리시키는 방법인 집합(aggregation) 혼란 기법이 있고, 셋째, 객체와 변수, 메소드와 배열의 순서를 변경시키는 순서(ordering) 혼란 기법이 있다.

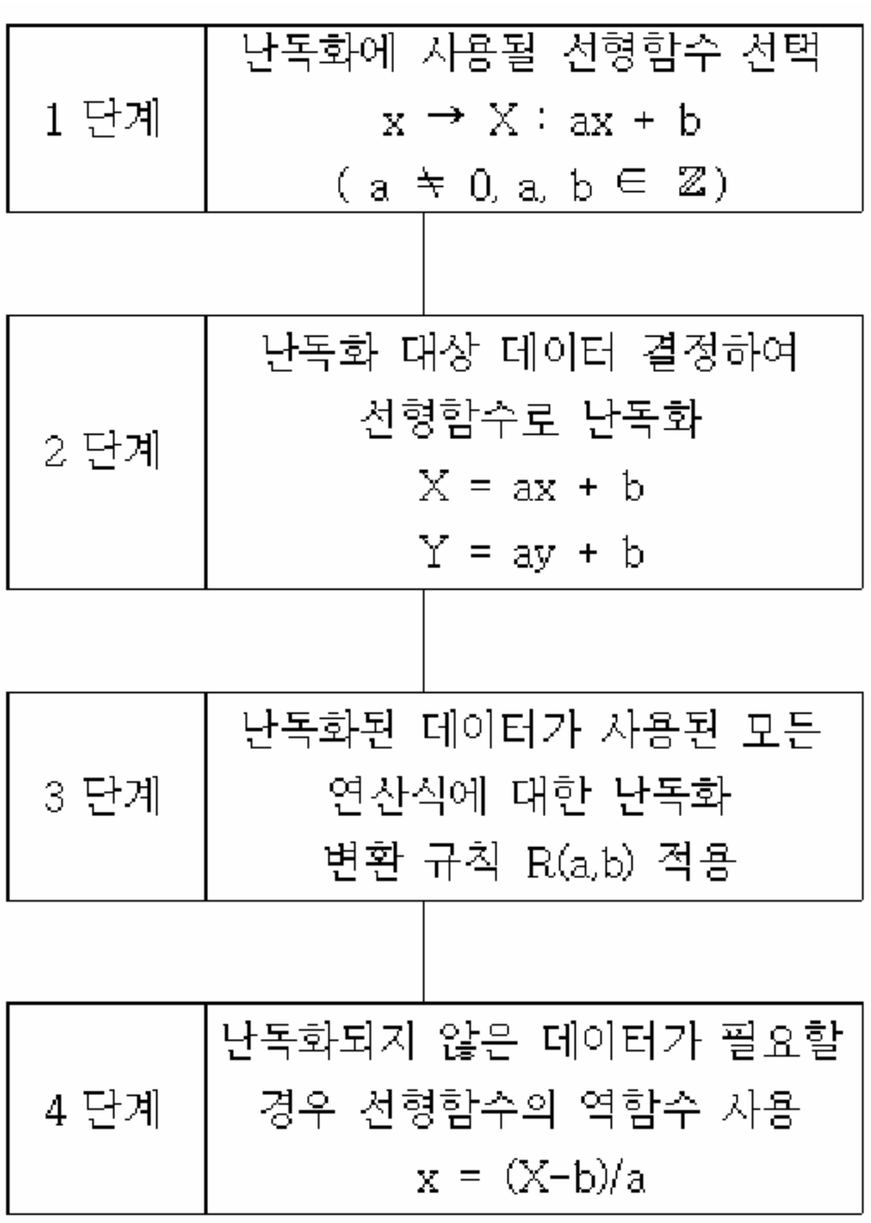
항목(d)의 제어 혼란 변환도 크게 세 가지로 다시 분류할 수 있다. 첫째, 메소드를 인라인(inline) 시키거나 문장들을 아웃라인(outline) 시키거나, 메소드를 복제(clone)하거나 루프를 해제(unroll loop)하는 기법인 집합(aggregation) 혼란 기법이 있고, 문장이나 루프, 표현식 등의 순서를 바꾸는 순서(ordering)변환 기법이 있고, 제어 흐름 그래프를 변경시키거나 루프 조건을 확장시키거나 프로그램 해석 테이블을 이용하는 등 계산(computation) 혼란 기법이 있다[1].

3. 선형함수를 이용한 연산 난독화

3.1. 연산 난독화 기법의 개요

선형함수를 이용한 데이터와 연산에 대한 난

독화[4]의 개요는 다음과 같다. 첫 번째, 데이터 난독화에 사용될 선형함수를 결정한다. 두 번째, 난독화의 대상이 되는 데이터를 결정하여 선형함수를 사용하여 난독화를 한다. 세 번째, 난독화된 데이터가 사용된 모든 연산에 대해서 연산의 난독화를 한다. 만일 이 과정을 거치지 않는다면 난독화된 데이터가 사용된 연산은 원본 프로그램과 다른 결과를 얻게 된다. 네 번째, 난독화되지 않은 원래의 계산 결과를 얻고자 하는 경우에 첫 번째 단계에서 사용한 선형함수의 역함수를 이용해서 난독화된 값을 본래의 값으로 되돌린다. 이 과정은 간략하게 (그림 5)와 같이 표현할 수 있다.



(그림 5) 선형함수를 이용한 연산 난독화 과정

(그림 5)의 1단계에서 선택되는 선형함수를 이용해서 데이터가 난독화되는데, 데이터의 난독화를 위한 변환 함수가 반드시 선형함수일 필요는 없다. 역함수가 존재하는 함수인 경우에는 어떠한 함수를 사용해도 상관없지만 난독화의 실행에 소요되는 비용을 고려해서 선형함수를 선택한 것이다. 또한, 선형함수를 선택했을 경우에는 3단계에서 사용되는 연산식 변환규칙에서 축약된 표현을 얻을 수 있는 장점이 있다. 데이터 난독화에 사용되는 변환 규칙을 C(a,b)라고 표현하고 다음과 같은 의미가 있다.

$$C(a,b) : x \rightarrow X : X = ax+b$$

2단계에서는 프로그램중에서 보안의 중요도가 높은 데이터를 선택하여 1단계에서 결정한 데이터를 난독화한다. 선형함수로 'ax+b'를 선택했다면 'x'를 난독화한 결과 'X=ax+b'의 계산값을 가진다.

3단계에서는 난독화된 데이터가 사용된 모든 연산식을 난독화한다. 연산식의 난독화에서 사용되는 변환 규칙은 (그림 6)과 같다. 나열된 규칙들에 대한 유도 방법은 간단하다. 변환된 식으로부터 원래의 식을 얻어낼 수 있는 형태를 유지하기 위한 연산이 추가된 형태라고 볼 수 있다[4].

$$R_1(a,b) : X + Y \rightarrow X + Y - b$$
$$R_2(a,b) : X - Y \rightarrow X - Y + b$$
$$R_3(a,b) : X * Y \rightarrow (X*Y - b*(X + Y - b - a))/a$$
$$R_4(a,b) : X / Y \rightarrow (a*X + b*Y - b*(b + a))/(Y - b)$$
$$R_5(a,b) : X + c \rightarrow X + a*c$$
$$R_6(a,b) : X - c \rightarrow X - a*c$$
$$R_7(a,b) : c - X \rightarrow a*c - X + 2*b$$
$$R_8(a,b) : c * X \rightarrow c*X - b*c + b$$
$$R_9(a,b) : X / c \rightarrow X/c - b/c + b$$
$$R_{10}(a,b) : c / X \rightarrow (a^2*c)/(X - b) + b$$

(그림 6) 선형함수 난독화 변환규칙

4단계는 난독화되지 않은 데이터값이 필요할 경우에 1단계에서 결정한 선형함수의 역함수를 이용하는 단계이다.

이 변환규칙을 D(a,b)라고 표현하고 다음과 같은 의미가 있다.

$$D(a,b) : X \rightarrow x : (X - b) / a = x$$

3.2. 연산 난독화 기법의 적용

지금까지 설명한 선형함수를 이용한 연산 난독화를 서론에서 다루었던 예제 프로그램에 적용하여 각각의 단계를 살펴보고자 한다.

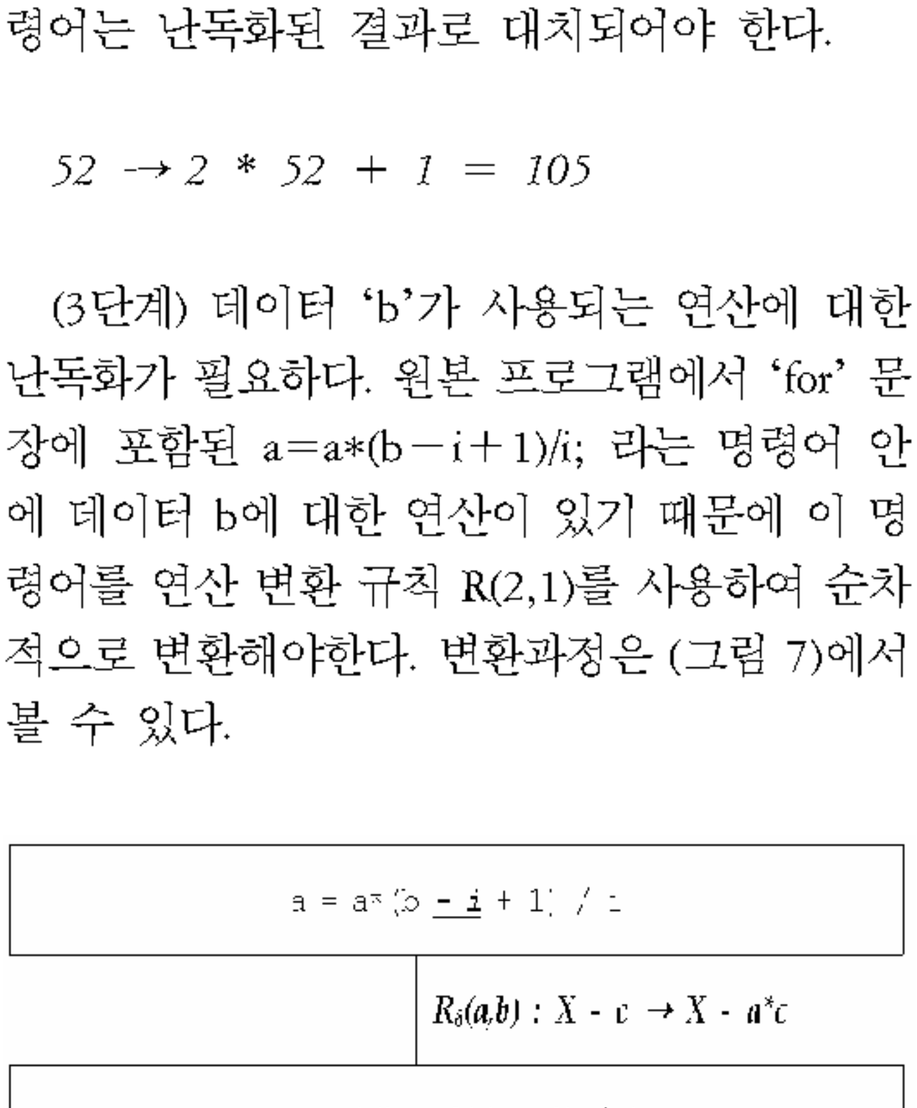
(1단계) 선형함수 C(a,b)를 결정해야 한다. 여기서는 a=2, b=1로 정하였다.

$$C(2,1) : x \rightarrow X : 2*x + 1$$

(2단계) 난독화 대상 데이터는 'b'로 정하였다. 따라서 원본 프로그램의 'b = 52;' 라는 명령어는 난독화된 결과로 대치되어야 한다.

$$52 \rightarrow 2 * 52 + 1 = 105$$

(3단계) 데이터 'b'가 사용되는 연산에 대한 난독화가 필요하다. 원본 프로그램에서 'for' 문장에 포함된 a=a*(b−i+1)/i; 라는 명령어 안에 데이터 b에 대한 연산이 있기 때문에 이 명령어를 연산 변환 규칙 R(2,1)를 사용하여 순차적으로 변환해야한다. 변환과정은 (그림 7)에서 볼 수 있다.



(그림 7) 연산 난독화 과정

(4단계) 변환된 연산식에서 난독화되지 않은 데이터값을 되돌리기 위해서 선형함수의 역함수를 사용한다.

$$D(2,1) : X \rightarrow x : (x - 1) / 2$$

연산 난독화의 모든 단계를 거쳐서 완성된 프로그램은 (그림 8)과 같다.

```
class Test {
    static int a = 12;
    static int b = 105;
    public static void main(String args[])
    {
        for(int i = 1; i <= 50; i--)
            a = ((( a*(b-2*i - 2)-a+1 )/2 - 1)/i + 1)-1)/2;
    }
}
```

(그림 8) 데이터와 연산이 난독화된 프로그램

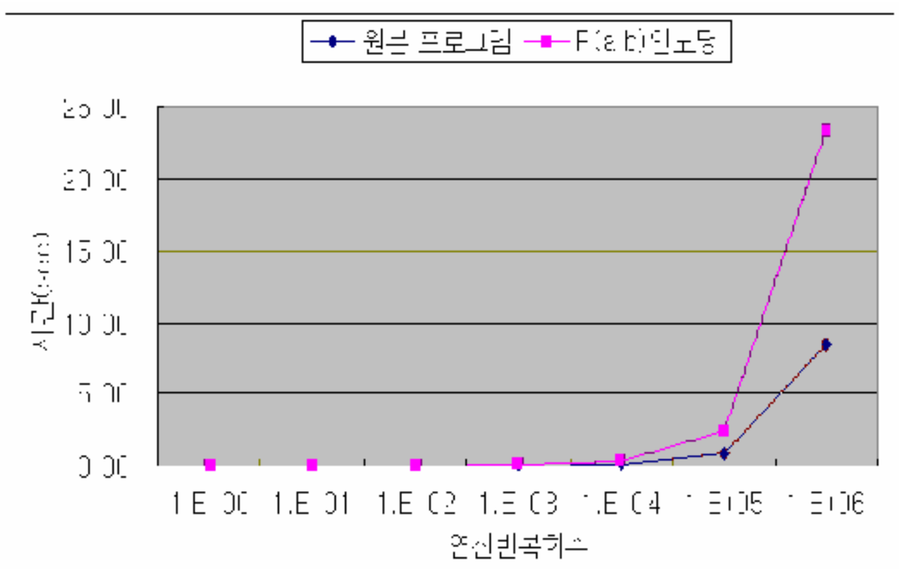
3.3. 선형함수를 이용한 난독화 기법의 평가와 문제점

난독화 기법의 평가는 세 가지 측면에서 이루어진다. 첫째, 프로그램에 얼마나 많은 애매성이 존재하는지에 대한 평가(Potency)와, 둘째, 자동화된 해석도구(deobfuscator)로부터 얼마나 잘 견딜 수 있는지 강인성에 대한 평가(Resilience)와, 셋째, 프로그램 실행에서 생기는 난독화의 오버헤드에 대한(Cost)평가이다[1,2].

선형함수를 사용한 난독화 기법은 하나의 연산에 난독화를 여러번 반복하여 적용하는 것이 가능하기 때문에 Potency가 매우 높다. 또한, 이미 변환된 연산을 자동화된 도구에 의해서 난독화되기 이전 상태로 되돌릴 수 없기 때문에 Resilience도 매우 높다.

원본 프로그램과 난독화된 프로그램의 실행 속도를 실험을 통해서 비교하였다. 본 논문에서 사용한 실험 환경은 Intel Pentium Ⅲ 933Mhz, Microsoft Windows 2000 Professional, 256MB memory, Sun JDK 1.4.1이다. (그림 9)의 실험 결과로부터 선형함수를 이용하여 연산 난독화

된 프로그램은 원본 프로그램과 비교하여 속도 저하에 의한 Cost가 높다는 것을 알 수 있다.



(그림 9) 선형함수를 사용한 난독화의 실행 속도

4. 비트 연산자를 이용한 연산 난독화

4.1. 비트 연산자의 특징

Axil Ultra 1 Benchmarks
Sun Ultra-1 Clone, 1x200MHz UltraSPARK CPU at 200MHz
256MB memory, Sun JDK 1.1.5 (no JIT)

Java	JVM	time(ns)
+, -	iadd, isub	70
^	ixor	65
*	imul	134
/	idiv	262
<<, >>	ishl, ishr	75
%	irem	287

(그림 10) 연산자 실행 속도 비교

(그림 10)[7]의 실험 결과로부터 알 수 있듯이, 비트 연산자는 사칙 연산자에 비하여 실행 속도가 빠르고, 2진법에 기초를 하고 있기 때문에 10진법의 사칙연산에 익숙한 역공학자들의 분석을 어렵게 한다는 특징이 있다. 따라서 데이터와 연산의 난독화를 위해서 비트 연산을 사용한다면 연산 난독화의 단점이었던 실행 속도를 향상시키면서 분석을 더욱 어렵게 만들 수 있다.

4.2. 비트 연산을 이용한 난독화 기법

본 논문에서 제안하는 비트 연산을 사용한 난독화 기법이 관련 연구인 선형 함수를 사용한 연산 난독화 기법에 구별되는 차이점은 데이터 변환에 사칙연산을 사용하지 않고 비트 연산을 사용하는 것이다. 따라서 데이터를 난독화하는 C(a,b)와 원래의 상태로 되돌리는 D(a,b)를 새롭게 정의해야 하고, 또한, 난독화된 데이터에 대한 연산 규칙도 다시 유도해야 한다.

먼저, 데이터 난독화 규칙은 BC(a,b)라고 이름짓고, 다음과 같이 정의할 수 있다.

$$BC(a,b) : x \rightarrow X : x << a \ \hat{} \ b = X \ (a,b \in \mathbb{N})$$

규칙 BC(a,b)에 의해서 x, y 는 각각 X, Y로 난독화 될 수 있다.

$$X = x << a \ \hat{} \ b$$
$$Y = y << a \ \hat{} \ b$$

또한, 난독화된 데이터에 대한 연산을 수행하기 위해서 선형함수를 이용한 관련연구와 비슷한 방식으로 연산 변환규칙을 유도할 수 있다. (그림 11)에서는 10개의 연산에 대한 변환 규칙만을 보여주고 있지만, 실제로 프로그램에 적용하기 위해서는 자바언어에서 제공하는 모든 연산자에 대한 연산 규칙이 필요하다. 나머지 연산자에 대한 연산 규칙과 변환규칙을 유도해내는 방법은 부록으로 첨부하였다.

(연산자 우선순위 : 사칙연산 > 쉬프트 연산 > XOR연산)

$$BR_1(a,b) : X + Y \rightarrow (X \hat{} b) + (Y \hat{} b) \ \hat{} \ b$$
$$BR_2(a,b) : X - Y \rightarrow (X \hat{} b) - (Y \hat{} b) \ \hat{} \ b$$
$$BR_3(a,b) : X * Y \rightarrow (X \hat{} b) * (Y \hat{} b) >> a \ \hat{} \ b$$
$$BR_4(a,b) : X / Y \rightarrow (X \hat{} b) / (Y \hat{} b) << a \ \hat{} \ b$$
$$BR_5(a,b) : X + c \rightarrow (X \hat{} b) + (c<<a) \ \hat{} \ b$$
$$BR_6(a,b) : X - c \rightarrow (X \hat{} b) - (c<<a) \ \hat{} \ b$$
$$BR_7(a,b) : c - X \rightarrow (c<<a) - (X \hat{} b) \ \hat{} \ b$$
$$BR_8(a,b) : c * X \rightarrow c*(X \hat{} b) \ \hat{} \ b$$
$$BR_9(a,b) : X / c \rightarrow (X \hat{} b)/c \ \hat{} \ b$$
$$BR_{10}(a,b) : c / X \rightarrow c/(X \hat{} b) << 2*a \ \hat{} \ b$$

(그림 11) 비트 연산을 사용한 연산 변환 규칙

난독화된 변수를 다시 원래의 상태로 되돌리는 방법은 다음과 같은 식을 이용한다.

$$BD(a,b) : X \rightarrow x : (X \hat{} b) >> a = x$$

4.3. 비트 연산을 이용한 연산 난독화 기법의 적용

지금까지 설명한 비트 연산을 사용한 연산 난독화를 서론에서 다루었던 예제 프로그램에 적용하여 각각의 단계를 살펴보고자 한다.

(1단계) 선형함수 BC(a,b)를 결정해야 한다. 여기서는 a=2, b=101(2)로 정하였다.

$$BC(2,1) : x \rightarrow X : x << 2 \ \hat{} \ 5$$

(2단계) 난독화 대상 데이터는 'b'로 정하였다. 따라서 원본 프로그램의 'b=52;' 라는 명령어는 난독화된 결과로 대치되어야 한다.

$$52 \rightarrow 52 >> 2 \ \hat{} \ 5 = 213$$

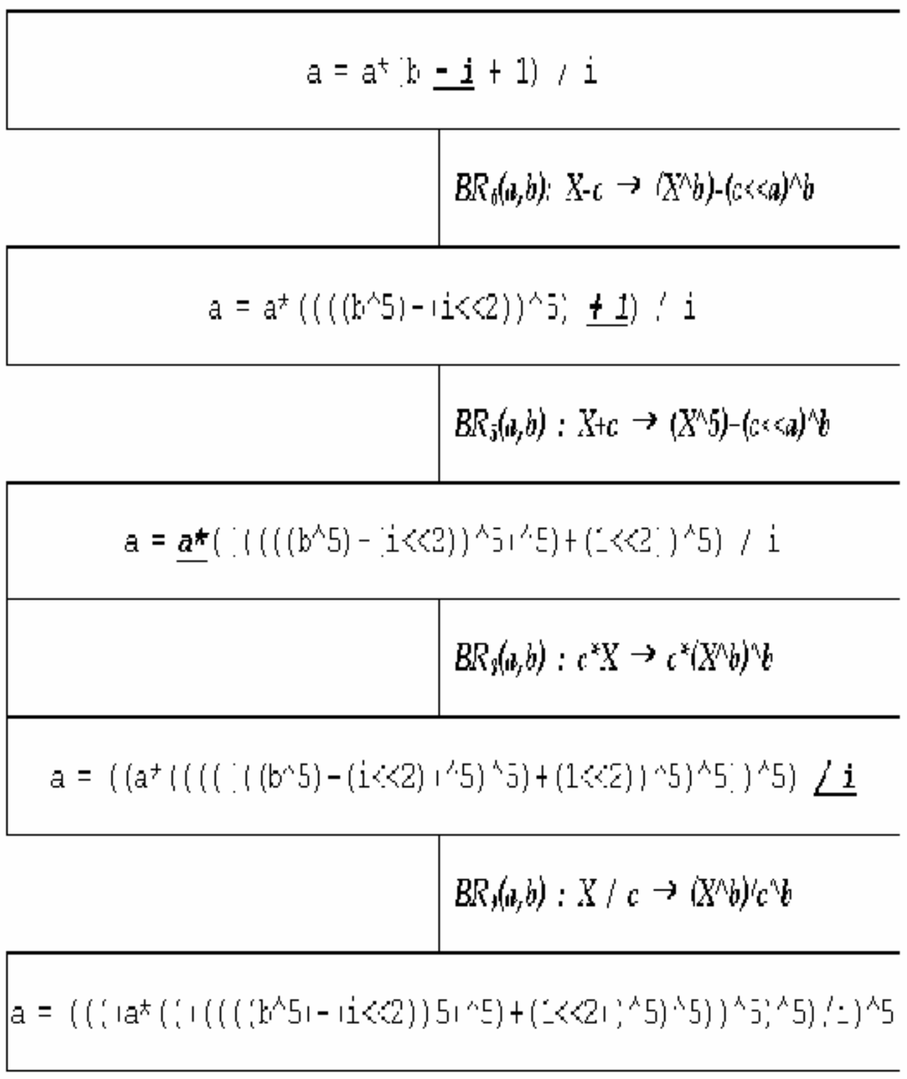
(3단계) 데이터 'b'가 사용되는 연산에 대한 난독화가 필요하다. 원본 프로그램에서 'for' 문장에 포함된 a=a*(b-i+1)/i; 라는 명령어 안에 데이터 b에 대한 연산이 있기 때문에 이 명령어를 연산 변환 규칙 BR(2,5)를 사용하여 순차적으로 변환해야한다. 변환과정은 (그림 12)와 같다.

(4단계) 변환된 연산식에서 난독화되지 않은 데이터값을 얻기 위해서 비트연산의 역연산을 사용한다.

$$BD(2,1) : X \rightarrow x : (x \ \hat{} \ 5) >> 2$$

연산식 난독화의 모든 단계를 거쳐서 완성된 프로그램은 (그림 13)과 같다.

원본 프로그램의 단순했던 연산식이 쉬프트 연산과 'xor'연산을 포함하는 복잡한 연산식으로 바뀐 것을 알 수 있다.



(그림 12) 비트 연산을 사용한 연산 난독화 과정

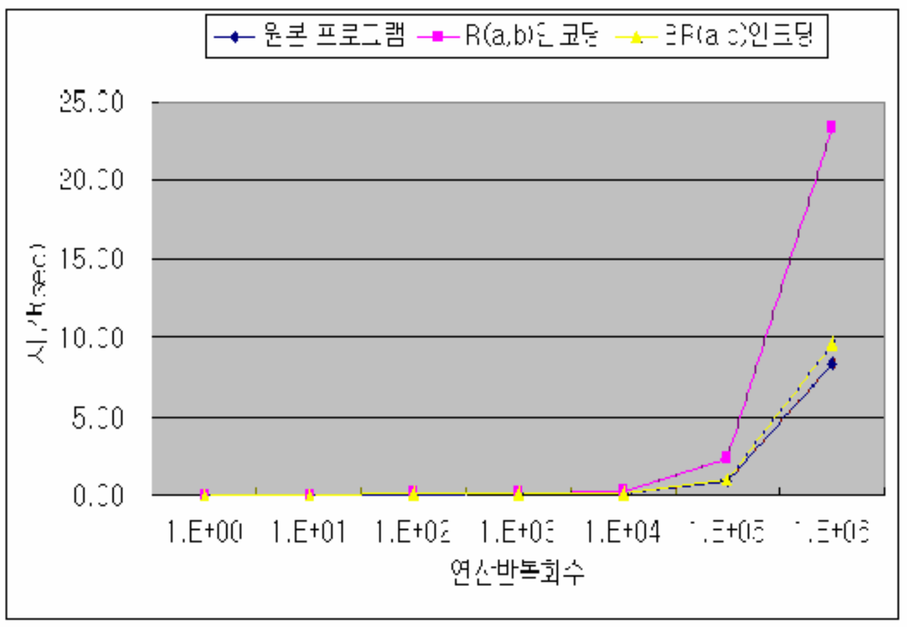
```
class Test {
    static int a = 12;
    static int b = 213;
    public static void main(String args[])
    {
        for (int i = 1; i <= 50; i++)
        a = ((((((a*(((((((b^5)-(i<<2))^5)^5)+(1<<2))^5)^5))^5)^5)/i)^5)^5)>>2;
    }
}
```

(그림 13) 비트 연산으로 난독화된 프로그램

4.4. 비트 연산을 이용한 난독화 기법의 평가

비트 연산을 사용한 난독화 기법도 세 가지 평가 기준으로 검토해볼 수 있다.

Potency를 살펴보면 하나의 연산에 난독화 기법을 여러번 반복하여 적용하는 것이 가능하기 때문에 매우 높다. 또한, 이미 변환된 연산을 자동화된 도구에 의해서 난독화되기 이전 상태로 되돌릴 수 없기 때문에 Resilience도 매우 높다.



(그림 14) 선형함수와 비트연산 난독화의 실행 속도 비교

(그림 14)의 실험 결과와 같이 연산을 연산의 반복 회수를 높여서 클래스 초기화에 사용되는 오버헤드를 무시할 수 있을 정도가 되었을 경우에 실행 속도는 선형함수를 사용한 연산 난독화 기법보다 2배 이상 빠른 결과를 얻을 수 있었고, 난독화를 거치지 않은 프로그램과 비교하면 10%정도의 속도 저하를 나타내었다. 그러나, 비트 연산을 사용한 난독화 기법은 연산 변환 규칙이 복잡하기 때문에 더 많은 연산이 필요하고, 따라서 클래스 크기는 (그림 15)를 통해서 알 수 있듯이 선형함수를 사용한 경우보다 커진다는 문제점이 있다. 이 문제점을 해결하기 위해서는 난독화된 연산식을 최적화하는 과정이 필요하다.

5. 난독화된 연산의 최적화

5.1. 최적화의 필요성

프로그램	크기(byte)
변환되지 않은 상태	344
선형함수 난독화	373
비트연산 난독화	784

(그림 15) 클래스 크기 비교

연산 난독화를 하면 간단한 연산식이 복잡한 식으로 바뀐다. 변환된 연산식에는 불필요하게 추가된 연산이 포함될 수 있기 때문에 프로그램 크기를 줄이고, 실행 시간 오버헤드를 줄이기 위해서 연산식의 최적화 단계가 필요하다.

연산식의 최적화는 소스 레벨과 바이트코드 레벨에서 각각 이루어질 수 있다. 소스 레벨에서의 최적화는 연산식의 우선순위를 고려해서 연산식을 파싱한 다음 불필요한 연산을 삭제해야 하기 때문에 과정이 복잡하다. 그러나 바이트코드 레벨에서의 최적화는, 자바 컴파일러에 의해서 연산식의 우선순위가 이미 고려된 상태이고 자바 가상머신이 스택기반으로 동작하기 때문에 바이트코드상에서 서로 인접하면서 상쇄되는 연산을 함께 삭제하면 된다. 따라서 소스 레벨보다 과정이 단순하지만 최적화된 결과가 완벽하지 못하다는 단점이 있다.

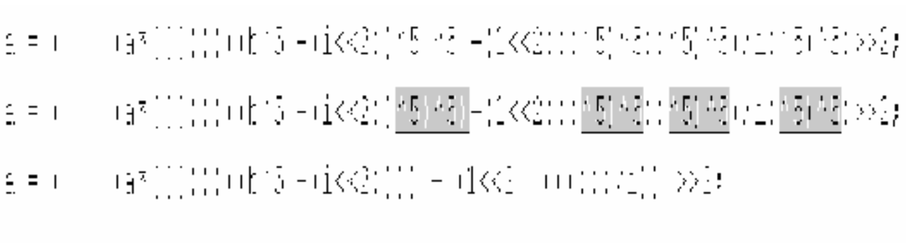
5.2. 소스 레벨에서의 최적화

(그림 16, 17)은 각각 선형함수 난독화 기법과 비트연산 난독화 기법을 적용한 연산식의 소스 레벨 최적화 과정을 보여준다. 최적화 과정을 거치고 나면 연산식을 난독화하는 과정에서 추가된 불필요한 연산들이 삭제된 연산식을 얻을 수 있다.

```
a = (((( a*(b-2*i - 2)-a+1 )/i - 1/i - 1)-1)/2;
a = (((( a*(b-2*i - 2)-a)/i - 1/i - 1/i + 1)-1)/2;
a = (((( a*(b-2*i - 2)-a)/i - 1)-1)/2;
a = ((( a*(b-2*i + 2)-a)/i + 1 - 1)/2;
```

최적화된 연산식 : `a = (((a * (b - 2 * i - 2) - a) / i) / 2;`

(그림 16) 선형함수 난독화된 연산식의 소스 레벨 연산 최적화



최적화된 연산식 :

(그림 17) 비트연산 난독화된 연산식의 소스 레벨 연산 최적화

5.3. 바이트코드 레벨에서의 최적화

(그림 18, 19)는 각각 선형함수 난독화 기법과 비트연산 난독화 기법을 적용한 연산식의 바이트코드 레벨 최적화 과정을 보여준다. 최적화 과정을 거치고 나면 연산식을 난독화하는 과정에서 추가된 불필요한 연산들이 삭제된 연산식을 얻을 수 있다.

(그림 18)에서는 상쇄되는 연산인 {iconst_1, iadd}와 {iconst_1,isub}가 서로 삭제되었고, 그림19에서는 {iconst_5,ixor}와 {iconst_5,ixor}가 서로 삭제되었다. 최적화된 결과로 알 수 있듯이 비트연산 난독화 기법은 'xor'연산의 특성상 바이트코드 레벨에서 상쇄되는 부분이 많다. 즉, 연산식을 분석하지 않고서도 바이트코드 레벨에서 상쇄되는 연산을 없애는 과정을 거쳐서 최적화된 결과를 쉽게 얻을 수 있다.

// 1: iload_1 : 지역 변수 a	// 10: iload_1	// 0: isub
// 2: iload_2 : 지역 변수 b	// 11: isub	// 20: iconst_1
// 3: iconst_2	// 12: iconst	// 21: iadd
// 4: iload_3 : 지역 변수	// 13: idiv	// 22: iconst_1
// 5: imul	// 14: iload_3	// 23: isub
// 6: isub	// 15: ixor	// 24: iconst_2
// 7: iconst_2	// 16: iconst	// 25: idiv
// 8: iadd	// 17: iload_3	// 26: istore_1
// 9: imul	// 18: ixor	

(그림 18) 선형함수 난독화된 연산식의 바이트코드 레벨 연산 최적화

선형함수 난독화된 연산식의 바이트코드 레벨에서 최적화된 연산식은 다음과 같다.

a=(((((a*((b−2*i)+2)−a)+1)/i−1/i)/2;

// 1 iload_1 : 지역 변수 a	// 13: iconst_4	// 25: idiv
// 2 iload_2 : 지역 변수 b	// 14: iadd	// 26: iconst_5
// 3 const_5	// 15: iconst_5	// 27: ixor
// 4 xor	// 16: ixor	// 28: iconst_5
// 5 iload_3 : 지역 변수 i	// 17: iconst_5	// 29: ixor
// 6 const_2	// 18: ixor	// 30: iconst_2
// 7 shl	// 19: imul	// 31: ishr
// 8 sub	// 20: iconst_5	// 32: istore_1
// 9: iconst_5	// 21: ixor	
// 10: ixor	// 22: iconst_5	
// 11: iconst_5	// 23: ixor	
// 12: ixor	// 24: iload_3	

(그림 19) 비트연산 난독화된 연산식의 바이트코드 레벨 연산 최적화

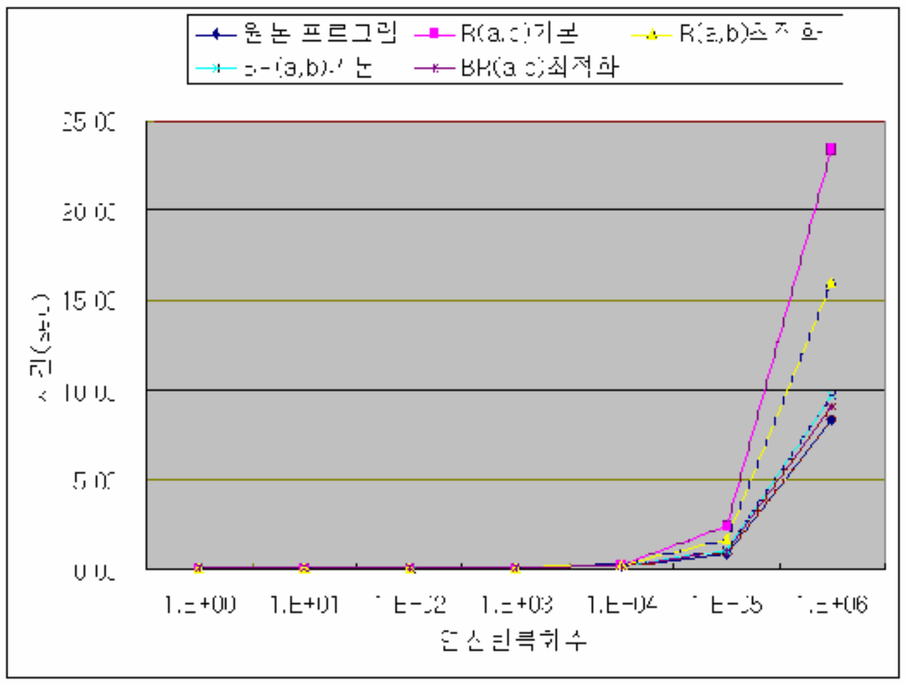
비트연산 난독화된 연산식의 바이트코드 레벨에서 최적화된 연산식은 다음과 같다.

a=a*((b^5)−(i<<2)+4)/i>>2;

5.4. 최적화된 난독화 기법의 평가

(그림 20)의 실험 결과에서 최적화에 의한 난독화 프로그램의 실행속도 향상을 볼 수 있다. 선형함수 난독화 기법을 적용한 프로그램은 최적화 과정을 거친 후 실행 속도가 빨라졌지만 여전히 원본 프로그램과 비교하면 속도가 오버헤드로 작용할 수 있다.

비트연산 난독화 기법을 적용한 프로그램이 최적화 과정을 거치면 실행 속도가 향상되어서 더욱 원본 프로그램에 근접한 성능을 보여준다.



(그림 20) 최적화된 난독화 기법의 실행 속도

프로그램		크기(byte)
변환되지 않은 상태		344
선형함수 난독화	기본	373
	최적화	361
비트연산 난독화	기본	784
	최적화	351

(그림 21) 최적화된 클래스 크기 비교

비트연산 난독화 기법의 최적화는 실행 속도로부터 얻는 이익과 함께 클래스 크기 감소로 얻을 수 있는 이익이 더 크다. (그림 21)에서와

같이 비트연산 난독화 기법은 최적화 과정을 거치고 나면 선형함수 난독화 기법보다 클래스 크기가 더 작아지는 것을 확인할 수 있다.

6. 결론 및 향후 연구 과제

6.1. 결론

자바 프로그램은 클래스 파일에 원본 자바 소스 프로그램의 심볼릭 정보를 그대로 유지하고 있어서 역공학 도구들에 의해서 쉽게 분석될 수 있다. 단순한 역컴파일러를 시작으로 수많은 자바 클래스 분석 도구들이 개발되어왔고, 이러한 도구를 이용하면 자바 클래스 파일로부터 원본 소스와 거의 동일한 소스 코드를 얻어낼 수 있다.

이러한 문제점을 방지하기 위해서 난독화라는 방법이 제안되었다. 난독화란 프로그램의 기능을 그대로 유지한채 소스코드를 해석하기 어려운 형태로 변환하는 방법이다. 난독화를 사용하면 프로그램이 역공학 도구들에 의해서 쉽게 분석되는 것을 방지할 수 있다.

지금까지 개발된 난독화 도구들은 의미있는 변수들의 이름을 무의미하게 바꾸는 레이아웃 변환을 중점적으로 다루어왔지만 프로그램에 포함된 알고리즘 구조 자체를 변경시키지는 못한다. 따라서 중요한 상수 데이터나 알고리즘을 보호하려고 할때 난독화를 적용하는 것은 무의미했다.

따라서 본 논문에서는 관련연구로서 선형 함수를 사용하여 데이터와 그 연산을 보호하는 난독화 기법을 소개하고, 선형 함수 난독화 기법의 단점인 실행 속도 저하와 클래스 크기 증가에 대한 단점을 보완한 비트 연산자를 사용한 연산 난독화 기법을 새롭게 제안했다.

비트 연산자를 사용해서 연산을 난독화하면 일반적인 레이아웃 난독화에 의해서 변환되지 않는 데이터와 연산들에 대해서 복잡한 난독화를 수행할 수 있다. 또한 후처리 과정으로서 연산의 최적화 과정을 제안하여 클래스 크기 증가와 프로그램 실행 속도 저하의 단점을 보완할 수 있게 하였다.

6.2. 향후 연구 과제

본 논문의 연구를 바탕으로 앞으로 해결해야 할 문제점을 세 가지로 요약하였다.

첫째, 연산 난독화에 기법에 의해서 변환된 연산에 있어서 오버플로우 발생 위험성에 대한 검증이 있어야 한다. 난독화에 의해서 오버플로우가 일어난다면 프로그램의 의미가 변경되는 것이기 때문에, 그것을 예측하여 데이터의 타입을 확장하거나, 오버플로우를 발생시키지 않는 난독화 규칙 BC(a,b)를 만들어야만 한다.

둘째, 현재는 하나의 메소드 안에서만 이루어지는 난독화를 고려했지만, 메소드 호출시에도 난독화된 데이터를 메소드 인자로 전달해주는 것도 가능하다. 이러한 경우에는 난독화된 실인자로부터 형식인자가 난독화되고 호출된 메소드에서도 난독화된 형식인자에 대한 연산이 난독화되어야 한다.

셋째, 클래스 필드의 난독화를 고려해야 한다. 클래스 필드는 클래스 메소드가 서로 공유하는 변수이다. 따라서 클래스 필드를 난독화하면 모든 메소드는 난독화된 필드에 대한 연산을 한다. 따라서 난독화된 필드임을 가정하고 클래스 메소드들이 호출되기 전에, 미리 필드에 대한 난독화 작업을 반드시 처리해야만 한다.

덧붙여, 정수형이 아닌 실수형 연산에 대한 난독화가 불가능하다는 점도 고려해야 할 사항이다.

참고문헌

[1] Christian Collberg, Clark Thomborson, Douglas Low, *A Taxonomy of Obfuscating Transformation*, Thechnical Report #148, Department of Comuter Science, The University of Auckland, 1997

[2] Christian Collberg, Clark Thomborson, Do

uglas Low, *Manufacturing Cheap, Resilient, a nd Stealthy Opaque Constructs*, SIGPLAN-SIG ACT POPL98, ACM Press , San Diego, C A, January 1998

[3] Christian Collberg, Clark Thomborson, Do uglas Low, *Breaking Abstractions and Unstruc tureing Data Structures*, IEEE International Conference on Computer Languages, ICCL 98

[4] Ahirotsugu S., Akito M., *Program obfuscation by coding data and its operation*, Dept. of Inf ormation Science, Nara Institute of Science and Technology, Japan, Symposium on Inf ormation Security Mar. 2002

[5] Pavel Kouznetsov, *Jad - the fast JAva Decom piler*, http://kpdus.tripod.com/jad.html

[6] Retrologic Systems, *RetroGuard*, http://ww w.retrologic.com/

[7] *Axil Ultra 1 Benchmarks*, http://www.protom atter.com/nate/java-optimization/yoda.html

[8] Joshua Engel, *Progrmmaing for the Java Virt ual Machine*, Addison-Wesley, 1999

[9] David Flanagan, *Java in a Nutshell*, O'Reilly, 1997

부록

1. BR(a,b) 연산들의 변환 규칙 유도

1.1 축약 가능한 연산들의 변환 규칙 유도

$BR_1(a,b) : X+Y \rightarrow (X \hat{} b)+(Y \hat{} b) \hat{} b$

$\{(x+y)<<a \hat{} b\}\{x=(X \hat{} b)>>a,\ y=(Y \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)+((Y \hat{} b)>>a))<<a \hat{} b$

$=\underline{((X \hat{} b)+(Y \hat{} b)) \hat{} b}$

$BR_2(a,b) : X-Y \rightarrow (X \hat{} b)-(Y \hat{} b) \hat{} b$

$\{(x-y)<<a \hat{} b\}\{x=(X \hat{} b)>>a,\ y=(Y \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)-((Y \hat{} b)>>a))<<a \hat{} b$

$=\underline{((X \hat{} b)-(Y \hat{} b)) \hat{} b}$

$BR_3(a,b) : X * Y \rightarrow (X \hat{} b) * (Y \hat{} b)>>a \hat{} b$

$\{(x * y)<<a \hat{} b\}\{x=(X \hat{} b)>>a,\ y=(Y \hat{} b)>>a\}$

$=(((X \hat{} b)>>a) * ((Y \hat{} b)>>a))<<a \hat{} b$

$=\underline{((X \hat{} b) * (Y \hat{} b))>>a \hat{} b}$

$BR_4(a,b) : X/Y \rightarrow (X \hat{} b)/(Y \hat{} b)<<a \hat{} b$

$\{(x/y)<<a \hat{} b\}\{x=(X \hat{} b)>>a,\ y=(Y \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)/((Y \hat{} b)>>a))<<a \hat{} b$

$=\underline{((X \hat{} b)/(Y \hat{} b))<<a \hat{} b}$

$BR_5(a,b) : X+c \rightarrow (X \hat{} b)+(c<<a) \hat{} b$

$\{(x+c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)+c)<<a \hat{} b$

$=\underline{((X \hat{} b)+c<<a) \hat{} b}$

$BR_6(a,b) : X-c \rightarrow (X \hat{} b)-(c<<a) \hat{} b$

$\{(x-c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)-c)<<a \hat{} b$

$=\underline{((X \hat{} b)-c<<a) \hat{} b}$

$BR_7(a,b) : c-X \rightarrow (c<<a)-(X \hat{} b) \hat{} b$

$\{(c-x)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(c-((X \hat{} b)>>a))<<a \hat{} b$

$=\underline{(c<<a-(X \hat{} b)) \hat{} b}$

$BR_8(a,b) : c * X \rightarrow c * (X \hat{} b) \hat{} b$

$\{(c * x)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(c * ((X \hat{} b)>>a))<<a \hat{} b$

$=\underline{(c} * \underline{(X \hat{} b)) \hat{} b}$

$BR_9(a,b) : X/c \rightarrow (X \hat{} b)/c \hat{} b$

$\{(x/c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=((X \hat{} b)>>a/c)<<a \hat{} b$

$=\underline{((X \hat{} b)/c) \hat{} b}$

$BR_{10}(a,b) : c/X \rightarrow c/(X \hat{} b)<<2 * a \hat{} b$

$\{(c/x)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(c/(X \hat{} b)>>a)<<a \hat{} b$

$=((c/(X \hat{} b))<<a<<a) \hat{} b$

$\underline{=((c/(X \hat{} b))<<2 * a) \hat{} b}$

$BR_{11}(a,b)$: $X<<c \rightarrow (X \hat{} b)<<c \hat{} b$

$\{(x<<c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)<<c)<<a \hat{} b$

$=\underline{(X \hat{} b)<<c \hat{} b}$

$BR_{12}(a,b)$: $X>>c \rightarrow (X \hat{} b)>>c \hat{} b$

$\{(x>>c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=(((X \hat{} b)>>a)>>c)<<a \hat{} b$

$=\underline{(X \hat{} b)>>c \hat{} b}$

1.2 축약 불가능한 연산들의 변환 규칙 유도

@ ∈ { ++, --, %, ~, !, &, ^, | }인 연산자 @에 대하여,

$BR(a,b): X@Y \rightarrow ((X \hat{} b)>>a)@((Y \hat{} b)>>a))<<a \hat{} b$

$\{(x@y)<<a \hat{} b\}\{x=(X \hat{} b)>>a,\ y=(Y \hat{} b)>>a\}$

$=\underline{(((X \hat{} b)>>a)\ @\ ((Y \hat{} b)>>a))<<a \hat{} b}$

$BR(a,b)$: $X@c \rightarrow (((X \hat{} b)>>a)\ @\ c)<<a \hat{} b$

$\{(x@c)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=\underline{(((X \hat{} b)>>a)@c)<<a \hat{} b}$

$BR(a,b)$: $c@X \rightarrow (c@((X \hat{} b)>>a))<<a \hat{} b$

$\{(c@x)<<a \hat{} b\}\{x=(X \hat{} b)>>a\}$

$=\underline{(c@((X \hat{} b)>>a))<<a \hat{} b}$

2. Spec JVM 98 벤치마크 프로그램에 대한 적용 예

2.1. Spec JVM 98 205_raytrace

2.1.1. Canvas.java 원본프로그램

```
class Canvas {
    private int Width;
    private int Height;
    private int[] Pixels;

    void SetPixel(int x, int y, int component)
    {
        int index = y * Width + x;
        Pixels[index] = (0x000000FF & Pixels[index])
                | (component << 8);
    }
}
```

2.1.2. RetroGuard에 의해서 난독화된 프로그램

```
class a {
    private int a;
    private int _fldo;
    private int _fldif[];

    void a(int i, int j, int k)
    {
        int l = j * a + i;
        _fldif[l] = 0xffff & _fldif[l] | k << 16;
    }
}
```

2.1.3. 비트 연산을 이용한 연산 난독화된 프로그램

```
class a {
    private int a;
    private int _fldo;
    private int _fldif[];

    void a(int i, int j, int k)
    {
        int l = 0x3fff6;
        int i1 = j * a + i;
        _fldif[i1] = (((_fldif[i1]^0xa)>>2&_fldif[i1]<< 2
            ^ 0xa | k << 16) ^ 0xa) >> 2;
    }
}
```

2.2. Spec JVM 98 201_compress

2.2.1. Decompressor.java 원본 프로그램

```
public Decompressor(In_Buffer in, Out_Buffer out)
{
    maxbits = Input.getbyte();
    block_compress = maxbits & compress.BLOCK_MASK;
    maxbits &= Compress.BIT_MASK;
    maxmaxcode = 1 << maxbits;
}
```

2.2.2. RetroGuard에 의해서 난독화된 프로그램

```
public a.a.a.a(g g1, e e1)
{
        _fldint = _fldbyte.a();
        a = _fldint & 0x80;
        _fldint &= 0x1f;
        _fldelse = 1 << _fldint;
}
```

2.2.3. 비트 연산을 이용한 연산 난독화된 프로그램

```
public a.a.a.a(g g1, e e1)
{
        _fldint = _fldbyte.a();
        _fldint = _fldint<<4 ^ 0xa; // BC(4,0xa)
        a = (_fldint^0xa) >> 4 & 0x80 ) ;
        _fldint = (_fldint^0xa) >> 4 & 0x1f;
        _fldelse = 1 << ((_fldint^0xa) >> 4);
        _fldint = (_fldint ^0xa) >> 4; // BD(4,0xa)
}
```


박 희 완

1993년~1997년 동국대학교 컴퓨터공학과(학사)
1997년~1999년 한국과학기술원 전산학과(석사)
1999년~현재 한국과학기술원 전자전산학과 전산학전공 박사과정
관심분야 : 소프트웨어 보호, 디버깅, 슬라이싱

최 석 우

1994년~1998년 한국과학기술원 전산학과(학사)
1998년~2000년 한국과학기술원 전산학과(석사)
2000년~현재 한국과학기술원 전자전산학과 전산학전공 박사과정
관심분야 : 최적화, 소프트웨어 보호, 지식기술언어

한 태 숙

1972년~1976년 서울대학교 전자공학과(학사)
1976년~1978년 한국과학기술원 전산학과(석사)
1990년~1995년 Univ. of North Carolina at Chapel Hill(박사)
1996년~현재 한국과학기술원 전자전산학과 부교수
관심분야 : 프로그래밍 언어론, 함수형 언어