韓國컴퓨터情報學會 論文誌
第16卷 第6號, 2011. 6.

2011-16-6-1-5

제어 흐름 난독화를 효과적으로 수행하기 위한 전략

김 정 일*, 이 은 주**

A strategy for effectively applying a control flow obfuscation to programs

Jung-il Kim* , Eun-joo Lee*

요 약

악의적인 소프트웨어 역공학으로부터 프로그램이 가지는 코드를 보호하기 위해서 코드 난독화가 제안되었다. 이것은 기존에 존재하는 프로그램 코드를 어렵게 변환시키는 것으로 프로그램 코드에 대한 악의적인 정적 분석을 어렵게 만든다. 코드 난독화는 난독화 목적에 따라 레이아웃, 데이터, 제어 난독화로 분류되어진다. 이 중 제어 난독화는 프로그램이 가지는 제어 흐름에 대한 추상적인 정보를 보호하는 것으로 다양한 종류의 개별 제어 흐름 난독화 변환이 제안되었지만, 이를 효과적으로 적용할 수 있는 방법은 제안되지 않았다. 본 논문에서는 제어 흐름 난독화 변환을 프로그램에 효과적으로 적용할 수 있는 난독화 전략을 제안하고, 실험을 통해서 제안한 난독화 전략의 효용을 보였다.

▸ Keyword : 소프트웨어 보안, 코드 난독화, 제어 흐름 난독화, 난독화 알고리즘

Abstract

Code obfuscation has been proposed to protect codes in a program from malicious software reverse engineering. It converts a program into an equivalent one that is more difficult to understand the program. Code obfuscation has been classified into various obfuscation technique such as layout, data, control, by obfuscating goals. In those obfuscation techniques, control

---

• 제1저자 : 김정일 교신저자 : 이은주
• 투고일 : 2011. 01. 25, 심사일 : 2011. 02. 14, 게재확정일 : 2011. 03. 17
* 경북대학교 대학원 전자전기컴퓨터(Graduate School of Electrical Engineering and Computer Science, Kyungpook Natioanl University) 박사과정
** 경북대학교 IT대학 컴퓨터학부(School of Computer Science and Engineering, College of IT Engineering, Kyungpook National University) 조교수
※ 이 논문은 2007년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업 연구임 (NRF-2007-331-D00407)

www.kci.go.kr

obfuscation is intended to complicate the control flow in a program to protect abstract information of control flow. For protecting control flow in a program, various control obfuscation transformation techniques have been proposed. However, strategies for effectively applying a control flow obfuscation to program have not been proposed yet. In this paper, we proposed a obfuscation strategy that effectively applies a control flow obfuscation transformation to a program. We conducted experiment to show that the proposed obfuscation strategy is useful for applying a control flow transformation to a program.

▸ Keyword : software security, code obfuscation, control flow obfuscation, obfuscation algorithm

Ⅰ. 서 론

소프트웨어 역공학(software reverse engineering)은 소프트웨어 개발 공정을 역으로 수행하는 것으로 코드 재구조화, 추상화 단계를 거쳐서 소프트웨어의 설계와 명세서를 추출해내는 프로세스이다. 일반적으로 소프트웨어 역공학은 자신이 소유권을 가진 소프트웨어에 대해서 수행하는 것이 원칙이지만, 타인 또는 경쟁사 프로그램의 내부 정보를 불법적인 방법으로 추출 및 변경하는 악의적인 목적으로도 사용되어진다[1]. 한 가지 예로 소프트웨어 크래킹(software cracking)에 의한 악의적인 역공학을 들 수 있다. 소프트웨어 크래킹은 소프트웨어에 존재하는 보호 루틴을 제거하여 소프트웨어에 대한 불법적인 사용 및 배포를 목적으로 하는 작업으로 역분석자는 소프트웨어 분석 도구를 이용하여 이진 실행 파일에서 코드를 추출한 후 크래킹을 위한 소프트웨어 분석을 수행한다.

악의적인 역공학으로부터 소프트웨어를 보호하기 위한 방법으로는 크게 법률적인 방법과 기술적인 방법으로 분류할 수 있다[2]. 법률적인 방법은 소프트웨어의 이용에 관한 배타적인 권리를 부여 받고, 타인에 의한 침해 행위에 대해 법적인 보호를 받을 수 있는 방법이다. 기술적인 방법으로는 암호화, 서버 측 실행, 코드 난독화 등과 같이 프로그램 코드를 직접적으로 보호하는 방법으로 역공학 작업을 어렵게 만든다.

여러 가지 보호 기법 가운데 코드 난독화(code obfuscation)는 소프트웨어 배포자가 악의적인 역공학으로부터 자신의 프로그램 코드를 보호하기 위해서 기능적으로 동일하면서 이해하기 어려운 코드로 변환하는 것이다. 이것은 악의적인 역공학 작업을 어렵게 만드는 여러 가지 기법 가운데 비용 대비 효과가 우수한 보호 기법에 속한다[1][2]. Colleberg 등은 코드 난독화를 변환 목적에 따라 레이아웃(layout), 데이터(data), 제어(control)와 같이 크게 3가지로 분류하였고, 자바(Java) 프로그램에 난독화를 적용하기 위한 난독화 도구인 난독기(obfuscator)를 제안했다[2][3][4].

제어 난독화는 프로그램의 실제 제어 흐름을 숨기는 것으로 프로그램이 가지는 제어 흐름 구조에 대한 정보를 보호하는 난독화이다[5]. 코드 난독화와 관련된 앞선 연구들에서는 여러 가지 개별 제어 난독화 변환들[2][3][4][5][6][7][8]이 제안되었다. Colleberg 등은 코드 사이에 더미 코드와 불분명 술어를 추가하여 프로그램에 대한 정적 제어 흐름 분석을 어렵게 만들어주는 분기삽입변환(Branch Insertion Transformation : 이후 BIT로 표기)을 제안했다[2][3]. 또한, [4]에서 역난독기의 정적 분석을 피할 수 있는 복잡한 구조의 불분명 술어를 생성하는 방법을 제시했다[4]. [5]에서는 코드 블록을 몇 개의 부분으로 분할하고, 분할된 코드 사이에 더미 코드와 원래 프로그램의 실행에 아무런 영향을 미치지 않는 제어문들을 추가하는 기본 블록 분할(Basic block fission obfuscation) 변환을 제시했다. [6]에서는 프로그램 코드들을 여러 개의 함수들로 나누어 정의하고, 조건문을 이용하여 정의된 함수를 함수 포인터 변수에 할당하여 실행하는 것으로 프로그램의 실행 흐름에 대한 분석을 어렵게 할 수 있는 제어 난독화를 제안했다. [7]에서는 자바와 같은 객체지향언어로 작성된 프로그램의 소스 코드에 대한 난독화를 위해서 객체지향 언어의 특징인 메소드 오버로딩(method overloading)을 기반으로 하는 방법으로 클래스에 포함된 메소드들의 이름을 동일하게 정의하고, 각 메소드가 가지는 매개변수에 따라 실행되어 지는 코드가 결정되는 형태의 난독화 변환을 제안했다. [8]에서는 복잡한 랜덤 수(random number)를 반복적으로 생성하고, 생성된 랜덤 수를 평가하여 코드의 실행을 결정하는 것으로 제어 흐름에 대한 분석을 어렵게 만드는 난독화 방법을 제시했다.

www.kci.go.kr

난독화와 관련된 앞선 연구들은 난독화 변환 방법만을 제안하고, 이를 프로그램에 효과적으로 적용하는 방법에 대해서는 제안하지 않았다. 여기서 말하는 효과적이라는 것은 정해진 횟수의 난독화 변환으로 결과 코드를 최대한 복잡하게 만드는 것을 의미한다. 코드 난독화를 적용하는 것은 프로그램이 가지는 악의적인 역분석에 대한 위험성을 줄일 수 있지만, 그대신 난독화로 인한 코드량의 증가로 인해 오버헤드가 증가한다[2]. 따라서, 오버헤드를 고려하여 난독화를 적용할 필요가 있다.

본 논문에서는 제어 난독화 가운데 [2][3][4][5] 등과 같이 계산 난독화로 분류되어진 난독화 변환을 효과적으로 적용할 수 있는 새로운 전략을 제안한다. 즉, 제안하는 난독화 전략은 불분명 술어를 기반으로 수행되는 난독화 변환을 프로그램에 적용하는데 유용하게 사용될 수 있다.

계산 난독화를 적용하기 위해서 먼저 난독화 대상 코드들을 다수의 코드 집합으로 분할하고 분할된 코드 영역 가운데 계산 난독화 변환이 적용되었을 때 코드가 가장 복잡해 질 수 있는 코드 영역에 계산 난독화 변환을 적용한다. 난독기는 이러한 전략을 이용하여 계산 난독화를 효과적으로 적용하는 것이 가능하다.

본 논문의 구성은 다음과 같다. 먼저 2장에서 코드 난독화와 제어 난독화에 관련된 연구들에 대해서 알아보고, 3장에서 난독화된 프로그램을 평가하는 메트릭과 제안하는 난독화 전략이 변환 위치를 결정하기 위해서 사용하는 메트릭에 대해서 설명한다. 4장에서는 제안하는 제어 흐름 난독화 전략에 대해서 알아본 후, 5장의 실험을 통해서 제안하는 난독화 전략의 효용을 알아본다.

II. 관련 연구

2.1 난독화 알고리즘

자바 프로그램은 다음과 같은 소스 코드 객체(source code object)들로 구성되어 있다[3].

- 응용프로그램
- 클래스
- 각 클래스의 필드
- 각 클래스의 메소드
- 각 메소드의 기본 블록

Collberg 등은 자바 코드의 난독화를 위한 자바 난독기(Java Obfuscator)인 Kava(Konfused Java)를 제안했다[2]. 그리고 난독기가 수행하는 난독화 알고리즘을 다음 <그림 1>과 같이 정의하고 있다.

```
WHILE NOT Done(A) DO
  S := SelectCode(A);
  T := SelectTransform(S);
  A := Apply(T,S);
END
```

그림 1. 일반적 난독화 알고리즘
Fig. 1. General obfuscation algorithm

여기서, A는 난독화 대상 프로그램(application), S(source code object)는 소스 코드 객체, T(transformation)는 난독화 변환이다. <그림 1>의 난독화 알고리즘은 먼저 A에 존재하는 난독화 대상 코드들 가운데 난독화를 적용할 소스 코드 객체를 선택한다. 적용 대상 코드의 선택은 프로그래머를 통해 직접적으로 또는 발견적인 방법(heuristic approach)들을 이용하여 결정될 수 있다. 그 다음, 여러 가지 난독화 변환 가운데 선택된 소스 코드 객체에 적용 가능한 난독화 변환을 선택하여 이를 대상 소스 코드 객체에 적용하는 방법으로 사용자가 요구하는 난독화 조건(프로그램의 복잡도 및 크기의 증가)을 만족할 때 까지 반복적으로 위의 절차를 수행한다.

2.2 제어 난독화

제어 난독화에 속하는 난독화 변환들은 변환 목적에 따라 집합 난독화, 계산 난독화, 순서 난독화로 분류된다[2]. 이 절에서는 본 논문과 관련된 계산 난독화에 속하는 난독화 변환에 대해서만 소개한다.

• 불분명 술어(opaque predicate)

불분명 술어(opaque predicate)는 계산 난독화에 속하는 제어 흐름 변환의 중요한 기본 요소이다[1][2]. 불분명 술어는 조건문의 결과가 참(true) 또는 거짓(false)만으로 평가되는 논리문이다. <그림 2>는 거짓으로 평가되는 불분명 술어의 기본적인 모델을 나타내고 있다.

www.kci.go.kr

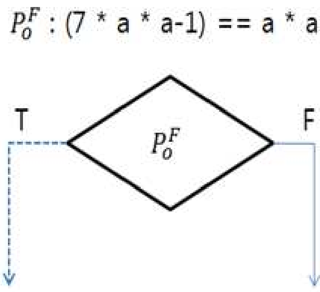

그림 2. 불분명 술어
Fig. 2. Opaque predicate

일반적으로 불분명 술어는 단독으로 쓰이지 않고, 더미 코드들과 함께 프로그램 코드에 추가되어 새로운 제어 구조를 생성하는 것으로 역분석자가 수행하는 제어 흐름에 대한 이해를 어렵게 만들 수 있다[2].

• 계산 난독화(Computation Obfuscations)

계산 난독화는 프로그램이 가지는 제어 흐름 구조를 숨기는 목적으로 실행 흐름에 영향을 주지 않는 더미 코드들을 프로그램 코드사이에 삽입한다[2]. <그림 3>은 일련의 연속된 코드들 사이에 불분명 술어를 기반으로 수행되는 계산 난독화를 적용한 예를 보인다. <그림 3>에서 P_o^F는 반드시 거짓으로 평가되는 불분명 술어, DS_i는 더미 코드, S_i는 프로그램 코드를 각각 나타낸다. <그림 3>의 예와 같이 계산 난독화는 불분명 술어를 기반으로 수행되며, 이와 관련된 대표적인 난독화 변환은 앞에서 언급했던 Colleberg 등이 제안한 BIT와 Low 등이 제안한 기본 블록 분할 변환이 있다. BIT는 일련의 연속된 코드들 사이에 불분명 술어와 더미 코드를 삽입하는 형태로, 기본 블록 분할 변환은 코드 블록을 분할하고 분할된 코드 사이에 불분명 술어를 추가하는 형태로 프로그램 코드에 대한 난독화 변환을 수행한다[2][5].

다른 난독화 변환들과 마찬가지로, 계산 난독화 변환들 역시 프로그래머에 의해서 수동으로 직접 프로그램에 적용하거나, <그림 1>의 난독화 알고리즘에 의해서 자동으로 적용될 수 있다. 수동으로 적용될 경우 중요하다고 판단되는 목적 코드 객체에 직접 난독화 변환을 적용할 수 있지만, 이것은 코드 난독화를 위해서 프로그래머에게 많은 노력을 필요로 한다. 반면 계산 난독화가 난독기의 난독화 알고리즘을 통해 자동적으로 적용될 경우 난독화 변환의 목적 코드 객체를 적절히 판단하고 적용할 수 있는 전략적인 방법이 필요하다.

본 논문에서 제안하는 난독화 전략은 계산 난독화 변환을 적용하기 위해 변환 대상이 되는 목적 소스 코드 객체들 중에 변환이 적용되었을 때 변환 후 가장 복잡한 결과를 초래할 수 있는 소스 코드 객체를 선택하는 방법으로 코드난독화 알고리즘은 이것을 이용해서 계산 난독화를 효과적으로 수행할 수 있다.

제안하는 난독화 전략에 대한 자세한 설명은 4장에서 알아보고, 다음 장에서는 제어 흐름 난독화 결과에 대한 평가와 제안하는 난독화 전략에서 목적 코드 객체들이 가지는 복잡도를 평가하기 위해 본 논문에서 사용되는 평가 메트릭들에 대해서 알아본다.

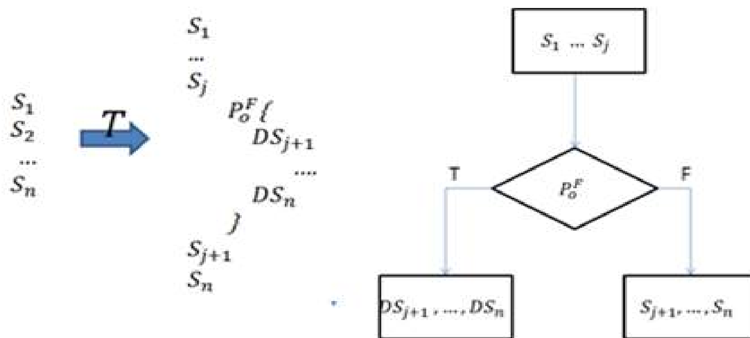

그림 3. 불분명 술어를 기반으로 수행되는 제어 흐름 난독화 변환의 예
Fig. 3. An example of applying control flow obfuscation transformation based on opaque predicate

III. 평가 메트릭

소프트웨어 복잡도는 대상 시스템의 유지 보수성을 평가하는 주요한 메트릭이지만, 코드 난독화에서는 난독화 결과의 효용을 평가하기 위해서 이용될 수 있다[2]. 복잡도 메트릭은 주로 절차적 및 객체지향 시스템의 평가를 위해서 제안

www.kci.go.kr

되었지만, 최근에는 웹 기반의 복잡도 메트릭 역시 증가하는 추세이다[9][10].

일반적으로 복잡도가 낮은 소프트웨어가 이해하기 쉽고, 유지보수성이 뛰어나기 때문에 높은 품질의 소프트웨어로 평가되지만, 코드 난독화에서는 반대의 기준으로 평가된다. 즉, 코드 난독화가 적용된 2개의 프로그램에 가운데 높은 복잡도를 가지는 프로그램이 높은 난독화 품질을 가진다고 평가된다.

난독화 품질에 대한 평가는 코드에 적용되는 난독화 변환의 종류에 따라 여러 가지 복잡도 메트릭으로 평가될 수 있다. 즉, 난독화 변환에 따라 측정 가능한 적절한 메트릭을 결정해야 하며, 일반적으로 제어 난독화는 프로그램이 가지는 제어 구조를 어렵게 만드는 것이 목적이기 때문에 이에 대한 평가는 제어 흐름 복잡도 메트릭을 이용한다[11].

이 장에서는 본 논문에서 사용되는 두 가지 복잡도 메트릭을 에 대해서 설명한다.

• N-Scope 복잡도 메트릭

[11]에서는 여러 가지 제어 흐름 복잡도 메트릭 가운데 N-Scope 복잡도을 이용하여 제어 흐름 난독화 변환의 품질을 평가하였다.

N-Scope 복잡도 메트릭은 제어 흐름 그래프가 가지는 여러 가지 제어문의 중첩 수준을 측정 하기 위해 제안된 메트릭으로 [11]에서는 다음 <식 1>과 같이 나타내고 있다.

$$ns(g) = \sum_{B_i \in B} \frac{|R(g,B_i)|}{|R(g,B_i)| + |NC|} \qquad \text{식 1}$$

여기서,

g : 제어 흐름 그래프

B_i : 분기 노드

B : g 가 가지는 분기 블록 집합

$|NC|$: g 안의 모든 노드의 수

$|R(g,B_i)|$: B_i 의 중첩 수준, 중첩 수준은 가 가지는 경로의 수 또는 루프 내부의 노드의 수

위의 <식 1>은 제어 흐름 그래프(g)가 가지는 분기 노드의 숫자와 위치에 따라서 결정된다. 만약 제어 흐름 그래프에 다수의 분기 노드들이 존재하고, 그들이 서로 중첩되었을 경우 N-Scope 복잡도는 높은 값을 가지게 된다. 반대로, 제어 흐름 그래프에 분기 노드가 존재하지 않는다면 N-Scope 복잡도 값은 0의 값을 가진다.

본 논문의 실험에서도 제어 흐름 난독화 변환 결과에 대한 효용을 위의 <식 1>로 평가한다.

• 엔트로피 기반의 복잡도 메트릭

샤논의 엔트로피는 정보가 가지는 정보량을 정의하기 위한 방법으로 프로그램의 복잡도를 측정하는데 유용하게 사용될 수 있다[12].

[12]에서는 일련의 연속된 코드들을 노드와 간선을 가지는 방향 그래프로 표현하고, 그래프 내부의 각 노드가 가지는 이웃관계, 내부, 외부 간선의 수를 기준으로 그래프가 가지는 노드 집합들을 결정하고, 이렇게 결정된 노드 집합을 다음 <식 2>를 이용해서 그래프의 복잡도를 측정하는 방법을 제안했다[12].

$$H(G) = -\sum_{i=1}^{n} P(A_i)\log P(A_i) \qquad \text{식 2}$$

여기서, G : 그래프

A_i : G가 가지는 노드들의 집합

$P(A_i)$: A_i가 가지는 노드의 수를 G에 있는 전체 노드의 수로 나눈 값.

만약 두 개의 그래프가 가지는 노드와 간선의 수가 동일하더라도, 그래프의 구조가 다르다면 이 두 개의 그래프는 동일한 복잡도를 가진다고 볼 수 없다. 따라서, <식 2>는 그래프 구조의 복잡도를 측정하는데 유용하게 사용될 수 있다.

본 논문에서 제안하는 난독화 전략에서는 <식 2>를 난독화 변환 위치를 결정하기 위해서 이용한다.

Ⅳ. 제어 흐름 난독화 전략

• 제어 흐름 난독화 전략의 필요성

<그림 4>는 계산 난독화에 속하는 여러 가지 난독화 변환 가운데 하나인 BIT를 수행한 2가지 경우를 보인다.

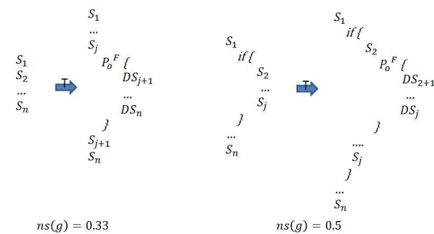

그림 4. 2가지 BIT 적용 예
Fig. 4. two examples of BIT

<그림 4>의 왼쪽은 일련의 연속된 코드 사이에 BIT를 적용한 결과이고, 오른쪽은 하나의 분기문으로 이루어진 코드

www.kci.go.kr

사이에 BIT를 적용한 결과이다. 이 경우, 오른쪽의 난독화 결과가 왼쪽의 난독화 결과 보다 더 높은 제어 흐름 복잡도 값을 가지는 것을 볼 수 있다. 즉, 계산 난독화에 속하는 난독화 변환은 변환 적용 위치에 따라서 변환 후 코드가 가지는 가독성에 차이를 가진다. 그러므로 제어 흐름 난독화 변환을 효과적으로 적용하기 위해서는 코드 상에서 적절한 변환 위치를 결정하는 방법이 필요하다.

• 코드 영역 분할

제어 흐름 난독화 변환의 난독화 대상 소스 코드 객체는 함수 내부에 존재하는 단일 기본 블록(basic block)이다 [2][3][11]. 따라서, 코드 영역 분할은 단일 함수를 기준으로 함수 내부에 존재하는 코드들을 대상으로 수행된다. 분할 방법은 다음과 같다. 만약 함수 내부에 여러 개의 분기들이 존재할 경우, 이것들 가운데 최상위 분기들을 영역 분할 기준으로 결정하고, 그 외의 분기들은 무시한다. <그림 5>는 3개의 분기들로 이루어진 함수의 분할 예를 보인다. 여기서, 2개의 분기(if, for)가 최상위 분기에 속하므로, 이들을 기준으로 코드 영역을 나누고, 나누어진 코드 영역을 다음과 같이 제어 흐름 난독화 후보 영역으로 정의한다.

[정의 1] 제어 흐름 난독화 후보 영역

$$f = \{sub_1, sub_2, \ldots, sub_n\}$$

여기서,

f : 함수

sub_i : 난독화 후보 영역

n : 전체 난독화 후보 영역의 수

함수가 가지는 코드 구조에 따라 난독화 후보 영역의 수는 다르며, 각 난독화 후보 영역은 1개 이상의 난독화 대상 코드들, 즉 1개의 기본 블록을 가진다.

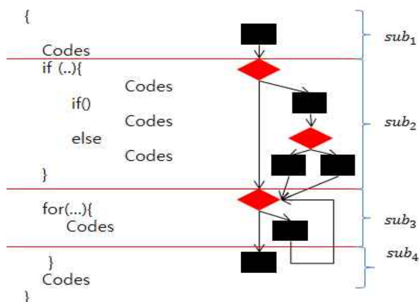

그림 5. 최상위 분기로 구분된 코드 영역들
Fig. 5. Sub areas divided by top level branch

• 변환 적용 위치 선택

<그림 4>에서 알아본 바와 같이 함수가 가지는 제어 구조에 대한 복잡도는 제어 흐름 난독화 변환의 적용 위치에 따라 결정된다. 따라서, 난독화 변환 후 함수가 가지는 제어 흐름 복잡도의 증가량을 최대한으로 높이기 위해서 분할된 난독화 후보 영역들 가운데 가장 복잡한 코드 영역에 난독화 변환을 적용할 필요가 있다. 즉, 단일 함수가 가지는 n개의 변환 대상 후보 영역 가운데 가장 복잡한 위치에 적용되어야 한다. 또한, 제어 흐름 변환이 동일한 코드 영역에 연속적으로 적용될 경우 함수 전체에 대해서 제어 흐름에 대한 난독화를 보장하지 못하는 문제를 고려할 필요가 있다.

결론적으로, 제어 흐름 난독화 변환을 적용할 위치를 선택하는 기준을 다음 두 가지 요소로 결정한다.

- 후보 영역이 가지는 복잡도
- 후보 영역에 적용된 난독화 비율

<그림 5>와 같이 분할된 각 후보 영역은 노드와 간선으로 이루어진 부분 그래프로 표현된다. 따라서 각각의 부분 그래프가 가지는 구조를 기준으로 부분 그래프의 복잡도를 결정할 필요가 있다. 여기서는 각 부분 그래프에 대한 복잡도를 2장에서 알아본 <식 2>를 이용하여 결정한다.

2장에서 기술한 바와 같이, 제어 흐름 난독화는 변환 위치에 불분명 술어와 임의의 더미 코드들을 추가하여 불분명한 코드 구조를 생성하는 것으로 제어 흐름 난독화를 수행한다. 즉, 더미 코드가 많이 존재하는 코드 영역은 다른 코드 영역에 비해 난독화 변환이 많이 적용되었다는 것을 의미하며, 높은 난독화 적용 비율을 가진다고 볼 수 있다. 따라서, 후보 영역에 대한 난독화 비율은 코드 영역이 가지고 있는 원본 코드와 난독화로 인해 추가된 더미 코드의 양으로 평가하고, 다음 <식 2>로 정의한다.

[정의 2] 난독화 비율

$$Obfratio(sub_i) = \frac{OrigCodes}{OrigCodes + DummyCodes} \quad \text{식 3}$$

여기서,

sub_i : 부분 영역

$OrigCodes$: 원본 코드

$DummyCodes$: 더미 코드

임의의 부분 영역에 난독화가 적용될수록 그 부분 영역이 가질 수 있는 난독화 적용 확률은 낮아져야 한다. 따라서, <식 3>은 난독화가 적용될수록 낮은 값을 가지게 되고, 이것은 <식

www.kci.go.kr

2>로 측정할 수 있는 부분 영역의 복잡도와 함께 각 부분 영역이 가질 수 있는 난독화 우선 순위를 결정하는데 사용된다.

[정의 3] 부분 영역 우선순위

$$Priority(sub_i) = H(sub_i) \times ObfRatio(sub_i) \quad \text{식 4}$$

여기서,

$H(sub_i)$: 코드 영역이 가지는 복잡도[12]

$ObfRatio(sub_i)$: 코드 영역에 적용된 난독화 비율

최종적으로 제어 흐름 변환은 가장 높은 우선 순위 값을 가지는 난독화 후보 영역에 적용된다.

다음 <정의 4>는 지금까지 설명한 난독화 전략의 의사코드이다.

[정의 4] 제어 흐름 난독화 전략

```
- TF: 대상 함수
- T: 난독화 변환 (본 논문에서는 BIT를 이용)

divide TF into subareas SUB={sub1, sub2, ..., subm}
select sub_j = argmax_{sub_k ∈ SUB} Priority(sub_k), 1 ≤ k ≤ m
apply T into sub_j
```

난독화 전략은 먼저 최상위 분기를 기준으로 대상 함수에 존재하는 코드들을 m개로 분할한다. 그 다음 <정의 3>의 <식 3>으로 각각의 코드 영역이 가지는 우선순위를 계산하고, 가장 높은 우선순위 값을 가지는 코드 영역을 선택하여 선택된 코드 영역에 제어 흐름 변환을 적용한다. 이 전략은 제어 흐름 난독화 변환을 적용하기 위한 난독화 알고리즘의 한 부분으로 사용될 수 있다. 예를 들어, 2.2절에서 알아본 일반적인 난독화 알고리즘에서 BIT를 적용하기 위해서 이 전략을 이용할 수 있다. 즉, 특정 평가 메트릭을 사용하여 그 값이 더 이상 향상되지 않을 때까지 사용한다거나, 지정된 회수만큼 반복할 때 적용된다거나 하는 다양한 제어 난독화 프로세스의 일부로 사용할 수 있다.

<그림 6>은 실험 프로그램인 cflow[13]에 포함된 여러 사용자 정의 함수들 가운데 expand_argcv 함수를 대상으로 수행된 난독화 결과의 예를 추출한 제어 흐름 그래프를 통해 보여준다. <그림 6>에서 "B"는 분기, "S"는 기본 블록, "D"는 변환으로 추가된 더미 코드들을 각각 나타내며, 가)는 변환 전, 나)는 변환 후에 가지는 제어 흐름 그래프이다. expand_argcv 함수는 코드 구조에 따라 2개의 코드 영역 (sub_1, sub_2)으로 구분되고, 그 중 sub_2의 우선순위 값이 sub_1의 우선순위 값보다 높게 평가되어 sub_2에 제어 흐름 변환이 적용되었다.

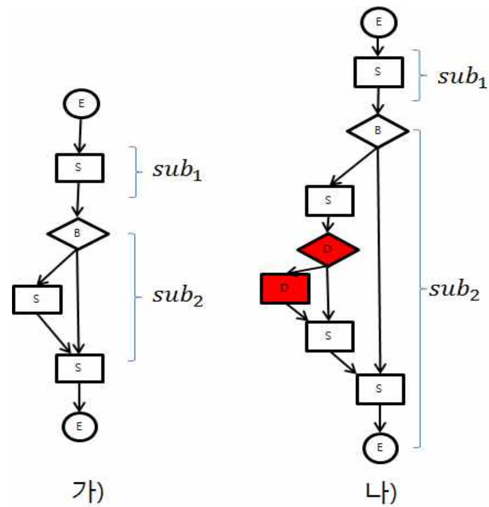

그림 6. expand_argcv의 난독화 적용 예
Fig. 6. Example of applying obfuscation to expand_argcv.

V. 실험 및 평가

본 논문에서 제안하는 난독화 전략의 효용은 C로 구현된 cflow를 대상으로 실험을 통해서 알아본다. 실험은 Windows 7, Core2 Duo CPU 2.40Ghz memory 2G 의 환경에서 수행하였다. cflow 는 리눅스 오픈소스 프로그램으로 프로그램이 가지는 정적 호출 정보를 기반으로 호출 그래프 정보를 추출하는 분석 도구이다.

실험의 목적은 제안하는 난독화 전략의 효용을 알아보는 것이기 때문에 아무런 방법론 없이 수행한 난독화 결과와 본 논문에서 제안하는 전략으로 수행한 난독화 결과의 비교를 통해서 평가한다. 평가는 두 가지 방법으로 수행한 BIT의 결과로 제어 흐름 그래프가 가지는 복잡도를 N-Scope 복잡도 메트릭으로 평가하여 비교한다.

실험 프로그램이 가지는 제어 흐름 정보는 understand 2.6[14]을 이용하여 추출하였다. understand는 크로스 플랫폼, 다중 언어, 유지보수 기반의 통합 개발 환경(IDE)로써, 개별 소스코드는 물론 프로젝트 단위의 큰 소스 코드를 수정하거나, 이해하는데 사용되는 코드 분석 도구이다.

<표 1>은 무전략(랜덤 수행)과 제안한 방법(전략적 방법)으로 수행한 BIT 결과로 cflow가 포함하는 각 함수가 가지는 제어 흐름 그래프의 N-Scope 값이 변환 전 복잡도이며,

www.kci.go.kr

이후 각각 랜덤 및 제안된 방법으로 5회 적용한 후의 변환 후 복잡도를 보여준다.

표 1. 무전략과 전략적 방법의 난독화 수행 결과 제어 흐름 복잡도
Table 1. $ns(g)$ of random and strategic obfuscation

함수명	복잡도($ns(g)$)			함수명	복잡도($ns(g)$)		
	변환전	랜덤	전략적 방법		변환전	랜덤	전략적 방법
dcl	0.407	0.631	0.687	print_level	0.259	0.551	0.551
declare_type	0.4	0.602	0.686	print_symbol	0.391	0.629	0.629
expand_argcv	0.167	0.39	0.679	print_symbol_type	0.25	0.632	0.656
fake_struct	0.636	0.864	0.871	putback	0.313	0.527	0.6
gnu_output_handler	0.462	0.592	0.592	restore	0.111	0.574	0.603
init	0.333	0.567	0.567	skip_struct	0.608	0.818	0.822
init_parse	0	0.5	0.5	source	0.3	0.486	0.591
main	0.479	0.522	0.552	source_rc	0.358	0.484	0.505
newline	0	0.483	0.483	tokpush	0.111	0.425	0.452
output	0.387	0.5	0.595	xref_output	0.409	0.582	0.627
output_init	0	0.5	0.5	yyparse	0.5	0.584	0.61
parse_declaration	0.25	0.604	0.656	parse_dcl	0.182	0.5	0.586
parse_typedef	0.222	0.547	0.67	parse_declaration_free	0.182	0.4	0.4
parser_rc	0.517	0.667	0.731	scan_tree	0.294	0.55	0.585
posix_output_handler	0.389	0.587	0.587	yy_create_buffer	0	0.55	0.55
pp_option	0.417	0.69	0.714	yy_flex_alloc	0.25	0.656	0.677
print_function	0.167	0.459	0.556	-	-	-	-

그 결과, 난독화 결과 33개의 함수들 가운데 23개가 랜덤한 방법보다 전략적 방법으로 BIT를 적용했을 때 더 높은 복잡도 값을 가지는 것을 보였다. 그 외 10개의 함수는 무전략(랜덤) 및 전략적 방법의 결과가 같았다. 그 이유는, 해당 10개의 함수 내부에 BIT변환 적용이 어려웠기 때문이다. 구체적으로 살펴보면, 분기는 존재하지만 내부 코드 량이 작거나(gnu_output_handler외 5개), 분기가 존재하지 않는 경우들(init_parse외 3개)이었다. 여기서 분기가 존재하지 않는 경우의 복잡도는 N-scope 복잡도의 특성상 0이 된다.

<표 2>는 각각의 방법의 평균 복잡도 증가율을 보이고 있다. 변환 전 0의 복잡도 값을 가지는 함수들은 유효한 증가량을 가지지 않기 때문에 계산 대상에서 제외시켰다. 즉 전체 33개의 함수들 중 29개의 유효한 복잡도 증가량을 가지는 함수들을 대상으로 "복잡도 증가율 1"은 변환 후 동일한 복잡도 증가율을 가지는 함수들을 모두 포함한 경우(29개)의 변환 전 대비 평균 복잡도 증가율이고, "복잡도 증가율 2"는 변환 후 동일한 복잡도 증가율을 가지는 함수를 제외한 경우(23개)의 평균 복잡도 증가율이다.

결론적으로 <표 2>에서와 같이 전략적 방법이 랜덤한 방법에 비해 각각 20%, 26%의 향상된 복잡도 증가를 보인다. 따라서 제안된 난독화 전략을 적용 시에, 무전략으로 수행하는 경우에 비하여 전체적으로 프로그램의 평균 제어 흐름 복잡도가 더 높아진다고 볼 수 있다.

표 2. 랜덤 & 전략적 방법의 평균 복잡도 증가율
Table 2. Average complexity increasing rate of random and strategic obfuscation.

	랜덤	전략적 방법
복잡도 증가율 1	98%	118%
복잡도 증가율 2	104%	130%

VI. 결 론

프로그램이 가지는 제어 흐름에 대한 추상적인 정보는 프로그램을 이해하는데 중요한 요소이다. 코드 난독화에서는 실행과 관계없는 새로운 제어 구조를 프로그램에 추가하는 것으로 프로그램이 가지는 제어 흐름에 대한 악의적인 정적 분석을 어렵게 하는 방법인 제어 흐름 난독화 변환을 제안했다. 하지만, 제어 흐름 난독화 변환을 효과적으로 적용하는 방법론은 제시되지 않았다. 본 논문에서는 제어 흐름 난독화 변환을 효과적으로 적용하기 위한 새로운 난독화 전략을 제안했다. 새롭게 제안하는 난독화 전략은 함수에 존재하는 코드들을 최상위 분기들을 기준으로 n 개의 코드 영역으로 분할하고, 이것들을 난독화 후보 영역으로 결정한다. 그리고 분할된 코드 영역들이 가지는 복잡도와 난독화 비율을 고려

www.kci.go.kr

하여 변환 우선 순위를 결정하고, 가장 높은 우선 순위를 가지는 코드 영역에 난독화 변환을 적용한다.

실험에서는 제안하는 난독화 전략과 무전략을 이용하여 BIT를 수행한 결과를 통해 제안한 난독화 전략의 효용을 보였다. 실험 결과 기본적으로 단순한 제어 구조 및 코드량이 적은 함수들을 제외하고 새롭게 제안하는 난독화 전략이 무전략으로 수행하는 방법보다 더 높은 제어 흐름 복잡도 값을 가지는 것을 보였고, 실험 프로그램이 가지는 평균 복잡도 증가율 또한 20%, 26% 향상된 결과를 보였다.

본 논문에서 제안하는 난독화 전략은 분할된 코드 영역 전체에 대해서 난독화 우선 순위를 평가하고 적용 유무를 결정했다. 그렇지만, 분할된 코드 영역이 가지는 기본 블록들 사이의 연산자, 데이터 등을 분석하고, 이들 사이의 난독화 우선 순위를 결정할 수 있다면 더 나은 난독화 결과를 기대할 수 있을 것으로 예상된다. 또한 현재 제어 구조 상의 최상위 분기를 대상으로 적용하였으나, 이후 분기의 계층 구조까지 고려하여 보다 정교하게 적용할 수 있을 것이다. 향후 연구로는, 기술한 바와 같이 연산자 및 데이터 분석 결과와 분기 계층 구조를 활용하도록 하며, 최종적으로는 제어 난독화에서 계산 난독화 이외의 기법까지 고려한 제어 난독화 프레임워크로 확장할 것이다.

참고문헌

[1] E. Eilam, "Reversing: Secrets of Reverse Engineering," Wiley Publishing, Inc., pp. 327-357, Apr. 2005.

[2] C.Colleberg, C.Thomborson, D.Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, University of Auckland, Jul. 1997.

[3] D. Low, "Java control flow obfuscation," Master's Thesis, Department of Computer Science, University of Auckland, New Zealand, Jun. 1998.

[4] C.Collberg, C.Thomborson, and D.Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," In Principles of Programming Languages 1998, POPL'98, San Diego, CA, Jan. 1998.

[5] T. Hou, H. Chen, and M. Tsai, "Three control flow obfuscation methods for Java software,"Proc. Inst. Elect. Eng. Software, vol. 153, no. 2, pp. 80 - 86, Jan. 2006.

[6] T. Ogiso, Y.Sakabe, M.Soshi, and A. Miyaji, "Software obfuscation on a theoretical basic and its implementation," IEICE Trans. Fundamentals, vol. E86-A(1), no.1, pp.176-186, Jan. 2003.

[7] Y. Sakabe, M. Soshi, A.Miyaji, "Java Obfuscation Approaches to Construct Tamper-Resistant Object-Oriented Programs," IPSJ Digital Courier, vol.1, pp. 134-146, Dec. 2005.

[8] T. Toyofuku, T. Tabata, and K. Sakurai, "Program obfuscation scheme using random numbers to complicate control flow," EUC Workshops, IEIC Tech. Rep, pp. 916-925, Jan. 2005.

[9] Woosung Jung, Eunjoo Lee,"A Structural Complexity Metric for Web Application based on Similarity," Journal of the Korea Society of Computer and Information, vol.15, no.8, pp.117-126, Aug. 2010.

[10] Sungkyun Oh, Mijin Kim, " A Study of Estimation for Web Application Complexity," Journal of the Korea Society of Computer and Information , vol.9, no.3, pp.27-34, Sep. 2004.

[11] H.Yi Tsai, Y.Lun Huang, D.Wagner, "A graph approach to quantitative analysis of control-Flow obfuscating transformations," IEEE Transactions On Information Forensics and Security, vol.4, pp 257-267, Jun. 2009

[12] J. S. Davis, R. J. Leblanc, "A Study of the Applicability of Complexity Measures," IEEE Transactions on Software Engineering archive, vol. 14, No.9, pp. 1366-1372, Sep. 1988.

[13] GnuCflow, http://www.gnu.org/software/cflow/

[14] SciTools, http://www.scitools.com/

www.kci.go.kr

저 자 소 개

김 정 일

2009-2011 : 경북대학교 대학원 전자전기컴퓨터 (석사과정)

2011-현재 : 경북대학교 대학원 전자전기컴퓨터(박사과정)

관심분야 : 소프트웨어 분석, 설계 및 개발 방법론.

Email : 2009307043@knu.ac.kr

이 은 주

2005 : 서울대학교 전기컴퓨터공학부 (공학박사)

2006-현재 : 경북대학교 IT대학 컴퓨터학부 조교수

관심분야 : 웹공학, 재공학, 메트릭, 소프트웨어 유지보수

Email : ejlee@knu.ac.kr

www.kci.go.kr