韓國IT서비스學會誌
第10卷 第3號
2011年 9月

유전 알고리즘에 기반한 코드 난독화를 위한 인라인 적용 기법*

김정일** · 이은주***

A Technique to Apply Inlining for Code Obfuscation based on Genetic Algorithm*

Jung-Il Kim** · Eun-Joo Lee***

■ Abstract ■

Code obfuscation is a technique that protects the abstract data contained in a program from malicious reverse engineering and various obfuscation methods have been proposed for obfuscating intention. As the abstract data of control flow about programs is important to clearly understand whole program, many control flow obfuscation transformations have been introduced. Generally, inlining is a compiler optimization which improves the performance of programs by reducing the overhead of calling invocation. In code obfuscation, inlining is used to protect the abstract data of control flow. In this paper, we define new control flow complexity metric based on entropy theory and N-Scope metric, and then apply genetic algorithm to obtain optimal inlining results, based on the defined metric.

Keyword : Code Obfuscation, Genetic Algorithm, Inline, Complexity Metric

논문투고일 : 2011년 01월 19일 논문수정완료일 : 2011년 02월 28일 논문게재확정일 : 2011년 03월 16일

* 이 논문은 2010 년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임 (2010-0002820).

** 경북대학교 대학원 전자전기컴퓨터 박사과정

*** 경북대학교 IT대학 컴퓨터학부 조교수, 교신저자

1. 서 론

소프트웨어 역공학(reverse engineering)은 합법적으로 소유권을 가지고 있는 프로그램의 성능을 개선하기 위해서 수행되는 분석 기법이지만, 타인의 프로그램 코드를 추출하여 무단으로 수정 및 재사용하기 위한 악의적인 목적으로 사용되기도 한다. 대표적인 역공학 사례로 소프트웨어 크래킹(software cracking)을 생각할 수 있다. 소프트웨어 크래킹은 코드를 보호하기 위해서 프로그래머가 프로그램 내부에 설계한 보호 루틴을 제거하고, 데이터 및 특별한 알고리즘 등과 같은 프로그램 내부에 있는 중요한 정보들을 무단으로 조작 및 획득하는 악의적인 역공학 행위이다[11].

이와 반대로 코드 난독화(code obfuscation)는 프로그램의 기능은 동일하게 하면서 코드 구조를 복잡하게 만드는 것으로 악의적인 역공학 작업을 어렵게 만드는 보호 기법이다[2]. Colleberg 등은 난독화 목적에 따라서 수행될 수 있는 난독화 종류를 레이아웃, 데이터, 제어로 분류하였다[6]. 레이아웃 난독화는 프로그램에 존재하는 식별자 및 디버깅 정보를 변경하거나 삭제하여 이에 대한 정보를 쉽게 획득하지 못하게 하는 난독화이다. 데이터 난독화는 프로그램에서 사용하는 데이터 구조들을 변경하는 것으로 데이터 구조에 대한 분석을 어렵게 하는 난독화이다. 마지막으로 제어 난독화는 프로그램의 제어 구조를 변경하여 프로그램이 가지는 실행 흐름에 대한 이해를 어렵게 하는 난독화이다.

일반적으로 함수 인라인(function inline)은 함수 호출로 인해 발생되는 실행시간 오버헤드를 감소시켜 프로그램의 성능을 향상시키는 목적으로 컴파일러 최적화 기법으로 사용되지만[5, 10], 인라인으로 인해 프로그램으로부터 추상적인 함수 호출 정보를 제거하는 것이 가능하기 때문에 프로그램의 난독화를 위해서 유용하게 사용될 수 있다[6]. Colleberg 등은 인라인 변환을 집합 변환으로 분류하고, 이를 프로그램이 가지는 제어 구조를 보호하기 위해 사용할 수 있는 것을 제안했다.

[6]에서는 코드 난독화를 위해 사용될 수 있는 일반적인 난독화 알고리즘이 제안되었으나, 인라인 변환을 효과적으로 적용하는 방법을 고려하지 않았다. 여기서 인라인 난독화를 효과적으로 적용한다는 것의 의미는 컴파일러가 프로그램 성능 향상을 위해서 함수 인라인을 효과적으로 적용하는 것과 정반대의 관점으로 볼 수 있다. 즉, 함수 인라인으로 프로그램 코드를 최대한 어렵게 만드는 것이 목적이다. 이것은 최적화를 위해 함수 인라인을 수행하는 것과 반대의 목적을 가지지만 유사한 크기의 문제 범위를 가진다. [5, 10]에서는 프로그램 성능 향상을 위해 가질 수 있는 인라인 조합이 프로그램에 존재하는 함수의 수에 따라서 지수적인 크기를 가진다고 보고, 유전 알고리즘을 이용하여 최적의 인라인 조합을 결정하는 것을 제안했다. 이처럼 목적은 다르지만, 코드 난독화에서 프로그램을 복잡하게 만들기 위해 가질 수 있는 인라인 변환 조합 또한 이와 동일한 크기를 가진다고 볼 수 있다. 따라서 여기서는 난독화를 목적으로 최적의 인라인 조합을 찾기 위해서 유전 알고리즘을 사용하는 것을 제안하고, 난독화의 목적에 맞게 염색체 및 평가 함수를 새롭게 정의한다. 또한 인라인의 결과에 대한 효용을 평가할 수 있는 정량적인 방법이 필요한데, 본 논문에서는 기존에 제안되었던 복잡도 메트릭들을 기반으로 새로운 제어 흐름 복잡도 메트릭을 정의한다.

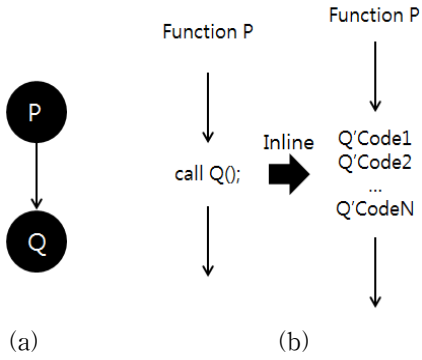
본 논문의 구성은 다음과 같다. 제 2장에서는 인라인 변환과 일반적으로 난독화 효용을 평가하기 위해서 사용되는 복잡도 메트릭에 관련된 연구들에 대해 알아보고, 제 3장에서는 난독화 효용을 평가하기 위해서 본 논문에서 새롭게 제안하는 제어 흐름 복잡도 메트릭을 설명한다. 제 4장에서는 본 논문의 핵심인 유전 알고리즘을 이용한 인라인 변환 적용 방법을 설명하고 제 5장에서 실험 및 결과를 보인다. 마지막 제 6장에서는 결론 및 향후 연구를 제시한다.

2. 관련 연구

다양한 종류의 난독화 변환들이 제안되었지만, 여기서는 본 논문의 주제와 관련된 인라인 난독화 변환에 대해서만 알아본다. 그리고 함수 인라인을 이용하여 프로그램 최적화를 수행하기 위해 유전 알고리즘을 이용한 관련 연구들에 대해서 알아본다. 마지막으로 난독화의 효용을 평가하기 위해 사용될 수 있는 복잡도 메트릭에 관련된 앞선 연구들에 대해서 알아본다.

2.1 인라인 변환(Inline Transformation)

코드 난독화에서 인라인은 프로그래머가 프로그램 작성 중에 생성한 하이 레벨 구조를 파괴하여 프로그램 코드의 하이레벨 구조를 더 이상 의미 없게 만드는 목적으로 수행되는 난독화 변환이다[6]. 즉 인라인 변환의 결과로써 코드 상에 존재하는 함수 호출 문장이 함수가 가지는 내부 코드들로 치환되고, 치환된 함수에 대한 정보는 프로그램에서 제거되기 때문에 인라인을 프로그램에 존재하는 절차적 추상화(procedural abstraction)를 제거할 수 있는 난독화 변환으로 정의하고 있다[6]. 예를 들어 [그림 1](a)와 같이 임의의 두 함수 P, Q가 있고, 함수 P가 함수 Q를 호출 하는 호출 관계를 가지고 있을 때 Q함수가 인라인 된다면 [그림 1](b)와 같이 Q의 코드는 P 함수 내부에 삽입되고,



[그림 1] 인라인 변환 결과

함수 P와 함수 Q 간의 호출 관계는 제거된다. 또한 인라인 결과로 추가되어진 코드들이 원래 Q 함수의 코드라는 것을 추적하는 것은 불가능하기 때문에 더 이상 함수 P와 함수 Q의 호출 관계에 대한 추상적인 정보는 알 수 없다.

2.2 인라인 최적화

일반적으로 함수 인라인은 프로그램의 성능을 향상시키기 위해서 수행되는 컴파일러 최적화 기법이다. 대부분의 컴파일러에서 인라인은 프로그래머가 각각의 함수들에게 키워드('inline')를 명시적으로 지정하거나, 인라인 조건으로 함수가 가지는 코드 크기와 호출 횟수에 대한 임계치를 설정하는 것으로 수동적 또는 자동적으로 대상 함수들에 대해서 인라인을 수행한다. 함수 인라인은 함수 호출 문장을 함수 내부의 코드들로 치환하기 때문에 코드 증가에 대한 프로그램의 오버헤드를 초래할 수 있다. 따라서 성공적으로 인라인을 수행하기 위해서는 프로그램에 존재하는 인라인 가능한 모든 함수들의 코드 량과 호출 횟수를 고려하여 인라인을 수행해야 하지만 이는 너무 많은 비용을 필요로 한다. 예를 들어 프로그램에 존재하는 인라인 가능한 함수의 개수가 50개라면 여기에 대한 최적의 인라인 결정 문제는 2^{50}개의 탐색 범위를 가지게 된다[10].

이와 같은 인라인 결정 문제의 탐색 범위를 고려하기 위해서 유전 알고리즘은 좋은 해결책을 제시할 수 있다[5, 10].

[10]에서는 프로그램이 가지는 함수 종속성 그래프(accurate functions dependency graph)로 각 함수의 호출 관계를 표현하는 염색체를 정의했다. 그리고 인라인 후 발생될 수 있는 크기 증가에 대한 임계치만을 고려하여 프로그램 최적화를 위한 최적의 인라인 조합을 유전 알고리즘을 통해서 찾아냈다. [5]에서는 컴파일러가 인라인을 결정하기 위해 일반적으로 고려하는 정보들인 "인라인 가능한 함수의 크기", "피 호출자 함수의 크기", "호출자

함수가 가질 수 있는 최대 코드 량", "함수의 호출 깊이" 등에 대한 속성 값들을 염색체로 설계하고, 각각의 속성 값들은 유전 알고리즘을 통해서 결정된다. 그리고 평가 함수는 결정된 속성 값들을 기반으로 인라인을 수행했을 때 프로그램이 가지는 컴파일 시간 및 실행시간을 평가하여 가장 우수한 경우를 찾아낸다.

위의 [5, 10]관련 연구들에서는 프로그램 최적화를 위해 최적의 인라인 조합을 결정하기 위한 목적으로 유전 알고리즘을 이용하였다. 하지만 본 논문에서는 프로그램 난독화를 목적으로 가장 높은 복잡도를 가질 수 있는 인라인 조합을 결정하기 위해서 유전 알고리즘을 설계하고 사용한다.

2.3 복잡도 메트릭

소프트웨어 복잡도는 대상 시스템의 이해도, 유지보수성을 평가하는 주요한 메트릭이다. 일반적으로 소프트웨어 메트릭은 이해하기 쉽고, 최적화된 프로그램을 높은 품질의 소프트웨어로 평가하지만, 코드 난독화에서는 반대로 평가된다. [6, 11]에서는 기존에 존재하는 여러 가지 복잡도 메트릭을 이용하여 난독화 변환의 품질을 평가하는 것을 제안하였다. 이 절에서는 인라인 변환의 효용을 평가하기 위해서 본 논문에서 제안하는 복잡도 메트릭과 관련된 연구들에 대해서 알아본다.

2.3.1 정보이론

정보이론에서는 임의의 메시지가 가지는 정보량은 그 메시지의 등장 빈도수로 결정한다. 즉, 정보이론에서 정의하는 정보(information)는 놀라움의 정도(amount of surprise)이며, 새롭게 등장하는 정보가 자주 접하게 되는 정보 보다 높은 정보량을 가진다고 정의하고 있다[8]. 이는 정보의 발생 확률이 적을수록 정보의 가치가 높다는 것을 의미하고, 메시지의 집합 $M=\{m_1, m_2, \cdots, m_n\}$에서 각 m_i의 정보량은 다음 식 (1)과 같다[8].

$$I(m_i) = -\log_2 P(m_i) \quad (1)$$

여기서,

$P(m_i)$: m_i의 참조 확률

$I(m_i)$: m_i의 정보량

또한 엔트로피(entropy)를 정보원(information source)이 가지는 평균 정보량으로 정의하고 있다. 예를 들어, 메시지 집합 M이 가지는 평균 정보량은 다음 식 (2)와 같다[8].

$$H(M) = \sum_{i=1}^{n} I(m_i) \times P(m_i) \quad (2)$$

여기서,

n : 전체 정보의 수

이러한 엔트로피의 정의는 소프트웨어의 복잡도를 평가하기 위한 척도로 많이 사용되어 왔다[1, 3]. [1]에서는 클래스 내의 함수와 데이터들 간의 관계를 나타내는 DFR(Data and Functions Relationship Di-Graph) 그래프를 정의하고, DFR 그래프는 클래스내부에 존재하는 데이터 및 함수간의 참조관계를 표현한 그래프로써 이들 참조관계를 기반으로 각 클래스가 가지는 복잡도를 엔트로피 함수로 결정한다[1]. [3]는 웹 어플리케이션에서 웹 페이지들 간의 유사도를 평가하기 위한 메트릭을 엔트로피를 이용하여 정의했다.

2.3.2 제어 흐름 복잡도

프로그램이 가지는 제어 흐름은 제어 흐름 그래프(Control Flow Graph : CFG)로 표현되며, 이것은 프로그램의 제어 흐름 복잡도를 측정하는 기준이 될 수 있다. 제어 흐름 복잡도 메트릭은 제어 흐름 난독화 결과를 평가하는데 유용하게 사용되었다[4, 9]. [4]에서는 제어 흐름에 대한 난독화 변환의 평가를 위해서 McCabe의 사이클로메틱 복잡도 메트릭을 사용하였다. 사이클로메틱 복잡도 메트릭은 제어 흐름 그래프가 가지는 노드와 간선

의 수를 고려하는 측정 방법으로 노드들 간의 중첩 수준에 대한 복잡성을 고려하지 못한다. [9]은 제어 흐름 그래프가 가지는 중첩 수준을 고려할 수 있는 N-Scope 복잡도 메트릭을 이용하는 것을 제안했다. N-Scope 복잡도는 다음 식 (3)으로 표현된다[16].

$$ns(g) = \sum_{B_i \in B} \frac{|R(g, B_i)|}{|R(g, B_i)| + |NC|} \quad (3)$$

여기서,

g : 제어 흐름 그래프

B_i : 분기 노드

B : g가 가지는 분기 블록 집합

$|NC|$: g안의 모든 노드의 수

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

N-Scope 복잡도 메트릭은 제어 흐름 그래프(g)가 가지는 분기 노드의 숫자와 위치에 따라서 결정된다. 만약 제어 흐름 그래프에 다수의 분기 노드들이 존재하고, 그들이 서로 중첩되어 있을 경우에 N-Scope 복잡도는 높은 값을 가지게 된다. 반대로, 제어 흐름 그래프에 분기 노드가 존재하지 않는다면 N-Scope 복잡도 값은 0의 값을 가진다.

다음 장에서는 인라인 변환 결과 프로그램이 가질 수 있는 복잡도를 평가하기 위한 새로운 제어 흐름 복잡도 메트릭을 엔트로피와 N-Scope 복잡도 메트릭을 이용하여 정의한다.

3. 평가 메트릭

3.1 배경

프로그램은 하나의 시작 함수(main function)와 n개의 사용자 정의 함수들의 집합으로 구성되어 있으며, 각 함수들은 자신 또는 다른 함수들을 호출하는 호출 관계를 가지고 있다.

[정의 3-1] 함수의 집합

$$P = \{f_e, f_1, \cdots, f_n\}$$

여기서,

P : 프로그램

f_e : 시작 함수

f_i : 함수

[정의 3-2] 호출 관계

$$R = \{r_1, r_2, \cdots, r_k\}$$

$$r_i = \langle f_a, f_b \rangle, 1 \leq i \leq k$$

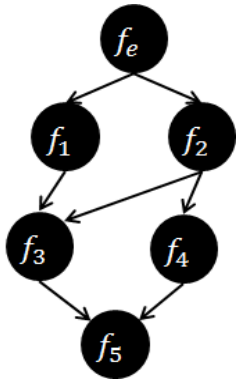
여기서,

k : 함수에 존재하는 모든 호출 관계의 수

R : 프로그램이 가지고 있는 모든 함수의 호출 관계 집합

$r_i = \langle f_a, f_b \rangle$: f_a는 caller, f_b는 callee

일반적으로 프로그램의 제어 흐름은 시작 함수에서부터 각 함수의 정적 호출 관계를 따라 이동하며, 이러한 제어 흐름 구조는 [그림 2]와 같은 호출 그래프로 표현할 수 있다.



[그림 2] 호출 그래프

[그림 2]에서 노드는 함수를 간선은 호출 방향을 나타내고 있다. 이 경우, 호출 그래프에서 각 노드가 가지는 in-link 개수는 전체 제어 흐름 중에 사용자가 해당 함수를 접할 수 있는 횟수로 볼 수 있다.

개발자 또는 분석자가 함수를 분석할 때, 각 함

수가 가지는 코드 구조와 참조 횟수에 따라 그 함수에 대한 이해도가 결정된다. 극단적으로 1000줄의 코드를 가지는 함수를 1번 볼 경우와, 10줄의 코드를 가지는 함수를 1번 볼 경우에 두 함수를 이해하는데 드는 비용이 같다고 할 수 없다. 또한 1000줄의 코드를 5번 볼 경우보다 10줄의 코드를 1번 볼 경우가 이해하는데 드는 비용이 더 높다고 볼 수 없다. 따라서 각 함수에 대한 복잡도를 적절하게 평가하는데 전체 제어 흐름 가운데 참조 횟수와 그 함수가 가지는 내부 코드 구조를 함께 고려할 필요가 있다.

3.2 엔트로피 기반 제어 흐름 복잡도 메트릭

본 절에서는 프로그램이 가지는 제어 흐름 복잡도를 엔트로피를 기반으로 하여 새롭게 정의한다. 프로그램에 존재하는 각 함수는 자신 또는 다른 함수에 의해서 1번 이상 호출되어진다. 따라서 각 함수가 가지는 참조확률은 다음과 같이 정의한다.

[정의 3-3] 함수의 참조확률

$$P(f_i) = \frac{Fanin(f_i)}{\sum_{j=1}^{n} Fanin(f_j)} \quad (4)$$

여기서,

n : 프로그램에 존재하는 모든 함수의 개수

f_i : 함수

$Fanin(f_i)$: 정적 호출 그래프에서 f_i로의 in-link 개수

시작함수 역시 시스템에 의해서 호출되어지므로 1개의 in-link를 가지고 있다고 본다. 따라서 [그림 2]에서 전체 in-link의 개수는 8개가 되고, 2개의 in-link를 가지는 f_3의 참조확률은 0.25이다.

다음은 각 함수가 가지는 정보량을 결정한다. 참조 확률만을 고려한 정보량은 모든 정보가 동일하다는 전제하여 정의되었다. 하지만 프로그램에 존재하는 모든 함수들이 가지는 코드 구조는 차이를 가지기 때문에 모든 함수가 동일한 정보량을 가지고 있다고 볼 수 없다. 따라서 각 함수가 가질 수 있는 새로운 정보량을 정의한다. 프로그램의 제어 흐름에 대한 평가를 위해서 함수가 가지는 참조 확률과 제어 흐름 복잡도를 정보량의 측정 요소로 결정하고 다음으로 정의한다.

[정의 3-4] 함수의 정보량

$$I_{new}(f) = I(f) \times ns(f) \quad (5)$$

여기서,

f : 함수

$I(f)$: 참조 확률에 기반을 둔 f의 정보량 (식 (1))

$ns(f)$: f의 제어 흐름 복잡도(식 (3))

최종적으로 각 함수가 가지는 새로운 정보량은 호출 횟수와 제어 흐름 복잡도에 의해서 결정된다.

프로그램이 가지는 전체 제어 흐름 복잡도는 모든 함수가 가지는 정보량의 평균으로써 다음 식 (6)으로 정의된다.

[정의 3-5] 프로그램의 제어 흐름 복잡도

$$C(P) = \sum_{i=1}^{n} I_{new}(f_i) \times P(f_i) \quad (6)$$

여기서,

P : 프로그램

n : 전체 함수의 수

$C(P)$: 프로그램이 가지는 제어 흐름 복잡도

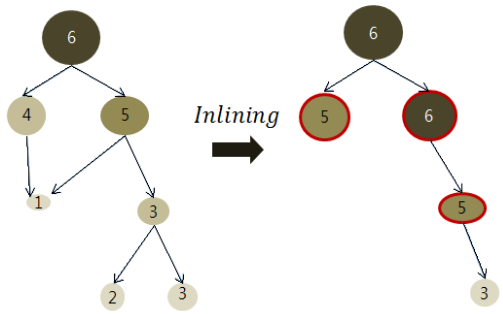
$C(P)$는 제 2장에서 소개한 엔트로피에 각 함수의 제어 흐름에 대한 복잡도가 반영된 새로운 정보량 및 각 함수가 가지는 참조확률을 적용하여 정의되었다.

4. 유전알고리즘을 이용한 인라인 적용 기법

4.1 유전 알고리즘의 필요성

제 2장에서 알아본 것 같이 코드 난독화에서 인

라인 변환은 프로그램에 존재하는 추상적인 호출 정보를 제거할 수 있는 난독화 변환이다. 또한, 인라인의 결과로 기존에 존재하는 함수의 코드 가독성을 떨어뜨리는 효과를 초래할 수 있다. 예를 들어, 함수 P와 Q의 호출 관계 $r_i = (f_p, f_q)$에 대한 인라인 변환이 수행될 경우 Q 함수가 가지는 코드들이 P의 함수로 복사되어 P 함수의 코드 량이 증가한다. 이때 Q 함수의 내부 코드 구조가 복잡할수록 인라인 후 P 함수가 가지는 코드 가독성이 낮아질 수 있다. 또한 복사되어지는 코드 량에 비례하여 전체 프로그램의 크기가 켜지는 부정적인 효과가 발생한다.



[그림 3] 인라인 후 정적 호출 그래프의 변화

위의 [그림 3]의 왼쪽은 인라인 전, 오른쪽은 인라인 후, 프로그램이 가지는 각 함수의 복잡도와 크기의 변화를 정적 호출 그래프로 나타낸 것이다. [그림 3]에서 노드의 색은 각 함수가 가지는 복잡도로 색이 진할수록 높은 복잡도를 가지는 것을 나타내고, 노드안의 숫자는 그 함수의 크기를 나타낸다. [그림 3]은 인라인의 결과로 전체 프로그램의 복잡도가 증가하였지만, 프로그램이 가지는 크기 역시 증가하는 부정적인 효과의 예를 보여준다.

따라서 최적의 인라인 조합을 결정하기 위해서는 함수들 간의 호출관계와 더불어 함수가 가지는 코드 구조를 고려할 필요가 있다. 하지만, 프로그램이 가지는 함수들의 호출 관계와 각 함수의 내부 코드들은 프로그램에 따라 다양하게 구성되기 때문에 최적의 인라인 변환 조합을 결정하는 문제는 많은 계산 시간이 필요하다. [10]에서는 함수 인라인 문제가 프로그램이 가지는 함수의 수에 따라 지수 범위의 문제 크기를 가지는 것을 고려하여 프로그램 최적화를 위한 인라인 결정 문제를 해결하기 위해서 유전 알고리즘을 이용하는 것을 제안했다. 유전 알고리즘은 진화의 원리를 문제 해결에 이용하는 대표적인 방법론 중 하나로 조합 최적화(Combinatorial optimization problem) 문제에 주로 사용된다. 코드 난독화를 위한 인라인 변환 문제 역시 최적화를 위한 인라인 문제와 동일한 크기의 문제 범위를 가지므로 유전 알고리즘을 이용하여 해결하는 것을 제안한다.

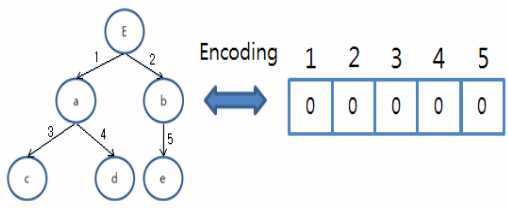
4.2 염색체 인코딩

유전 알고리즘에서 사용되는 염색체의 설계는 프로그램이 가지는 정적 호출 그래프로 결정한다. 염색체는 0 또는 1의 값을 가지는 L개의 크기의 이진 스트링이 된다. 염색체의 크기 L은 프로그램이 가지는 호출 관계 집합의 크기이고, 각 비트의 값(0, 1)은 호출 관계에 있는 함수들 간의 인라인 유무를 나타낸다.

예를 들어 임의의 프로그램이 아래 [그림 4]의 왼쪽과 같은 정적 호출 그래프를 가진다면,

$$r_1 = (f_e, f_a), r_2 = (f_e, f_b), r_3 = (f_a, f_c), r_4 = (f_a, f_d)$$

$r_5 = (f_b, f_e)$와 같은 호출 관계 집합을 가지게 되고, 염색체는 길이가 5인 이진 스트링이 된다. 이때 염색체의 값이 [10010]의 값을 가진다면, 함수 'a'와 함수 'd'가 인라인 되었다는 것을 의미한다.



[그림 4] 염색체 인코딩

여기서 프로그램에 존재하는 메인 함수, 재귀 함수, 동적 호출 함수들에 대해서는 인라인 후 프로그램에 부정적인 효과를 초래할 수 있으므로 인라인 변환 후보에서 제외된다.

4.3 평가 함수

평가 함수를 정의하기 전에 인라인 변환 후 프로그램이 가지는 복잡도 증가율과 크기 임계치를 먼저 결정한다.

프로그램이 가지는 제어 흐름 복잡도의 평가는 식 (6)으로 결정 한다. 복잡도 증가 비율은 염색체가 가지는 인라인 조합 결과를 인라인 전의 복잡도로 나눈 값으로 결정한다.

[정의 4-1] 복잡도 증가 비율

$$CIR(T(P)) = \frac{C(P')}{C(P)} \quad (7)$$

여기서,

P : 변환 전 프로그램

P' : 변환 후 프로그램

$T(P)$: 인라인 변환이 적용된 P

$C(P)$: P가 가지는 제어 흐름 복잡도(식 (6))

다음은 인라인으로 인해 발생되는 크기 증가를 고려하기 위해 크기 임계치를 설정한다. 난독화 변환은 코드에 새로운 정보를 추가하는 것으로 프로그램을 보호하는 효과를 얻음과 동시에 프로그램 코드 크기의 증가에 대한 오버헤드를 초래하기 때문에 난독화 변환으로 발생되는 오버헤드를 고려해야 된다[6]. 코드 난독화와 관련 된 앞선 실험들에서 평균적으로 난독화의 결과로 원본 프로그램이 가지는 크기의 160% 증가하는 것을 관찰하고 이를 허용했다[9, 10, 14]. 따라서 유전 알고리즘이 고려할 수 있는 크기 증가에 대한 임계치를 원래 프로그램 코드의 160%로 설정한다.

따라서 각 염색체의 품질을 나타내는 적합도 값을 평가하는 위해 유전 알고리즘의 평가 함수는 다음으로 정의한다.

[정의 4-2] 평가 함수

$$F(P) = \begin{cases} CIR(T(P)), & \text{if}(S < ST) \\ 0, & Otherwise \end{cases} \quad (8)$$

여기서,

S : 변환 후 프로그램 크기

ST : 크기 임계치

식 (8)로 인해서 각 해 집단의 염색체들이 가지는 적합도 값은 인라인 변환 후에 크기 임계치를 만족하는 프로그램의 복잡도 증가 비율이 된다.

5. 실험 및 평가

본 장에서는 C언어로 구성된 cflow[13]를 실험 대상 프로그램으로 하여 인라인 변환 수행 결과를 평가한다. cflow는 프로그램의 호출 구조를 분석하는 코드 분석도구로써 리눅스 오픈 소스 프로그램으로 전체 8개의 소스 파일과 33개 사용자 정의 함수로 구성되어 있다. 실험 프로그램에 대한 호출 그래프와 제어 흐름 그래프에 대한 분석은 코드 분석 도구인 understand2.6[15]를 이용하였다. understand는 크로스 플랫폼, 다중 언어, 유지보수 기반의 통합 개발 환경으로써 개별 소스코드는 물론 프로젝트 단위의 큰 소스 코드를 수정하거나 이해하는데 유용하게 사용되는 코드 분석 도구이다.

[6]에서는 인라인 변환을 프로그램에 적용하기 위한 적절한 알고리즘을 제안하지 않았다. 따라서 본 실험에서는 크기 제한 임계치를 동일하게 설정하고, 랜덤 방법으로 수행한 결과와 유전 알고리즘으로 수행한 결과를 $C(P)$ 값으로 비교하여 제안하는 방법의 효용에 대해서 평가한다.

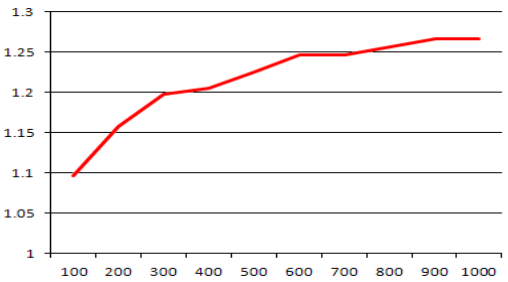
랜덤 방법은 프로그램에 존재하는 인라인 가능한 함수들에 대해 무작위로 인라인을 수행하기 때문에 결과로 얻을 수 있는 $C(P)$ 값은 일정하지 않다. 따라서 본 실험에서는 랜덤 방법으로 100번의 인라인을 수행한 결과의 평균 $C(P)$ 값으로 유전 알고리즘으로 수행한 결과 $C(P)$ 값과 비교하여 제안하는 방법의 효용을 평가한다. 100번의 랜덤 방

법으로 인라인 난독화를 수행한 결과들의 평균 $C(P)$ 값은 1.74이고, 이것은 인라인 전의 $C(P)$ 값인 1.46에 비해 19% 증가하였다.

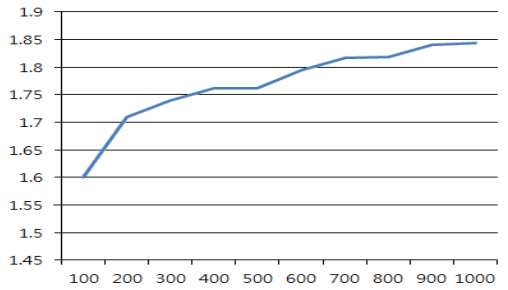
다음 <표 1>는 실험에서 설정한 유전 알고리즘 인자들의 값을 나타내고 있다.

〈표 1〉 유전 알고리즘의 인자

인자	값
해 집단의 크기	10
선택연산	임의 선택
교차연산	2점 교차
변이연산(확률)	50%
대치	최소 품질의 부모 해와 대치
세대 수	1000



[그림 5] 세대 간 해집단에서 가장 우수한 해의 적합도 값($Fit(P)$)



[그림 6] 각 세대에서 얻은 우수한 해를 통한 인라인 난독화 결과($C(P)$)

[그림 5]은 유전 알고리즘이 1000세대까지 반복 수행되면서 각 세대의 해집단에서 가장 우수한 해가 가지는 적합도 값($Fit(P)$)을 나타내고, [그림 6]는 각 세대에서 찾은 가장 우수한 해를 통해서 인라인 변환을 수행했을 때 얻을 수 있는 제어 흐름 복잡도 값($C(P)$)을 각각 나타내고 있다.

두 개의 그림 모두 가로축은 세대를, 세로축은 염색체의 적합도 값과 제어 흐름 복잡도 값을 각각 나타낸다. 가장 높은 적합도 값을 가지는 우수한 해는 전체 세대 가운데 800세대 정도에 유전 알고리즘이 수렴하는 결과를 보였다.

최종적으로 유전 알고리즘이 전체 세대를 반복해서 수렴한 해를 이용하여 인라인 난독화를 수행했을 때 프로그램에 존재하는 인라인 후보 함수들에 대해 27번의 인라인 변환이 적용되었고, 변환 후 전체 프로그램이 가지는 제어 흐름 복잡도 값($C(P)$)은 1.84로 원본 프로그램 대비 26% 증가한 결과를 보였다. 이는 100번의 랜덤 방법으로 수행한 인라인 난독화 결과들의 평균 값(1.74)이 원본 프로그램 대비 19% 증가한 것보다 7% 높은 $C(P)$ 값을 가진다.

〈표 2〉 인라인 변환 전, 후의 $C(P)$

	변환 전	랜덤	유전 알고리즘
$C(P)$	1.46	1.74	1.84
증가율	-	19%	26%

6. 결 론

인라인은 프로그램이 가지는 제어 흐름 구조에 대한 분석을 어렵게 하기 위한 목적으로 사용될 수 있다. 본 논문에서는 코드 난독화를 목적으로 인라인 변환을 적용하기 위해 유전 알고리즘을 이용하는 것을 제안했다. 이를 위해 먼저 엔트로피와 제어 흐름 복잡도를 이용하여 각 함수가 가질 수 있는 제어 흐름 복잡도를 새로운 정보량으로 정의하고, 이를 바탕으로 프로그램이 가지는 전체 제어 흐름 복잡도 메트릭을 제안했다. 그리고 제안한 메트릭을 기준으로 평가함수를 정의하고, 유전 알고리즘은 이를 통해 가장 높은 제어 흐름 복잡도를 가질 수 있는 최적의 인라인 조합을 찾아

낸다. 제안하는 방법의 효용은 난독화 수행 조건을 동일하게 설정하고, 원본 프로그램이 가지는 $C(P)$ 값에서 무전략(랜덤 방법)으로 난독화를 수행한 경우와 유전 알고리즘으로 난독화를 수행한 경우에 얻어지는 결과 $C(P)$ 값들의 증가율을 각각 비교하여 유전 알고리즘의 난독화 결과가 7%높은 $C(P)$ 값을 가지는 것을 확인했다.

본 논문에서 제안하는 인라인 변환 전략은 프로그램 크기 증가 임계치만을 고려하여 난독화를 수행한 결과 160%의 프로그램 크기 증가가 발생하였다. 이는 앞선 연구의 난독화 실험 결과에서 허용한 범위에 속하지만 인라인 변환 외 다른 난독화 변환을 추가적으로 적용할 경우 제한적으로 수행될 수 있다. 따라서 인라인 변환으로 발생되는 프로그램 크기 증가에 대한 오버헤드를 함께 고려할 수 있도록 유전 알고리즘의 평가 함수를 확장할 필요가 있다.

참 고 문 헌

[1] 김갑수, 신영길, 우치수, “엔트로피를 이용한 객체 지향 프로그램의 복잡도 척도”, 「한국정보과학회논문지」, 제22권, 제12호(1995), pp.1656-1666.

[2] 엘다드 에일람, 「리버싱 : 리버스 엔지니어링 비밀을 파헤치다」, 1판, 에이콘, 2009.

[3] 정우성, 이은주, “유사도 기반의 웹 어플리케이션 구조 복잡도”, 「한국컴퓨터정보학회논문지」, 제9권, 제8호(2010), pp.117-126.

[4] 채영현, “소프트웨어 보안을 위한 코드 난독화 도구의 개발”, 「Master Thesis, Dep. Electrical Engineering and Computer Science. KAIST」, (2007), pp.1-38.

[5] Cavazos, J. and M. F. P. O'Boyle, “Automatic Tuning of Inlining Heuristics”, *Proc. of the ACM/IEEE SC 2005 Conference*, (2005), p.14.

[6] Colleberg, C., C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations”, *Technical Report 148, University of Auckland*, (1997), pp.2-36.

[7] Collberg, C., C. Thomborson, and D. Low, “Manufacturing cheap, resilient, and stealthy opaque constructs”, In Principles of Programming Languages, POPL, San Diego, CA, (1998), pp.184-192.

[8] Harrison, W., “An entropy-based measure of software complexity”, *IEEE Transactions on Software Engineering*, Vol.18(1992), pp. 1025-1029.

[9] Hsin-Yi, T., H. Yu-Lun, and D. Wagner, “A graph approach to quantitative analysis of control-Flow obfuscationg transformations”, *IEEE Transactions On Information Forensics and Security*, Vol.4(2009), pp.257-267.

[10] Li, M. and H. Wang, “GA based inlining optimization in front-end synthesis of embedded software”, *5th International Conference on ASIC*, Vol.1(2003), pp.341-343.

[11] Low, D., “Java control flow obfuscation”, *Master's Thesis, Department of Computer Science*, (1998), pp.13-69.

[12] http://www.scitools.com/.

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

[14] Ogiso, T., Y. Sakabe, M. Soshi, and A. Miyaji, “Software obfuscation on a theoretical basic and its implementation”, *IEEE Trans. Fundamentals*, Vol.1, No.1(2003), pp. 176-186.

[15] Sakabe, Y., M. Soshi, and A. Miyaji, “Java Obfuscation Approaches to Construct Tamper-Resistant Object-Oriented Programs”, *IPSJ Digital Courier*, Vol.1(2005), pp.134-146.

[16] Zuse, H., “Software Complexity : Measures and Methods”, Hawthorne, NJ : Walter de Gruyter Co., 1991.

◈ 저 자 소 개 ◈

김 정 일 (jikim424@gmail.com)

경북대학교 전자전기컴퓨터학과에서 석사과정을 취득하고, 현재 경북대학교 전자전기컴퓨터학과 박사과정 중에 있다. 소프트웨어 공학을 전공으로 하고 있으며, 주요 관심분야는 소프트웨어 분석, 설계 및 개발방법론 등이다.

이 은 주 (ejlee@knu.ac.kr)

서울대학교 전기컴퓨터공학부에서 박사학위를 취득했으며, 현재 경북대학교 IT대학 컴퓨터학부 조교수로 재직 중이다. 주요 관심분야는 웹 공학, 재공학, 메트릭, 소프트웨어 유지보수 등이다.