유전 알고리즘을 이용한 코드 난독화에서의 효율적 함수 호출 인라인 기법

김 정 일 ○ 이 은 주
경북대학교 전자전기컴퓨터학부
2009307043@knu.ac.kr, ejlee@knu.ac.kr

Effective Technique for Inlining Function Calls in Code Obfuscation Using Genetic Algorithm

Jungil Kim ○ Eunjoo Lee
School of Electrical Engineering and Computer Science, Kyungpook National University

요 약

코드 난독화 기법 중의 하나인 인라인(Inline)은 코드의 복사를 통하여 함수의 호출 구조를 파괴하여서 코드의 복원과 이해를 어렵게 만든다. 하지만 적절한 전략 없이 인라인 기법을 적용하게 되면, 프로그램 성능이 저하되며 난독화의 결과도 기대 이하일 가능성이 존재한다. 따라서 지나친 성능의 저하를 막으면서 결과적으로 코드의 복원과 이해를 최대한 어렵게 하기 위한 인라인 수행 전략이 필요하다. 이를 위하여 본 논문에서는 정적 함수 호출 그래프를 기반으로 인라인의 적용 여부를 유전 알고리즘을 사용하여 결정하도록 한다. 그리고 인라인 전후의 효용을 보여주기 위하여 정보이론 및 제어 흐름의 복잡도에 기반하여 전체 프로그램의 복잡도를 정의하였다. 마지막으로 해당 기법의 효용을 실험을 통해 보였다.

1. 서 론

코드난독화(Code Obfuscation)는 프로그램의 기능을 그대로 유지하면서 프로그램 소스코드를 이해하기 어렵게 변환하는 것으로, 플랫폼에 상관없이 소프트웨어를 보호하기 위해서 적용할 수 있는 소프트웨어 보호 기술이다[1]. 하지만 난독화 변환은 기존 코드의 크기, 실행 속도, 메모리의 사용량과 같은 프로그램 성능의 저하를 초래하기도 한다. 그러므로 프로그램의 특징에 따라 난독화 적용 정도를 조절할 필요가 있다.

코드 난독화 시 수동 변환은 프로그램의 변환을 위해서 노력에 비해 결과가 비효율적인 경우가 많으므로 일반적인 난독화 변환은 난독기를 사용한 자동 변환을 통하는 경우가 많다. 그러므로 자동 난독화 변환을 프로그램에 효율적으로 적용 할 수 있는 방법이 고려되어야 한다. 여러가지 난독화 변환 가운데 인라인은 호출관계에 있는 두 개의 함수를 병합함으로 인해 잘 짜여진 함수의 호출 구조를 파괴하고, 함수의 내부 코드량이 커지면서 프로그램의 이해도를 떨어뜨리는데 매우 효과적인 난독화 변환이다[1][2]. 하지만 적절한 전략 없이 인라인을 하게 되면 프로그램 성능이 저하될 뿐 아니라 난독화의 결과도 좋지 않을 가능성이 높으므로 인라인 수행 여부를 신중하게 결정할 필요가 있다.

난독화 변환을 목적으로 하는 인라인 결정 문제는 프로그램내에 존재하는 인라인 가능한 함수의 수에 비례하여 지수적인 수행 시간을 가지기 때문에 일반적인 알고리즘으로 문제를 해결하기 위해서 많은 시간이 요구된다. 따라서 본 논문에서는 유전 알고리즘을 이용하여 인라인 적용 여부를 결정하도록 한다. 또한 인라인 결과의 효용을 판단하기 위한 메트릭이 필요한데 일반적으로 프로그램 복잡도 메트릭이 많이 적용되었다[2]. 하지만 본 논문에서는 인라인의 수행으로 변하는 프로그램의 구조에 대해서 더 자세한 평가를 위하여 정보이론에서 정의하는 정보량과 프로그램 복잡도 메트릭을 병합한 새로운 복잡도 메트릭을 제안하여 인라인 결과의 효용을 판단하는 근거로 사용한다.

본 논문의 구성은 다음과 같다. 2장에서 관련연구에 대해서 알아보고, 3장에서는 인라인 수행의 결과를 평가하기 위해 정보 이론과 제어흐름 복잡도 측정 척도를 함께 고려한 새로운 복잡도 메트릭을 제안한다. 4장에서 직접적인 실험을 통해 인라인 난독화 변환 문제를 해결하는데 유전 알고리즘이 좋은 해결책이 될 수 있다는 가능성을 보인다. 마지막으로 본 연구에 대한 결론과 향후 연구 과제에 대해서 설명한다.

2. 관련 연구

2.1. 프로그램 최적화

인라인은 컴파일타임(Compile Time)에 수행되는 프로그램 최적화 기능 중 하나로 컴파일러는 프로그램의 실행시간을 줄이기 위한 목적으로 인라인을

사용할 수 있다. 하지만 너무 많은 인라인의 사용은 컴파일 시간과 프로그램의 크기를 증가시키는 부정적인 효과를 가져오기도 한다[3]. [4]에서는 임베디드 프로세서(Embedded Prosesor)에 의해서 실행되는 프로그램의 최적화를 위해 함수 호출의 최소화를 목적으로 인라인을 사용하는 방법을 제안한다. 인라인으로 인한 프로그램의 크기의 증가를 제한하고, 제한된 범위내에서 최적의 인라인 적용 범위를 결정하기 위해서 Branch and bound(B&B)알고리즘을 제안한다. [5]에서는 [4]와 동일한 문제에 대해서 유전 알고리즘을 사용한 해결법을 제안하고, 유전 알고리즘과 B&B 알고리즘을 비교하여 프로그램 최적화를 위한 인라인 결정 문제에 대해 유전 알고리즘이 더 좋은 해결책이 된다는 것을 결과를 통해 보였다. [3]은 동적 컴파일이 가능한 언어(Java, C#)로 작성된 프로그램의 컴파일 시간과 실행 시간에 대한 프로그램의 전체적인 성능 향상을 목적으로 하는 인라인 수행 여부를 컴파일러가 스스로 판단하기 위한 방법으로 유전 알고리즘을 적용하였다. 이처럼 위의 연구들은 유전 알고리즘이 인라인을 이용한 프로그램 최적화 문제를 해결하는데 좋은 해결책이 될 수 있다는 것을 보여주었다. 프로그램 최적화와 비슷한 문제로 분류할 수 있는 난독화의 효과를 극대화 하기 위한 목적의 인라인 수행 여부를 결정하기 위해서 유전 알고리즘이 적용된 사례는 찾아볼 수 없었다. 본 연구에서는 프로그램 난독화를 목적으로 하는 효율적인 인라인을 위해서 유전 알고리즘을 적용하여 해결하는 방법을 제안한다.

2.2 정보이론 & 제어흐름 복잡도

정보이론에서는 “자주 발생되는 메시지가 그렇지 않은 메시지 보다 정보량이 적다” 라고 정의하고 있다. 전체 메시지의 집합 M = $\{m_1, m_2, \cdots m_n\}$으로 가정할 때, 임의의 메시지 m 의 정보량은 m이 나타나는 확률로 아래 <식 1>로 정의한다[6].

$$I(m) = -\mathrm{Log}_2 P(m) \quad \cdots \text{ 식(1)}$$

여기서
I(m): m의 정보량
P(m): m이 나타날 확률.

전체 메시지 집합 M의 평균 엔트로피는 모든 메시지 정보량의 평균으로 계산되며 아래 <식 2>와 같다.

$$H(M) = \sum_{i=1}^{N} I(m_i) \times P(m_i) \quad \cdots \text{ 식(2)}$$

이 이론은 프로그램의 복잡도를 측정하기 위해서 이용되었다[6][7]. 앞선 연구들에서는 프로그램의 복잡도를 측정하기 위해서 프로그램내에 사용되어지는 연산자(Operators)들을 프로그램이 가지는 정보라고 판단하고, 각 연산자의 빈도수에 따른 정보량을 계산하여 프로그램의 복잡도를 결정하였다.
프로그램의 복잡도를 측정하기 위한 방법으로 제어흐름 그래프(Control Flow Graph)를 기반으로 하는 복잡도 척도가 일반적으로 많이 사용되었다[8][9][10]. 프로그램의 제어흐름이 복잡할수록 코드의 가독성, 이해도, 유지보수가 어려워진다. 따라서 이러한 척도들은 간결하고 이해하기 쉬운 프로그램의 제작을 목표로 얼마나 복잡도가 낮은지를 측정하기 위해 제안되었지만, 난독화의 효과를 확인하기 위해서 프로그램 복잡도의 변화를 측정하는데 유용하게 사용될 수 도 있다[11].
본 논문에서 프로그램에 대한 난독화 적용 후의 복잡도를 측정하기 위해서 프로그램에서 사용 되어지는 함수의 호출 빈도수와 각 함수 내부 코드의 제어흐름 복잡도를 함께 고려하여 정적분석을 통한 난독화의 평가를 위해 각각의 함수와 프로그램 전체의 복잡도를 측정하는 새로운 척도를 제안한다.

3. 프로그램 복잡도

이 장에서는 인라인으로 인한 난독화 결과의 효용을 평가하기 위한 프로그램의 복잡도 측정에 대해서 설명한다. 먼저 정보이론의 개념을 기반으로 각 함수의 호출 빈도수에 의한 참조확률로 정보량을 계산하고, 이어서 함수 내부 코드의 제어 흐름 복잡도를 측정하는 메트릭에 대해 차례로 알아본다. 마지막으로 함수의 정보량과 제어 흐름 복잡도를 병합한 새로운 복잡도 메트릭을 제안한다.

3.1 함수의 정보량

코드상에서 함수간의 호출 관계를 정적분석을 통해서 <그림 1>과 같이 호출 그래프로 표현할 수 있다.호출 그래프는 프로그램이 가진 함수를 나타내는 m개의 노드 집합 N = { $n_1, n_2, \ldots, n_m$ }. 함수들간의 호출 관계를 나타내는 k개의 간선 집합 E = $\{e_1, e_2, \ldots e_k\}$로 구성된다. 두 노드간의 관계를 표현하는 간선 e_i 는 $e_i = < n_k, n_j >$로 표현 할 수 있으며 이것은 n_k가 간선 e_i의 소스 노드(Caller)가 되고, n_j 를 간선 e_i 의 목적 노드(Callee)라고 정의한다.
본 연구에서는 호출 그래프에서 표현된 하나의 노드를 프로그램이 가지는 정보라고 정의하고, 노드가 가지는 간선의 수를 이용해 노드의 참조횟수에 따른 정보량을 계산한다.
노드가 받는 간선(InEdge)을 Fanin 간선이라고 하며, 호출 그래프를 구성하는 모든 노드는 1개 이상의 Fanin 간선을 가진다.(Main 함수 역시 시스템에 의해 호출되기 때문에 1개의 Fanin을 가진다.) 호출 그래프가 가지고 있는 전체 노드의 Fanin 간선과 각 노드가 가지고 있는 Fanin 간선으로 노드에 대한 참조확률을

아래 <식 3>으로 구할 수 있다.

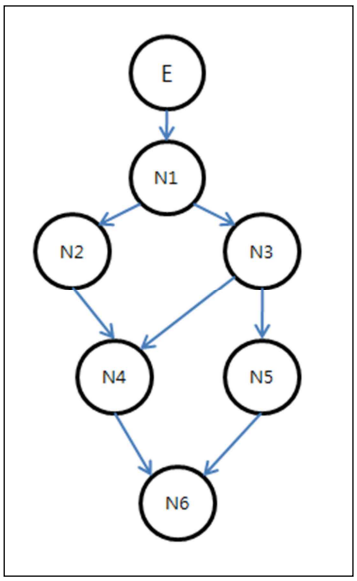
$$P(n_i) = \frac{Fanin\ (n_i)}{\sum_{j=0}^{N} Fanin\ (n_j)} \quad \cdots \text{ 식(3)}$$

여기서
N: 전체 노드의 수
$Fanin(n_i)$: 노드 n_i가 가지는 Fanin 간선의 수
$P(n_i)$: 노드 n_i의 참조확률

$P(n_i)$ 는 자신이 가지는 Fanin의 수와 모든 Fanin 간선의 수로 나누어서 계산된다. 여기서 각 노드의 정보량은 <식 1>을 이용하여 아래 <식 4>와 같이 정의된다.

$$I(n_i) = -Log_2\ P(n_i) \quad \cdots \text{ 식(4)}$$

여기서
$I(n_i)$: 노드 n_i의 정보량
$P(n_i)$: 노드 n_i의 참조 확률



[그림 1]. 일반적인 프로그램 호출 그래프

3.2 함수의 제어 흐름 복잡도

소프트웨어 복잡도는 정적 분석을 통한 프로그램 코드의 이해수준과 유지보수 비용을 평가하기 위해 사용되었다[8][9][10]. 프로그램 코드가 가지는 제어흐름에 대한 복잡성을 계산하여서 제어흐름이 복잡할 수록 사람이 이해하는데 드는 비용이 증가한다는 기준을 제시하였다. 이러한 복잡도 척도는 프로그램의 이해 수준을 평가할 수 있으므로 코드 난독화 이후의 프로그램 이해도를 평가하는데 사용되어진다[11].

제어흐름 복잡도를 측정하기 위해서 [8]가 제안한 사이크로메트릭 복잡도가 가장 많이 사용되지만,[8]은 흐름 그래프의 복잡도가 추가되어지는 노드와 간선의 수에 비례하여 증가되지 않으므로, 인라인으로 인한 코드의 증가량에 따른 복잡도 측정에 적합하지 않다. 따라서 본 연구에서는 []에서 제안한 TAC(Total Adjusted Complexity)를 이용하여 코드의 제어 흐름에 대한 복잡도를 측정한다. 또한 각 함수의 독립적인 복잡도 값을 할당하기 위해서 측정 범위를 해당 함수의 수행 범위로 제한할 필요가 있다.

제어흐름 복잡도의 측정의 범위를 함수 블럭 단위로 제한한다면, 하나의 제어흐름 그래프는 아래 <식 5>로 계산되어 진다.

$$C(N) = \sum_{i=1}^{M} Comp(n_i) \quad \cdots \text{ 식(5)}$$

여기서
M: 제어 흐름 그래프내의 전체 노드의 수
$Comp(n_i)$:노드 n_i의 복잡도

<그림 2>는 TAC를 이용하여 측정된 제어흐름 그래프의 복잡도이다. 제어흐름 그래프의 복잡도는 모든 노드의 복잡도의 합이므로 a)는 3, b)는 1의 복잡도 값을 가진다.

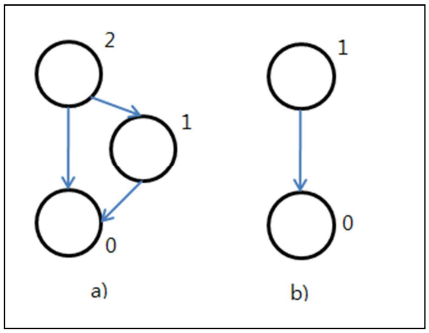


그림 2. TAC를 이용한 흐름 그래프의 복잡도

3.3. 새로운 정보량

프로그램의 정적 분석을 통해서 나타날 수 있는 여러 가지 정보(연산자, 예약어, 데이터구조)들 가운데 함수 호출 문장은 프로그램의 특징을 쉽게 분석할 수 있게 해 주는 매우 중요한 정보로 분류될 수 있다[6][7]. 또한 함수의 내부 코드량에 따라 분석을 하기 위해 들 수 있는 비용의 차이가 생긴다. 따라서 본 논문에서는 난독화에 대한 척도의 단위를 정적 분석을 통해서 구별할 수 있는 코드상에 존재하는 함수 호출 문장으로 결정하고, 각 함수의 호출 빈도수와 내부 제어 흐름 복잡도를 계산하여 프로그램에서 함수가 가지는정보량을 결정하고, 이렇게 결정된 모든 함수의 정보량의 평균으로 전체 프로그램의 복잡도를 계산한다. 각 함수의 새로운 정보

량은 아래 <식 6>으로 계산된다.

$$I_{new}(n_i) = I(n_i) \times C(N_i) \quad \cdots \text{ 식(6)}$$

여기서
$I_{new}(n_i)$: 노드 n_i의 새로운 정보량
$I(n_i)$: 노드 n_i의 정보량
$C(N_i)$: 노드의 제어 흐름 복잡도

프로그램을 구성하는 모든 함수의 새로운 정보량이 계산되었다면, 전체 프로그램의 복잡도 C(P)는 모든 함수의 새로운 정보량의 평균으로 다음 <식 7>으로 계산된다.

$$C(P) = \sum_{i=1}^{N} I_{new}(n_i) \times P(n_i) \quad \cdots \text{ 식(7)}$$

4. 실 험

이 장에서는 실험을 위한 가상의 그래프 모델과 유전 알고리즘 인자 설계에 대해서 설명한다.

4.1 그래프 모델 설계

구조적 프로그램은 내부의 함수 호출로 인한 함수간 호출 관계를 반영할 수 있는 호출 그래프로 표현할 수 있다. 본 실험에서는 실제로 존재하는 프로그램이 아닌 가상의 호출 그래프만을 정의하여 실험을 수행한다. 설계된 실험 모델은 시작 노드(Main 함수)를 포함한 11개의 노드(E,1,2,3,…,10) 와 노드간의 호출(참조) 관계를 나타내는 21개의 간선(시스템에 의한 Main 함수의 호출도 포함)으로 구성되어지며, <그림 3>에 나타나있다. <그림 3>에서 원은 노드(함수)를 나타내며, 방향을 가진 간선은 참조(호출)되는 노드를 가리킨다. 원 내부에 표시된 숫자는 노드를 가리키는 그래프 상의 유일한 번호이고, 간선 옆에 표시된 두 개의 숫자는 각각 간선의 식별 번호와 (참조 횟수)를 나타낸다.

그래프의 복잡도를 측정하기 위해 각 노드에 할당되는 참조 확률, 제어 흐름 복잡도, 무게가 필요하다. 참조확률은 그래프에 표현되는 노드가 참조되어지는 간선의 수로 결정할 수 있지만, 제어 흐름 복잡도와 무게(크기)는 임의의 값으로 할당 하였다.

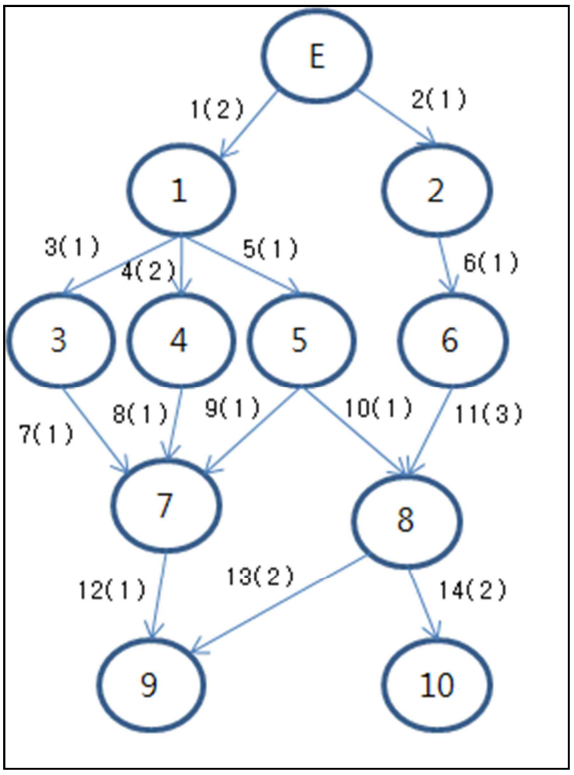


그림 3. 실험 대상 호출 그래프 모델

실험 그래프의 각 노드에 정보는 <표 1>에 표시되어 있다.

실험 그래프의 평균 복잡도는 3.3절에서 살펴본 <식 7>로 계산되어지며, 그래프의 크기는 모든 노드가 가지는 무게의 합으로 결정한다. 따라서 이 실험 그래프의 인라인 수행 전의 평균 복잡도는 '13.8'이고, 크기는 '100'이다.

표 1. 각 노드의 상태 정보

	복잡도	참조확률	무게	새로운 정보량
E	5	1 / 21	10	21.9
1	2	2 / 21	5	6.7
2	4	1 / 21	5	17.5
3	5	1 / 21	7	21.9
4	3	2 / 21	9	10.1
5	8	1 / 21	17	35.1
6	8	1 / 21	13	35.1
7	3	3 / 21	7	8.4
8	5	4 / 21	10	11.9
9	3	3 / 21	8	8.4
10	4	2 / 21	9	13.5

4.2 유전 알고리즘

그래프에 존재하는 모든 함수의 인라인 수행 여부는 2^n 개의 경우를 가지고, n(함수의 수)의 값이 커질수록 그 수는 지수적으로 증가한다. 이런 유형의 문제를

일반적인 알고리즘을 사용하여 풀 경우 n의 수가 작을 경우에는 별 문제가 되지 않지만, n의 수가 커질 경우 문제를 해결하는데 많은 시간이 필요해지고, 또한 최적의 해를 찾는다는 보장이 없다. 인라인 유무의 문제처럼 해결해야 될 문제의 범위가 큰 경우 일반적으로 잘 알려진 유전 알고리즘을 사용하여 문제에 대한 답을 찾아낸다[3][5]. 따라서 본 실험에서도 함수의 인라인 수행 여부에 대한 결정을 유전 알고리즘 적용하여 해결하는 방안을 제시한다.

4.2.1 염색체 인코딩

염색체(chromosome)는 인라인이 가능한 간선의 수로 L개의 비트를 갖는 이진 스트링으로 나타낸다. 각 비트의 값은 인라인 수행 유무를 나타내며 0 또는 1의 값을 가진다. 이 실험 그래프 모델은 노드간에 20개의 연결된 간선을 가지고 있으므로 염색체 배열의 길이 L = 20의 값을 가지므로, 하나의 염색체는 <그림 4>와 같이 표현한다.

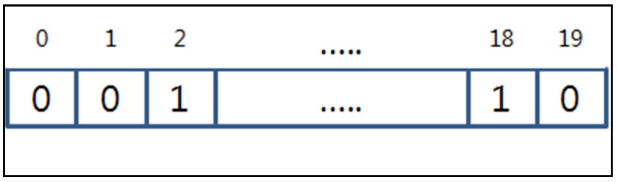


그림 4. 염색체 배열 구조

4.2.2. 임계치 설정

인라인의 수행은 기본적으로 크기의 증가로 인한 성능의 저하를 가져 올 수 있다. 기존의 연구들에서 여러가지 난독화 기법을 적용한 후 변화되는 코드의 크기를 관찰했을 때 평균적으로 원래 크기의 1.6배 정도의 증가 보였다[11][12]. 따라서 본 연구에서는 크기의 증가를 고려하여서 유전 알고리즘을 인한 크기의 변화를 60%로 제한하기 위한 임계치를 적용하여 수행 한다.

4.2.3. 유전자 알고리즘 인자

알고리즘의 수행을 위해 초기해 집단 P의 크기는 20으로 정하였으며, 임의의 값을 발생시켜 무작위로 초기해의 생성을 결정하였다. 세대형 알고리즘을 기반으로 하여 알고리즘의 정지조건을 10000 세대까지의 반복 수행으로 정하였다. 해집단 P의 크기가 20이므로 한 세대에 20개의 자손을 생성하여 부모 해집단과 비교된다. 선택연산은 룰렛 휠 선택방법을 적용하여 해집단 P 에서 전체 해들의 품질의 평균값을 구하고, 품질값에 따라 선택되어질 확률이 높아진다. 교차연산은 단순한 3점 교차 연산을 수행한다. 염색체의 변이 확률은 0.6의 값을 주었다.

생성된 자손들과 해집단을 이루는 부모해와의 비교를 위해서 각 해의 품질을 나타내는 평가값이 필요하다. 여기서 평가함수는 다음 <식 8>으로 표현하며, 염색체의 값에 따른 인라인 후의 증가하는 복잡도의 비율로 결정된다.

$$\mathrm{Fit(p)} = \begin{cases} \dfrac{C_{New}(G)}{C_{Ori}(G)}, & \mathrm{If}(S < T) \\ 0, & \mathrm{Otherwise} \end{cases} \quad \cdots \text{ 식(8)}$$

여기서

S : 인라후 증가된 크기

T : 임계치

$C_{Ori}(G)$: 인라인 전의 그래프의 복잡도

$C_{New}(G)$: 인라인 후의 변화된 그래프 복잡도

$\mathrm{Fit(p)}$: 염색체에 할당되는 적합도 값

Fit는 인라인 후 복잡도 증가 비율이 되며, 만약 증가된 크기가 설정된 임계치를 넘는다면 Fit는 '0'의 값을 가진다. '0'의 적합도값을 가진 염색체는 해집단에 포함되지 못하고 제거된다.

위와 같이 파라미터를 설정하고 1만 세대까지 수행한 결과 47%의 복잡도 증가로 수렴하였다.

5. 결 론

본 논문에서는 난독화를 위해 적용할 수 있는 인라이닝을 효율적으로 수행하기 위한 방법을 찾기 위해서 유전 알고리즘을 적용하는 것을 제안하였다. 또한 정보량과 제어흐름 복잡도를 병합하여 새로운 정보량을 정의하여, 난독화 결과의 효용을 평가하기 위한 척도를 제안했다.

하지만 본 연구에서 실험한 모델의 대상이 실제 프로그램의 코드가 아닌, 프로그램의 추상적인 개념을 나타내는 호출 그래프이다. 또한 인라인 시 내부 복잡도의 변경을 단순히 100% 증가된다고 가정하고 알고리즘을 적용하였다. 따라서 향후 과제로는 실제 프로그램을 대상으로 제안한 기법의 유용성을 평가하며, 내부 복잡도의 변경에 대해서도 실제와 유사한 조건으로 100% 이하의 다양한 변수를 주어 실험을 할 것이다. 또한 본 문제에 적합한 다른 최적화 알고리즘을 적용하여 비교하도록 한다.

참고 문헌

[1] Eldad Eilam, "Reversing: Secrets of Reverse Engineering", WILEY, 617p, April 2005.

[N.Naeem2006]N.Naeem, M.Batchelder, L.Hendren, "Metrics for Measuring the Effectiveness of Decompilers and Obfuscators", In Program Comprehension, 15th IEEE International Conference (ICPC'07), pp. 253-258, Jun 2007.

[2] C. Collberg, C.Thomborson and D.Low, "A Taxonomy of Obfuscating Transformations", Technical Report 148, University of Auckland, July 1997.
[3] J.Cavazos and M.O'Boyle, "Automatic Tuning of Inlining Heuristics", In Proceedings of the Supercomputing Conference on High Performance Networking and Computing, Nov. 2005.
[4] R.Leupers, P.Marwedel, "Function Inlining under Code Size Constraints for Embedded Processors", In Proc of the Intl.Conf. on Computer-Aided Design, San Jose, CA. 253-256. 1999.
[5]M. Li, H. Wang, "GA Based Inlining Optimization in Front-End Synthesis of Embedded Software",ASIC 2003, Proceedings, 5th International Conference on , Page : 341-343 Vol1, 2003
[6] W.Harrison, "An Entropy-Based Measure of Software Complexity", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO 11, NOVEMBER 1992
[7] J.S.Davis and RRichard J. Leblanc, "A Study of the Application of Complexity Measures", IEEE Transactions on Software Engineering, Vol.14, No.9, pp.1366-1372, 1988
[8]T.J. McCabe, "A complexity measure", IEEE Trans. SoftwareEng., pp. 308-320. Dec 1976
[9] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, "Properties of Control-Flow Complexity Measures," IEEE Trans. Software Eng., vol. 17, no. 12, pp. 1289-1295, Dec. 1991.
[10] W.Harrison, K.I.Magel, R.Kluczny and A. Dekock, "Applying software complexity metrics to Program maintenance",Computer, vol.15, no. 9, pp. 65-79, 1982.
[11] 채영현, "소프트웨어 보안을 위한 코드 난독화 도구의 개발", Master Thesis, Dep. Electrical Engineering and Computer Science. KAIST, Feb. 2007
[12] O.Toshio, S.Yusuke, S.Masakazu and M.Atsuko, "Software Obfuscation on a Theoretical Basis and Its Implementation", IEEE Trans. Fundamentals, E86-A(1), pp.176-186 January 2003.