고급레벨 Obfuscation에서 자료구조 변환 및 제어흐름 변환의 비교

장혜영° 조병민 조성제
단국대학교 컴퓨터과학전공
hychang@dankook.ac.kr° onlyhr98@gmail.com sjcho@dankook.ac.kr

Comparison of Data Structure and Control Flow Transformations in High-level Obfuscation

Hyeyoung Chang° ByoungMin Cho, Seongje Cho
Dept. of Computer Science, Dankook University

요 약

자동화된 obfuscation은 보안 목적으로 코드를 이해하기 어렵게 만들어 역공학 공격을 방어하는데 가장 효과적인 방식이라고 알려져 있다. 본 논문에서는 역공학 공격과 같은 소프트웨어 지적재산권의 침해로부터 마이크로소프트사의 비주얼 C++소스 프로그램을 보호하기 위한 obfuscator를 구현하였다. 그리고 obfuscation 알고리즘에 따라 potency(복잡도)를 측정하여 분석하여 각 알고리즘을 평가하였다. 또한 cost(비용)과 어셈블리 코드를 비교하여 obfuscator의 성능과 유효성을 평가하였다. 그 결과, 변환된 소스 코드가 실행시간 오버헤드를 일부 유발시키긴 하지만 프로그램 보호에는 효과적임을 알 수 있었다.

1. 서 론

국가/군사 기밀 데이터, 기업 데이터, 병원 기록, 개인 사생활 정보 등의 대부분은 소프트웨어에 의해 처리된다. 소프트웨어 자체 또한 데이터의 형태로 관리되며 인터넷 기반의 컴퓨팅 환경에서 소프트웨어는 절도(theft) 및 오용(misuse)공격에 취약하다. 많은 경우, 기밀 데이터를 훔치거나 변조하기 위해 공격자는 그 데이터를 보호하는 소프트웨어 프로그램을 공격한다. 실제로 한 응용으로부터 추출된 귀중한 코드 일부가 경쟁자의 코드에 통합된 사례가 있다[1]. 이러한 위협은 최근 계산 자원들의 증가, Objdump, IDA Pro, SoftICE와 같은 역공학 도구(기술)의 발전으로 더 일반화 되었다[2,3].

소프트웨어의 지적재산권을 보호하는 대표적인 기법에는 불법복제를 방어하기 위한 워터마킹(watermarking), 변조 공격을 방어하기 위한 변조방지(tamper-proofing)와 암호화, 역공학을 방어하기 위한 obfuscation 등이 있다[1,4-9]. 이들 기법 중 본 논문에서는 obfuscation에 초점을 맞춘다.

Obfuscation은 공격자가 이해할 수 없도록 코드를 변환하여 최대한 복잡하고 혼란스럽게 만드는 것으로 언어 구조(language constructs), 데이터, 알고리즘, 제어흐름 등을 숨겨서 분석하기 어렵게 하면서 프로그램의 본래 기능(semantics)은 그대로 유지된다.

본 논문에서는 소스 분석 공격 및 역 어셈블 공격으로부터 마이크로소프트사의 비주얼 C++ 소스 프로그램을 보호하기 위한 obfuscation 도구, 즉 obfuscator를 개발하였다. 이때, obfuscation을 데이터 변환(data obfuscation)과 제어 변환(control obfuscation) 알고리즘 각각 두 개씩 구현한다. Obfuscation의 대상이 데이터만 변환(data obfuscation)될 때와 제어만 변환(control obfuscation) 그리고 두 개다 적용했을 때, 해독 및 역공학 공격에 얼마나 강한지를 평가해 본다.

이 논문의 구성은 다음과 같다. 2장에서는 obfuscation과 관련된 연구를 기술하고, 3장에서는 obfuscation 분류하고 사용된 알고리즘들에 대해 설명한다. 4장에서는 Obfuscation의 성능을 측정하는 방법과 측정 결과를 기술하고 5장에서 결론을 맺는다.

2. 관련 연구

소프트웨어를 읽고 이해하기 어렵게 만들어 역공학 공격을 방어하여 소프트웨어에 포함된 비밀 정보를 보호하는 것이 obfuscation이다[1-11]. Obfuscation은 그 적용 레벨에 따라 C++ 소스 프로그램과 같은 고급 언어 수준에서 적용되는 '고급 obfuscation', Java 바이트코

드나 마이크로소프트 중간언어(MSIL)에서 적용되는 ' 중급 obfuscation ', 기계어 코드에서 적용되는 ' 저급 obfuscation '으로 분류할 수 있다[9].

소프트웨어 보호 도구로는 Cloakware, DashO 및 Dotfuscator, Semantic Designs의 소스 코드 obfuscator, Kava(Konfused Java), JHide 등이 있다[1, 10, 12]. Cloakware는 C 소스 코드에 대한 제어 및 데이터 흐름 변환을 수행한다. DashO 및 Dotfuscator는 각각 Java와 MSIL에 대해 무의미 코드를 제거하고(dead code removal) 식별자의 이름을 변경하여 준다. 마이크로소프트사의 MSIL의 경우 식별자와 알고리즘 등이 메타데이터 형태로 되어 있어 역공학 분석을 통해 소스 코드를 유추할 수 있다. 따라서 소프트웨어의 지적재산권을 보호하기 위해 마이크로소프트사는 Dotfuscator[13]를 사용할 것을 권장하고 있다. Semantic Designs사의 소스 코드 obfuscator는 여러 고급 언어에 대해 식별자 이름 변경 및 불필요한 공백 제거 등과 같은 단순한 기능을 수행한다.

Java 바이트 코드는 플랫폼 독립적이라는 장점이 있으나 바이트 코드를 구성하는 메타언어가 내포하는 정보들로 인해 프로그램의 핵심 알고리즘들이 역공학에 취약하다는 단점을 갖는다. 따라서 Kava와 JHide 등을 포함하여 자바 바이트 코드에 대한 obfuscator 연구가 활발하며 상용화된 도구도 다수 개발되었다[10, 12, 14].

C++ 등의 고급언어는 문법이 복잡한데 비해 어셈블리 코드는 각 코드들이 기계어와 1대1로 대응되고 제한적인 명령어들 및 구조를 갖기 때문에 어셈블리 코드의 obfuscation을 위한 연구가 활발히 진행되어 왔다[15][16]. 그러나 이 방법은 시스템에 매우 종속적이라는 문제점을 가지므로 이를 보완하기 위해 시스템 독립적인 어셈블리 코드로 변환하여 처리된 소스 코드를 획득하는 연구도 수행되었다.

본 논문에서는 마이크로소프트사의 C++ 소스를 변환하는 고급레벨의 obfuscation을 다룬다.

3. Obfuscaton의 분류 및 Obfuscation 알고리즘

Obfuscation은 대상에 따라 레이아웃 변환(Layout obfuscation), 데이터 변환(Data obfuscation), 제어 변환(Control obfuscation), 변환 예방(Preventive transformation)으로 분류할 수 있다. 레이아웃 변환은 포맷 변경, 주석 제거 등 레이아웃을 목표로 변환하는 방법이고, 데이터 변환은 데이터의 구조를 목표로 변환하며, 제어 변환은 개별 프로그램 함수들에서 제어 흐름을 이해하기 어렵게 만드는 방식이다. 이러한 방식들은 사람이 코드를 판독하기 어렵게 만드는 방법이고 변환 예방은 이와 달리 역공학 도구들이 코드에 접근하기 어렵도록 하는 것이다. 이들 중 obfuscation에 가장 많이 사용되는 레이아웃 변환과 데이터 변환의 알고리즘을 구현하여 실제 얼마나 obfuscation이 적용됐는지를 확인한다.

3.1 데이터 변환

소스레벨 프로그램의 특정 데이터(변수)를 대상으로 하며 그림1과 같이 데이터의 저장(Storage), 인코딩(encoding), 통합(aggregation), 순서화(ordering) 등에 영향을 주어 obfuscation을 달성한다.

본 논문에서는 변수 분할 알고리즘과 배열차원 늘리기 알고리즘을 구현하여 적용시켰다.

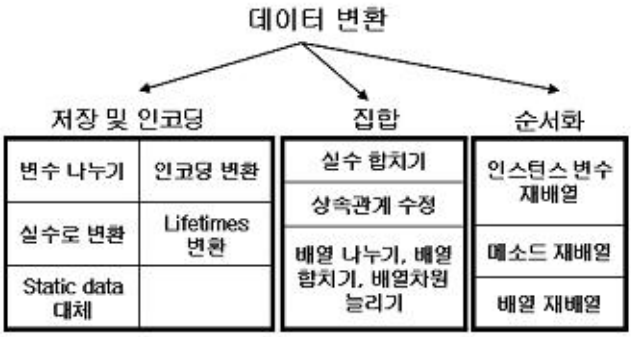

그림 1 데이터 변환의 분류

3.1.1 변수 분할 알고리즘

변수 분할(split variable) 알고리즘은 논리형(boolena)이나 크기가 제한적인 다른 기본타입 변수들을 여러 개의 변수로 나누거나 변수의 사용자 정의 구조체 또는 값을 반환하는 함수로 대체하는 방법이다. 본 논문에서는 32비트의 정수형 변수를 16비트의 unsigned short형의 두 변수로 나누어 값을 할당하는 부분을 함수로 대체하는 방식으로 구현했다. 아래의 그림 2는 변수 분할 알고리즘의 적용 전과 후의 코드 변화 모습이다.

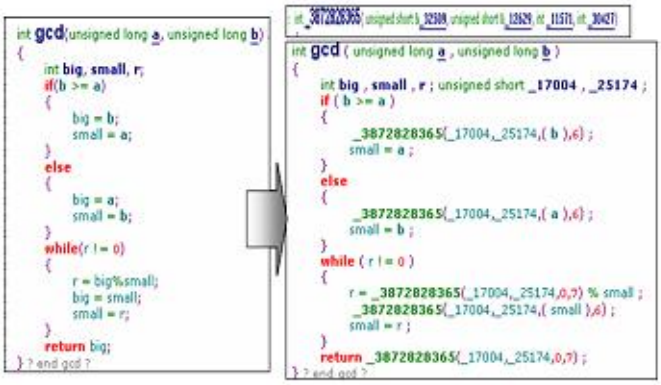

그림 2 변수 분할 알고리즘의 예

3.1.2 배열차원 늘리기 알고리즘

배열차원 늘리기(fold array) 알고리즘은 배열의 특성 및 용도를 공격자와 공격도구가 분석하기 어렵도록 만드는 기법이다. 아래의 그림 3과 같이 1차원 배열을 2차원 배열로 차원을 증가시킴으로써 두 개의 배열이라는 혼란을 줄 수 있다.

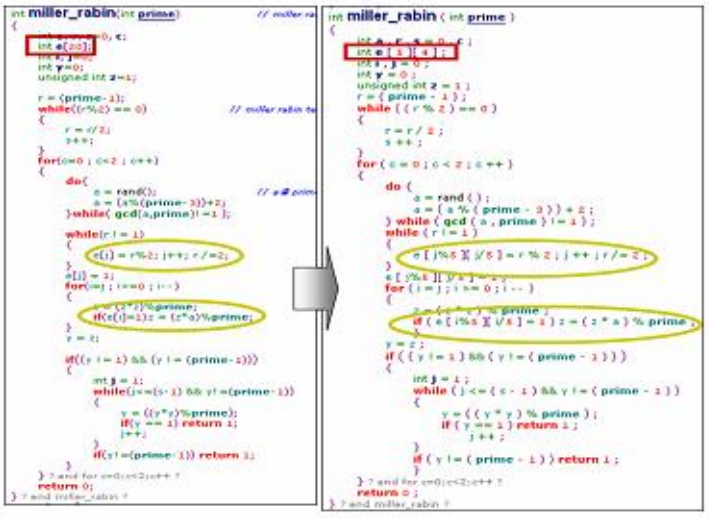

그림 3 배열차원 늘리기 알고리즘의 예

3.2 제어 변환

제어 변환은 처리 순서의 변경이나 기능상에 변화를 주지 않는 제어 흐름을 추가하여 이해하기 어렵게 만드는 방식이다. 제어 변환의 대표적 예인 '불투명한 조건가(opaque predicates)'는 조건부가 항상 참이거나 거짓인 조건문을 사용한다. 이때 항상 수행되는 조건부는 의미 있는 코드를 포함하는 반면, 다른 조건부는 임의 코드를 포함하게 만든다. 따라서 제어 변환은 자동분석 공격에 대항할 수 있게 구현해야 한다.

본 논문에서는 집합의 제어를 변환하는 루프조건 확장 알고리즘(loop Unrolling)과 계산의 제어를 변환하는 루프 조건 확장 알고리즘을 구현한다.

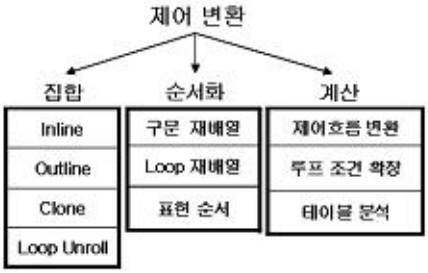

그림 4 제어 변환의 분류

3.2.1 루프조건확장 알고리즘

루프조건확장(Extend loop condition) 알고리즘은 조건문이나 반복문에 조건을 추가하여 종료 조건을 복잡하게 만드는 기법이다. 이때 어떠한 조건식을 추가하더라도 본래 의도한 종료시점에는 변화가 없어야 한다.

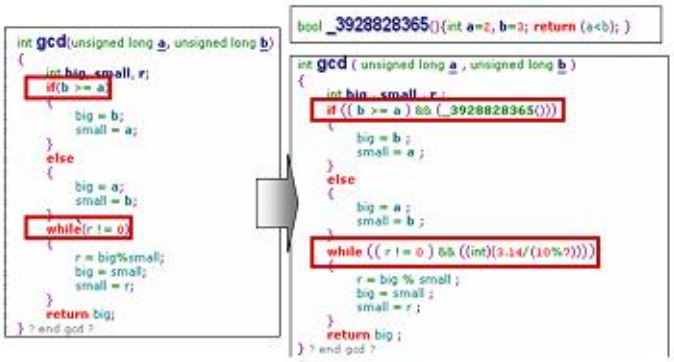

그림 5 루프조건확장 알고리즘의 예

3.2.2 루프 Unrolling 알고리즘

루프 Unrolling 알고리즘은 조건 분기문이 포함될 수 밖에 없는 루프의 카운트를 줄임으로써 조건 분기문을 최대한 회피하는 방법이다. 즉, 아래의 그림 6과 같이 루프의 몸체를 여러 번 반복시켜 루프가 반복되는 횟수를 줄임으로써 혼란을 야기 시킬 수 있다.

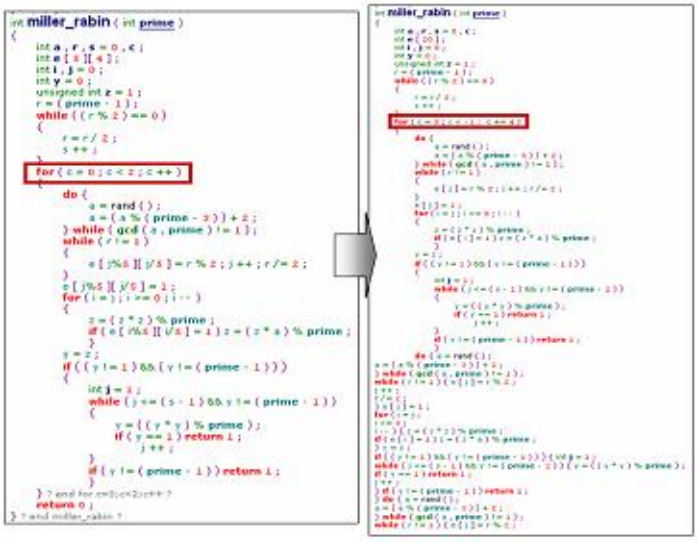

그림 6 루프 Unrolling 알고리즘의 예

4. Obfuscation 성능 측정

Obfuscation의 결과를 평가하는 기준으로는 크게 potency, resilience, 비용(오버헤드) 등이 있다 [10][15].

Potency는 원본 코드보다 변환된 코드가 얼마나 이해하기(분석하기) 어려운 가를 측정하는 정도로 클수록 좋다. Potency는 변환된 프로그램을 역공학하여 이해하는데 드는 시간이 원래 프로그램을 역공학하여 이해하는데 드는 시간 보다 더 길다는 의미인 obscurity를 포함한다.

Resilience는 역변환하는 자동화된 도구를 구현하는 것의 어려운 정도, 또는 자동화된 역변환(deobfuscation) 도구를 실행하는데 드는 시간의 정도를 나타내는 것으로 클수록 좋다. 바이너리 코드를 자동 deobfuscation하는 도구가 일부 있긴 하지만 보통 역공학 도구를 이용하여 공격자가 소스를 일일이 확인하며 역추적하고 있기 때문에 Resilience 값을 측정하기란 쉽지 않다. 비용(cost)은 실행시간과 오브젝트 파일의 크기를 비교하여 측정한다.

본 논문에서는 potency를 중심으로 성능측정을 해본다. 실험에 사용된 C++ 소스 프로그램은 RSA 알고리즘과 Deap 트리 알고리즘을 이용했다.

4.1 Potency

Potency를 측정하기 위해서는 아래와 같이 다섯 가지의 항목을 계산하게 된다.

- **p(1)** : 프로그램의 길이로 바이트의 수를 측정한다.
- **p(2)** : 함수의 순환복잡도를 계산하는 것으로 Edges(에지수) − Nodes (노드수) + connected component (연결 요소)를 계산한다.
- **p(3)** : 각 함수의 if, while, for 등으로 인한 scope의 레벨을 계산한다.
- **p(4)** : 함수에서 참조되는 지역변수의 증가를 계산한다.
- **p(5)** : 함수에서 사용되는 파라미터 및 전역변수의 증가를 계산한다.

표 1. 각 obfuscation 측정 결과

	원본	변수분할	배열차원 늘리기	데이터 Obfuscation	루프조건 확장	루프Unrolling	제어 Obfuscation	데이터 + 제어
p(1)	10782	11730	14611	**14669**	15202	14293	**15397**	**18525**
p(2)	93	93	95	**93**	114	114	**111**	**97**
p(3)	29	34	34	**37**	77	34	**69**	**74**
p(4)	58	71	62	**77**	127	61	**61**	**148**
p(5)	12	12	12	**12**	12	12	**12**	**12**
potency	**10,974**	**11,940**	**14,814**	**14,888**	**15,532**	**14,514**	**15,650**	**18,856**

위의 표1의 결과를 살펴보면 데이터 obfuscation보다 제어 obfuscation이 순환복잡도와 scope를 현저히 증가시켜 복잡하게 만듦을 확인할 수 있다. 데이터 obfuscation과 제어 obfuscation을 각각 적용할 때 보다 같이 적용했을 때가 더욱 복잡해짐을 확인할 수 있는데 이때 p(2), 즉 순환 복잡도가 줄어드는 이유는 데이터 obfuscation을 적용함으로써 코드가 변형되어 obfuscation 알고리즘의 적용 대상에서 제외됨으로써 줄어들었다. 하지만 그 외의 항목들은 증가한 것을 확인할 수 있다. 아래의 그림 7의 위의 그림은 obfuscation을 적용하기 전의 프로그램이고 아래의 그림은 데이터와 제어 obfuscation 4가지 모두 적용했을 때의 함수 호출도이다. IDA Pro는 공격자들이 역공학을 위해 사용되는 대표되는 프로그램 중 하나이다. 이를 이용하여 함수 호출 그래프를 살펴봤다. 그림 7을 보는 것과 같이 obfuscation을 적용했을 때가 적용전보다 더욱더 복잡해짐을 확인할 수 있다.

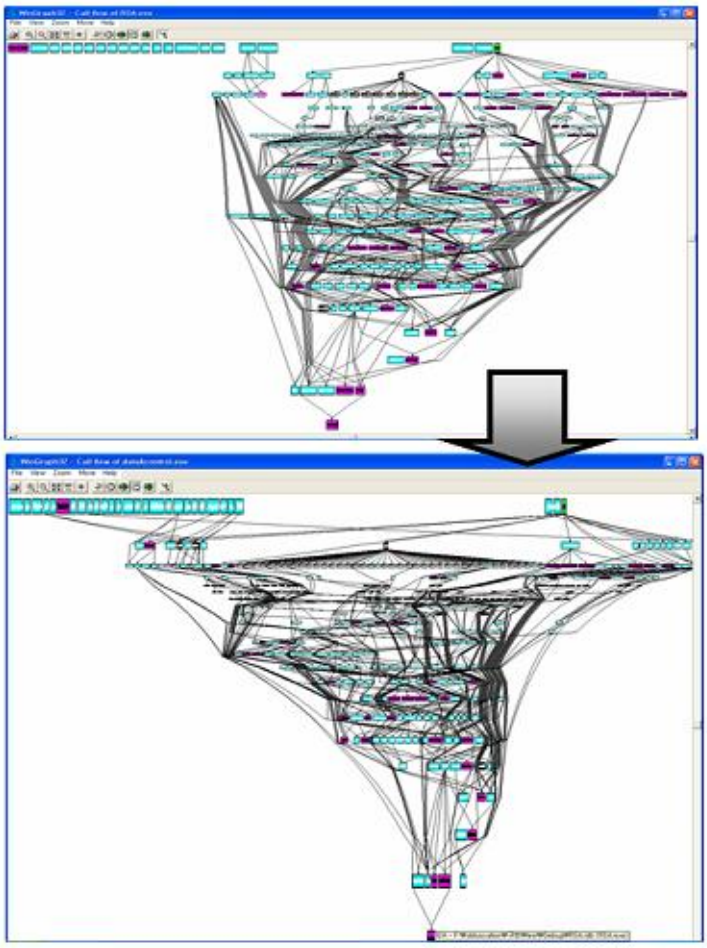

그림 7 Obfuscation 적용 전·후의 함수 호출 흐름도

4.2 비용

비용은 obfuscation을 적용 전후의 프로그램 실행 시간과 공간을 계산하고 Deap 트리 프로그램을 대상으로 비용을 계산해 보았다. 실행시간은 Deap 트리 프로그램에 500회 삽입, 삭제 연산을 10회 수행한 후 평균값을 계산하였다. 그 결과 원 프로그램은 1.783초, obfuscation이 적용된 프로그램은 2.980초가 측정되었고 약 1.7배 증가함을 확인할 수 있었다.

공간에 대한 비용은 아래의 표2와 같이 실제 메모리에 적재되는 오브젝트 파일 및 중요한 파일들의 크기를 측정하여 비교했다. 그 결과 파일의 크기는 전체적으로 증가하고 실제 프로그램의 기계어 코드를 담고 있는 오브젝트 파일의 크기는 약 1.3배 증가한 것을 확인할 수 있다.

표 2 적용 전후의 파일 크기 비교

파일 종류	원 프로그램	Obfuscated 프로그램
소스파일 크기(KB)	4.56	7.27
어셈블리 파일 크기(KB)	28.40	44.60
오브젝트 파일 크기(KB)	6.74	9.62
EXE 파일 크기(KB)	48.00	52.00

4.3 어셈블리 코드 비교

루프조건 확장 알고리즘이 적용되기 전후의 소스를 컴파일 할 때 최적화시켜 어셈블리 코드를 서로 비교해봤다. 비교된 코드는 Deap 트리 알고리즘 중 $\log_2 X$의 값을 계산하기 위한 함수이 일부분으로 비교결과가 그림 8에서 확인할 수 있다.

	대상	코드

원 프로그램	소스	...
		int log2(int p)
		{
		int product = 1
		int i = 0;
		while (product <= p)
		{
		product += 2;
		i++;
		}
		i--;
		return i;
		}
		...

	어셈블리	... mov eax, DWORD PTR _product$[ebp] cmp eax, DWORD PTR _p$[ebp] jg SHORT $L1728 ...

Obfuscated 프로그램	소스	...
		int log2 (int p)
		{
		int product = 1 ;
		int i = 0 ;
		while ((product <= p) && (_156296810197112@@)
		{
		product += 2 ;
		i ++ ;
		}
		i -- ;
		return i ;
		}
		...

	어셈블리	... mov eax, DWORD PTR _product$[ebp] cmp eax, DWORD PTR _p$[ebp] jg SHORT $L1841 call ?_156296810197112@@YAHXZ test eax, eax je SHORT $L1041 ...

그림 8 obfuscation 적용 전·후의 어셈블리 코드 비교

위의 결과를 살펴보면 while문에 조건식을 확장해주는 루프 조건 확장 알고리즘이 적용된 것을 볼 수 있고 어셈블리 코드에서 보는 것과 같이 최적화시켜 컴파일해도 obfuscation 알고리즘이 적용 되어 있음을 확인 할 수 있다. 이는 본 논문에서 적용한 obfuscation이 기계어 코드에 영향을 미쳐 역공학 공격을 어렵게 만든다는 것을 보여준다.

5. 결론

소프트웨어의 지적재산권을 침해하는 공격 중의 하나가 역공학 공격이고 이 역공학 공격을 방어하는 가장 좋은 방법이 obfuscation이다. 본 논문에서는 C++ 소스 프로그램을 대상으로 변환하는 obfuscator를 개발하였다. 그리고 데이터를 변환시키는 데이터 obfuscation과 제어흐름을 변환시키는 제어 obfuscation 각각 두개씩 네 개의 알고리즘을 적용시켜 그 결과를 분석해 보았다. 실험결과 obfuscation 알고리즘의 적용으로 일부 오버헤드가 유발되었으나 프로그램 보호를 위한 도구의 기능적 측면은 충분히 성취된 것으로 확인되었다.

참고문헌

[1] C. Collberg and C. Thomborson, "Watermarking, Tamper-proofing, and Obfuscation-Tools for Software Protection," *IEEE Trans. Software Eng.*, Vol. 28, no. 8, pp. 735-746, 2002.

[2] Colin W. Van Dyke, "Advances in Low-Level Software Protection," Ph. D. Thesis, Oregon State University, Jun. 2005.

[3] C. Collberg, C. Thomborson, and D. Low, "*A Taxonomy of Obfuscating Transformations*," Tech. report 148, Dept. of Computer Science, University of Auckland, New Zealand, 1997; www.cs.arizona.edu/~collberg/Research/Publications/ColbergThomborsonLow97a/

[4] C. Collberg, G. Myles, and A. Huntwork, "Sandmark - A Tool for Software Protection Research," IEEE Security & Privacy (Software Protection), pp. 40-49, Jul./Aug. 2003.

[5] G. Naumovicb and N. Memon, "Preventing Piracy, Reverse Engineering, and Tampering," IEEE Computer, pp. 64-71, Jul. 2003.

[6] Bin Fu, Golden G. Richard III, Yixin Chen, and Adbo Husseiny, "Some New Approaches For Preventing Software Tampering," Proc. of the 44th ACM Southeast Regional Conference (ACM SE'06), pp. 655-660, Mar. 2006.

[7] C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings", *Proceedings of POPL '99 of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 311-324, Mar. 1999.

[8] P. C. van Oorschot, "Revisiting Software Protection", 6th ISC 2003, Springer-Verlag LNCS 2851, pp. 1-13, Oct. 2003

[9] M. R. Stytz and J. A. Whitaker, "Software Protection: Security's Last Stand?", IEEE Security & Privacy, 1(1), pp. 95-98, Jan. 2003.

[10] C. Collberg, C. Thomborson, and D. Low, "*A Taxonomy of Obfuscating Transformations*," Tech. report 148, Dept. of Computer Science, University of Auckland, New Zealand, 1997; www.cs.arizona.edu/~collberg/Research/Publications/ColbergThomborsonLow97a/

[11] B. Barak et al., ""On the (Im)possibility of Obfuscating Programs," *Advances in Cryptology-Crypto 2001, Proc. 21st Ann. Int'l. Cryptology Conf.*, LNCS 2139, Springer-Verlag, pp. 1-18, 2001.

[12] Levent Ertaul, and Suma Venkatesh, "JHide-A Tool Kit for Code Obfuscation", Proceedings of the 8th IASTED International Conference Software Engineering and Applications (SEA 2004), Nov. 2004.

[13] .NET Obfuscator (Dotfuscator), http://www.preemptive.com/products/dotfuscator/index.html

[14] Christian Collberg, Clark Thomborson, and Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", ACM SIGPLAN-SIGACT POPL'98, Jan. 1998.

[15] G. Wroblewski, "A General Method of Program Code Obfuscation," Ph.D. Dissertation, Wroclaw University, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, Jun. 2002.