Visual C++ 소스코드를 위한 Obfuscation 도구 구현

조병민*○, 장혜영*, 노진욱*, 오현수*, 정민규**, 이승원**, 박용수+, 우제학++, 조성제*
*단국대학교, **서울대학교, +한양대학교, ++㈜코어트러스트
onlyhr98@gmail.com○, {neocastle, pyxis3, chenil}@dankook.ac.kr, april18.mg@gmail.com,
{leesw, yspark}@ssrnet.snu.ac.kr, jhwoo@coretrust.com, sjcho@dankook.ac.kr

Implementation of an Obfuscation Tool for Visual C++ Source Code

ByoungMin Cho*○, Hye-Young Chang*, JinUk Noh*, Hyunsoo Oh*, Mingyu Jung**,
Seungwon Lee**, Yongsoo Park+, Jehak Woo++, Seongje Cho*
*Dankook University, **Seoul National University, +Hanyang University, ++Coretrust, Inc.

요 약

최근 소프트웨어의 주요 알고리즘 및 자료구조 등의 지적재산권을 역공학 분석과 같이 악의적인 공격들로부터 보호하기 위한 연구가 이루어지고 있다. 본 논문에서는 산업 현장에서 많이 사용되는 Visual C++ 또는 MFC로 작성된 프로그램의 소스 코드를 역공학 공격으로부터 보호하기 위한 Obfuscation 도구를 구현하고 그 성능을 평가한다. 구현된 도구는 3가지 Obfuscation 알고리즘을 적용하여 소스 코드를 생성하며 생성된 소스 코드들은 가독성이 떨어지고 역공학 분석이 어렵도록 변환되지만, 프로그램의 본래 기능은 그대로 유지하며 성능상의 변화가 크지 않음을 실험을 통해 확인할 수 있었다.

1. 서 론

IT기술의 발전과 더불어 다양한 SW들이 개발되어 사용되고 있다. 그러나 이러한 SW들은 현재 해킹과 불법복제로 인한 지적재산권 침해의 위협에 직면해 있는 실정이다. 실제로 국내에서 개발 및 판매되는 SW의 경우에도 그 피해 규모가 매년 수백억에 이르는 것으로 집계되고 있다[1]. 또한 인터넷을 통해 광범위하게 유통되는 웹 어플리케이션의 경우에도 시스템 독립적인 특성을 유지하기 위해 SW의 정보를 정형화된 형태로 저장하는 경향이 있어 결코 안전하다고 할 수 없다.

본 논문에서는 소스 분석 공격 및 역공학(reverse engineering) 공격으로부터 소스코드를 보호하기 위해, C++ 소스 코드를 입력으로 받아 스크램블(scramble)하여 기능적으로 동일한 또 다른 소스 코드를 생성하는 Visual C++ 소스 코드 obfuscator를 설계하고 구현한다. 이러한 연구는 소스코드의 지적재산권을 보호하고 목적코드에 대한 역공학적 분석을 어렵게 한다.

Obfuscation이란 컴퓨터 과학의 측면에서 기존코드의 기능은 그대로 유지하면서 역공학 분석이 어렵도록 코드를 재구성하는 작업을 말한다[8][9][10]. 완전한 역공학 분석 공격의 방지는 불가능하지만 암호화 기법처럼 분석에 필요한 비용과 시간을 증가시켜 난해하게 만드는데 그 의의가 있다.

본 논문에서는 구성은 다음과 같다. 2장에서는 지적 재산권 보호를 위한 기술과 현재까지 연구된 obfuscation 기법에 대해 살펴보고 그 기법들의 대상을 기준으로 한 연구에 대해 간략한 설명과 본 논문에서 구현한 도구의 차이점을 소개한다. 3장에서는 구체적인 obfuscator 전체 구조와 기반 기술에 대해 설명하고 4장에서는 obfuscation 알고리즘을 구현한 내용을 기술한다. 5장에서는 실험 및 성능평가에 대한 결과를 다룬다. 마지막 6장에서는 결론 및 향후 과제에 대해서 논의한다.

2. 관련 연구

소프트웨어의 지적 재산권을 보호하는 기술로는 Encryption, Server-side execution, Trusted native Code, Obfuscation 등이 있다. 여기서 Server-side execution 기법은 서블릿과 같이 모든 컴퓨팅은 서버에서 수행하고 그 결과만 클라이언트에게 제공하는 것으로 악의적인 역공학 공격을 사전에 막는 효과가 있다. 또 Trusted native code 기법은 어플리케이션을 개개의 사용자의 시스템에 맞추어 제공하는 것으로 역공학 분석을 어렵게 만드는 효과가 있다. 본 논문에서 다루고자 하는 Obfuscation은 어플리케이션의 기능은 원본 그대로 유지

하고 복잡도를 증가시켜 분석에 필요한 시간과 비용을 증가시키는 기법으로 Server-side execution 기법처럼 공격을 원천 봉쇄할 수는 없지만 서버의 자원으로 수행하기 힘든 어플리케이션의 보호엔 효과적이다[2].

Obfuscation은 크게 Layout, Data, Control Obfuscation과 Preventive Transformation 등의 4가지로 구분된다. Layout Obfuscation은 소스코드의 주석이나 포맷, 사용된 변수의 이름 등을 변형하는 기법이고, Data Obfuscation은 메모리의 스택이나 힙에 저장되는 데이터의 구조를 변형시키는 기법이다. Control Obfuscation은 어플리케이션의 일련의 수행 과정을 변형하는 기법이다[10].

2.1 어셈블리 코드 Obfuscation

어셈블리 코드는 어셈블러를 통해 기계어로 번역되어 실행 가능한 프로그램이 된다. 어셈블리어는 CPU 제조사, 컴파일러 제작사에 따라 각기 다른 특징을 갖게 된다. 복잡한 문법의 C, C++언어에 비해 어셈블리 코드는 각각의 코드들이 기계어와 1대1로 대응되고 제한적인 명령어들과 구조를 갖기 때문에 Obfuscation을 위한 연구가 비교적 활발히 진행되고 있다[3]. 그러나 C나 C++언어로 작성된 프로그램을 하나의 시스템에서 컴파일할 경우 동일한 어셈블리 코드가 생성될 수 있으며, 컴파일된 기계어도 역공학 공격이나 디어셈블러를 통한 분석에 쉽게 노출될 수 있다.

2.2 자바 바이트 코드 Obfuscation

자바 언어로 작성된 프로그램은 컴파일 시 자바 바이트 코드를 생성한다. 이 바이트 코드는 JVM(Java Virtual Machine)을 통해 시스템 독립적으로 실행할 수 있다. 여기서 JVM은 기계어 코드가 아닌 자바 바이트 코드를 해석하는 interpreter역할을 수행한다. 그러나 자바 바이트 코드는 시스템 독립적이라는 장점과 함께 내포하고 있는 정보들로 인해 프로그램의 핵심 알고리즘들이 노출될 수 있는 단점을 갖는다. JHide[4]등의 도구들은 자바 바이트 코드를 대상으로 Obfuscation을 수행한다.

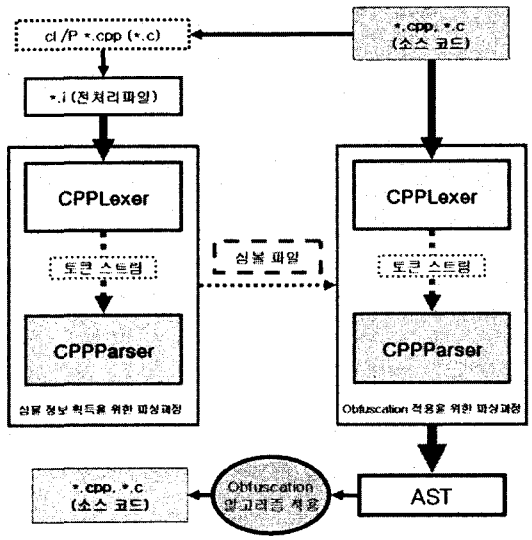
2.3 MSIL 코드 Obfuscation

Microsoft는 .Net 플랫폼을 다양한 프로그래밍 언어가 조화를 이루어 상호 작용 할 수 있도록 개발하였다. 따라서 어떠한 언어로 개발을 하더라도 java의 바이트 코드처럼 시스템에 독립적인 프로그램이 되도록 MSIL(Microsoft Intermediate Language) 이라는 중간언어의 개념을 도입하였다. MSIL은 식별자와 알고리즘 등을 메타데이터 형태로 담고 있으며 이 정보들은 궁극적으로 인간이 이해할 수 있다는 문제점을 갖는다. 따라서 지적 재산권의 보호를 위해 Microsoft는 Dotfuscator[5]라는 도구를 개발하였다. Dotfuscator는 MSIL의 어셈블리 코드를 대상으로 Obfuscation을 수행하게 된다.

본 연구에서 구현하고자 하는 Obfuscation 도구는 Visual C++(MFC 포함) 소스 코드를 대상으로 하며 컴파일된 기계어의 복잡도 증가와 함께 소스 코드의 가독성을 낮추는데 그 목적을 둔다. C 또는 C++ 소스 코드를 대상으로 하는 Obfuscation 도구[6][8][9]에 대한 연구가 일부 이루어지고 있지만 국내 연구는 활발하지 않다.

3. 비주얼 C++ 소스 코드 obfuscator 구조

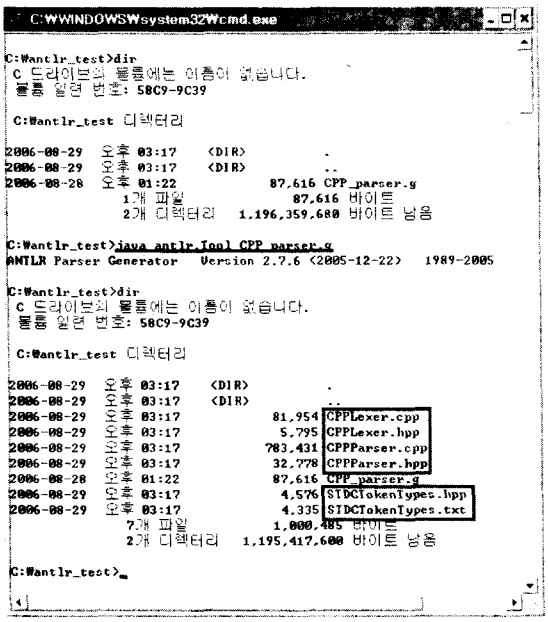
본 연구에서는 C++로 작성된 프로그램을 인식하기 위해 ANTLR (ANother Tool for Language Recognition)이란 파서 생성기를 사용한다. 파서는 프로그램을 구조적으로 분석하기 위한 정보를 추출하게 된다. 특히 소스코드의 여러 심벌들을 파일, 또는 메모리에 저장하고 사용자가 직접 필요한 심벌들을 추가할 수 있도록 설계하였으며, 최종에는 분석된 프로그램을 소스코드로 재생성하는 소스 대 소스 변환 기능을 수행하게 된다. Obfuscation 알고리즘은 파싱과정에서 생성되는 AST (Abstract Syntax Tree)를 통하여 구현되며, Obfuscator의 전체 구조가 (그림 1)나타나 있다.



(그림 1) C++ obfuscator의 전체 구성도

3.1 ANTLR

ANTLR은 ANTLR Meta Language라는 특화된 언어로 작성된 lex나 yacc와 유사한 문법 명세서를 통해 언어인식기(프로그램)를 생성해주는 도구이다[7]. 언어인식기는 C, C++, Java, C#등의 다양한 언어로 생성될 수 있으며 문법 명세서에 기술된 문법으로 작성된 문서들을 인식한다. 생성된 언어 인식기는 크게 Lexer와 Parser 등 두 부분으로 나눌 수 있으며 각각은 컴파일러의 어휘 분석과 구문분석, 의미분석을 담당하게 된다. (그림 2)는 "CPP_parser.g"라는 사용자 정의 문법을 통해 C++문법의 언어인식기를 생성하는 과정을 보여준다.



(그림 2) C++ 언어인식기 생성 과정

3.2 AST (Abstract Syntax Tree)

AST는 입력되는 토큰 스트림(stream)의 구조적인 표현을 위해 ANTLR에서 정의한 트리구조이며 컴파일러 이론에서 구문트리와 유사하다. 트리를 구성하는 노드는 이진트리 형태로 구성되며 어휘분석을 통해 나뉘는 토큰들의 이름, 타입, 행 번호 등의 정보를 저장하게 된다. 위에서 간략히 언급한 바와 같이 Obfuscation 알고리즘의 구현 및 적용은 AST를 통해 이루어진다. 알고리즘의 적용은 AST에 저장된 정보를 토대로 알고리즘 적용 조건에 맞는 부분을 검색하고 분석하여 노드 삽입, 수정 및 재구성을 통해 수행된다. 소스 재생성해내는 기능 또한 AST를 통해 구현되었다.

4. Obfuscation 알고리즘 설계 및 구현

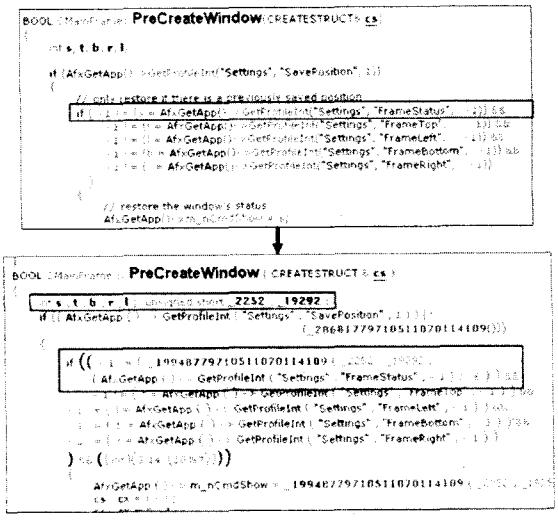
본 논문에서는 먼저 Layout Obfuscation을 적용하여 C++ 프로그램의 변수 이름 변경 및 주석 제거를 수행한다. 또한 Data Obfuscation 알고리즘 한가지와 Control flow Obfuscation 알고리즘 두 가지를 적용한다. 실험을 위해 FTP 소스 프로그램을 사용한다.

4.1 식별자 스크램블 및 주석 제거 (Layout obfuscation)

Layout obfuscation은 소스 코그의 포맷을 변경시키는 것으로 소스 코드 주석과 디버그 정보들을 없애고, 클래스, 멤버 변수, 로컬 변수 등의 이름을 변경한다. 본 논문에서는 항상 변수(식별자) 이름을 변환하는 알고리즘과 주석을 삭제하는 알고리즘을 기본적으로 적용한다.

4.2 Split Variable

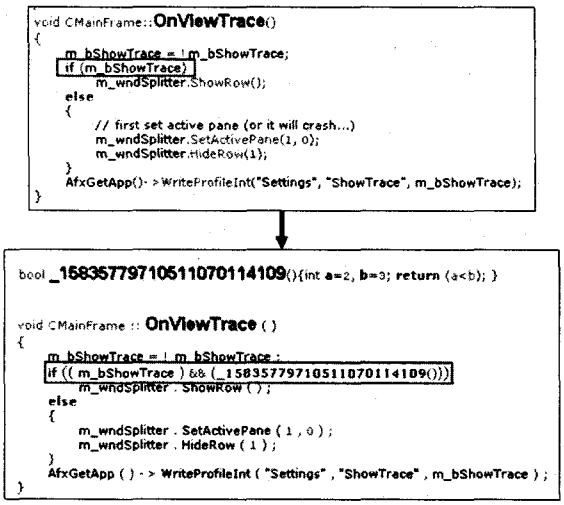
Split Variable 알고리즘은 기본타입의 변수들을 여러 개의 변수로 나누거나 변수의 사용자 정의 구조체 또는 값을 반환하는 함수로 대체함으로써 해석을 어렵게 만드는 기법이다. 본 논문에서는 32bit 정수형 변수를 16bit의 unsigned short형의 두 변수로 나누어 단항 연산자, 다항 연산자, 값의 할당부분을 함수로 대체하는 방식으로 구현하였다. (그림 3)은 Split Variable 알고리즘의 적용 후 코드 변화를 나타낸다.



(그림 3) Split Variable 알고리즘 적용

4.3 Extend Loop Condition

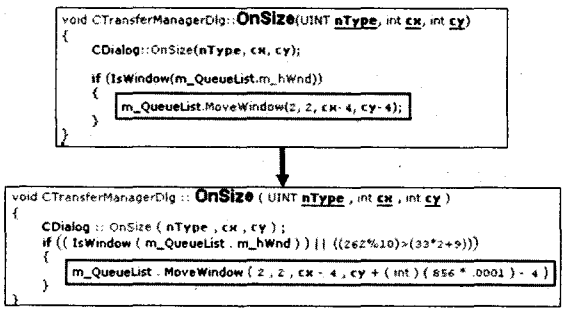
Extend Loop Condition 알고리즘은 반복문 또는 조건문에 조건식을 추가하여 복잡도를 증가시키는 기법이다. (그림 4)처럼 원 소스의 조건식에는 영향을 주지 않으면서 '&&' 연산자와 함께 무조건 참이 되는 조건식을 추가하거나, '||' 연산자와 함께 무조건 거짓이 되는 조건식을 추가하게 된다. 조건식은 함수나 수식으로 구성된다.



(그림 4) Extend Loop Condition 알고리즘 적용

4.4 Add Redundant Operand

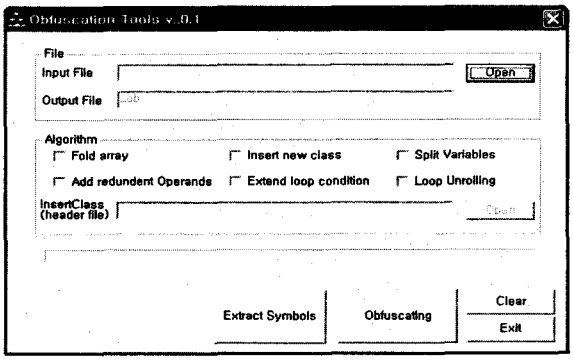
Add Redundant Operand 알고리즘은 간단한 연산 식에 operand를 추가하여 수식을 좀 더 복잡하게 만드는 기법이다. operand는 기존 수식에는 어떠한 영향도 미치지 않는다. 때문에 형변환으로 인한 데이터의 손실을 특별히 고려하여 적용한다(그림 5 참조).



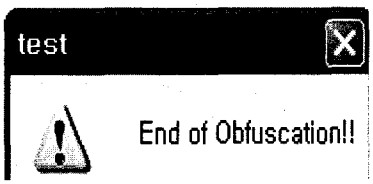
(그림 5) Add Redundant Operand 알고리즘 적용

4.5 구현 화면

본 시스템의 사양은 펜티엄 4 1.7GHz, 576MB RAM, 운영체제로는 Windows XP Professional SP2이며 개발 툴로는 Visual Studio 6.00이 사용되었다. 사용자의 편의성을 위해 사용자 인터페이스를 (그림 6)과 같이 구현하였다. 그림에서 우선 obfuscation을 적용하고자 하는 파일을 선택하면 obfuscation이 적용된 파일은 파일이름_ob로 같은 폴더에 생성이 된다. 그리고 "Extract Symbols" 단추를 눌러서 컴파일에 필요한 심벌 파일을 생성한다.



(그림 6) Obfuscation을 위한 사용자 인터페이스



(그림 7) obfuscation 완료 메시지

심벌 파일이 만들어지면 메시지가 뜨고 적용하고자 하는 obfuscation 알고리즘을 선택한 후 "Obfuscating" 단추를 클릭한다. 동시에 여러 개의 알고리즘을 선택하여 적용시킬 수도 있다. (그림 6)에는 본 논문에서 소개되지 않은 알고리즘도 나타나지 않았는데 현재 구현(테스트) 중에 있다. Obfuscation 적용이 끝나면 (그림 7)과 같이 메시지가 나타난다.

5. 성능평가

Obfuscator의 성능은 크게 potency, resilience, 비용 등의 항목으로 평가할 수 있다[2]. 본 논문에서는 MFC

로 작성된 API 통신 프로그램을 대상으로 평가하였다.

5.1 Potency

Potency는 변경된 코드가 원래의 것에 비해서 얼마나 이해하기 어려워졌는지를 나타낸다. 좋은 소프트웨어 디자인의 목적은 이러한 척도에 따라 복잡도를 최소화하는 것이지만 obfuscation의 목적은 복잡도를 최대화하는 것이다. potency를 측정하기 위해서는 크게 다섯 부분을 계산하여 potency(Obfuscation이 적용된 소스)/potency(원본소스)-1 의 값이 0보다 크면 potency가 좋아졌다고 평가한다. 평가용 측정 요소는 다음과 같다.

- ▷ p(1) 프로그램의 길이 (바이트 수)
- ▷ p(2) 매트릭스의 복잡도
- ▷ p(3) 중첩 복잡도
- ▷ p(4) 데이터 플로우 복잡도
- ▷ p(5) Fan-in/out 복잡도

<표 1>의 결과를 토대로 potency를 계산하면 0.7480으로 potency가 좋아졌음을 알 수 있다.

<표 1> potency 측정 결과

	원본소스	적용된 소스
p(1)	8133	14172
p(2)	51	89
p(3)	10	25
p(4)	50	81
p(5)	48	128
potency	8292	14495

5.2 Resilience

이는 변경된 코드가 자동화된 deobfuscation 공격에 대해서 얼마나 잘 견뎌낼 수 있는지를 측정하는 것이다. 사람이 코드를 읽기 혼란스럽다고 하면 potency가 좋은 것이고 deobfuscator들이 re-transformation을 할 수 없으면 resilience가 좋은 것이다. resilience 측정은 관련 연구 [2]의 측정표를 참고하였고 transformation할 때 one-way일 경우 30점, strong~weak에 대해서는 20점, trivial은 10점으로 계산하였다.

5.3 비용 (실행시간 측정)

원래의 코드와 비교하여 필요한 실행 시간을 측정한 것으로 결과는 (표 3)과 같다.

<표 2> Resilience 측정 결과

Target	Obfuscation	Value
Data	Split variable: Weak	20
Control	Extend loop condition: Strong~weak	20
	Add redundant: Strong~weak	20
Layout	Scramble identifier	30

<표 3> 실행시간 측정 결과 (단위 : ms)

원본소스	Obfuscation 적용 소스
0.139	0.18

6. 결론 및 향후과제

본 논문에서 구현한 Obfuscator는 전처리 파일을 통해 심볼 정보를 획득하기 때문에 Obfuscator 수행속도가 약간 느린 편이며 C++문법의 난해함으로 인해 다양한 obfuscation 알고리즘의 적용이 어려웠다. 따라서 심볼 정보획득 과정을 최적화시키고 좀 더 많은 obfuscation 알고리즘을 추가하는 것이 필요하다. 또한 향후에 obfuscation 알고리즘 적용 전후의 실행파일 또는 어셈블리 코드 수준에서 비교·분석할 계획이다.

참고문헌

[1] http://www.spc.or.kr

[2] C. Collberg, Clark Thombsorson, Louglas Low, " A Taxonomy of Obfuscating Transformations", technical Report 148, Dept of Cpmputer Science, Univ. of Auckland, July 1997

[3] Chenxi Weng, "A Security Architecture for Survivability Mechanisms", October 2000, University of Virginia

[4] Levent Ertaul, Suma Venkatesh, "JHide-A Tool Kit for Code Obfuscation", November 2004, LNCS

[5] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dotfuscator/dotf3e5x.asp

[6] http://www.semdesigns.com

[7] http://www.antlr.org

[8] Akers, R., Baxter, I. and Mehlich, M., "Reengineering C++ Components Via Automatic Program Transformation," ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, 2004.

[9] M. Gomathisankaran and A. Tyagi, "Architecture Support for 3D Obfuscation," IEEE Transactions on Computers, Vol. 55, No. 5, pp. 497-507, May 2006

[10] C.S. Collberg, et al., "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection", IEEE Trans. S/W Eng., August 2002.