비주얼 C++소스 코드를 위한 obfuscator 구현
(Implementation of an Obfuscator for Visual C++ Source Code)

장 혜 영 † 조 성 제 ††
(Hye-Young Chang) (Seong-Je Cho)

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 obfuscation 기법을 제안하고 구현하였다. 즉, 원본 비주얼 C++ 소스 프로그램을 기능은 동일하지만 이해하기에는 훨씬 힘든 또 다른 프로그램으로 변환시켜 주는 도구인 *코드 obfuscator*를 구현하였다. 비주얼 C++ 소스를 다루기 위해 ANTLR이라는 파서 생성기를 도입하여, '주석 제거', '식별자 스크램블', '변수 분할', '배열 중첩', '클래스 삽입', '루프 조건 확장', '부가 피연산자 삽입', '무의미 코드 삽입' 등의 변환 방식들을 구현하였다. 또한, 복잡도, 복원력, 비용 등의 측면에서 본 obfuscator의 성능과 유효성을 평가하였다. 원본 소스 코드와 비교하여 실험한 결과, 변환된 소스 코드가 실행시간 오버헤드를 일부 유발시키긴 하지만 프로그램 보호에는 효과적임을 알 수 있었다.

키워드 : Obfuscator, 비주얼 C++ 소스 프로그램, 역공학 공격

Abstract Automatic obfuscation is known to be the most viable method for preventing reverse engineering intentionally making code more difficult to understand for security purposes. In this paper, we study and implement an obfuscation method for protecting MS Visual C++ programs against attack on the intellectual property in software like reverse engineering attack. That is, the paper describes the implementation of a *code obfuscator*, a tool which converts a Visual C++ source program into an equivalent one that is much harder to understand. We have used ANTLR parser generator for handling Visual C++ sources, and implemented some obfuscating transformations such as *'Remove comments', 'Scramble identifiers', 'Split variables', 'Fold array', 'Insert class', 'Extend loop condition', 'Add redundant operands'*, and *'Insert dead code'*. We have also evaluated the performance and effectiveness of the obfuscator in terms of potency, resilience, and cost. When the obfuscated source code has been compared with the original source code, it has enough effectiveness for software protection though it incurs some run-time overheads.

Key words : Obfuscator, Visual C++ Source Program, Reverse Engineering Attack

1. 서 론

국가/군사 기밀 데이타, 기업 데이타, 병원 기록, 개인 사생활 정보 등의 대부분은 소프트웨어에 의해 처리된다. 소프트웨어 자체 또한 데이타의 형태로 관리되며 인터넷 기반의 컴퓨팅 환경에서 소프트웨어는 절도(theft) 및 오용(misuse) 공격에 취약하다. 많은 경우, 기밀 데이타를 훔치거나 변조하기 위해 공격자는 그 데이타를 보호하는 소프트웨어 프로그램을 공격한다. 소프트웨어에 대한 주요 공격으로는 불법복제(piracy), 변조(tampering), 역공학(reverse engineering) 등이 있다[1-3].

이들 공격으로부터 소프트웨어를 방어하는 기법을 연구할 필요가 있는데, 그 이유는 다음과 같다. 첫째, 소프트웨어 판매 수익은 많은 소프트웨어 벤더들의 생존과

· 본 연구는 2006년도 단국대학교 대학연구비의 지원으로 연구되었음

† 학생회원 : 단국대학교 정보컴퓨터과학
hychang@dankook.ac.kr
†† 정 회 원 : 단국대학교 정보컴퓨터과학부 교수
sjcho@dankook.ac.kr

논문접수 : 2007년 4월 23일
심사완료 : 2007년 11월 27일

정보과학회논문지: 소프트웨어 및 응용 제35권 제2호(2008.2)

직결되어 있다. 둘째, 최근 많은 벤더들이 공격자가 쉽게 조작할 수 있는 형태[1]로 소프트웨어를 배포하고 있다. 셋째, DRM(Digital Rights Management, 디지털 저작권 관리) 시스템과 같은 새로운 유형의 소프트웨어는 매우 주요한 비밀 정보를 포함한다. 예로 DRM 미디어 플레이어에 저장된 암호화 키를 추출할 수 있는 공격자는 비용을 지불하지 않고 불법으로 임의 미디어를 재생할 수 있다. 최근, "모바일 에이전트" 시스템처럼 클라이언트가 작성한 코드가 호스트에 다운로드 및 설치되어 사용되는 경우가 많아짐에 따라 프로그램 일부를 추출하거나 파괴하는 악의적인 호스트 공격(malicious host attack)을 방어하는 기법에 대한 관심도 증가하고 있다[1,2].

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[1-7]. 이들 기법 중 본 논문은 obfuscation에 초점을 맞춘다. 많은 소프트웨어 개발자들은 그들의 프로그램들이 역공학 대상이 되는 것을 걱정하고 있다. 실제로 한 응용으로부터 추출된 귀중한 코드 일부가 경쟁자의 코드에 통합된 사례가 있다[2]. 이러한 위협은 최근 계산 자원들의 증가, 그리고 Objdump, IDA Pro, SoftICE와 같은 역공학 도구(기술)의 발전으로 더 일반화 되었다[8,9]. 따라서 역공학으로 기계어로부터 소스 수준의 정보를 자동으로 추출하는 것이 가능하다.

여러 역공학 도구 및 기술로부터 소프트웨어를 보호하는 방법들이 많이 제안되었지만, 가장 강력하고 널리 연구되고 있는 방법 중의 하나가 obfuscation이다[7-9]. Obfuscation은 역공학을 방어하기 위해 공격자가 이해할 수 없도록 코드를 변환(transformation)하여 최대한 복잡하고 혼란스럽게 만드는 것이다. 이를 위해 언어 구조(language constructs), 데이타, 알고리즘, 제어흐름 등을 숨겨서, 원본 소프트웨어보다 변환된 소프트웨어(obfuscated software)를 이해(분석)하기 어렵게 만든다. 물론 프로그램의 본래 기능(semantics)은 그대로 유지되어야 한다.

본 논문에서는 소스 분석 공격 및 역 어셈블 공격으로부터 마이크로소프트사의 비주얼 C++ 소스 프로그램을 보호하기 위한 obfuscation 도구, 즉 obfuscator를 개발하였다. 구현에 적용된 변환 알고리즘은 '주석 제거', '식별자 이름 변경', '변수 분할', '배열 중첩', '클래스 삽입', '루프 조건 확장', '부가 피연산자 삽입', '무의미 코드 삽입' 등이다. 구현된 obfuscator의 효용성을 확인하기 위해 변환된 프로그램을 복잡도(해독 및 역공학 공격에 강한 정도), 복원력(자동화된 deobfuscation 공격에 강한 정도), 비용(오버헤드) 등의 면에서 성능을 평가하였다.

이 논문의 구성은 다음과 같다. 2장에서는 관련연구에 대해 기술하고, 3장에서 우리가 구현한 obfuscator 전체 구조에 대해 설명한다. 4장에서는 코드 변환에 사용된 알고리즘들에 대해 설명하고, 5장에서는 실험 결과와 성능 평가에 대해 기술한다. 6장에서 결론을 맺는다.

2. 관련 연구

불법복제, 변조, 역공학 등의 소프트웨어에 대한 "악의적인 호스트 공격"은 지적 재산권을 침해하여 재정적인 손실을 유발하는 공격으로, 주요 공격 대상은 핵심 알고리즘, 암호화 키, 프로그램 등록 검사 등이다[1-3]. 이러한 공격으로부터 소프트웨어를 보호하는 기술로는 암호화, 워터마킹, 변조 방지, 서버측 실행(server-side execution), Trusted native code, Self-checking, 바이너리 변경(Binary modification), obfuscation 등 매우 다양한 방법들이 있다[1-6,9,10]. 이들 방법 중 역공학 공격을 방어하는데 가장 효과적인 방법이 obfuscation이며, 본 논문에서는 obfuscation에 초점을 둔다.

최근에 모바일 코드나 ANDF(Architecture Neutral Distribution Format)처럼 아키텍처에 독립적인 포맷으로 배포되는 소프트웨어가 증가하고 역공학 기술들이 발전함에 따라, 역어셈블 공격이나 역컴파일 공격이 더 쉬워졌다[2,8-10]. 따라서 소프트웨어에 숨겨진 설계, 알고리즘, 암호화 키와 같은 데이타, DRM의 경우 소비자가 콘텐츠를 재생하거나 재배포할 수 있는 방식을 나타내는 *비즈니스 규칙(business rule)* 등의 비밀 정보가 불법 침해받을 수 있다. 소프트웨어를 읽고 이해하기 어렵게 만들어 역공학 공격을 방어하여 소프트웨어에 포함된 비밀 정보를 보호하는 것이 obfuscation이다[1-3, 6-11].

Obfuscation 알고리즘이 적용되어 변환된 프로그램이라도 그 프로그램의 기능 및 최종 결과는 원본 프로그램과 동일하다. 즉, 변환은 semantics-preserving하다. Obfuscation은 그 적용 레벨에 따라, C++ 소스 프로그램과 같은 고급 언어 수준에서 적용되는 '고급 obfuscation', Java 바이트코드나 마이크로소프트 중간언어(MSIL)에서 적용되는 '중급 obfuscation', 기계어 코드에서 적용되는 '저급 obfuscation'으로 분류할 수 있다[9]. 또한 obfuscation(변환)의 대상에 따라 어휘 변환(lexical transformation), 레이아웃 변환, 데이타 변환,

1) Java 바이트코드 또는 마이크로소프트의 MS IL(Microsoft Intermediate Language)이라는 중간언어 등은 배포 형태가 소스 코드와 거의 동일

제어 변환(control transformation), 예방차원의 변환(preventive transformation) 등으로 분류하는데 연구진마다 일부 차이가 있다[2,3,9,10]. 본 논문에서는 고급 obfuscation 레벨에서 레이아웃 변환, 데이타 변환, 제어 변환 등의 방식을 적용하며, 세부적인 내용은 3장과 4장에서 기술한다.

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 데이타 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 메타데이타 형태로 되어 있어 역공학 분석을 통해 소스 코드를 유추할 수 있다. 따라서 소프트웨어의 지적 재산권을 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. 여기서 어셈블리 코드는 변환을 위해 특별히 고안된 문법으로 제작된 것을 말하는데 SUIF(Stanford University Intermediate Format)[17]이 그 대표적인 예이다. SUIF 컴파일러 시스템은 C, C++, Fortran, Java 등의 다양한 언어로 작성된 프로그램을 SUIF 어셈블리 코드로 변환하여 이를 C 언어 소스코드로 변환해준다. Wang은 악의적인 의도를 가진 호스트들로부터 네트워크상에서 동작하는 소프트웨어를 보호하고자 핵심이 되는 모듈들을 보호하기 위한 변환 기법을 연구하면서 SUIF 시스템을 도입했다[18]. Wang은 단방향 변환(one-way translation)이라는 기법을 사용하여 변환된 코드를 다시 역변환 할 경우에 그 비용이 급격히 증가되게 하였다. 이 방법의 단점은 어셈블리 수준에서의 변환을 위해 추가적인 작업이 필요하다는 것이다.

소프트웨어 기반으로 소프트웨어 보호 방법들의 유효성을 측정하는 도구에 대한 연구의 결과물로 Sandmark가 있다[1]. Sandmark의 목적은 최소한의 오버헤드를 유발시키면서 공격에 가장 강한 소프트웨어 보호 알고리즘들을 실험적으로 결정하는 기술을 개발하는 것이다. 또한 Barak 등은 2001년에 [11]에서 이론적으로 obfuscation 개념을 달성하는 것이 불가능함을 주장하였으나, 최근에도 obfuscation에 대한 연구는 매우 활발하게 이루어지고 있다.

3. Obfuscator 구조

이 장에서는 MFC를 포함한 비주얼 C++ 소스 프로그램을 보호하기 위한 obfuscator의 구조에 대해 기술한다. C++ 소스 코드 obfuscator는 프로그램 C++ 소스 파일을 입력으로 받아, 이해하기 힘들고 역공학하기 어려워지지만 기능은 동일한 소스 파일로 변환하여 준다. 이러한 기술은 다음의 경우에 지적 재산권을 보호하는데 사용될 수 있다.

- 소스 코드가 공개 실행 목적으로 배포되어야 할 때,
- 상업적인 컴포넌트가 소비자에 의해 최종 제품(C나 PHP로 작성된 이식 가능한 응용, Verilog나 VHDL로 작성된 코드 라이브러리 또는 하드웨어 컴포넌트)으로 직접 통합될 수 있도록 소스 형태로 배포되어야 할 때,
- 소유권이 있는 코드(proprietary code)로부터 유도된 테스트 사례들(test cases)을 벤더들에 보낼 때,
- 오브젝트 코드들이 여전히 너무 많은 단서를 포함하고 있을 때

이런 이유로 인해, obfuscator를 구축하는 가장 신뢰성이 있는 방법은 어휘 및 구문(syntax, 문법) 규칙에 따라 소스 언어를 컴파일 단계와 유사한 자료구조들로 파싱하여 obfuscation을 수행한 후, 다시 소스 언어로 역파싱(un-parsing)하는 것이다.

본 논문에서는 Visual C++로 작성된 프로그램을 인식하기 위해 ANTLR (ANother Tool for Language Recognition)이란 파서 생성기를 도입하였다. 파서는 Visual C++ 소스 프로그램을 입력으로 받아 그 프로그램을 구조적으로 분석할 수 있도록 필요한 정보를 추출한다. 특히 소스 코드의 여러 심벌들을 파일이나 메모리에 저장하고 사용자가 직접 필요한 심벌들을 추가할 수 있도록 작성하였다. obfuscation 알고리즘은 파싱과정에서 생성되는 AST (Abstract Syntax Tree)를 통하여 구현되었다.

3.1 ANTRL

ANTLR은 ANTLR 메타 언어라는 특화된 언어로 작성된 문법 명세서를 통해 언어인식기 역할을 하는 프로그램을 생성해주는 도구이다[19]. 언어인식기는 다양한 언어로 생성될 수 있으며 문법 명세서에 기술된 문법으로 작성된 문서들을 인식한다. 본 논문에서는 C++ 소스 코드에 obfuscation 알고리즘을 적용하기 위해 어휘 분석기와 구문 분석기를 통해 나온 토큰(token)을 트리 형태로 만드는데 ANTLR을 사용한다.

언어인식기는 크게 어휘 분석기(Lexer)와 구문 분석기(Parser)로 나눌 수 있다. 어휘 분석기는 원시 프로그램을 읽어 들여 일련의 토큰을 생성하는 일을 수행한다. 구문 분석기는 어휘 분석기의 출력인 토큰을 받아 원시 프로그램에 대한 오류를 검사하고 올바른 문자에 대한 구문구조를 트리형태로 만든다[20]. 어휘 분석기와 구문 분석기는 본 논문에서는 도구 구현을 위해 기본이 되는 ANSI C++ 문법명세서를 수정하고 비주얼 C++ 언어에서만 사용되는 매크로 및 심벌들을 추가했다.

3.2 AST

AST(Abstract Syntax Tree)는 입력되는 토큰 스트림의 구조적인 표현을 위해 ANTLR에서 정의한 트리구조이고 컴파일러 이론에서의 구문트리와 유사한다. AST는 트리를 구성하는 노드는 이진트리와 유사하며 어휘분석을 통해 나뉘는 토큰들의 이름, 타입, 행 번호 등의 정보를 저장하게 된다. Obfuscation 알고리즘은 AST에 저장된 정보를 토대로 알고리즘 적용 조건에 맞는 부분을 검색하고 분석하여 노드 삽입, 수정 및 재구성을 통해 수행된다. 소스 코드를 재생성 하는 기능 또한 AST를 통해 구현되었다. 그림 1은 생성된 AST의 한 예이다.

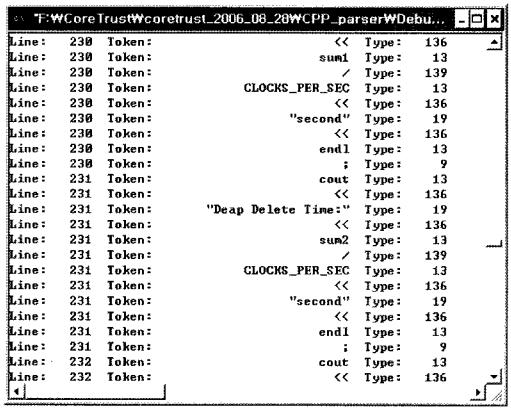

그림 1 AST 노드 출력 화면

3.3 Obfuscator 구조

Obfuscator는 크게 C++ 소스 코드로부터 심벌(symbol)정보를 획득하는 부분과 추출된 심벌정보를 통해 실제로 obfuscation 알고리즘을 적용하는 부분으로 구분할 수 있다. 심벌정보는 원시 프로그램에 나타난 모든 데이타(식별자)에 대한 정보들로 변수의 타입에 관한 정보나 배열의 크기에 관한 정보 등을 가지게 된다.

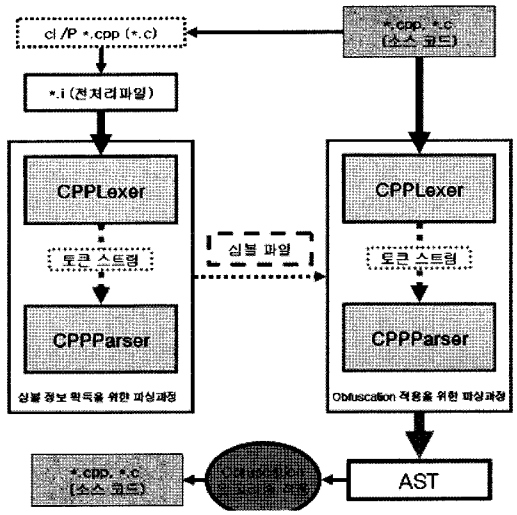

그림 2 Obfuscator 전체 구성도

본 논문에서 제안하는 obfuscator의 전체 구성도는 그림 2와 같다. 심벌정보 추출은 소스코드의 헤더 정보가 모두 추가된 전처리 파일로부터 이루어진다. 비주얼 스튜디오에서 컴파일 시 '/P' 옵션을 추가하면 확장자 'i'를 갖는 전처리 파일이 생성된다. 전처리 파일을 입력하여 심벌정보를 그림 2의 왼쪽처럼 획득을 한다. 그림 2의 오른쪽처럼 심벌파일을 가져와 실제 C++ 소스 코드를 파싱하여 AST를 생성한다. 생성된 AST 조작을 통해 obfuscation 알고리즘을 적용한 후 최종적으로 소스 코드를 생성한다.

4. Obfuscation 알고리즘 구현

Obfuscation은 그 대상에 따라 어휘 변환, 레이아웃 변환, 데이타 변환, 제어 변환 등으로 분류할 수 있다[2,3,9,10]. 어휘 변환은 프로그램 식별자의 이름을 변경하거나 스크램블(Scrambling of identifiers) 하는 것을 말하는데, 프로그램 문맥(context)으로부터 식별자의 의미를 추론할 수 있으므로 이 변환은 충분하지 않다. 레이아웃 변환은 소스-레벨 프로그램의 물리적인 구조를 대상으로 하며, 예로는 한 프로그램을 여러 프로시저들로 분할하는 방법, 포맷 변경(공백 제거), 주석 제거(Remove comments) 등이 있다[3]. 이 방법은 한번 수행되면 본래 코드로 복원할 수 없으며 소스 프로그램의 가독성을 떨어뜨리지만 기계어 프로그램의 복잡도를 증

가시키지는 못한다. 본 논문에서는 클래스 및 멤버 변수, 지역 변수 등의 식별자들의 이름을 변경하고, 소스 코드의 주석을 제거하였다.

데이타 변환은 소스-레벨 프로그램의 특정 데이타(변수)를 대상으로 하며 데이타의 저장, 인코딩, 통합(aggregation), 순서화(ordering) 등에 영향을 주어 obfuscation을 달성한다. 데이타 변환에 관련된 구체적인 방식은 매우 많은데, 이 논문에서는 '변수 분할(variable splitting)', '배열 중첩(array folding)', '클래스 삽입(class insertion)' 등의 세 가지 방식을 구현하였다. 이 데이타 변환은 실행시간 오버헤드가 크지 않다고 알려져 있다[3].

제어 변환은 개별 프로그램 함수들에서 제어 흐름을 이해하기 어렵게 만드는 방식으로 처리 순서의 변경(제어 흐름의 변환)이나 기능상에 변화를 주지 않는 제어 흐름을 추가하는 방법을 말한다. 제어 변환의 대표적 예인 '불투명한 조건자(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 자동 분석 공격에 대항할 수 있게 구현해야 한다. 이 논문에서는 '루프 조건 확장(extending loop condition)', '부가 피연산자 삽입(adding redundant operands)', '무의미 코드 삽입(inserting dead code) 등의 세 가지 제어 변환 방식을 구현하였다.

제어 변환의 경우 실행 시 추가적인 오버헤드가 불가피하고 소스-레벨 obfuscation의 경우 컴파일러의 최적화 기능들로 인해 제거될 수 있다. 즉, 비주얼 C++ 소스 프로그램의 경우, 불필요한 변수나 코드가 삽입되며 컴파일러가 최적화 과정에서 해당 변수나 코드를 제거할 수 있다. 이를 고려하여 본 논문에서는 컴파일러의 최적화 옵션을 적용할 경우에도 어셈블리 코드에서 제거되지 않고 남을 수 있는 변환 알고리즘만을 적용하였다.

본 논문에서 구현한 알고리즘들은 "A Taxonomy of Obufscationg Transformations" 논문 등[2,3,9,10]의 중급 obfuscation에서 적용한 알고리즘을 참조하여 구현했고 디프트리 프로그램, 거품정렬 프로그램, AES 암호화 프로그램 등을 대상으로 실험하였다.

4.1 변수 분할

변수 분할 알고리즘은 논리형(boolena)이나 크기가 제한적인 다른 기본타입 변수들을 여러 개의 변수로 나누거나 변수의 사용자 정의 구조체 또는 값을 반환하는 함수로 대체하는 방법이다. 변수를 분할함으로써 해석을 어렵게 하고 복잡도와 복원력이 증가된다. 그림 3은 변수 분할 중 논리형 변수를 분할하는 방법의 예를 보여준다. 그림의 (a)에서 p, q는 0과 1의 값을 가지고 함수 f는 두 변수의 값에 대해 참과 거짓을 반환한다. 이들의 상태를 0~3의 숫자로 표현하여 구성한 표이다. (b)의 VAL는 p, q의 XOR 값을 저장한 배열의 표이다. (c), (d)는 논리형 변수 A와 B의 AND, OR 연산의 결과를 (a)의 상태 표처럼 참과 거짓의 값을 숫자로 적용한 표이다. (a)~(d)를 알고리즘으로 적용한 예가 (e)에 나타나 있다[10].

본 논문에서는 32비트의 정수형 변수를 16비트의 unsigned short형의 두 변수로 나누어 단항 연산자, 다항 연산자, 값을 할당 부분을 함수로 대체하는 방식으로 구현했다. 그림 4는 변수 분할 알고리즘의 적용 전과 후의 코드 변화를 나타낸다.

4.2 배열 중첩

배열 중첩 알고리즘은 배열의 특성 및 용도를 공격자와 공격도구가 이해(분석)하기 어렵도록 만드는 것으로 이러한 기법들이 그림 5에 나타나 있다. 그림 5에서, 하나의 배열 A를 두 개의 배열 A1과 A2로 나눔(split)으로써 두 개의 배열이라는 혼란을 줄 수 있다. 그리고 B와 C처럼 여러 개의 배열을 하나로 합칠(merge) 수도 있다. D처럼 배열을 중첩(fold)시킴으로써 차원을 증가

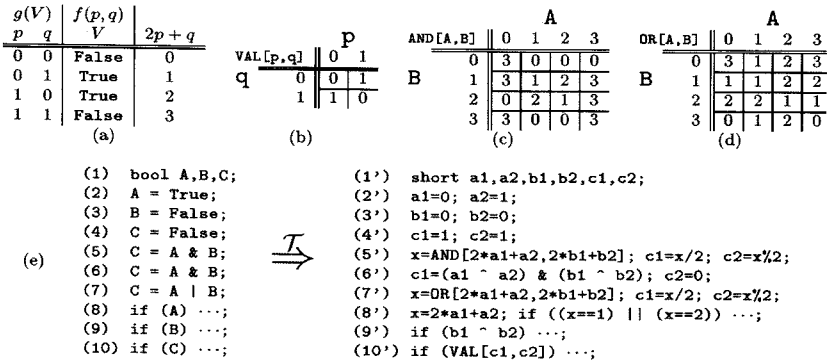

그림 3 논리형 변수의 분할

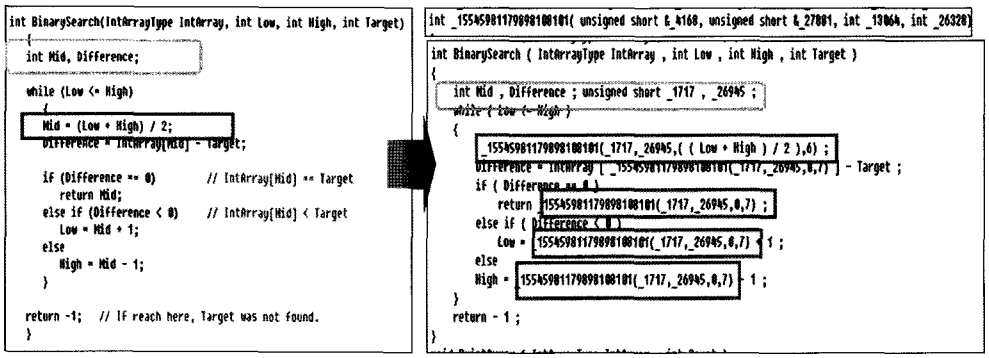

그림 4 변수 분할 알고리즘 적용

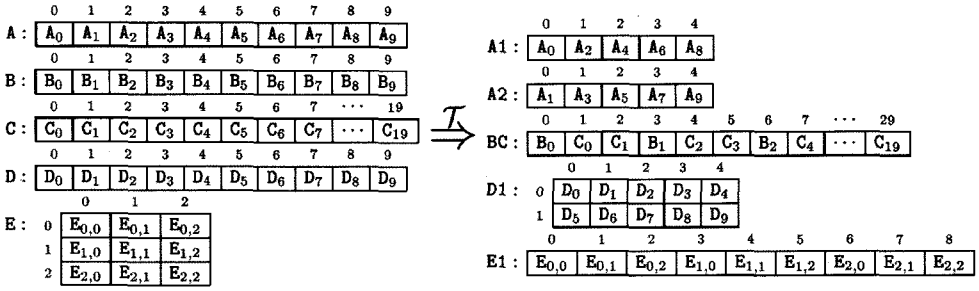

그림 5 배열 재구성 기법

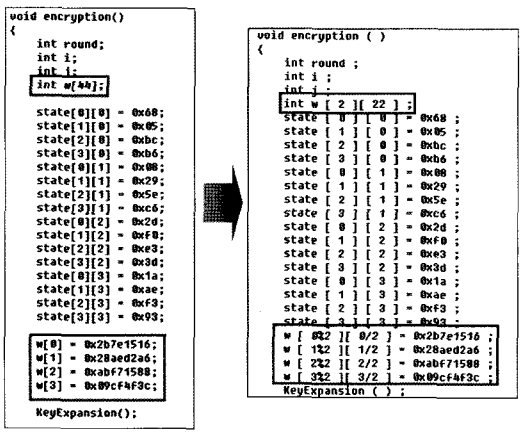

그림 6 배열 중첩 알고리즘 적용

시키거나 E처럼 배열의 차원을 감소(flatten)시킬 수도 있다.

본 논문에서는 1차원 배열을 2차원 배열로 차원을 증가시키는 배열 중첩 알고리즘을 구현하였고 그림 6에서 알고리즘의 적용을 보인다.

4.3 클래스 삽입

클래스의 복잡도는 상속 트리의 루트로부터 거리를 나타내는 깊이(depth)가 커질수록 증가한다. 클래스 obfuscation 알고리즘은 깊이의 증가나 클래스의 상속

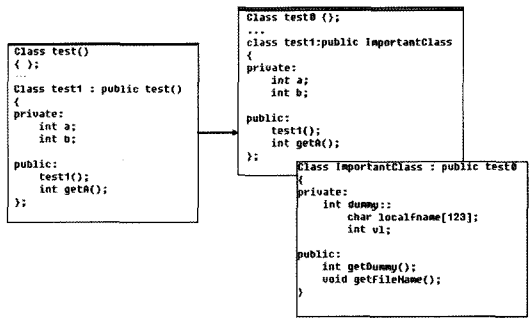

그림 7 클래스 삽입 알고리즘 적용

자의 수를 증가시킴으로써 복잡도, 복원력, 비용을 높이게 된다. 그중 클래스 삽입 기법은 상속 관계 수정(Modify Inheritance Relations) 기법들 중 하나로 객체지향 언어인 C++, 자바 등의 상속개념을 이용한 것이다. 하나의 클래스가 다른 클래스로부터 상속을 받게 되면 부모, 자식 클래스의 관계가 성립된다.

본 논문에서는 하나의 상속관계에 있는 클래스를 대상으로 중간에 더미 클래스를 추가하는 알고리즘을 구현하였다. 그림 7의 왼쪽 클래스 test1은 클래스 test0을 상속받는 클래스이다. 여기에 ImportantClass라는 더미 클래스를 하나 추가함으로써 상속 수준을 한 단계 증가시켰다.

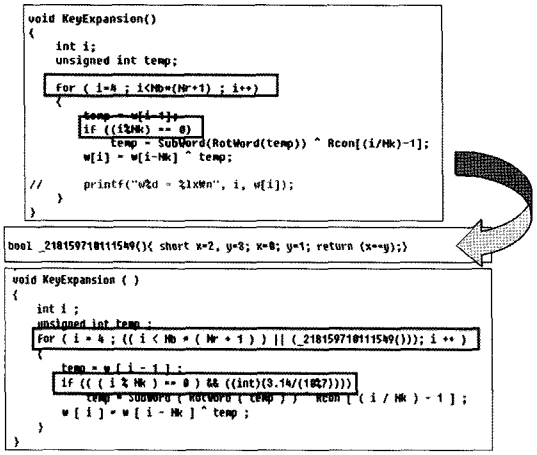

그림 8 루프 조건 확장 알고리즘 적용

4.4 루프 조건 확장

루프 조건 확장(Extend loop condition) 알고리즘은 조건문이나 반복문에 조건을 추가하여 종료 조건을 복잡하게 만드는 기법이다. 이때 어떠한 조건식을 추가하더라도 본래 의도한 종료시점에는 변화가 없어야 한다. 본 논문에서 '&&' 연산자에 무조건 참이 되는 조건식을 추가하고 '||' 연산자는 무조건 거짓이 되는 조건식을 추가하여 원 소스의 조건식에는 영향을 주지 않게 구현하였다.

4.5 부가 피연산자 삽입

부가 피연산자 삽입(Add redundant operand)알고리즘은 간단한 연산식에 피연산자를 추가하여 수식을 좀 더 복잡하게 만들어주는 기법이다. 삽입된 피연산자는 기존 수식의 결과에는 어떤 영향도 미치지 않는다. 알고리즘을 구현할 때 형변환으로 인한 데이타의 손실을 특별히 고려하였고 알고리즘을 적용한 결과는 그림 9와 같다. 함수의 인자 값으로 사용된 수식 "cy-4"에 형 변환된 추가적인 수식이 삽입된 것을 확인할 수 있다.

이 외에도, 주석 제거, 식별자 이름 변경, 무의미한 코드 삽입 방식을 구현하였으며, 총 8가지의 변환 방식을 적용하였다. 이에 반해 Semantic Designs사가 소스 프로그램의 obfuscation에 적용한 변환 방식은 주석 제거, 들여쓰기(indentation) 및 공백 제거, 상수 인코딩, 식별자 이름 변경 등이다. 따라서 본 논문에서 구현한 알고리즘이 변환 방식도 다양하고, 그 방식 또한 훨씬 복잡하여 obfuscation에 효과적이라고 할 수 있다.

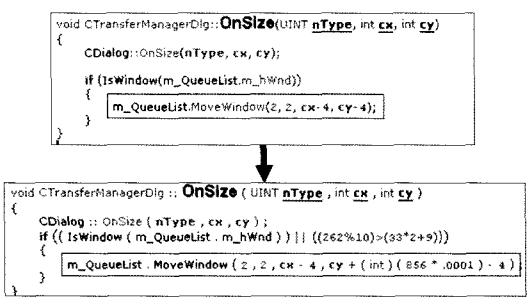

그림 9 부가 피연산자 삽입 알고리즘 적용

5. 실험 및 성능평가

제안한 obfuscation 기법은 펜티엄 4 1.7GHz, 576MB RAM, Windows XP Professional SP2의 시스템 상에 구현되었으며 개발 툴로 비주얼 스튜디오 6.0을 사용하였다. 사용자의 편의성을 위해 사용자 인터페이스를 그림 10과 같이 구현하였다. 먼저 'Open' 버튼을 사용하여 obfuscation을 적용하고자 하는 파일을 선택한다. 그 다음 "Extract Symbols" 버튼을 눌러서 컴파일에 필요한 심벌 파일을 생성한다. 심벌 파일을 생성하고, 적용하고자 하는 변환 알고리즘(들)을 선택한 후 "Obfuscating" 버튼을 클릭한다. 이때 여러 개의 알고리즘을 선택하여 동시에 적용시킬 수 있다. obfuscation 적용이 끝나면 확인 메시지가 출력된 후, obfuscation이 적용된 파일이 '*파일이름.cpp_ob*'으로 입력 파일과 같은 폴더에 생성이 된다.

Obfuscation의 결과를 평가하는 기준으로는 크게 복잡도, 복원력, 비용(오버헤드) 등이 있다[10,16]. 복잡도(Potency)는 원본 코드보다 변환된 코드가 얼마나 이해하기(분석하기) 어려운 가를 측정하는 정도로 클수록 좋다. 복잡도는 변환된 프로그램을 역공학하여 이해하는데 드는 시간이 원래 프로그램을 역공학하여 이해하는데 드는 시간 보다 더 길다는 의미인 obscurity (complexity, un-readability)를 포함한다. 복원력(Resilience)은 역변환하는 자동화된 도구를 구현하는 것의 어려운 정도, 또는 자동화된 역변환(deobfuscation) 도구를 실행하는데 드는 시간의 정도를 나타내는 것으로 클수록 좋다. 사람이 코드를 읽고 이해하기 힘들도록 복잡하게 만든다면 'potent'한 것이고 자동화된 deobfuscation이 어

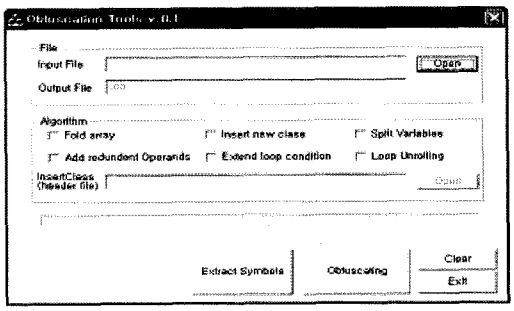

그림 10 Obfuscator 실행화면

렵다면 'resilient'한 것이다. 비용을 계산하기 위해서는 실행시간과 오브젝트 파일의 크기를 비교하였다.

실험대상으로 디프트리 프로그램, 거품정렬 프로그램, AES암호화 프로그램 등을 구현하여 성능을 평가하였다.

5.1 복잡도 기준 평가

복잡도를 측정하기 위해서는 표 1과 같이 크게 다섯 가지 항목을 계산하게 된다. 원 소스와 obfuscation된 소스를 각각 계산하여 다음의 식 (1)을 만족할 경우 복잡도가 증가한 것으로 본다.

$$\frac{potency(Obfuscated\,source\,code)}{potency(Original\,source\,code)} - 1 > 0 \qquad (1)$$

표 1의 다섯 가지 측정 요소를 각각 계산한 결과가 표 2, 3에 나타나 있다. 표 2는 실험에 사용한 세가지 알고리즘의 복잡도를 각각 원 소스 코드와 obufscation 한 소스 코드를 비교한 표이다. 그중 디프트리 알고리즘을 각각 알고리즘을 적용했을 때의 복잡도를 표 3에 나타내었다. 아래의 표 2와 같이 식 (1)을 계산하면 각각 0.7194, 0.7885, 0.5198로 복잡도가 향상되었음을 확인할 수 있다.

5.2 복원력 기준 평가

구현된 알고리즘들의 정량적인 복원력 값을 측정하기는 힘들지만 그림 12와 같이 trivial~one-way 등급까지 5단계로 나누어 측정할 수 있다.

본 논문에서는 구현된 변환 방식과 이에 대응되는 복원력 단계를 수치(value)로 환산하여 표 3에 나타내었다. 이때 각 알고리즘의 측정기준은 Collberg 등이 제안한 연구 결과[10]에서 소개된 측정 기준을 참조했다.

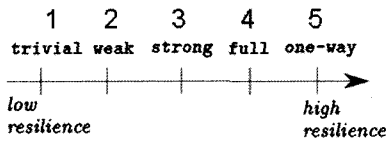

그림 11 복원력 측정 등급

표 1 복잡도 측정 요소

항목	설 명
p(1)	프로그램 길이 (데이터 수)
	프로그램의 피연산자와 연산자의 증가
p(2)	Cyclomatic Complexity
	함수의 순환복잡도, CFG(Control Flow Graph)의 노드, 에지 개수의 합
p(3)	Nesting Complexity
	함수에서 for, if, while등으로 인한 Scope의 레벨 증가
p(4)	데이타흐름 복잡도
	함수에서 참조되는 지역변수의 증가
p(5)	Fan-in/out Complexity
	함수에서 사용되는 파라미터 및 전역변수 개수의 증가

표 2 각 소스별 복잡도 측정 결과

항목	디프트리 프로그램		거품정렬 프로그램		AES 프로그램	
	원 코드	obfuscated 코드	원 코드	obfuscated 코드	원 코드	obfuscated 코드
p(1)	10782	18625	3001	3335	11660	17797
p(2)	93	97	13	19	8	14
p(3)	29	74	16	32	39	67
p(4)	58	143	58	133	193	282
p(5)	14	29	14	29	172	187
복잡도	0	0	0	0	0	0
복잡도측정	0.7194		0.7885		0.5198	

표 3 obfuscation 알고리즘의 각각의 복잡도 측정 결과

	원본	변수분할	배열차원 늘리기	데이타 Obfuscation	루프조건확장	부가피연산자 삽입	제어 Obfuscation	데이타 + 제어
p(1)	10782	11730	14611	14669	15202	14293	15397	18625
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	143
p(5)	14	23	14	23	16	14	16	29
복잡도	0	0	0	0	0	0	0	0

5.3 비용 기준 평가

비용(Cost)은 obfuscation을 적용 전후의 프로그램 실행 시간과 공간을 계산한다. 공간에 대한 비용은 표 4와 같이 실제 메모리에 적재되는 오브젝트(object) 파일과 소스파일의 크기를 측정하여 비교하였다. 표 4와 같이 파일의 크기는 전체적으로 증가했다. 특히 실제 프로그램의 기계어 코드를 담고 있는 오브젝트 파일의 크기는 약 1.3배 증가한 것을 확인할 수 있다.

실행시간은 디프트리 알고리즘은 500회 삽입, 삭제 연산을 10회 수행한 후 평균값을 계산하였고 거품정렬 알고리즘도 500회 삽입하여 10회 수행한 평균값을 계산했다. 암호화 알고리즘은 128bit AES 알고리즘을 10회 수행한 평균값을 실행시간으로 계산하였다.

5.4 어셈블리 코드 비교

'루프 조건 확장' 기법이 적용되기 전과 후의 소스 각각을 컴파일 시 최적화시켜 어셈블리 코드를 서로 비교해봤다. 비교된 코드는 디프트리 알고리즘 중 log2X의 값을 계산하기 위한 함수의 일부분으로 비교결과가 표 5에 나타나 있다. While 문에 조건식을 확장해주는 루프 조건 확장 알고리즘이 적용된 것을 볼 수 있고 어셈블리 코드에서 보는 것과 같이 최적화시켜 컴파일 하여도 obfuscated 알고리즘이 적용 되어있음을 확인할 수 있다. Obfuscated 알고리즘이 적용될 때 추가된 전역함수가 그림 13에 나타나 있다. 이는 본 논문에서 적용한 변환알고리즘이 기계어 코드에 영향을 미쳐 역공학 공격을 어렵게 만든다는 것을 보여준다.

```
...
int _1562968101971120
{ int i, j; i=920%2; j=i+1; return _1381868101971120(i, j); }
...
nt _1381868101971120(int a, int b)
{ int temp; temp = a; a = b; b = temp; return a-b; }
...
```

그림 13 표 5의 obfuscated 알고리즘 적용 시 추가된 전역 함수

6. 결론 및 향후연구

소프트웨어의 지적재산권을 침해하는 공격 중의 하나가 역공학 공격이며 이 역공학 공격을 방어하는 가장 좋은 방법이 obfuscation이다. 본 논문에서는 MFC 기반의 원본 비주얼 C++ 소스 프로그램을 대상으로 어휘, 자료구조, 제어 등을 변환하여 공격자가 분석하기 어려운, 그러나 기능은 동일한 또 다른 비주얼 C++ 소스 프로그램을 생성하여 주는 obfuscator (obfuscation 도구)를 구현하고 실험을 통해 성능을 평가하였다. 본 obfuscator에 적용된 변환 방식은 '주석 제거', '식별자 이름 변경', '배열 중첩', '클래스 삽입', '루프 조건 확장', '부가 피연산자 삽입', '무의미 코드 삽입' 등이다. 어셈블리 코드나 자바 바이트 코드, MSIL 등과 같이 정형화된 형태의 코드가 아닌 프로그래머가 직접 작성하는 소스 레벨에서 obfuscation을 적용함으로써 소스 코드 가독성 저하 및 기계어 코드에 대한 역공학 방어라는 두 가지 효용성을 동시에 추구하였다. 구현된 obfuscator를 복잡도, 복원력, 비용 측면에서 실험하여 성능을 평가하였다. 실험결과 obfuscation 알고리즘의 적용으로 일부 오버헤드가 유발되었으나 프로그램 보호를 위한 도구의 기능적 측면은 충분히 성취된 것으로 확인되었다.

향후, 소스 레벨의 데이타 obfuscation 및 제어 obfuscation 각각이 복잡도 및 복원력 등의 성능에 미치는 영향을 좀 더 체계적으로 분석하고, MFC 및 C++ 문법의 복잡성을 극복하여 좀 더 많은 변환 알고리즘을 적용할 계획이다. 임베디드 비주얼 C++ 소스 obfuscator를 개발하여 코드 최적화 기법과 통합하는 방안에 대해서도

표 4 구현된 알고리즘의 복원력

구분	대상	구현 알고리즘	복원력	값
레이아웃		주석 제거	One-way	5
제어흐름	연산	무의미 코드 삽입	Weak~Strong	2~3
		루프 조건 확장	Weak~Strong	2~3
		부가 피연산자 삽입	Weak~Strong	2~3
자료구조	스토리지 및 인코딩	변수 분할	Weak	2
	배열 및 클래스 상속	배열 중첩	Weak	2
		클래스 삽입	Trivial	1

표 5 적용 전후의 파일 크기 비교

	디프트리 알고리즘		거품정렬 알고리즘		AES 알고리즘	
파일 종류	원 프로그램	Obfuscated 프로그램	원 프로그램	Obfuscated 프로그램	원 프로그램	Obfuscated 프로그램
소스파일 크기(KB)	4.56	7.27	3	5	12	18
오브젝트파일 크기(KB)	6.74	9.62	94	98	13	19
실행시간(초)	1.783	2.980	0.0386	0.0417	0.3098	0.321

표 6 어셈블리 코드 비교

	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 $L1841** ...

연구할 필요가 있다.

참 고 문 헌

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] M. R. Stytz and J. A. Whitaker, "Software Protection: Security's Last Stand?," IEEE Security & Privacy, 1(1), pp. 95-98, Jan. 2003.
[8] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna, "Static Disassembly of Obfuscated Binaries," Proc. of the 13th USENIX Security Symposium, pp. 255-270, Aug. 2004.
[9] Colin W. Van Dyke, "Advances in Low-Level Software Protection," Ph. D. Thesis, Oregon State University, Jun. 2005.
[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] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," Proc. of the 10th ACM Conference on Computer and Communications Security (CCS), pp. 290-299, Oct. 2003.
[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.
[16] SUIF Compiler System, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/
[17] Chenxi Wang, "A Security Architecture for Survivability Mechanisms," Ph.D. Dissertation, University of Virginia, Oct. 2000.
[18] ANTLR, http://www.antlr.org
[19] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools," Addision-Wesley Publishing Company, 1998.

장 혜 영
2003년 단국대학교 이과대학 전산통계학과 학사. 2005년 단국대학교 컴퓨터과학 및 통계학과 석사. 2007년 단국대학교 정보컴퓨터과학과 박사과정수료. 관심분야는 컴퓨터 보안, 임베디드 시스템, DRM, Obfuscation

조 성 제
1989년 서울대학교 컴퓨터공학과 학사 1991년 서울대학교 컴퓨터공학과 공학석사. 1996년 서울대학교 컴퓨터공학과 공학박사. 1996년~1997년 서울대학교 컴퓨터신기술연구소 연구원. 2001년~2002년 미국 University of California, Irvine 객원연구원. 1997년 3월~현재 단국대학교 정보컴퓨터학부 부교수. 관심분야는 컴퓨터 보안, 시스템 소프트웨어, 실시간 시스템, 임베디드 시스템 등