모바일 소프트웨어를 위한 고급수준 난독처리 기법의 전력 소모량 분석

(Power Consumption Analysis of High-Level Obfuscation for Mobile Software)

이 진 영 † 장 혜 영 ††

(Jin-Young Lee) (Hye-Young Chang)

조 성 제 †††

(Seong-Je Cho)

요 약 난독처리(obfuscation)는 프로그램의 의미를 그대로 유지하면서, 프로그램 코드를 이해/분석하기 어렵게 만드는 기술로, 악의적인 역공학(reverse engineering) 공격으로부터 소프트웨어를 방어하는 가장 효과적인 기술 중의 하나이다. 하지만, 난독처리로 인해 원본 프로그램에 비해 난독처리된 프로그램의 코드 크기 및 실행 시간이 증가될 수 있다. 모바일 기기에서 코드크기 및 수행시간 증가는 전력소모 증가 등 자원낭비로 이어진다. 본 논문에서는 ARM 프로세서가 장착된 임베디드 보드 상에서 몇 가지 고급수준 난독처리 알고리즘을 구현하고, 각 난독처리 알고리즘의 유효성 및 전력 소모량을 분석하여, 프로그램의 특성에 따라 실행시간이나 전력소모 면에서 효율적인 난독처리 기법이 있음을 보였다.

키워드 : 난독처리, 역공학, 모바일, 전력 소모, 고급수준

· 본 연구는 2008년 정부(교육과학기술부)의 재원으로 한국학술진흥재단의 지원(KRF-2008-313-D00821)을 받아 수행되었음

· 이 논문은 2009 한국컴퓨터종합학술대회에서 '모바일 소프트웨어를 위한 고급수준 난독처리 기법의 구현 및 전력 소모량 분석'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 단국대학교 컴퓨터학과 컴퓨터과학
windofme@naver.com

†† 학생회원 : 단국대학교 정보컴퓨터과학과 컴퓨터과학
hychang@dankook.ac.kr

††† 정 회 원 : 단국대학교 공과대학 컴퓨터학부 교수
sjcho@dankook.ac.kr
(Corresponding author임)

논문접수 : 2009년 8월 14일
심사완료 : 2009년 10월 5일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제15권 제12호(2009.12)

Abstract Obfuscation is known as one of the most effective methods to protect software against malicious reverse engineering transforming the software into more complicated one with still preserving the original semantic. However, obfuscating a program can increase both code size of the program and execution time compared to the original program. In mobile devices, the increases of code size and execution time incur the waste of resources including the increase of power consumption. This paper has analyzed the effectiveness of some high-level obfuscation algorithms as well as their power consumption with implementing them under an embedded board equipped with ARM processor. The analysis results show that there is (are) an efficient obfuscation method(s) in terms of execution time or power consumption according to characteristics of a given program.

Key words : Obfuscation, Reverse engineering, Mobile, Power consumption, High-level

1. 서 론

소프트웨어 역공학(reverse engineering)은, 기계어 코드를 분해하여 원래의 소스코드를 드러내는 작업을 수반한다. 소스 코드가 없을 경우에 역공학은, 특정 연산들이 어떻게 실행되는 지를 알아내어 프로그램 성능 개선이나 버그 정정을 위해 수행되기도 한다. 그러나 역공학은 프로그램의 지적재산권(Intellectual Property, IP)과 영업비밀, 핵심 알고리즘들을 훔치는데 악용되어, 소프트웨어 개발 회사의 신용에 악영향을 미치며, 경쟁사에 큰 이점을 줄 수도 있다[1].

모바일 기술의 발전과 휴대 인터넷의 일반화로 모바일 서비스에 대한 다양한 요구가 증가하고 있으며[2], 이에 비례하여 모바일 소프트웨어의 적용, 불법복제 및 역공학 공격도 증가할 것이다. 외국 스마트폰의 경우 보안상징인 USIM 칩에 대한 보안이 강화되고 있지만 지속적으로 락에 대한 해킹이 이루어지고 있다. 또한, 국내 모바일 단말에 의무적으로 탑재되던 통합형 플랫폼이 개방됨에 따라 소프트웨어 산업에 큰 변화가 예고된다[3]. 이러한 모바일기기의 발전과 개방화 추세는 소프트웨어에 대한 더욱 다양한 위협 증가로 이어질 것이다.

이러한 위협에 대한 방어책의 일환으로 난독처리(obfuscation) 기술이 도입되었다[4-13]. 난독처리는 코드의 기능은 그대로 유지하면서 공격자가 이해할 수 없도록 코드를 변환(transformation)하여 최대한 복잡하고 혼란스럽게 만들어, 역공학 공격으로부터 소프트웨어를 보호하는 가장 강력한 방법 중의 하나이다. 하지만 난독처리로 소프트웨어 성능이 저하될 수 있으며[5], 이점은 자원이 한정된 모바일 기기에서 난독처리 기술의 적용

에 걸림돌이 되고 있다.

본 논문에서는 ARM 기반의 임베디드 보드에서 소스 분석 공격 및 역 어셈블 공격으로부터 C와 C++ 소스 코드를 보호하기 위한 난독화 기술을 구현하여, 난독처리기술을 적용하기 전과 후의 프로그램 코드를 복잡도(potency), 복원력(resilience), 비용, 소모전력 등의 측면에서 비교·분석하였다.[1] 이를 통해 모바일 환경에 적절한 난독처리 알고리즘이 무엇인지 조사하고, 전력 소모와 수행 시간 등의 면에서 더 효율적인 난독처리 방법이 있음을 보인다.

이 논문의 구성은 다음과 같다. 2장에서는 난독처리기술에 대한 관련 연구들을 기술한다. 3장과 4장에서는 본 연구진이 구현한 C/C++ 소스 코드 용 난독처리 시스템과, 시스템의 성능 평가에 대해 기술한다. 5장에서 향후 연구 계획을 언급하고 결론을 맺는다.

2. 관련 연구

난독처리는 그 기술을 C나 C++ 등의 소스 수준에 적용하는 고급수준 난독처리, Java나 C# 등의 바이트코드 수준에 적용하는 중급수준 난독처리, 어셈블리 코드 수준에 적용하는 저급수준 난독처리로 분류할 수 있다[4,6]. 이 중 바이트코드가 플랫폼 독립적이면서 소스코드의 정보를 그대로 담고 있어, 중급수준 난독처리 도구가 많이 주목 받고 있다[7-9]. 저급수준 난독처리 기술[8-10]은 플랫폼에 종속적이고 이식성이 낮다는 문제점을 가진다. 이를 보완하기 위해 시스템 독립적인 어셈블리 코드로 변환하여 처리된 소스코드를 획득하는 연구[11]도 수행되고 있으며, 이 경우 어셈블리 수준에서의 변환을 위해 추가적인 작업이 필요하다. 고급수준 난독처리는, 고급언어의 문법이 복잡하므로 다른 난독처리 기술에 비해 연구가 미진한 편이다.

난독처리기술은 난독처리(변환)되는 대상에 따라 다음 네 가지로 분류할 수 있다[4,8,12,13]. 첫째, 소스 프로그램의 물리적 구조를 변환하는 레이아웃(layout) 난독처리로, 한 프로그램을 여러 프로시저들로 분할, 포맷 변경, 주석 제거 등이 있다[12]. 이 방법은 한번 수행되면 본래 코드로 복원될 수 없어 프로그램의 가독성을 떨어뜨리지만 기계어 프로그램의 복잡도를 증가시키지는 못한다. 둘째, 데이터 난독처리로, 데이터의 메모리 저장 방법 혹은 저장된 데이터의 해석방법을 바꾸거나 데이터의 배열 순서를 변경한다. 셋째, 제어흐름(control) 난독처리로 조건자를 사용하여 조건문이나 반복문 등을 이해하기 어렵게 만든다. 넷째, 예방차원의 변환 방지(preventive transformation)로, 역변환도구(de-obfuscator)나 역컴파일러(decompiler)들의 알려진 문제점을 활용하여 역변환 기술을 적용하기 더 어렵도록 만드는 방식이다.

이러한 난독처리 방식들은 각각 장단점이 있어 혼용하여 사용할 수도 있지만, 현재까지의 다양한 기술들을 어떻게 조합하면 이상적인 난독처리가 되는지에 대한 체계적인 연구가 거의 이루어지지 않았다. 최근에 인텔 펜티엄4 프로세서 및 Windows XP Professional SP2 시스템에서 MS사의 Visual C++ 언어를 대상으로 난독처리를 하는 난독기(obfuscator)를 구현한 논문[5]이 발표되기도 하였지만, 모바일 환경을 고려하지 않았다.

본 논문에서는 ARM 프로세서 기반 임베디드 보드 상에서 C와 C++ 소스를 대상으로 난독처리 알고리즘들을 적용하고 성능 및 효율성을 분석한다.

3. 소스수준의 난독처리

3.1 난독처리기의 구조

C와 C++로 작성된 프로그램을 인식하기 위해 ANTLR (ANother Tool for Language Recognition)이란 파서 생성기를 사용하였다[14]. 난독처리 알고리즘은 파싱과정에서 생성되는 AST(Abstract Syntax Tree)를 통하여 구현되었다. AST는 입력되는 토큰 스트림의 구조적인 표현을 위해 ANTLR에서 정의한 트리구조이고 컴파일러에서의 구문트리와 유사한다.

난독처리기는 먼저 심벌정보를 추출하는데, 이는 소스코드의 헤더 정보가 모두 추가된 전처리 파일로부터 이루어진다. 컴파일 시 '/P' 옵션을 추가하면 확장자 'i'를 갖는 전처리 파일이 생성된다. 전처리 파일을 입력하여 심벌정보를 획득한다. 그다음 심벌파일을 가져와 실제 C++ 소스 코드를 파싱하여 AST를 생성한다. 생성된 AST 조작을 통해 난독처리 알고리즘을 적용한 후 최종적으로 난독처리 된 소스 코드를 생성한다[5].

3.2 실험에 사용된 난독처리 알고리즘

레이아웃변환(주석제거), 데이터 변환(변수분할, 배열 중첩), 제어흐름변환(루프 조건 확장, 부가 피연산자 삽입) 등을 구현하여 실험하였다.

3.2.1 데이터 난독처리의 예

변수분할 알고리즘은, 논리형(boolean)이나 크기가 제한적인 다른 기본타입 변수들을 여러 변수로 분할하거나 사용자 정의 구조체/값을 반환하는 함수로 대체하는 방법이다. 이로 인해 해석이 어렵게 되고 복잡도와 복원력이 증가된다. 본 논문에서는 그림 1과 같이 32비트의 정수형 변수를 16비트의 unsigned short형의 두 변수로 나누어 단항 연산자, 다항 연산자, 함수를 이용한 값의 할당 등으로 대체하게 구현하였다.

1) 난독처리 품질 평가 척도에 대한 설명은 4장을 참조 바람

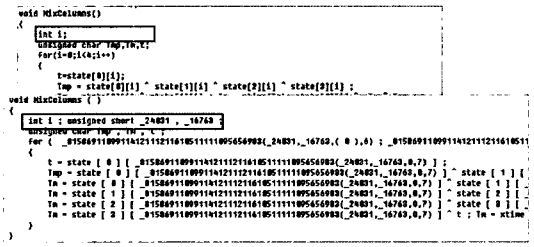

그림 1 데이터 난독처리 알고리즘 적용 전후

3.2.2 제어흐름 난독처리의 예

"루프 조건 확장"(Extend loop condition) 알고리즘은 조건문/반복문에 조건을 추가하여 종료 조건을 복잡하게 만드는 기법이다. 이때 어떠한 조건식을 추가하더라도 본래 의도한 기능대로 수행되어야 한다. 그림 2와 같이 '&&' 연산자에 무조건 참이 되는 조건식을 추가하고 '||' 연산자에 무조건 거짓이 되는 조건식을 추가하여 원 소스의 조건식에는 영향을 주지 않게 구현하였다.

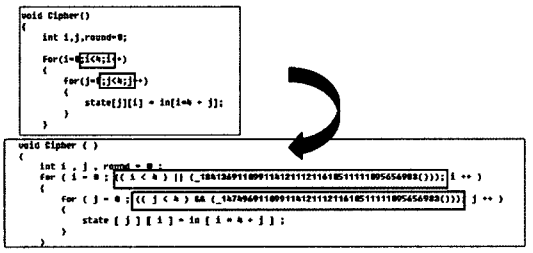

그림 2 제어흐름 난독처리의 예

부가 피연산자 삽입은 연산식에 피연산자를 추가하여 수식을 좀 더 복잡하게 만드는 기법이다. 삽입된 피연산자는 원래 수식의 결과에는 영향을 미치지 않는다.

3.3 실험 대상 소프트웨어

본 논문에서 난독처리 효과를 분석하기 위해 다음 두 가지 유형의 프로그램에 난독처리를 적용하여 실험하였다. 첫 번째는 지역변수와 전역변수 참조 및 변경이 빈번히 발생하는 AES(Advanced Encryption Standard) 대칭키 암호화 알고리즘[15]을, 두 번째는 메모리 참조 후 조건 분기가 많은 트리 탐색 알고리즘을 선택하였다.

4. 실험 및 성능평가

S3C6400(ARM1176JZF-S 533 MHz/667MHz) CPU, DRAM 128MB, NAND 128MB 등이 장착된 임베디드 보드 상에, Embedded Linux(커널 2.6.21), g++ 컴파일러를 구축하여 실험하였다. 난독처리 효과를 평가하는 기준으로 복잡도(potency), 복원력(resilience), 비용(오버헤드)[6,12], 그리고 전력소모 등을 채택하였다.

4.1 복잡도 면에서 평가

복잡도는 원본 코드보다 변환된 코드가 얼마나 이해하기 어려운 가를 측정하는 정도로 클수록 좋다. 이는 변환된 프로그램을 역공학하여 이해하는데 드는 시간이 원래 프로그램을 역공학하여 이해하는데 드는 시간 보다 더 길다는 의미인 obscurity(complexity, un-readability)를 포함한다. 복잡도를 측정하기 위해서는 표 1과 같은 측정요소들을 계산하여 각 계산 값을 합하여 복잡도를 구한다. 원 소스와 난독처리된 소스를 대상으로 측정요소들을 계산하여 다음의 식 (1)을 만족할 경우 복잡도가 증가한 것으로 본다[5].

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

표 1의 측정요소를 각각 계산한 결과가 표 4에 나타나 있다. 즉, AES 프로그램에 대해 원 소스 코드를 기준으로 난독처리된 소스 코드의 데이터 난독화, 제어흐름 난독화, 데이터+제어흐름 난독화의 복잡도를 비교하였다. 식 (1)을 통해 계산한 결과, 각각 0.7933, 0.3549, 1.0205로 복잡도 비율이 향상되었음을 확인할 수 있었다.

표 1 복잡도 측정 요소[5,6]

항목	설 명
p(1)	프로그램 길이 (바이트 수)
p(2)	함수의 순환복잡도, CFG(Control Flow Graph)의 노드, 에지 개수의 합
p(3)	함수에서 for, if, while등으로 인한 Scope의 레벨 증가
p(4)	함수에서 참조되는 지역변수의 증가
p(5)	함수에서 사용되는 파라미터 및 전역변수 개수의 증가

4.2 복원력 면에서 평가

난독처리 방식들의 복원력 값을 정량적으로 측정하는 것이 쉽지 않지만 그림 3과 같이 trivial~one-way 등급까지 5단계로 나누어 측정할 수 있다[6]. 구현된 난독처리 방식과 이에 대응되는 복원력 단계를 수치로 환산하여 표 2에 나타내었으며, 구현한 난독처리 기법들의 복원력 역시 증가하였다.

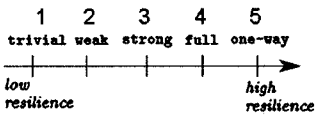

그림 3 복원력 측정 등급

4.3 비용 면에서 평가

난독처리 전과 후의 디프트리 프로그램으로 20회 삽입연산을 처리하고 탐색연산을 10,000회 수행하여 총 실행시간을 계산하였다. 또한 AES 프로그램에서 128 bit

표 2 구현된 알고리즘의 복원력[6]

구분	대상	구현 알고리즘	복원력	값
레이아웃		주석 제거	One-way	5
제어흐름	연산	루프 조건 확장	Weak~Strong	2~3
		부가피연산자 삽입	Weak~Strong	2~3
자료구조	스토리지 및 인코딩	변수 분할	Weak	2
	배열 및 클래스상속	배열 중첩	Weak	2

표 3 난독처리 전과 후의 파일크기 및 실행시간

	AES 암호화 프로그램				트리검색 프로그램			
	원본	데이터	제어	데이터+제어	원본	데이터	제어	데이터+제어
소스 파일 (byte)	8466	15208	11428	17088	778	2804	859	2890
오브젝트 파일 (byte)	13936	15779	16508	19486	6483	8543	6568	8624
실행시간 (s)	49.09	59.59	49.88	61.19	1.2	1.9	3.5	3.6

의 암호화키를 사용하여 128bit 평문 데이터를 20,000회 반복해서 암호화 연산을 수행한 시간을 계산하였다. 표 3에 측정된 실행시간이 second 단위로 나타나 있다.

결과를 보면, 트리검색의 경우 제어흐름 난독화로 인한 오버헤드가 더 많이 증가하였는데, 이는 최적화된 제어흐름 중심의 구성이 난독화로 인해 그 제어흐름이 흐트러졌기 때문이라고 추정된다. AES의 경우에는 데이터 난독화로 인한 오버헤드가 더 많이 증가하였는데, 이는 데이터 처리 중심의 코드 구성이 난독화로 인해 자료구조의 효율적인 구성이 깨졌기 때문이라고 추정된다.

4.4 소비전력 평가

소비전력은 멀티 메타를 사용하여 $W=V\cdot I\cdot t$ 로 측정하였다. 이때 사용한 멀티 메타는 전력측정 오차범위가 ∓2%이다. 소비전력 측정에 사용된 전압은 5V이며 전력 소비시간은 프로그램구동 시간과 비례한다.

표 4를 보면 AES는, 제어흐름 난독처리에 비해 데이터 난독처리된 프로그램이 더 많은 전력을 소비하였다. 데이터 난독화로 전력소모 및 연산시간이 증가된 원인을 두 가지로 추정할 수 있다. 첫 번째는, AES의 경우 정수 연산을 많이 수행하며 정수 연산에 최적화 되어 있는데, 변수 분할이라는 데이터 난독화를 적용하면 정수 연산이 short 연산으로 변환되어, 데이터 처리를 위한 실행횟수가 증가하고, 결국 실행시간이 증가된다. 두 번째는, ARM 코어 기반의 CPU는 동일한 계산 결과를 얻기 위해 short 연산보다는 정수 연산을 이용하는 것이 더 효율적인데, 난독처리 과정을 거치면서 short 연산이 더 많아져서 실행시간이 증가한 것으로 추정된다.

표 4 AES 프로그램에서 난독처리 방법별 복잡도, 실행시간, 소비전력

	원본	데이터	제어	데이터 + 제어
p(1)	8466	15208 (1.8)	11428 (1.35)	17088 (2.02)
p(2)	29	29(1)	49(1.68)	51(1.78)
p(3)	12	20(1.67)	25(2.08)	30(2.5)
p(4)	20	34(1.7)	46(2.3)	50(2.5)
p(5)	10	18(1.8)	19(1.9)	30(3.0)
측정요소 총합	8,537	15,309	11,567	17,249
복잡도	0	0.7933	0.3549	1.0205
시간(s)	49.090	59.590	49.880	61.189
전력(A)	1.69	1.70	1.71	1.69
전압(V)	5	5	5	5
전력량(Wh)	0.1152	0.1406	0.1185	0.1436
전력사용 증가율(%)	0	22.05 %	2.86 %	24.65 %

( ) 안의 숫자는 원본대비 변화율, 전력 오차범위 +-2%

4.5 종합 평가

전력소모에 큰 영향을 끼치는 것은 프로그램의 실행시간으로, 프로그램 실행시간을 단축해야 한다. 난독처리로 인해 프로그램의 실행시간과 전력소모 증가가 역공학 방지를 위해 불가피하지만, 모바일 컴퓨팅에서 추가적인 실행시간 및 전력소모를 줄일 수 있는 효율적인 난독처리 알고리즘을 도입하는 것이 필요하다.

표 4에서 난독화 방식에 따른 전력사용 증가율이 다른 점을 확인할 수 있다. AES의 경우 데이터 난독처리로 인한 전력 증가율이 22.05%에 달하나 제어흐름 난독처리로 인한 증가율은 2.86%에 불과하다. 트리검색에서는 제어흐름 난독처리로 인한 전력소모량이 더 크게 증가하였다. 이 결과를 보면, 프로그램의 어떤 특성에 따라 그에 적합한 난독처리 방식이 존재함을 알 수 있다.

AES 프로그램은 데이터 산술연산 처리의 양이 많은데, 데이터 난독화가 산술연산 처리에 필요한 데이터 처리 연산을 증가시켜 실행시간도 증가하였다. 그림 4를 보면, 원본 코드를 기준으로 볼 때 데이터 난독처리 후의 제어흐름 그래프가 크게 달라져 있음을 확인할 수 있다. AES 특성상 제안한 제어흐름 난독처리는 수행흐름에 큰 변화를 주지 못한 것으로 분석된다.

트리검색 프로그램의 경우 제어흐름에 따라 동적으로 검색 횟수가 증가하기 때문에 데이터 난독처리에 따른 성능저하보다 제어흐름 난독처리에 따른 성능저하가 더 크다. 따라서 프로그램 특성에 따라 난독처리 방식이 선택적으로 사용된다면, 난독처리로 인한 실행시간 및 전력소모의 증가를 최소화할 수 있을 것이다.

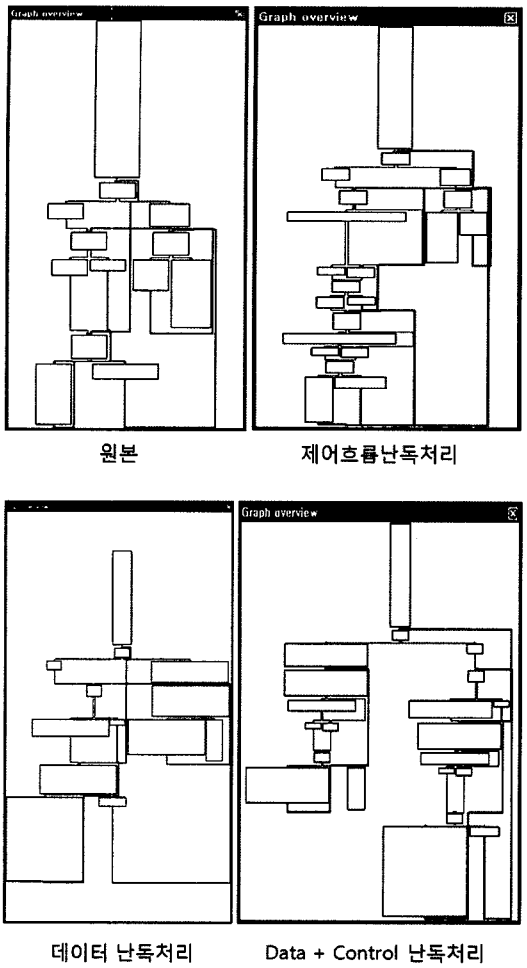

그림 4 AES 프로그램 난독화에 따른 제어그래프

5. 결론 및 향후연구

난독처리는 역공학 공격으로부터 임베디드 소프트웨어를 보호하는데 매우 효과적이다. 본 논문에서는 C/C++ 소스의 자료구조, 제어흐름, 레이아웃 등을 변형시키는 난독처리기술을 ARM 기반 임베디드 보드에서 구현한 후 실험하였다. 실험결과, 변수분할이나 루프조건확장 등의 난독처리 기술은 특정 프로그램에서 더욱 큰 실행 오버헤드와 전력소모를 유발하였다. 또한 임베디드 환경에서 난독처리 알고리즘은 복잡도 산정 요소 중 몇 가지 파라미터(자료구조 처리 연산 등)에 민감하게 반응한다. 이러한 난독처리 알고리즘의 특성을 적절히 활용한다면, 전력소모 등 자원 낭비를 줄이면서 역공학 공격으로부터 안전한 보호 기법을 개발할 수 있을 것이다. 제안한 기법에 대한 정확한 검증 및 실험 결과를 얻기 위해서, 향후 새로운 알고리즘을 구현하고, 복잡도 계산·분석에 필요한 자동화 도구에 대해 더 체계적으로 연구할 계획이다.

참 고 문 헌

[1] Avi Barir, "DRM: Protection of IP asset for earnings of company and advantage of competition," *Electronic Engineering Times-Korea*, design coner, June 1-15, 2007 (in korean)

[2] SW DC statistics and SW statistics of production, KIPA, 2008. http://www.software.or.kr (in korean)

[3] SW Industry white paper, KIPA, 2008. http://www.software.or.kr (in korean)

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

[5] H. Chang, S. Cho, "Implementation of an Obfuscator for Visual C++ Source Code," *Journal of KIISE : Software and Applications*, vol.35 no.2, 2008 (in korean)

[6] 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

[7] .NET Obfuscator. http://www.preemptive.com/products/dotfuscator

[8] C. Linn, and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *ACM Conference on Computer and Communications Security*, pp.290-299, October 2003.

[9] G. Wroblewski, "A general method of program code obfuscation," Ph.D. Dissertation, Wroclaw University, June 2002.

[10] SUIF Compiler System. http://suif.stanford.edu/suif/suif2/doc-2.2.0-4.

[11] C. Wang, "A security architecture for survivability mechanisms," Ph.D. Dissertation, University of Virginia, October 2000.

[12] 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.

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

[14] T. J. Parr, R. W. Quong, ANTLR: A Predicated-LL(k)Parser Generator, *SOFTWARE-PRACTICE AND EXPERIENCE*, vol.25(7), 789-810, July 1995.

[15] NIST FIPS, Announcing the ADVANCED ENCRYPTION STANDARD (AES) 2001. Nov. 26.