2006 한국컴퓨터종합학술대회 논문집 Vol. 33, No. 1(C)

C++ 언어를 위한 Control Flow Obfuscator 구현 및 평가

노진욱[0*], 조병민[*], 오현수[*], 장혜영[*], 정민규[**], 이승원[**], 박용수[+], 우제학[++], 조성제[*]
[*]단국대학교, [**]서울대학교, [+]한양대학교, [++](주)코어트러스트
{neocastle, cbm0041, pyxis3, chenil}@dankook.ac.kr, april18.mg@gmail.com,
{leesw, yspark}@ssrnet.snu.ac.kr, jhwoo@coretrust.com, sjcho@dankook.ac.kr

An Implementation of Control Flow Obfuscator for C++ Language

JinUk Noh[0], ByoungMin Cho[*], Hyunsoo Oh[*], Hye-Young Chang[*], Mingyu Jung[**], Seungwon Lee[**], Yongsoo Park[+], Jehak Woo[++], Seongje Cho[*]
[*]Dankook University, [**]Seoul National University, [+]Hanyang University, [++]Coretrust, Inc.

요 약

많은 소프트웨어 개발자들은 자신들의 프로그램들이 역공학 공격의 대상이 되는 것을 우려하고 있다. 특히 프로그램 내에 핵심적인 알고리즘이 내재되어 있는 경우 역공학 공격을 대비하는 기법이 필수적이다. 또한, 유비쿼터스 컴퓨팅 시대가 발전할수록 프로그램의 규모가 대형화됨에 따라 공동 개발자들 간에 프로그램 소스가 공유될 기회가 많아졌고, 그 결과 프로그램 소스 수준의 보호 기법도 필요하게 되었다. 본 논문에서는 C++ 프로그램 보호를 위해 Control Flow Obfuscator 도구를 구현하여 실험하였으며, 실험 결과 크지 않은 오버헤드로 악의적인 공격으로부터 프로그램을 분석을 어렵게 할 수 있음을 보인다.

1. 서 론

DRM (Digital Rights Management)은 디지털 콘텐츠 제공자의 권리와 이익을 안전하게 보호하는 기술로서, 디지털 콘텐츠를 건전하게 유통 보급할 수 있는 체계를 제공한다. 그러나, 악의적인 공격자들은 디지털 콘텐츠 및 관련 응용들을 불법 이용하기 위해, 그 응용의 자료구조를 알아내고, 대상 시스템의 알고리즘을 분석하기 위해 많은 역공학 기술을 발달시켰고, 그에 따른 역공학 도구들이 개발되었다.
본 논문에서는 역공학을 이용한 프로그램을 분석하는 것이 어렵고 많은 시간과 돈을 필요로 하게 만들기 위한 프로그램인 Code Obfuscator의 구현에 대해 기술한다. 현재 Java를 이용한 Code Obfuscation 프로그램은 많이 개발 되었고, 상용 프로그램도 많다. 하지만 C++을 이용한 Obfuscation 상업용 프로그램은 거의 없기 때문에 본 논문에서는 C++ 파일의 Code obfuscation 방법을 제안하여 구현 및 성능평가를 하였다.

본 논문의 구성은 다음과 같다. 2장에서는 현재 Obfuscation 관련 연구에 대해 설명한다. 3장은 구현한 Control Flow Obfuscation에 대해 설명한다. 4장은 프로그램 구현에 대한 설명이며, 5장은 테스트와 성능 평가에 대해 서술 한다. 그리고 6장은 결론 및 향후 연구 과제에 대해 설명하고 본 논문을 마무리 한다.

2. 관련 연구

최근 보안에 대한 인식이 높아지고 있으며, 역공학을 막기 위한 연구가 활발하다. 그 중 프로그램의 기능은 똑같지만 역공학자가 분석하기 힘들게 하기 위해 알아보기 힘든 소스로 바꾸는 Obfuscation의 연구도 활발히 진행되고 있다. 현재 JAVA Code obfuscation에 관련된 연구는 많이 진행 되어 왔으며, 상업용 프로그램으로 대표적인 것으로는 DashO, JMangle, JObfuscator 등 다양하게 개발되었다. 하지만 C++ 파일을 대상으로 하는 Obfuscator 연구는 많지 않고 상업용으로 개발된 것은 없다. 그 이유는 java 코드는 플랫폼에 독립적인 바이트코드로 만들어져서 대부분의 정보들은 자연어보다 디컴파일이 쉬운 바이트코드로 남아있기 때문이다. 디컴파일이 쉽다는 것은 개발자 입장에서는 심각한 취약점이 될 수 있다. 이러한 취약점 때문에 자바 Obfuscator의 연구가 필요하였고 많은Obfuscator 도구들이 만들어 졌다. 하지만 C++을 이러한 자바 프로그램 보다 안전하기 때문에 과거에는 Obfuscator의 필요성이 크지 않았으니 오늘날에는 역공학의 발달로 인해 C++ 파일도 더 이상은 안전하지 않기 때문에 개발하게 되었다.[1][2]

Code obfuscation 방법은 크게 다음과 같이 3가지로 나눌 수 있다.

- Layout Obfuscation
- Data Obfuscation
- Control Flow Obfuscation

Layout Obfuscation은 대상이 되는 것이 소스코드의 형식이나 변수이름 주석 등을 변환하는 것이다. Data Obfuscation은 프로그램의 데이터 구조의 변환을 나타낸다. 그리고 Control Flow Obfuscation은 프로그램에서 전체적인 제어 흐름을 변환하는 것이다. [2] 본 논문은 Control Flow Obfuscation 알고리즘을 이용한 새로운 도구를 설계/구현한다.

3. Control Flow Obfuscation 알고리즘

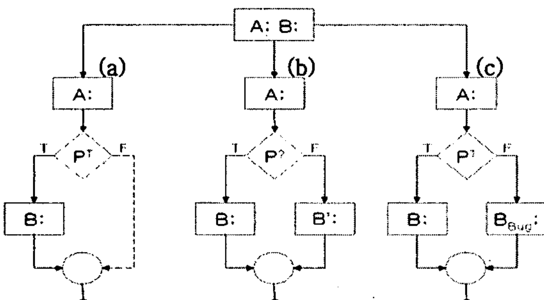
본 논문에서 구현한 프로그램의 제어 흐름(control Flow)을 대상으로 하는 Control Flow Obfuscation에는 다음과 같은 여러 가지 종류들이 있다.[3]

- Aggregation obfuscation: 문장들의 그룹을 변경하는 방법이다. 예를 들어 inlining을 들 수 있는데, 함수 콜을 함수 코드 자체로 변경하는 것이다.
- Ordering obfuscation: 문장들의 실행 순서를 변경하는 방법이다. 예를 들어 루프의 진행 순서를 반대로 하는 것을 들 수 있다.
- Computation obfuscation: 프로그램의 제어 흐름을 변경하는 방법이다. 예로 실행되지 않는 코드를 추가하거나, 불필요한 코드를 추가하는 것을 들 수 있다.

이러한 방법에는 논리적으로 관계 있는 연산들을 분리시키거나, 논리적으로 관계 없는 연산들을 합치거나, 또는 연산 순서를 바꾸거나, 관계없는 쓰레기 연산을 추가하는 방법 등이 있다[4]. 많은 방법 중 우리는 아래서 Insert Dead or Irrelevant Code, Extension Loop Condition, Add Redundant Operands에 대해 설명한다.

3.1 Insert Dead or Irrelevant Code

이 방법에 대해서 살펴보기 전에 predicate란 용어의 의미에 대해서 이해하고 있어야 한다. Predicate는 true 또는 false로 평가할 수 있는 함수 또는 조건을 말한다. Predicate는 프로그램 내의 조건 문에서 생성된다. PT는 항상 true인 predicate를 뜻하고, PF는 항상 false인 predicate를 뜻한다. P?는 조건문의 결과가 아직 정해지지 않은 상태를 나타낸다.

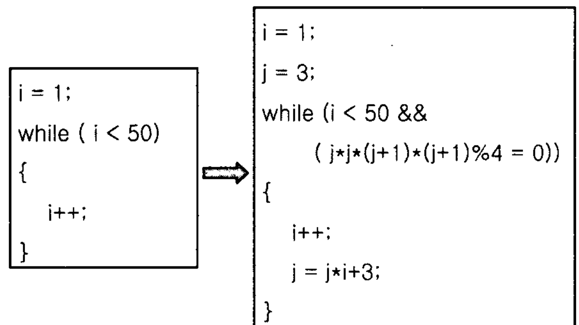


[그림 1] Insert Dead or Irrelevant Code

이 방법은 코드의 복잡도를 증가 시키기 위한 방법으로, 하나의 문장을 다음 [그림1] (a)와 같이 문장 A; B;를 둘로 나누고, 그 사이에 프로그램과 상관 없는 문장을 추가하여 분석을 어렵게 한다. 조건식 PT에서는 조건이 항상 참이 되어 결과는 같게 된다. [그림1] (b)는 B를 같은 기능을 하는 것을 B와 B` 두 개로 나누어 P?에서 참이 되든 거짓이 되든 결과는 같아진다. [그림1] (c)는 호출되지 않는 쓰레기코드를 추가하는 것이다. 조건식에서는 항상 참값을 갖는다. 이런 방법들은 실행 결과에는 영향을 주지 않지만 reverse engineer들은 쉽게 알지 못하게 된다.

3.2 Extension Loop Condition

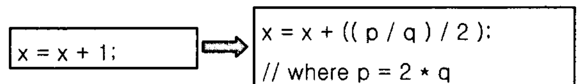
반복문의 조건절에 조건식을 추가하여 복잡하게 함으로써 복잡도를 증가시키는 방법이다. [그림2]에 나타나는 것처럼 전체적인 반복 횟수는 영향을 주지 않으면서 추가된 조건문은 항상 참이 되게 만들어 준다.



[그림 2] Extend Loop Condition

3.3 Add Redundant Operands

간단한 연산 식에 operand를 추가하여 좀 더 복잡하게 만들어 준다. 다음의 [그림3]에서 p는 사전에 q의 2배라고 정의하고 있기 때문에 결과적으로 x는 1을 더하는 것이지만 소스를 볼 때는 복잡하게 느껴진다.



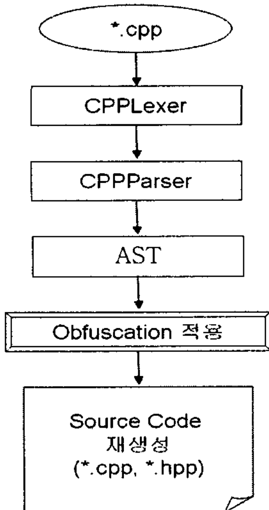
[그림 3] Add Redundant Operands

4. 구현 및 실험

4.1 구현환경

본 프로그램을 구현하기 위해 펜티엄4 1.7GHz, 512Mbyte 메모리, Windows XP Professional Service Pack 2 컴퓨터 시스템을 사용하였다. 프로그램은 MS Visual Studio 6.0을 사용해 구현하였다.

4.2 전체 구성도



[그림 4] 전체 구성도

프로그램의 전체적인 흐름을 살펴보면 먼저 C++로 만들어진 소스코드가 CPPLexer를 통해 어휘분석이 이루어진다. 그 다음, CPPParser에서는 CPPLexer를 통해 분석된 토큰을 받아들여 구문 분석 및 의미분석을 한다. 이후 심벌 테이블을 이용해서 AST(Abstract Syntax Tree)를 구성한다. 그 다음 제안 한 Obfuscation 알고리즘을 통해 새로운 *.cpp 파일이 만들어진다.

4.3 실험 결과

아래의 [그림5]는 원본 소스코드이고, [그림6]은 구현된 Obfuscator 프로그램을 통해 변환된 소스코드가 있다. [그림6]의 3, 8, 14라인에 있는 if문의 조건절이 좀 더 복잡해져 있는 것을 알 수 있다. 랜덤으로 추가된 연산이 or 연산이면 호출되는 함수의 반환값은 항상 0이 되고, and 연산이면 호출되는 함수의 반환값은 항상 1이 되게 만들어서 연산 결과에는 변함이 없다.

```
01:bool InitializeWinlo()
02:{
03:       if (IsNT)
04:       {
05:                  if
06:(!GetModuleFileName(GetModuleHandle(NULL),
07:                                   szExePath,
08:sizeof(szExePath)))
09:                        return FALSE;
10:                  pszSlash = strrchr(szExePath, 11:'₩₩');
12:                  int nn = nn + 1;
13:                  if (pszSlash)
14:                        pszSlash[1] = 0;
15:                  else
16:                        return FALSE;
17:       }
```

[그림 5] Original Source Code

```
01:bool InitializeWinlo ( )
02:{
03:       if (( IsNT ) || (_567()))       // Extend Loop Condition
04:       {
05:                  if    ((    !    GetModuleFileName
06:( GetModuleHandle ( NULL )
07:                                  ,    szExePath    ,
08:sizeof ( szExePath ) ) ) || (_358()))
09:                        return FALSE ;
10:                  pszSlash  =  strrchr  (  szExePath  ,
11:'₩₩' ) ;
12:                  int nn = nn * ( int ) ( _2487 + 80
13:* .0001 ) + 1 ;  // Add redundant Operand
14:                  if (( pszSlash ) || (_396()))
15:                        pszSlash [ 1 ] = 0 ;
16:                  else
17:                  return FALSE ;
18:       }
```

[그림 6] Obfuscated Source Code

[그림6]의 12라인을 보면 Add Redundant Operand 알고리즘이 적용된 것을 알 수 있다. 추가된 문장의 결과는 항상 1이기 때문에 곱하여도 결과에는 항상 같게 된다.

5. 성능 평가

제안된 프로그램의 성능을 측정하기 위해서 Process Explorer v10.06을 사용하였다. 예제로 MFC프로그램을 이용해 구현한 obfuscator 프로그램으로 변환 했을 때와 변환 전의 프로그램을 실행했을 때 시간, 메모리 사용량, 그리고 프로그램의 크기를 측정하였다. 시간측정은 클릭해서 프로그램이 실행될 때까지의 시간을 10^3회 검사해서 평균을 구했으며, 실험 결과 6% 정도의 시간이 더 필요로 했다. 메모리 사용은 1% 증가하는 것으로 나타나서 거의 차이가 없었다. 또한 프로그램의 크기는 10%정도 증가했다.

6. 결론 및 향후 연구 과제

본 논문에서는 불법적인 사용자가 공개된 소스코드를 역공학을 이용해 데이터구조, 알고리즘 등을 분석하여 이용할 수 있는 방지하는 기술중의 하나인 Control Flow Obfuscator를 C++ 파일을 대상으로 구현하였다. 향후 연구 과제로는 Code Obfuscation의 다른 방법인 Data Obfuscation 알고리즘을 결합하여 좀 더 분석이 어려운 Obfuscator를 만들 것이다.

7. 참고 문헌

[1] Douglas Low, "Java Control Flow Obfuscation", 1998
[2]http://palisade.plynt.com/issues/2005Oct/hiding-control-flows/
[3]http://www.macdevcenter.com/pub/a/mac/2005/04/08/code.html
[4] Christian Colleberg, Clark Thomborson, Douglas Low, "Manufacturing Cheap_ Resilient_ and Stealthy Opaque Constructs