Journal of Korea Multimedia Society Vol. 10, No. 5, May 2007(pp. 668-678)

가상기계 실행파일을 위한 보호 기법

박지우[†], 이창환[††], 오세만[†††]

요 약

유·무선 통신 기술의 발전은 다양한 정보에 쉽게 접근할 수 있도록 한다. 정보 접근의 용이성은 의도하지 않은 정보 유출이라는 문제점을 발생시킨다. 소프트웨어의 주요 알고리즘과 정보, 자원을 가지고 있는 실행파일은 보안에 매우 취약하다. 임베디드 시스템이나 가상기계의 실행파일은 알고리즘과 정보, 자원을 모두 가지고 있기 때문에, 정보 유출 문제는 더욱 심각하다. 본 논문에서는 정보 유출 문제를 해결하기 위해 암호화를 통한 실행파일 내용 보호 기법을 제안한다. 제안된 기법은 실험적으로 임베디드를 위한 가상기계인 EVM(Embedded Virtual Machine)에 적용하고 검증하였다. 또한, 실험을 통해 성능 오버헤드가 감내할 수 있는 수준이라는 것을 확인하였다.

Protecting Technique for the Executable File of Virtual Machines

Ji-Woo Park[†], Changhwan Yi[††], Se-Man Oh[†††]

ABSTRACT

The development of a wire and wireless communication technologies might permit easily accessing on various information. But, the easiness of accessing information has basically the problem of an unintended information outflow. An executable file which has key algorithms, data and resources for itself has very weak point in the security. Because the various information such as algorithms, data and resources is included in an executable file on embedded systems or virtual machines, the information outflow problem may appear more seriously. In this paper, we propose a technique which can be protecting the executable file contents for resolving the outflow problem through the encryption. Experimentally, we applied the proposed technique to EVM-the virtual machine for embedded system and verified it. Also, we tried a benchmark test for the proposed technique and obtained reasonable performance overhead.

Key words: Executable File Protection(실행파일 보호), Virtual Machine(가상기계), EVM:Embedded Virtual Machine(임베디드 가상기계)

1. 서 론

통신 기술의 발전에 따라 인터넷과 같은 네트워크 환경을 통해 저작권을 가진 소프트웨어의 유출, 초상권 침해 등과 같은 문제가 빈번히 발생하고 있다. 소프트웨어의 실행파일은 프로그램에 대한 많은 정보, 자원을 가지고 있기 때문에 보안에 매우 취약하다. 특히, 가상기계는 소프트웨어의 정보를 실행 이미지가 모두 가지고 있기 때문에 정보 유출 문제는 더욱 심각하다.

실행파일을 통한 정보 유출에는 다음과 같은 다양한 사례가 있다. 프로그램의 상수 정보를 통해 시스템 관리자의 아이디와 암호가 공개될 수 있다. 저작권 관리 시스템의 경우에는 실행파일을 통해 주요 정보가

※ 교신저자(Corresponding Author) : 오세만, 주소 : 서울시 중구 필동 3가 26번지 동국대학교(100-715), 전화 : 02)2260-3342, FAX : 02)2265-8742,
E-mail : smoh@dongguk.edu
접수일 : 2007년 2월 12일, 완료일 : 2007년 4월 23일

[†] 준회원, 동국대학교 컴퓨터공학과
(E-mail : jojaryong@dongguk.edu)
[††] 정회원, 링크젠
(E-mail : yich@dongguk.edu)
[†††] 정회원, 동국대학교 컴퓨터공학과

공개되어 저작권 관리 시스템이 무력화 된 적도 있다. 또한 실행파일의 명령어 분석과 같은 역공학(Reverse Engineering)을 통해 핵심 알고리즘이 알려질 수도 있다. 상업용 소프트웨어를 제작하는 기업에서 알고리즘의 노출은 막대한 피해를 발생시킨다.

위와 같은 정보 유출을 방지하기 위해 소프트웨어 개발자들은 많은 노력을 기울인다. 알고리즘 유출을 막기 위해 일반적으로 명령어의 난독화가 사용된다. 그러나 난독화는 알고리즘의 공개를 방지할 수 있지만, 상수 정보의 유출은 막지 못한다.

본 논문에서는 정보 유출 문제를 해결하기 위해 암호화를 통한 실행파일 내용 보호 기법을 제안한다. 제안된 기법은 실행파일 형식 확장과 가상기계 확장, 기법 적용기의 설계 및 구현으로 이루어져 있다. 기법은 임베디드 시스템을 위한 가상기계인 EVM (Embedded Virtual Machine)에 적용하였고, 작동 여부를 통해 실험적으로 검증하였다.

본 논문의 2장에서는 실험에 사용하는 가상기계인 EVM과 일반적인 실행파일 보호 기법에 대해 기술하였다. 3장에서는 가상기계를 위한 실행파일 보호 기법을 살펴보고, 기법이 적용된 시스템과 도구, 실행파일 형식을 설명한다. 4장 실험 및 결과에서는 기법이 가상기계인 EVM에 적용하여 구현되는 과정과 결과를 살펴본다. 또한 여러 예제 프로그램을 통해 기법의 오버헤드를 알아보고 결과를 분석하였다. 마지막 5장 결론에서는 본 논문에서 제안한 실행파일 보호 기법을 정리하고 향후 연구 방향을 언급한다.

2. 관련연구

2.1 EVM(Embedded Virtual Machine)

EVM은 스택 기반 가상기계로서 모바일 디바이스(Mobile Device), 셋톱박스(Set-top Box), 디지털 TV(Digital TV)등에 탑재되어 동적 응용 프로그램을 다운로드하여 실행하는 가상기계 솔루션이다. EVM은 그림 1과 같이 크게 변환기, 어셈블러, 가상기계의 세 부분으로 구성된다[1].

변환기 부분은 C#이나 자바 등의 고급 프로그래밍 언어로 작성된 프로그램을 어셈블리 형태로 번역한다. 어셈블러 부분은 가상기계의 어셈블리 형식인 SAF(Standard Assembly Format) 파일을 가상기계의 실행파일 형태인 EFF(Executable File Format)

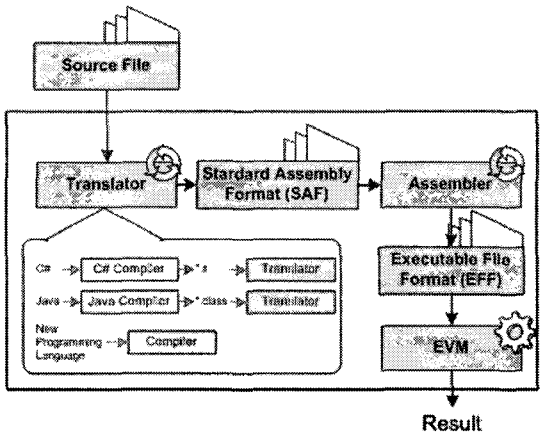

그림 1. EVM 시스템 구성도

파일로 변환하는 일을 한다[2-4]. 가상기계 부분은 실제 하드웨어에 탑재되어 실행파일을 실행하는 일을 한다[5].

2.1.1 SIL(Standard Intermediate Language)

EVM의 중간 언어인 SIL은 언어 독립성과 하드웨어 및 플랫폼 독립성을 갖도록 설계된 스택 기반의 명령어 집합이다. 명령어 집합은 그림 2와 같이 6개의 카테고리로 구성되어 있다.

다양한 프로그래밍 언어를 EVM에서 수용하기 위해 SIL은 바이트코드[6], .NET IL[7]과 같은 기존의 가상기계 코드들의 분석을 토대로 정의되었다. 또한 프로그램의 확장성을 위해 순차적 언어와 객체지

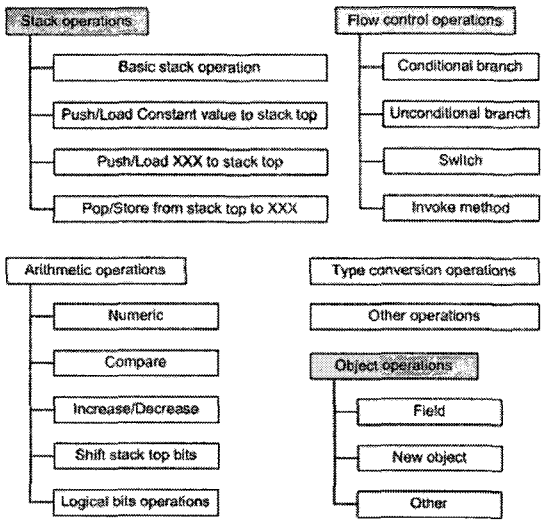

그림 2. SIL의 연산 카테고리

향 언어를 모두 수용할 수 있도록 설계되었다. 각 명령어는 EVM에서 실행되는 이진 코드와 1:1 대응된다[8].

2.1.2 SAF(Standard Assembly Format)

SAF는 임베디드 시스템을 위한 가상기계의 표준 어셈블리 형식으로 설계되었다. 그림 3은 SAF 문법을 EBNF(Extended Backus-Naur Form)로 표기한 것이다[9]. SAF는 클래스 선언 등 특정 작업의 수행을 의미하는 의사 코드와 가상기계에서 실행되는 실제 명령어에 대응되는 연산 코드로 이루어져 있다. 연산 코드로는 SIL을 사용한다. 따라서 연산 코드의 연상 기호는 특정 하드웨어나 소스 언어에 종속되지 않는 추상적인 형태를 지닌다.

EVM 시스템에서 고급 언어로 작성된 코드는 변환기를 통해서 의사 코드와 연산 코드로 구성된 EVM의 어셈블리 형식인 SAF로 변환되고 어셈블러에 의해 EFF 형태로 출력된다. 어셈블러의 출력인 EFF 파일은 그림 1과 같이 시스템의 운영 체제나 구조에 상관없이 EVM에 의해 실행된다[10,11].

2.1.3 EFF(Executable File Format)

EFF는 실행에 필요한 모든 정보를 가진 EVM의 실행파일 형식으로서 이진 형태의 바이트 스트림으로 되어 있다. 다양한 프로그래밍 언어를 수용하기 위해서 자바 클래스 파일[6], .NET PE 파일[7] 등 기존에 널리 사용되고 있는 가상기계 실행파일 형식들의 분석을 토대로 정의하였다.

EFF는 그림 4와 같이 크게 헤더, VM 코드, 메타데이터 세 부분으로 구성되어 있다[11].

```
<saf_program>       ::= { <class_dcl> }
<class_dcl>         ::= '.class' <modifiers> <class_name> [':' <super_class_name>]
                          '.bgn'
                             <class_body>
                          '.end'
<class_body>        ::= <member_dcl> { <member_dcl> }
<member_dcl>        ::= <field_dcl> | <method_dcl> | <class_dcl>

<field_dcl>         ::= '.field' <modifiers> <type_specifier> <filed_name>
<method_dcl>        ::= '.method' <modifiers> <type_specifier> <method_name> '(' [<formal_param>] ')'
                          '.bgn'
                             <method_body>
                          '.end'
<formal_param>      ::= <type_specifier> <formal_param_name> {',' <type_specifier> <formal_param_name>}
<method_body>       ::= { <dcl_statement> } { <instr_statement> }
<dcl_statement>     ::= <stack_dcl_st> | <local_var_dcl_st> | <exception_dcl_st> | <exception_proc_st>
<stack_dcl_st>      ::= '.maxstack' <number>
<local_var_dcl_st>  ::= '.locals' '(' [<local_var_list>] ')'
<local_var_list>    ::= <local_var> {',' <local_var> }
<local_var>         ::= <type_specifier> <local_var_name> { ',' <local_var_name> }
<exception_dcl_st>  ::= '.throws' <exception_type_name>
<exception_proc_st> ::=
'.catch' <exception_type_name> 'from' <start_lable_name> 'to' <end_label_name> 'using' <except_label_name>
<instr_statement>   ::= [ <label_name> ':'] ( <stack_op> | <arithmetic_op> | <flow_control_op>
                        | <object_op> | <typ_conversion_op> | <except_op> )
<modifiers>  ::= 'public' | 'private' | 'static' | 'terminal' | 'guarded' | 'interface' | 'concept'
<type_specifier>    ::= 'byte' | 'integer' | 'long' | 'float' | 'double' | 'short'
                        | 'char' | 'reference' | 'boolean' | 'void'
<*_name>            ::= '%ident'
<*_number>          ::= '%number'
<*_op>              ::= <opcode_name> [ <param_name> ]
```

그림 3. EBNF로 표기한 SAF 문법

```
EFF_File {
        U4 magic;               // Header
        U2 majorVersion;
        U2 minorVersion;
        U4 module;
        U4 language;
        U4 entryPoint;
        U4 VMCodesCount;  // Code
        VMCodeInfo VMCode[VMCodesCount];
        U4 metadataCount;  // Metadata
        MetadataInfo Metadata[metadataCount];
}
```

그림 4. EFF 파일 구조

헤더 부분은 해당 파일이 EFF 파일임을 나타내고 프로그램의 버전, 모듈 이름, EFF 파일을 생성한 언어, 시작점과 같은 정보를 가지고 있다. VM 코드 부분은 프로그램에서 실행되는 명령어 집합을 가지고 있다. 메타데이터 부분은 실행에 필요한 다양한 정보를 가지며, 표 1과 같이 저장되는 정보의 종류에 따라 총 15개의 테이블로 구성되어 있다.

표 1. 메타데이터의 종류

테이블 이름	태그 번호	설 명
MDTRefClass	0x0100	참조하는 클래스의 정보
MDTDefClass	0x0300	프로그래머가 정의한 클래스들의 정보
MDTNestedClass	0x0500	중첩 클래스들의 정보
MDTInterface	0x0600	인터페이스의 정보
MDTRefMember	0x0700	참조하는 클래스들의 멤버들
MDTField	0x1000	필드의 정보
MDTMethod	0x2000	메소드들의 정보
MDTDescriptor	0x3000	필드의 타입과 메소드의 시그네춰
MDTString	0x0400	시스템에서 사용하는 스트링 정보
MDTUserString	0x4500	사용자가 정의한 문자열
MDTInteger	0x5300	integer형 상수
MDTFloat	0x5400	float형 상수
MDTLong	0x5500	long형 상수
MDTDouble	0x5600	double형 상수
MDTException	0xe000	예외처리에 관한 정보

2.2 실행파일 보호

실행파일 보호는 내용이 임의로 변경되지 않고 외부에 공개되지 않도록 하는 것을 말한다. 실행파일 보호는 기밀성, 무결성, 사용자 인증, 부인 방지와 같은 보안의 4 가지 요구 사항에서 기밀성과 무결성 유지를 통해 이루어질 수 있다.

보안의 4 가지 요구 사항을 간략히 살펴보면 다음과 같다. 기밀성은 정보 이용이 허가되지 않은 사용자는 암호화된 데이터의 원문을 복원할 수 없는 성질을 말한다. 인증은 어떤 정보의 공급자나 취득자가 당사자가 맞는지, 그리고 그들 사이에서 행한 거래가 유효한지를 확인하는 수단이며, 무결성은 해당 데이터가 중간 전달 과정에서 위조, 변조되지 않고 원래의 형태로 보존되었는지 여부를 가리는 수단이다. 부인 방지는 메시지의 송·수신자가 송·수신 사실을 부인하지 못하도록 하는 방법이다[12].

2.2.1 실행파일 보호 기법

실행파일 보호 기법에는 명령어의 난독화, 내용의 암호화와 같은 방법이 있다. 명령어의 난독화를 통한 실행파일 보호는 명령어 코드에 대한 역공학을 힘들게 하여 중요 알고리즘을 보호하는데 사용한다. 이 방법은 성능상의 오버헤드(Overhead)가 작은 장점이 있지만, 실행파일 안에 있는 정보와 자원에 대한 노출을 막을 수 없는 단점을 가지고 있다. 암호화 기법을 통한 실행파일 보호는 암호화된 내용을 복호화하는 오버헤드가 발생하지만 실행파일의 모든 내용을 보호할 수 있는 장점을 가진다. 만약 일정 수준 이상의 성능을 요구하는 경우, 복호화 오버헤드가 작은 암호화 방법을 사용하여 성능을 올리는 방법을 사용할 수도 있다.

암호화를 통한 실행파일 보호 방법은 애플사의 Mac OS-X 운영체제에서 사용하고 있다[13]. Mac OS X의 실행파일은 여러 구역(Segment)로 이루어져 있다. Mac OS X에서 사용하는 실행파일 보호 기법은 모든 구역을 암호화하지 않고 보호가 필요한 구역만을 암호화한다. 그림 5는 실행파일 보호 구역이 들어간 실행파일을 실행하는 과정을 알고리즘의 형태로 정리한 것이다.

보호 구역은 구역 적재 명령 수행 과정에서 플래그 체크를 통해 일반 구역과 구분된다. 플래그가 보호된 구역을 표시하면, *LOAD_SUCCESS* 값을 반

```
1  // The execve system call handler
2  int execve(...) {
3        ...
4        // The case of a Mach-O file
5        error = exec_mach_imgact(...);
6        ...
7  }
8
9  load_return_t load_machfile(...)
10 {
11       ...
12       lret = parse_machfile(...);
13       ...
14 }
```

```
// Mach-O image activator
static int exec_mach_imgact(...) {
      ...
      lret = load_machfile(...);
      ... }
// Parse and handle Mach-O load commands
static load_return_t parse_machfile(...) {
      ...
      switch(lcp->cmd) {
      case LC_SEGMENT:
            ...
            ret = load_segment(...);
            ...
      } ... }
```

```
15 static load_return_t load_segment(...) {
16       ...
17       if(scp->flags & 0x8) {
18             // This is an apple-protected segment
19             ret = unprotect_segment_64((uint64_t)scp->fileoff, (uint64_t)scp->filesize, map,
20                         (vm_map_offset_t)map_addr, (vm_map_size_t)map_size)
21       } else ret = LOAD_SUCCESS;
22       return ret;  }
```

그림 5. 세그먼트 적재 알고리즘

환하기 전에 보호를 해제하는 작업을 수행한다. 구역 보호에 사용된 암호화 방법으로는 128비트 대칭키 암호화 알고리즘인 AES를 사용하고 있다.

2.2.2 암호 알고리즘

암호화 기술은 보안의 4가지 요구 사항인 데이터의 기밀성 유지와 사용자의 인증, 데이터 무결성 유지, 부인 방지 기술을 구현하는데 사용될 수 있다. 그러나 모든 암호화 알고리즘이 보안의 요구 사항을 해결하는데 사용할 수 있는 것은 아니다. 각 암호화 알고리즘의 특성에 따라 특정 요구 사항은 해결하기 어려운 경우도 있다. 실행파일 보호기법에 적당한 알고리즘을 선택하기 위하여, 암호화 알고리즘의 특성을 분석하고자 한다.

암호 알고리즘은 키의 분배 방법에 따라 크게 대칭키 암호 알고리즘과 공개키 암호 알고리즘으로 나눌 수 있다. 대칭키 암호 알고리즘은 수신자와 송신자가 동일한 비밀키를 사용하여 메시지를 교환하는 방법이다. 송신자와 수신자는 사전에 동일한 키를 분배 받아야하며 일반적으로 128비트의 키를 사용한다. 대칭키 암호 알고리즘은 인트라넷과 같은 폐쇄적인 소규모 네트워크에서는 사용이 용이하다. 그러나 인터넷과 같은 개방된 네트워크상에서는 n명의 사용자를 위해 n(n-1)/2 개의 서로 다른 비밀키가 요구된다. 대표적인 대칭키 암호 알고리즘으로는 DES, Triple-DES, SEED, AES 등이 있다.

대칭키 암호 알고리즘이 키 분배에서 문제점을 보이기 때문에 이를 해결하기 위해 Diffie와 Hellman은 공개키 암호 알고리즘을 제안하였다. 공개키 암호 알고리즘은 키를 두 개로 나누어 암·복호화 키로 사용한다. 암호화 키인 공개키는 모든 사용자에게 공개하고 복호화 키인 비밀키는 다른 사용자에게 공개되지 않도록 비밀리에 보관한다. 따라서 공개키 암호 알고리즘은 사전에 키를 분배할 필요가 없다. 대표적인 알고리즘으로는 RSA, ElGamal, ECC 등이 있다 [14,15]. 표 2는 암호 알고리즘별 보안 요구사항 만족 여부를 나타낸 것이다.

대칭키 암호 알고리즘이 인증, 부인방지와 관련된 보안 요소를 만족시키지 못하는데 비해 공개키 암호 알고리즘은 보안상의 모든 요구 사항을 충족시킬 수 있다.

표 2. 알고리즘별 보안 요구사항 만족 여부

구분	기밀성	무결성	인증	부인방지
DES	○	○	×	×
SEED	○	○	×	×
AES	○	○	×	×
RSA	○	○	○	○
ElGamal	○	○	○	○
ECC	○	○	○	○

3. 암호화를 통한 가상기계 실행파일 보호 기법

앞에서 말한 바와 같이 가상기계 실행파일을 보호하는 기법에는 여러 방법이 있지만, 본 논문에서는 암호화를 사용한 보호 기법을 제안하고 있다. 여기에서는 제안된 방법을 가상기계에 적용하기 위한 방법을 설명할 것이다.

일반적으로 가상기계 플랫폼에 기능을 추가하는 방법에는 두 가지 방법이 있다. 첫 번째 방법은 새로운 기능을 수행하는 정보와 명령어 코드를 실행파일에 추가하는 것이다. 이 방법은 실행파일 형식과 가상기계를 수정하지 않는 장점은 있다. 단점으로는 새로운 기능을 사용하는 실행파일에 관련 내용이 들어가기 때문에 크기가 커지는 문제점이 있다. 두 번째 방법은 실행파일에는 관련된 최소한 정보만을 추가하고 가상기계에서 그 정보를 사용하여 새로운 기능을 추가하는 것이다. 실행파일과 가상기계를 수정해야 하는 단점이 있지만, 실행파일 증가가 최소화되는 장점을 가진다.

본 논문에서는 후자의 방법을 사용하여 실행파일 보호 기법을 적용한 시스템을 설계하였다. 제안한 시스템은 실행파일 보호 기법을 구현하고, 이전 실행파일 형식과 가상기계와의 호환성을 유지, 최소한의 실행 오버헤드만을 가지는 것을 목표로 설계하였다. 이전 실행파일 형식과 가상기계와의 호환성을 위해서는 다음과 같은 사항을 만족시켜야 한다. 새로운 가상기계는 보호된 실행파일과 일반 실행파일을 실행할 수 있어야 한다. 또한 이전 가상기계는 일반 실행파일을 올바르게 실행하고 보호된 실행파일을 실행하는 경우에는 가상기계가 이상 동작을 하지 않아야 한다.

제안한 실행파일 보호 시스템의 개괄적인 구조는 그림 6과 같고 구성을 자세히 살펴보면 다음과 같다.

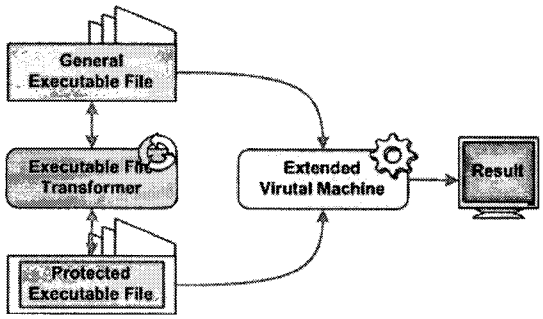

그림 6. 실행파일 보호 시스템 구성도

시스템은 크게 실행파일 보호 기법을 실행파일에 적용하기 위한 실행파일 보호 적용기와 실행파일 보호 기법이 적용된 실행파일을 실행할 수 있는 가상기계로 구성되어 있다. 실행파일 보호기법 적용기는 일반 실행파일에 보호된 실행파일로 변환하는 기능과 보호된 실행파일을 일반 실행파일로 변환하는 일을 한다. 가상기계는 보호된 실행파일뿐만 아니라 일반 실행파일도 실행할 수 있다.

다음 절에서는 위에 언급한 설계 목표와 구성을 가진 시스템을 위해 필요한 사항을 실행파일 형식 확장, 실행파일 보호 적용기, 가상기계 확장 순으로 살펴볼 것이다. 그리고 앞의 내용을 가상기계의 하나인 임베디드 가상기계(EVM: Embedded Virtual Machine)를 통해 설명할 것이다.

3.1 실행파일 형식 확장

실행파일 형식을 확장하는 경우에는 이전 가상기계나 도구와의 호환성 유지, 추가되는 정보의 최소화를 고려해야 한다. 호환성과 최소한의 정보 추가를 위해서는 전체적인 실행파일 구조를 변경하지 않는 방식을 찾아야 한다. 일반적인 실행파일의 경우, 상수 정보나 프로그램의 메타데이터를 저장하는 곳이 있다. 여기에 실행파일 보호를 위한 정보를 추가하여 위의 문제를 해결할 수 있다.

본 논문에서 예로 사용하는 EVM의 실행파일인 EFF인 경우에는 메타데이터를 위한 공간이 존재한다. 여기에 실행파일 보호에 필요한 정보를 가진 *MDTChiphertext* 항목을 추가하면 표 3과 같은 모양을 가지게 된다. 표 3과 같은 형식을 가지는 실행파일은 이전 가상기계와 도구에서는 해석할 수 없는 메타데이터만을 가진 실행파일로 인식하고, 이상 동

표 3. EVM을 위한 보호된 실행파일의 구성

EFF의 항목들	설 명
U4 magic	0×054C4CAB
U2 majorVersion	0×0001
U2 minorVersion	0×0000
U4 module	0×00000000
U4 language	0×00000000
U4 entryPoint	0×00000000
U4 VMCodesCount	0×00000000
VMCodeInfo VMCode [VMCodesCount]	없음
U4 metadataCount	0x00000001
MetadataInfo Metadata [metadataCount]	*MDTCiphertext* 테이블에 실행파일 보호 정보 저장

작을 하지 않고 종료하게 된다. 그러므로 설계 목표 중 하나인 호환성 문제를 해결할 수 있게 되었다.

좀 더 자세히 살펴보면 다음과 같다. 일반 실행파일에 암호화를 통한 실행파일 보호 기법을 단순히 적용하면, 생성된 실행파일은 EFF 고유의 형식을 가지지 않게 된다. 이 경우 이전 가상기계에서는 보호된 실행파일을 실행파일로 인식하지 않고 잘못된 파일로 인식하여 이상 동작을 할 가능성이 있다. 그리고 호환성 유지를 위해, 실행파일의 다른 요소는 EFF의 고유의 기본 값을 저장한다.

보호 기법에 필요한 정보를 저장하기 위해 추가한 *MDTCiphertext* 메타데이터의 구조는 표 4와 같다.

MDTCiphertext 메타데이터는 크게 보호 기법의 종류와 버전, 보호 기법에서 사용하는 키 정보, 보호 기법이 적용된 데이터로 구성되어 있다. 각 정보는 다시 정보의 길이를 나타내는 항목과 정보를 나타내는 항목으로 구성되어 있다. 자세히 살펴보면, *versionLen*은 보호기법 종류 및 버전 정보에 대한 길이를 나타내며, *versionString*은 보호기법 종류와 버전 대한 정보를 저장한다. *keyLen*은 키의 길이를 나타내고, *key*는 키 정보를 저장한다. *cipherLen*은 보호기법이 적용된 데이터의 길이를 나타내며, *ciphertext* 항목은 보호기법을 적용하기 위해 일반 실행파일의 모든 내용을 암호화하여 저장한 데이터를 나타낸다.

표 4. *MDTCiphertext* 메타데이터의 항목들

메타데이터 항목들	설 명
U1 versionLen	보호기법 종류 및 버전 정보 길이
U1 versionString [versionLen]	보호기법 종류 및 버전 정보
U1 keyLen	키의 길이
U1 key[keyLen]	키 정보
U4 cipherLen	보호기법이 적용된 데이터의 길이
U1 ciphertext[cipherLen]	보호기법이 적용된 데이터

3.2 실행파일 보호 적용기

실행파일 보호기법 적용기는 일반 실행파일에 보호 기법을 적용하여 보호된 실행파일을 생성하는 일과 보호된 실행파일을 일반 실행파일로 변환하는 일을 한다.

적용기가 보호된 실행파일을 변환하는 경우에는 그림 7과 같은 과정을 통해 실행파일을 변환한다.

과정은 암호화기, 재구성기, EFF 저장기(EFF Writer)와 같은 요소로 구성되어 있다. 각 요소는 다음과 같은 일을 한다. 암호화기는 실행파일 보호를 위해 일반적인 실행파일의 모든 정보를 암호화하여 내부 공간에 저장한다. 재구성기는 임시공간에 암호화된 정보를 사용하여 보호된 결과 실행파일을 생성하기 위한 사전 작업을 한다. 암호화된 정보를 EFF와 일대일로 대응되는 내부 자료 구조에 저장하는 작업을 한다. EFF 저장기는 재구성기의 결과인 내부 자료 구조를 읽어 EFF 파일로 저장하는 일을 한다.

적용기가 보호된 실행파일을 일반 실행파일로 변환하는 경우에는 그림 8과 같은 과정을 통해 실행파일을 변환한다.

변환 과정은 제거기, 복호화기, 재구성기, EFF 저장기의 4가지 모듈로 구성되며, 단계별 정보를 저장하기 위해 내부 버퍼와 내부 자료 구조를 사용한다.

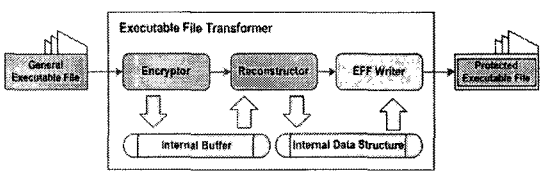

그림 7. 실행파일 보호기법 적용기(보호된 파일 생성시)

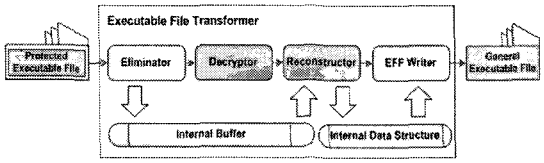

그림 8. 보호된 실행파일 복구를 위한 추출기

보호된 실행파일에 저장된 정보는 *MDTCiphertext* 메타데이터 테이블에 저장된 정보만이 유효한 정보이다. 따라서 필요 없는 정보는 모두 제거해야 하고 내부 버퍼에 저장한다. 복호화기는 내부 버퍼에 저장된 정보를 복호화하여 다시 내부 버퍼에 저장하는 일을 한다. 재구성기는 보호된 파일을 생성하는 경우와 같이 버퍼에 저장된 정보를 내부 자료 구조로 변환하는 일을 한다. 마지막으로 EFF 저장기는 내부 정보를 EFF 파일을 저장하는 일을 한다.

3.3 가상기계 확장

가상기계에서 제안한 보호된 실행파일을 실행하기 위해서는 가상기계의 기능을 확장해야 한다. 그림 9는 보호된 실행파일을 실행하기 위한 기능이 확장된 가상기계의 구조를 도식화한 것이다.

그림 9와 같이 보호된 실행파일을 실행하기 위해서는 가상기계의 입력으로 들어온 실행파일이 일반 실행파일인지 보호된 실행파일인지를 구분하고 이에 따라 처리해야 한다. 본 논문에서 일반 실행파일과 보호된 실행파일의 구분은 분석기가 수행하며, *MDTCiphertext* 메타데이터의 존재 여부로 구분한다. 그림 10은 확장된 가상기계에서 실행파일을 실행하는 과정을 의사코드 형태로 나타낸 것이다. 가상기계는 실행파일을 헤더와 메타데이터 영역을 순차적으로 읽는다. 메타데이터 영역의 각각의 테이블을 검

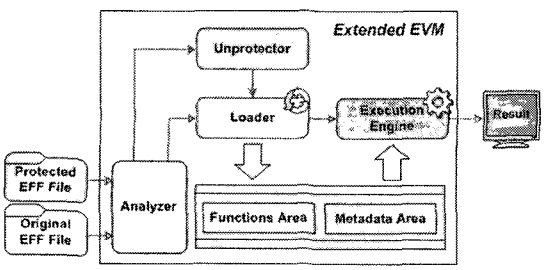

그림 9. 보호된 실행파일의 실행을 위해 확장된 EVM의 구조

```
1  while not EFF file's EOF
2      read all header, vmcode of EFF
3      read a metadata
4      if metadata's tag = Ciphertext metadata tag
5          then flag := 1, break
6  if flag = 1 then Unprotect EFF file
7  load EFF into memory
8  execute from memory
```

그림 10. EFF 파일의 실행 알고리즘

색하여 *MDTCiphertext* 메타데이터 테이블이 존재하면 보호된 실행파일이므로, 보호 해제 작업을 수행한다.

보호 해제된 실행파일 내용은 일반 실행파일 내용과 같으므로, 일반 실행파일을 처리하는 것과 동일한 과정으로 읽고 실행한다.

4. 실험 및 결과 분석

본 논문에서 제안한 실행파일 보호 기법은 임베디드를 위한 가상기계인 EVM에 적용하고 실험하였다. 또한 구현에 사용한 암호화 방식은 다양한 참고 구현이 존재하는 DES(Data Encryption Standard) 암호화 알고리즘을 사용하였다. 제안한 보호 기법은 DES 암호화 알고리즘이 아닌 다른 암호화 알고리즘을 사용하고자 하는 경우에는 쉽게 다른 암호화 알고리즘을 적용할 수 있도록 설계되어 있다.

보호 기법의 구현과 실험에 사용한 컴퓨터와 언어 환경은 다음과 같다. 사용한 컴퓨터는 펜티엄4 2.0 프로세서와 1기가의 메모리를 갖는 IBM 호환 컴퓨터이고 운영체제로서 마이크로소프트(Microsoft)사의 Windows XP SP2을 사용하였다. 언어 환경으로는 ANSI C를 사용하였고, 컴파일러로는 마이크로소프트사의 Visual C++ 2005를 사용하였다. 실행 시간 측정에는 컴퓨웨어(Compuware)사의 데브파트너(DevPartner)를 사용하였다.

4.1 구현된 시스템의 사용법

그림 11은 실행파일 보호를 지원하는 가상기계와 실행파일 보호를 위한 생성기, 추출기의 사용 방법을 나타낸 것이다.

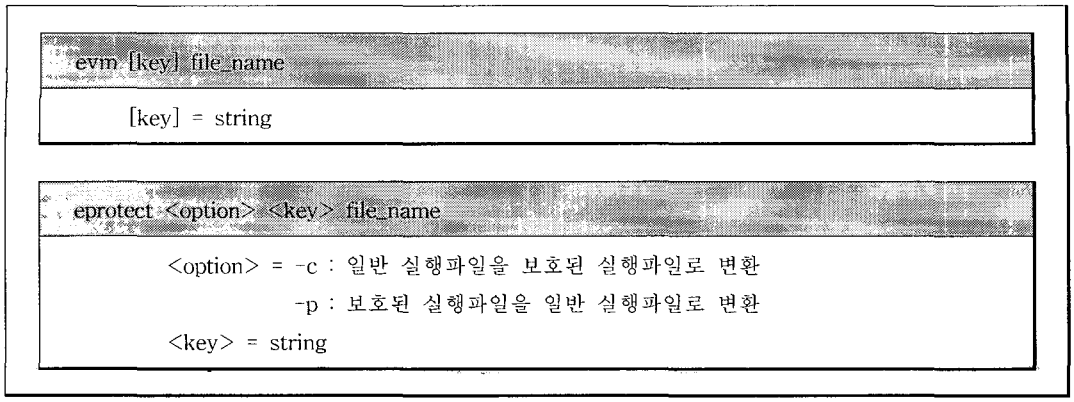

그림 11. 가상기계와 기법 적용기의 사용법

가상기계인 *evm*은 실행 대상인 파일과 키를 인자로 입력 받아, 실행하며 키의 입력은 선택 사항이다. 키가 필요한 보호된 실행파일의 경우에는 오류 메시지를 출력하고 실행을 멈춘다. 기법 적용기인 *eprotect*는 인자에 따라 보호 기법 적용과 해제 역할을 수행한다. 실행파일 보호를 위해 사용되는 키와 파일 이름을 인자로 받아 보호된 실행파일 또는 일반 실행파일을 생성한다.

4.2 보호 기법의 적용 결과

그림 12는 일반 실행파일의 한 부분으로 상수로 관리자 아이디와 암호가 들어가 있는 것을 확인할 수 있다.

그림 13은 본 논문에서 제안한 보호 기법이 적용된 실행파일의 한 부분을 나타낼 것이다. 그림 12에서 확인할 수 있는 여러 상수 정보들이 나타나지 않는 것을 확인할 수 있다.

```
00000110h: 00 00 51 00 52 00 A0 00 53 00 04 1B 00 0C 00 6C ; ..Q.R.?S......l
00000120h: 6F 67 69 6E 5F 73 65 72 76 65 72 0B 00 77 69 6E ; ogin_server..win
00000130h: 64 6F 77 73 32 30 30 33 00 45 65 00 07 00 61 63 ; dows2003.Ee...ac
00000140h: 63 6F 75 6E 74 08 00 70 61 73 73 77 6F 72 64 04 ; count..password.
00000150h: 00 61 75 74 68 05 00 61 64 6D 69 6E 08 00 72 68 ; .auth..admin..rh
00000160h: 6B 73 66 6C 77 6B 08 00 72 77 72 77 72 77 72 77 ; ksflwk..rwrwrwrw
00000170h: 05 00 61 6C 69 63 65 08 00 74 68 64 74 6C 73 77 ; ..alice..thdtlsw
00000180h: 6B 08 00 5F 5F 72 5F 72 5F 72 77 05 00 75 73 65 ; k..__r_r_rw..use
00000190h: 72 31 03 00 6E 6F 6E 08 00 5F 5F 5F 5F 5F 5F 72 ; r1..non..______r
000001a0h: 5F 00 53 08 00 88 39 00 00 12 0F 00 AB 00 54 08 ; _.S..?.....?T.
```

그림 12. 일반 실행파일

```
00000110h: 14 6A BE A9 CE 4B BC D4 88 A6 04 67 57 A9 D2 98 ; .j쑴?솎닰.gW(f)?
00000120h: 48 74 7A 7C BE 05 B1 DE E8 A0 5C AC FE 68 B2 3F ; Htz|?급?\?h?
00000130h: 8F 56 00 D5 32 AF EA FE 04 5A 49 A2 99 88 99 7A ; 퀟.???ZI줮넕z
00000140h: 69 FA 7B 89 57 7A 9E 93 10 4F 8C DB 1E E4 A8 BF ; i?뎉z옜.O똭.十?
00000150h: 61 8C E1 85 D3 49 85 2A FC 1E 2C 3E 04 A8 D7 C6 ; a뜾퓕I??,>.ⓚ?
00000160h: AF 99 1A D9 35 E5 E6 76 48 61 63 0C 77 50 8B 36 ; 캎.?膳vHa?wP?
00000170h: 9A E7 A0 79 D4 A5 44 6F 4C 5B 8D 79 C7 5F E9 E3 ; 싍쟹楹DoL[퓕?芊
00000180h: 26 77 CD A1 30 A6 37 EB 41 6F 61 4F 85 20 82 F4 ; &w榮0??oaO?급
00000190h: 9F 24 F7 54 8D AA A4 3B 4D FB F7 2F 78 50 1E 77 ; ??뚱?M訌/xP.w
000001a0h: D9 19 F1 0C 7F C7 A1 0F 99 56 62 4F FE 76 85 4D ; ??□퐉.쎌bO?꿒
```

그림 13. 실행파일 보호 기법을 적용된 실행파일

또한 가상기계에서도 제대로 실행되는 것을 확인할 수 있었다.

4.3 보호 기법의 오버헤드 분석

가상기계에서 보호된 실행파일은 일반 실행파일에 비해 처리해야 할 과정이 더 나타나므로 실행 속도 저하와 실행파일 크기 증가와 같은 오버헤드가 발생한다.

표 5는 여러 프로그램에서 실행파일 크기 증가량과 실행 시간 증가량과 같은 오버헤드를 측정하여 정리한 표이다.

논문에서 사용한 실행파일인 EFF 파일은 57 바이트 크기의 헤더를 가지고 있다. 보호된 실행파일의 경우에도 이전 가상기계와의 호환성을 위해서 헤더를 가지고 있으면 헤더 크기만큼의 고정된 오버헤드를 가지게 된다. 이 고정된 헤더 오버헤드는 크기가 큰 실행파일의 경우에는 무시할 수 있는 항목이 된다. 고정 헤더 오버헤드를 무시한 실행파일 크기 증가량은 평균 약 6% 이고, 일반 실행파일의 크기가 커질수록 증가량이 작아지는 것을 알 수 있다.

실행 시간 오버헤드는 주로 보호된 실행파일의 내용을 보호 해제하는데 사용되는 시간이다. 실험에서 사용한 시스템에서 DES의 경우에는 약 1초 이상의 수행 속도를 갖는 프로그램은 파일의 크기와 상관없이 수행 속도가 거의 차이가 나지 않음을 확인할 수 있었다.

5. 결론 및 향후 연구

정보에 대한 접근의 용이성은 사용자에게 유용한 정보를 쉽게 제공할 수 있는 장점이 있는 반면, 의도하지 않은 정보 유출과 같은 문제를 발생시킬 수 있는 양면성을 가지고 있다. 특히, 실행파일을 통한 소프트웨어의 주요 정보 유출은 경제적 측면에서 큰 손실을 가져올 수 있다. 따라서 실행 환경은 정보 유출 문제를 해결할 수 있는 방법을 제공할 필요가 있다.

본 논문에서는 실행파일에서의 정보 유출은 문제를 해결하기 위해 암호화를 이용한 실행파일 보호 기법을 제안하였다. 암호화는 기밀성과 무결성을 유지하는데 사용되는 방법으로 실행파일 분석을 통한 의도하지 않은 정보의 노출을 사전에 차단할 수 있게 한다. 제안된 기법을 임베디드 기기를 위한 가상기계인 EVM에 적용하여 올바르게 작동을 확인하였다. 또한 제안된 기법으로 인해 발생한 실행 시간과 실행파일 크기의 오버헤드를 다양한 작업을 수행하는 실행파일에 기법을 적용하여 측정하였다. 측정된 오버헤드 분석을 통해 기법 적용전의 실행파일의 크기가 클수록, 실행 시간이 길수록 상대적인 오버헤드가 적어지는 것을 확인하였다. 일반적인 실행파일은 실험에서 사용한 것보다 크기가 크고, 실행시간이 길기 때문에 보호 기법에 의해서 나타나는 오버헤드는 감내할 수 있는 수준임을 알 수 있었다.

향후 연구로는 본 논문에서 제안한 실행파일 보호 기법의 오버헤드가 감내할 수준이기는 하지만, 연산 능력이 부족한 기기에서는 이 정도 수준의 오버헤드도 문제가 될 수 있다. 따라서 연산 능력이 부족한 기기를 위해, 보호가 필요한 실행파일 영역만 부분적으로 보호 기법을 적용하여 오버헤드를 줄이는 방법을 연구할 것이다. 또한, 보호의 목적이 실행파일의 무결성만을 확보하는 것이라면 일방향 해쉬 함수와

표 5. 일반 실행파일와 보호된 실행파일간의 오버 헤드

프로그램 이름	일반 EFF		보호된 EFF		증가 비율	
	파일 크기 (바이트)	수행 속도 (초)	파일 크기 (바이트)	수행 속도 (초)	파일 크기 (배)	수행 속도 (배)
factorial.eff	232	0.017	315	0.089	1.36	5.225
magic.eff	487	0.215	559	0.338	1.15	1.572
fibonacci.eff	249	0.078	331	0.092	1.33	1.179
palindromic.eff	310	1.323	389	1.413	1.25	1.068
bubbleSort.eff	464	2.704	547	2.847	1.18	1.053
prime.eff	342	15.988	415	16.035	1.21	1.003
perfect.eff	329	57.923	409	57.942	1.24	1.000

같은 무결성과 관련된 알고리즘을 적용하여 문제를 해결하는 연구가 필요하다. 그리고 최근에 문제가 되고 있는 소프트웨어나 컨텐츠의 불법적인 사용 방지를 위한 저작권 관리 시스템을 가상기계에 추가하는 작업도 향후 연구로 고려해 볼 수 있다.

참 고 문 헌

[1] 오세만, 이양선, 고광만, "임베디드 시스템을 위한 가상기계의 설계 및 구현," 멀티미디어학회 논문지, 제 8권, 제 9호, pp. 1282-1291, 2005.

[2] 손윤식, 오세만, "실행 파일 포맷 생성기의 설계 및 구현," 한국정보처리학회 추계학술발표대회 논문집, 제11권, 제2호, pp. 623-626, 2004.

[3] 전병준, 이창환, 오세만, "퍼베이시브 컴퓨팅을 위한 가상기계의 어셈블러," 한국정보처리학회 추계학술발표대회 논문집, 제13권, 제2호, pp. 589-592, 2006.

[4] 최유리, 이창환, 오세만, "퍼베이시브 컴퓨팅을 위한 가상기계의 디스어셈블러," 한국정보처리학회 추계학술발표대회 논문집, 제13권, 제2호, pp. 585-588, 2006.

[5] 박지우, 이창환, 오세만, "퍼베이시브 컴퓨팅을 위한 가상 기계의 실행 엔진," 한국정보처리학회 추계학술발표대회 논문집, 제13권, 제2호, pp. 581-584, 2006.

[6] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification, 2nd Edition*, Addison Wesley, 1999.

[7] *MSIL Instruction Set Specification*, Microsoft Corporation, 2000.

[8] SIL Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

[9] SAF Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

[10] 정한종, 임베디드 가상기계를 위한 실행 파일 포맷, 동국대학교 석사학위논문, 2004.

[11] EFF Specification, 동국대학교 프로그래밍 언어 연구실, 2006.

[12] 남수현, "암호 시스템에 대한 이해", 마이크로소트웨어, 2월호, pp. 376-377, 2003.

[13] Amit Singh, *Understanding Apple's Binary Protection in Mac OS X*, http://osxbook.com/book/bonus/chapter7/binaryprotection/, 2006.

[14] Douglas R.Stinson, *Cryptography Theory and Practice, 2nd Edition*, CRC Press, 2002.

[15] 원동호, 현대 암호학, 도서출판 그린, 2004.

박 지 우

2004년 2월 동국대학교 전자공학과 졸업(학사)
2006년 9월~현재 동국대학교 컴퓨터공학과(석사과정)
관심분야 : 프로그래밍 언어, 컴파일러, 가상기계

이 창 환

1998년 2월 동국대학교 컴퓨터공학과 졸업(학사)
2000년 2월 동국대학교 대학원 컴퓨터공학과(공학석사)
2003년 2월~동국대학교 대학원 컴퓨터공학과(공학박사)
2006년 9월~현재 (주) 링크젠 책임연구원
2007년 3월~현재 동국대학교 산업기술연구원 겸임교수
관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

오 세 만

1985년 3월~현재 동국대학교 컴퓨터공학과 교수
1993년 3월~1999년 2월 동국대학교 컴퓨터 공학과 대학원 학과장
2001년 11월~2003년 11월 한국정보과학회 프로그래밍 언어연구회 위원장
2004년 6월~2005년 12월 한국정보처리학회 게임연구회 위원장
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅