개선된 가상 에뮬레이터를 이용한 다형성 바이러스 탐지 방법

김 두 현[†] · 백 동 현[††] · 김 판 구[†††]

요 약

프로그램 내 바이러스 코드 패턴을 탐색하는 현재의 백신 프로그램은 암호화 바이러스나, 다형성 바이러스를 탐지하는 데 어려움이 있다. 다형성 바이러스는 암호를 해제하는 코드 부분이 감염될 때마다 변형된다. 그래서, 이 바이러스를 탐지하기 위해서는 바이러스 본체를 해제하는 암호해제 코드의 행동을 추적해보아야 하며, 코드 분석시 많은 시간이 소요되는 것이 일반적이다. 특히, 바이러스 제작자가 바이러스 암호해제 코드의 반복 실행 수를 늘려 놓았다면 기존의 방식으로는 이를 발견하기 어렵다. 본 논문에서는 이러한 다형성 바이러스를 탐지하기 위해서 개선된 알고리즘을 이용한 에뮬레이터를 제안한다. 이를 이용하여 다형성 바이러스를 탐지해본 결과, 기존의 에뮬레이터에 비해 다형성 바이러스 진단율이 약 2%정도 향상되었다. 또한, 제안된 다형성 바이러스 진단 시스템은 MS-Windows 뿐만 아니라 Linux 등 Unix 계열 플랫폼에서도 동작할 수 있다는 장점이 있다.

A Detecting Method of Polymorphic Virus Using Advanced Virtual Emulator

Doo-Hyun Kim[†] · Dong-Hyun Baek[††] · Pan-Koo Kim[†††]

ABSTRACT

Current vaccine program which scans virus code patterns has a difficult to detect the encrypted viruses or polymorphic viruses. The decryption part of polymorphic virus appears to be different every time it replicates. We must monitor the behavior of the decryption code which decrypts the body of the virus in order to detect these kinds of viruses. Specially, it is not easy for the existing methods to detect the virus if the virus writer has modified the loop count of execution intentionally. In this paper, we propose an advanced emulator using a new algorithm so as to detect various kinds of polymorphic viruses. As a result of experiment using advanced emulator, we found that our proposed method has improved the virus detecting rate about 2%. In addition, our proposed system has a merit that it runs on not only MS-Windows but also Linux, and Unix-like platform.

키워드 : 바이러스(Virus), 에뮬레이터(Emulator), 다형성(Polymorphic)

1. 서 론

최근 네트워크 기술 발전에 힘입어 인터넷 보급이 확산됨에 따라 외국에서 발견된 바이러스가 몇 시간 뒤면 국내에서도 발견되고 있다. 이제 바이러스의 보안 문제는 국경을 초월하고 정보전의 양상을 띄는 등 그 심각성이 크게 대두되고 있다.

일반적인 바이러스 진단 방법으로는 평문 문자열을 비교(string comparison)하는 방법과 암호화 된 것을 풀어서 진단하는 복호화 방법이 있다. 문자열을 비교하는 방법은 다시 특정 위치 검색법과 전 프로그램 검색법으로 나뉜다. 즉, 첫 번째는 부트 섹터나 프로그램 내의 특정 위치에서 컴퓨터 바이러스의 문자열이 존재하는지를 검사하는 방법이며, 두 번째는 위치에 상관없이 부트 섹터나 프로그램 전체를 검색하여 그러한 문자열이 존재하는지를 검사하는 방법이다[9, 10].

복호화 알고리즘을 이용하는 방법은 다형성 바이러스(polymorphic virus)를 진단하기 위한 것이다. 다형성 바이러스는 암호화 된 바이러스의 일종인데, 암호화 위치나 방법이 일정한 단순 바이러스와는 달리, 다형성 바이러스는 감염시킬 때마다 암호를 해제하는 부분이 각기 다른 형태를 보이고 있다[7].

† 준 회 원 : 조선대학교 전자계산학과 대학원
†† 정 회 원 : (주)하우리 기술연구소 소장
††† 정 회 원 : 조선대학교 컴퓨터공학부 교수
논문접수 : 2001년 8월 20일, 심사완료 : 2002년 2월 8일

보통의 백신 프로그램은 바이러스 코드의 패턴을 알고 있어야 하는 방식으로 말미암아 다형성 바이러스 탐지의 어려움이 있다. 이를 해결하기 위해서 바이러스 코드의 행위를 추적해 볼 수 있는 가상실행 에뮬레이션 방법이 필요하다. 특히, 바이러스 제작자가 의도적으로 바이러스 내부에 실행 스텝수를 늘려 놓았다면 기존의 방법으로도 다형성 바이러스를 발견하지 못한다. 이에 본 논문에서는 바이러스 실행코드와 실행상의 특징을 이용하여 가상 메모리 상에서 다형성 바이러스를 진단하기 위한 새로운 가상 실행 에뮬레이션 방법을 제안한다. 실험 결과, 새로운 알고리즘을 이용한 에뮬레이터를 동작시킨 경우 기존의 방법에 비해서 다형성 바이러스 탐지율이 향상되었다.

본 논문의 구성은 다음과 같다. 서론에 이어 2장에서는 일반적인 바이러스 진단 방법에 대해 알아보고, 기존 에뮬레이터의 문제점에 대해서 살펴본다. 3장에서는 다형성 바이러스를 탐지하기 위한 새로운 알고리즘을 보이고, 4장에서는 다형성 바이러스 진단을 위한 시스템 구현과 새로운 알고리즘을 적용 한 후 실험 결과를 제시하고, 5장에서 결론을 제시하며 끝을 맺는다.

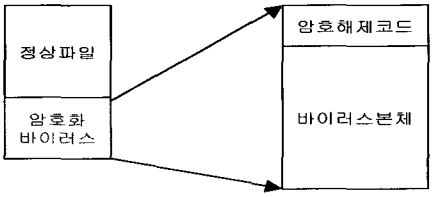
2. 관련 연구

본 장에서는 바이러스의 종류와 일반적인 바이러스 진단 방법에 대하여 알아보고, 가상 에뮬레이터 구현 방안에 대하여 논한다.

2.1 세대별 바이러스 발전양상과 다형성 바이러스

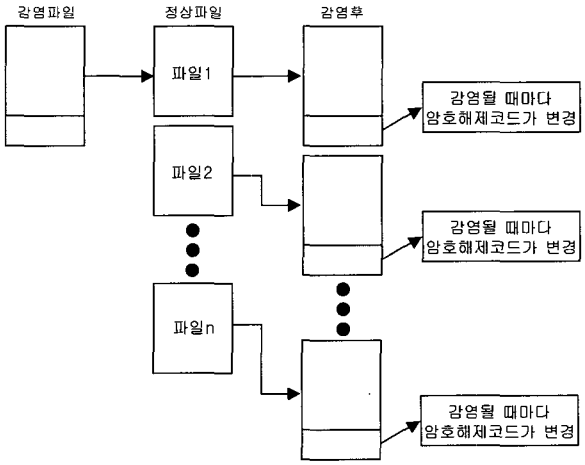
일반적으로 바이러스는 바이러스의 발전 단계에 따라 세대별로 분류한다. 처음 출현한 1세대 바이러스는 원시형 바이러스, 2세대 바이러스는 암호화 바이러스, 3세대 바이러스는 은폐형 바이러스, 4세대 바이러스는 다형성 바이러스, 그리고 5세대 바이러스는 스크립트형 바이러스로 나눌 수 있다.

원시형 바이러스는 초기에 제작된 만큼 가장 단순하고 분석이 쉬워 실력이 뛰어나지 않은 아마추어 프로그래머들에 의해 제작된 바이러스이다. 암호화(encryption) 바이러스는 어느 정도 실력을 갖춘 프로그래머들에 의해 만들어졌으며, 프로그램의 일부 또는 대부분을 암호화시켜 저장한다. 이 방법은 암호를 푸는 코드가 함께 내장되는 것이 일반적이다.



(그림 1) 암호화 바이러스

은폐형(stealth) 바이러스는 자신을 숨길뿐만 아니라 사용자나 백신 프로그램에게 거짓 정보를 제공하기 위해 다양한 기법을 사용한다. 즉, 기억장소에 존재하면서 감염된 파일의 크기가 변하지 않은 것처럼 보이게 하여 백신 프로그램을 속인다[9, 10]. 다형성(polymorphic) 바이러스는 암호화 바이러스의 일종으로 암호를 푸는 부분이 항상 일정한 암호화 바이러스와는 달리 암호를 푸는 부분조차도 감염될 때마다 변형된다. 그래서, 이 바이러스에 감염된 파일을 치료하기 위해서는 여러가지 형태로 변하는 행동을 모두 역추적해야 하며 보통 그 행위를 파악하는데 시간이 많이 걸리며 그 행위 또한 복잡하다. (그림 2)는 다형성 바이러스가 감염될 때마다 다른 암호해제코드가 들어가는 것을 보이고 있다. 즉, 같은 파일을 감염시킬 때에도 각기 다른 암호해제코드가 삽입됨을 의미한다[1, 2, 3].



(그림 2) 다형성 바이러스의 감염

2.2 바이러스 진단 방법

바이러스를 진단하기 위해서는 기본적으로 문자열을 비교한다. 다시 말해 바이러스에 감염된 파일의 일부분을 선택하여 비교 문자열로 삼고, 부트 섹터나 프로그램 내에 이것이 존재하는지를 검색하는 것이다. 문자열 비교 방법은 특정 위치 진단법과 전체 진단법으로 나뉜다. 특정 위치 진단법은 부트 섹터나 파일 내의 특정 위치에서 컴퓨터 바이러스의 문자열이 존재하는지를 검사하는 방법이며, 전체 진단법은 위치에 상관없이 부트 섹터나 파일 전체를 탐색하여 찾고자 하는 문자열이 존재하는지를 검색하는 방법이다[10].

바이러스 제작자들은 여러 가지 기법을 사용하여 자신이 만든 바이러스를 백신 프로그램이 진단하지 못하도록 회피하면서 감염시킨다. 초기에는 감염패턴이 일정한 바이러스들이 등장하다가 백신들이 쉽게 진단모듈을 만들어내자 이를 다시 암호화한 바이러스를 만들었다. 그러다가 나중에는 암호화한 내용도 알아보기 힘들게 만든 다형성 바이러스를 제작하였다. 여기서 다형성 바이러스란 진보된 암호화 바이

러스로서 감염될 때마다 바이러스 본체를 바꾸는 것은 물론 암호해제코드 부분까지도 바꾸게 되므로 단순한 패턴 진단방법만으로는 탐지해낼 수 없는 바이러스이다. 다형성 바이러스의 경우에는 일반적인 문자열 비교 방법으로는 진단할 수 없기 때문에 복호화 알고리즘을 이용하여 진단해야 한다. 이를 진단하기 위해서는 백신에 간단한 에뮬레이터를 탑재하여 바이러스 자체의 암호해제 코드를 모의실행한 후 나타나는 행위를 기반으로 하여 바이러스 본체를 패턴 진단하는 방법을 사용한다[9].

〈표 1〉 바이러스 진단방법의 특징

바이러스 종류	진단방법	비 고
원시형	단순한 문자열 비교로 진단 가능	파일 내 문자열 검색
암호형	실행이 시작되는 부분에 있는 암호해제코드가 항상 일정하므로 쉽게 분석 할 수 있음	암호 해제 쉬움
다형성	에뮬레이터에서 모의로 실행시켜 봄으로써 바이러스 패턴 진단 가능	가상 에뮬레이터 필요

2.3 가상 에뮬레이터의 구현방안

일반적으로 에뮬레이터는 어떤 하드웨어나 소프트웨어의 기능을 다른 종류의 하드웨어나 소프트웨어로 모방하여 실현시키기 위한 장치나 프로그램을 말한다.

프로그램 코드와 데이터가 실행되려면 CPU와 메모리가 필요하듯이 에뮬레이터를 구성하려면 기본적으로 실행 프로그램을 적재하고 실행할 가상의 메모리공간이 필요하다. 필요한 크기는 인텔 8086 CPU의 전체 메모리 공간인 1MB 만큼을 확보하는 것이 이상적일 수 있으나 바이러스의 동작 특성을 고려해서 바이러스 코드의 평균적인 크기를 최소 메모리로 확보하고, 추가적으로 소요되는 메모리는 캐쉬(cache)처리로 효율적인 공간을 설계한다. 또한, 적재된 프로그램을 실행시킬 수 있는 디코더(decoder)가 필요하다. 즉, 실행 프로그램은 기계어로 쓰여진 프로그램이므로 파일로부터 읽혀진 기계어를 해석하고 해당 동작을 수행할 수 있는 프로그램을 설계해야 한다[5].

3. 다형성 바이러스 진단

본 장에서는 2장에서 알아본 에뮬레이터를 이용한 기존의 다형성 바이러스 탐지 방법을 분석하여 문제점을 밝혀내어 그 문제점을 보완한 새로운 다형성 바이러스 진단 기술의 특징과 장점에 대하여 기술한다.

3.1 기존의 진단 방법

아래 (그림 3)의 순서도를 보면, 지정반복횟수(Nc) 만큼을 실행하면서 바이러스 본체라고 판단되는 부분부터 진단 패턴을 찾는다. 기존의 다형성 바이러스 진단 방법의 문제점은 Nc를 늘리면 처리속도가 늦어지고 Nc를 줄이면 바이러스 본체가 제대로 나타나지 않아 완벽한 진단을 하기 어렵다는 것이다. 그리고, 정상파일인 경우에도 Nc만큼 모의실행해야 하기 때문에 백신 전체 처리속도가 크게 떨어진다.

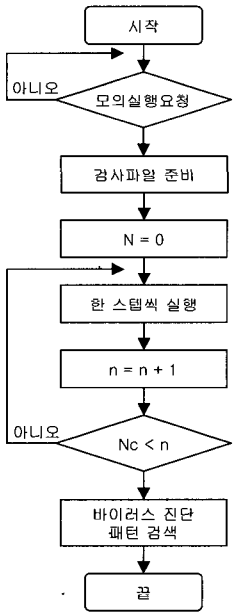
여기에서 Nv, Ni, 그리고, Nt를 아래와 같이 정의할 때,

- Nv : 암호화된 바이러스를 해제하는 데 필요한 반복회수,
- Ni : 경험적 통계치에 근거한 암호화 해제구간의 스텝수,
- Nt : 총스텝수,

암호화 바이러스를 완전히 해제하기 위한 총 스텝수는 아래와 같다.

$$Nt = Nv \times Ni$$

다시 말하면, 바이러스 패턴비교를 위하여 바이러스 본체 전체를 해제하려면 Nc는 Nt보다 충분히 크거나 같게 결정해야만 함을 의미한다.



(그림 3) 기존의 다형성 바이러스 진단 순서도

3.2 새로운 다형성 바이러스 진단 방법

지정반복횟수(Nc)를 정하는 이유는 다음과 같다. 바이러스 본체 전부가 해제될 때까지 가상으로 실행한다면 실제 실행이 아닌 모의실행이니 만큼 속도가 떨어지는 것을 막기 위해 본체 일부만 해제시킨 후 그 곳을 검사하기 위해서이다. 앞서 설명한 기존의 진단방법에서 백신 프로그램들마다 지정반복횟수를 지정하는 방법이 각각 다르다. 그런데, 바이러스 제작자가 바이러스 내부에 암호를 해제하기

앞서 실행 스텝수를 지정반복횟수 내에서 검사하지 못하도록 의도적으로 늘려 놓았다면 기존의 방법으로도 이 바이러스를 발견하지 못한다.

그래서 다음과 같은 새로운 진단 방법을 제안한다.

먼저, 바이러스의 암호를 해제할 때 실행코드에는 반드시 필요한 실행상의 특징이 있음을 발견할 수 있다. 나열하자면 다음과 같다.

① 특정 메모리 위치(주소)의 내용을 일정한 방법으로 변경하려는 실행코드가 반드시 존재한다.
② 특정 위치에 존재하는 ①항의 동작을 주어진 조건을 만족할 때까지 반복 동작한다.

실행코드로 말하자면 조건부 분기(conditional jump) 실행코드가 반드시 존재하며 특정주소에 있으면서 반복 실행된다.

(그림 4)의 순서도는 앞서의 진단방법을 위에 기술한 실행 코드를 발견하기 위해 개선한 것이다. 제1실행코드는 ①항을 말하고 제2실행코드는 ②항을 말한다.

여기서 지정횟수는 제1실행코드나 제2실행코드가 등장할 때까지의 경험적 통계치에 근거한 것으로 예상 스텝수(Nt')는 아래와 같이 표현할 수 있다.

$$Nt' \geq Nc'$$

여기에서 일정횟수(Nk)의 관계는 다음을 만족하도록 설정할 수 있다.

$$Nn < \text{바이러스 검사영역의 크기} \leq \text{일정횟수}(Nk)$$

여기서 Nn는 진단 모듈 내의 진단용으로 가지고 있는 바이러스 패턴의 최대 크기이다.

또한, Nk'와 Nt는 다음 성질을 만족한다.

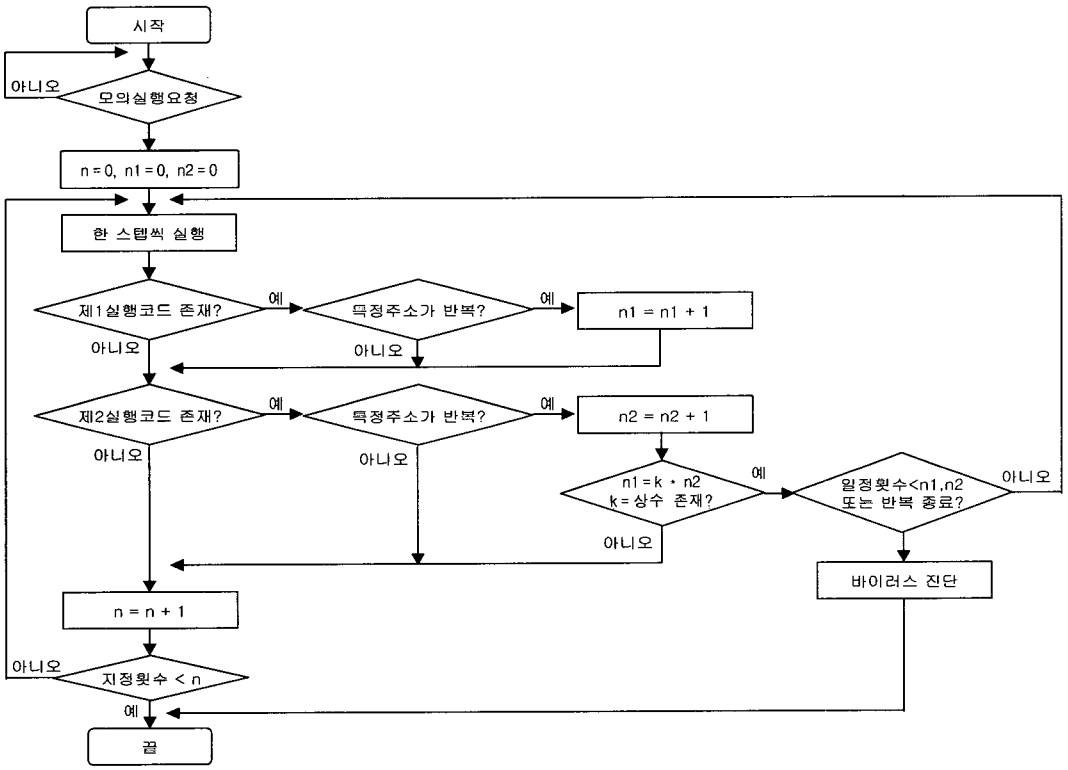
- $Nt = Nv \times Ni$
- $Nk' < Nt$

그러므로, Nc'를 기존의 Nc보다 대단히 작게 설정할 수 있다. Nc' 내에서 제1실행코드나 제2실행코드가 등장하지 않는다면, 즉 다형성 또는 암호화 바이러스가 아니라면 앞서의 기술보다 비교동작이 추가되는 것을 감안해도 Nc'이 Nc보다 매우 작으므로 진단속도가 빠르다.

그리고 Nc' 내에 제1실행코드나 제2실행코드가 등장하면 암호를 해제하기 위한 동작이라고 보고 지정횟수와 n을 비교하는 동작은 더이상 하지 않고 일정횟수와 비교하게 되는데 다음 두 가지를 추가로 만족해야 한다.

① 계속 같은 주소에 있는 실행코드가 반복되는가?
② '제1실행코드의 실행횟수와 K×제2실행코드의 실행횟수는 같다'를 만족하는 상수 K가 존재하는가?

즉, 계속 같은 위치에 있는 두 실행코드가 일정한 비율을 가지면서 반복실행 되는지를 확인했다면 암호를 해제하고



(그림 4) 새로운 다형성 바이러스 진단 순서도

있다고 보는 것이 일반적이다. 일정횟수(*Nk*)는 이런 동작이 몇 번 일어나고 있는지를 비교하는 수치이다. 의도적으로 스텝수를 늘렸다고 해도 제1실행코드나 제2실행코드가 등장하는 횟수만 측정하면 된다. 그러므로, 반복구간의 길이(= 스텝수)에 관계없이 일정한 길이(*Nk*)의 해제 영역을 얻을 수 있다. 또한, 일정횟수(*Nk*)의 측정이 시작되면 지정반복횟수(*Nc'*)와 n을 비교하는 동작은 더 이상하지 않기 때문에 지정반복횟수(*Nc'*)와 일정횟수(*Nk*)을 더한 값은 종래의 기술에 명시된 지정반복횟수(*Nc*) 보다 같거나 작다. 그러므로, 이 알고리즘을 이용하면 빠른 진단 속도를 유지하면서 추가적으로 다형성 바이러스를 발견할 수 있다.

4. 실험 및 평가

본 장에서는 운영체제의 종류에는 상관없이 동작할 수 있는 80386 에뮬레이터를 설계 및 구현한 내용을 보이고 있다.

실험을 위하여 대표적인 다형성 바이러스 5가지를 1,000개의 파일에 감염시킨 뒤 제안된 시스템과 국내에서 판매되고 있는 몇 개의 백신 프로그램을 이용하여 바이러스 진단을 실시하여 그 결과를 비교하였다. 연구의 실험을 위하여 사용된 시스템은 인텔 Pentium Ⅲ-700MHz CPU, 128MB RAM, 20GB HDD를 기본으로 구성하는 IBM PC이고, 개발언어로 Assembly와 C를 사용하였다.

4.1 16비트 에뮬레이터 설계 및 구현

프로그램이 실행되려면 CPU와 메모리가 필요한 것처럼 에뮬레이터를 구성하려면 실행 프로그램을 적재하고 실행할 가상의 메모리 공간과 메모리에 적재된 프로그램을 실행시킬 수 있는 디코더(decoder)가 필요하다.

가상 메모리 공간의 크기는 8086 CPU의 전체 메모리 공간인 1MB 만큼을 확보하는 것이 이상적일 수 있으나, 바이러스의 동작 특성을 고려하여 바이러스 코드의 평균적인 크기를 최소 메모리로 확보하면 된다. 그리고, 추가적으로 필요한 메모리는 캐쉬(cache)로 처리하여 메모리 공간을 효율적으로 사용하도록 한다.

4.1.1 가상 메모리 공간 설계

앞서 설명한 바와 같이 8086 CPU 전체 메모리 공간은 1MB이다. 본 에뮬레이터에서는 가상 메모리 개념을 도입, 4KB의 배열을 한 페이지로 하여 모두 16페이지(64KB)를 준비하였다. 그리고, 한 페이지마다 사용횟수를 기록하는 변수를 두어서 메모리 참조가 한번씩 일어날 때마다 1씩 증가하게 하여 나중에 스왑(swap)이 필요할 때 조회 할 수 있도록 하였다. 1MB의 메모리 대신 매핑 테이블을 메모리 참조가 일어날 경우, 먼저 참조되는 주소를 매핑 테이블의 배열 요소로 변환하고 그 위치의 페이지가 메모리에 존재하는지를 판단한다. 존재하면 페이지 번호를 얻고 참조되는 주소에서 페이지 내의 오프셋을 산출하여 해당 페이지의 해당 주소에 접근하게 한다.

이러한 매핑 테이블의 구조를 C언어로 표현하면 아래와 같다.

```
struct Dinfo {
        WORD numCache ;
        BOOL in_memory ;
} MappingTable[16] ;
```

여기에서 numCache는 캐쉬페이지 번호를 나타내고 in_memory는 페이지가 메모리에 있는가의 유무를 기록한다.

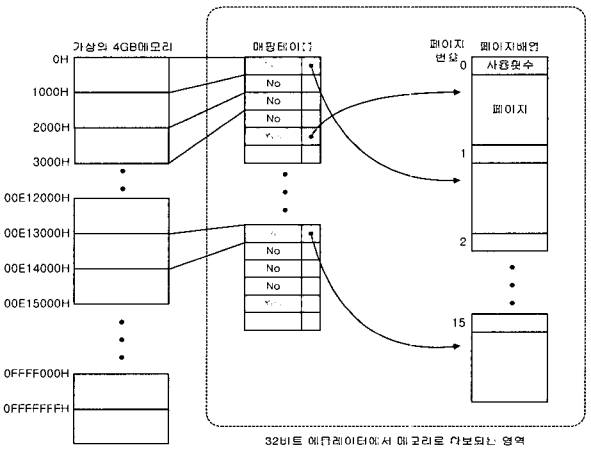
캐쉬페이지의 구조는 아래와 같다.

```
struct Cinfo {
        DWORD usedcount ;
        BYTE memSpace[4096] ;
} BufferCache[16] ;
```

실제 주소가 012345h라면 4096(= 01000h)로 나누어서 배열번호 012345h/01000h = 12h = 18을 얻고, MappingTable[18]에 접근한다. 그리고, MappingTable[18].in_memory가 TRUE 값을 가진다고 가정하면, BufferCache[MappingTable[18].numCache].memSpace 배열에 접근하고 여기서 memSpace 내의 배열요소는 012345h를 01000h로 나눈 나머지가 된다. 즉, memSpace[345h]를 접근하면 되는 것이다.

아래 (그림 5)는 매핑테이블 및 페이지 배열만 에뮬레이터 프로그램 내에 유지하면서 가상의 1MB 공간의 메모리를 접근하는 방식을 그림으로 표현한 것이다. 실행환경에서 메모리 참조가 일어나는 경우는 모두 3가지가 있다.

① CS : IP 주소에 있는 데이터를 CPU 내의 디코더로 가져와 실행하는 경우
② 디코더 내에서 연산수행 시 간접주소지정방식으로 다른 메모리로 참조하게 되는 경우
③ ②와 같은 의미이지만 스택(stack)에 저장 또는 읽어오는 경우

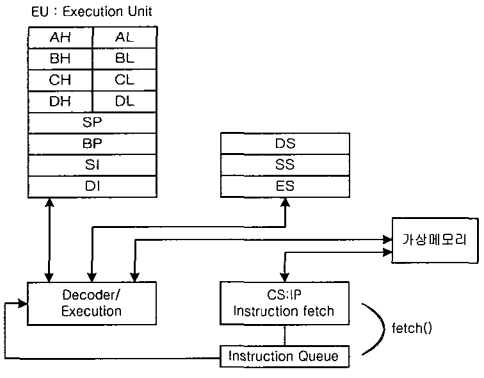


(그림 5) 에뮬레이터용 가상 메모리

4.1.2 에뮬레이터 설계

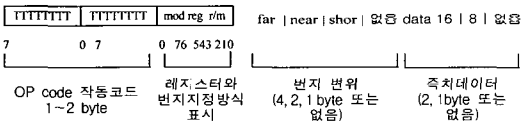
가. 에뮬레이터 구조

에뮬레이터의 구조는 CPU의 구성방식을 본 따서 설계하였다. 버스 대신 주어진 주소에 대하여 해당 페이지 내의 메모리를 매핑 시켜주는 가상메모리 처리기를 두었다. 가상메모리의 구조는 앞에서 설명한 바 있다. 가상메모리로부터 명령어를 가져와서 큐에 적재하는 fetch() 함수를 두었다. fetch() 함수 실행의 결과로 명령어가 Instruction Queue에 적재되면 디코더가 이를 입력으로 받아 명령을 수행한다.



(그림 6) 에뮬레이터의 구조

실행 프로그램은 기계어로 표현되어 있고 이진수로 구성되어 있다. 입력되는 이진값들은 OP Code의 종류, Operand 종류 및 크기의 정보가 들어있다. 인텔 8086의 인스트럭션 포맷은 아래 (그림 7)과 같다.



(그림 7) OP Code의 구조

명령어 스트링의 첫 번째 바이트에서 명령어의 종류가 결정됨을 알 수 있다. 그래서, 각각의 기능을 함수로 정의하고 이 함수들의 포인터 배열을 만든 다음, 명령어 큐의 첫 번째 바이트의 값을 배열의 인덱스로 처리한다. 그러면, 입력된 OP Code에 대하여 비교, 판단을 거치지 않고 해당되는 함수를 바로 수행할 수가 있다.

각각의 명령어를 수행하는 함수를 동일한 파라미터 구조로 만든다. 여기서 ipstr은 명령어 큐의 맨 꼭대기 주소이다. 그리고, 명령 수행 후 디코딩에 사용된 인스트럭션의 크기를 정하고 그 값을 isize에 돌려주어 다음 실행주소(CS : IP)를 결정하는 데 사용한다.

```
int decodeADD_0000000b(BYTE *ipstr)
        int isize ;
        isize = ADD 명령수행() ;
        return isize ; // 다음 실행주소 결정
}
```

```
int decodeADD_0000000w(BYTE * ipstr)
        int isize ;
        isize = ADD 명령수행() ;
        return isize ; // 다음 실행주소 결정
}
.
.
.
```

이렇게 작성된 각 함수들의 함수 포인터를 저장하는 구조를 아래와 같이 만들고 함수 포인터의 배열을 구성한다.

```
typedef struct iset {
VRINT (* Operation)(VRBYTE * ipstr) ;
} IsetList ;

IsetList IsetLists[ ] = {
{ decodeADD_0000000b },          /* 00 */
{ decodeADD_0000000w },          /* 01 */
{ decodeADD_0000001b },          /* 02 */
{ decodeADD_0000001w },          /* 03 */
{ decodeADD_accb },              /* 04 */
.
.
.
{ decodeSTD },                   /* FD */
{ decodeFE_b },                  /* FE */
{ decodeFF_w },                  /* FF */
}
```

에뮬레이터를 수행하는 방법을 C언어로 표현하면 아래와 같다.

```
while(1)
{
   opcode = fetchcode() ;
   // 명령어 큐에서 실행할 명령을 가져온다
   isize = IsetLists [ * opcode ].Operation(opcode) ;
   // * opcode는 맨 앞 OP code이므로
   // 이를 함수배열의 인덱스로 사용한다
}
```

4.2 32비트 에뮬레이터 설계 및 구현

4.2.1 80386 개요

인텔의 80386 프로세서는 32비트 프로세서이다.

현재 Pentium 4 마이크로프로세서까지 나와있는 상태이지만 80386의 아키텍처를 기본 골격으로 하고 있다. 이 장에서는 32비트 에뮬레이터 설계 관점에서 8086 CPU와의 차이점을 설명한다.

80386에서 적용된 레지스터 군은 기존의 16비트에서 32비트로 확장되었다. 1MB의 메모리 공간을 세그먼트 오프셋 방식으로 접근하던 16비트 환경과는 달리 32비트 선형 메모리 공간을 그대로 사용할 수 있어 규모가 큰 어플리케이션도 쉽게 만들 수 있게 설계되었다. 실행 프로그램의 크기는 도스환경에서 640KB 이하였던데 반해 그 크기의 제한이 없어졌으므로 설계시에 이를 반영하였다.

4.2.2 32비트 에뮬레이터 설계 및 구현

16비트 환경의 메모리 공간은 1MB이다. 32비트에서는 0~0FFFFFFFFH 구간의 32비트 선형번지를 사용하기 때문에 4GB의 방대한 메모리를 사용할 수가 있다. 16비트 에뮬레이터의 가상메모리 환경인 경우, 매핑테이블의 배열 크기는 최대 256개인데 반해 32비트인 경우 1,048,576개의 배열 요소를 가진 매핑테이블이 필요하다. 따라서 실행파일의 크기에 따라 매핑테이블의 개수 및 크기를 가변적으로 정의하였다.

매핑 테이블의 구조를 C언어로 표현하면 (그림 8)과 같다.

```
typedef struct Dinfo MEMORYCACHE
struct Dinfo {
        WORD numCache ;
        BOOL in_memory ;
} ;

typedef struct Minfo MEMORYMAP
struct Minfo {
        DWORD fromADDR, toADDR ;
        MEMORYCACHE map[256] ;
} ;

MEMORYMAP * mmap[12] ;
```

(그림 8) 매핑 테이블의 자료구조

여기서 fromADDR과 toADDR은 할당된 map이 어느 주소영역인지를 기록하고 numCache는 캐쉬페이지번호를, 그리고 in_memory는 페이지가 메모리에 있는가의 유무를 기록한다. 캐쉬페이지의 구조는 아래와 같다.

```
struct Cinfo {
        DWORD usedcount ;
        BYTE memSpace[4096] ;
} BufferCache[16] ;
```

BufferCache의 크기가 16으로 되어 있지만 상황에 따라 더 늘릴 수 있다.

4.3 실험 결과 및 평가

실험을 위하여 다형성 바이러스 5가지를 EXE파일 500개와 COM파일 500개에 감염시켜보았다. 다형성 바이러스의 특성상 감염될 때마다 암호를 푸는 코드가 변형된다. 즉, 하나의 바이러스가 1,000개의 바이러스 형태를 띄게 되는 것이다. 이 1,000개의 파일을 제안된 알고리즘이 사용된 진단 시스템과 국내에서 판매되고 있는 몇 개의 백신 소프트웨어를 이용하여 진단한 결과, <표 2>와 같은 결과를 얻을 수 있었다.

〈표 2〉 제안된 시스템의 성능분석

구 분 / 바이러스종류	A시스템	B시스템	Proposed System
Gifafe.2998 # TPE (Coffeshop.II)	901/1000	1000/1000	1000/1000
Natas.4746 (Natas.4744)	1000/1000	1000/1000	1000/1000
DSCE.Demo	1000/1000	1000/1000	1000/1000
FCL	988/1000	998/1000	1000/1000
Maltest_Amoeba	1000/1000	1000/1000	1000/1000
Rate	97%	99%	100%

그리고, 본 논문에서 제안한 에뮬레이터는 어느 플랫폼에서도 종류에 상관없이 동작할 수 있다는 장점이 있고, 기존의 에뮬레이터에 비해 다형성 바이러스 진단율이 약 2% 정도 향상됨을 알 수 있다.

바이러스와 같은 악성 프로그램의 경우 단 한개라도 놓치게 된다면 그 피해는 기하급수적으로 늘어나게 된다. 그러므로, 진단율을 조금이라도 향상시켜 바이러스 확산을 막는 것이 필수적이다.

5. 결론 및 향후 연구과제

본 논문에서는 바이러스 제작자가 의도적으로 바이러스 암호 해제 코드의 반복 실행 수를 늘려 놓았다 하더라도 다형성 바이러스를 탐지해 내는 새로운 알고리즘을 이용한 에뮬레이터를 제안하였다. 이를 이용함으로써 플랫폼에 구애받지 않으면서도 더욱 빠르게 다형성 바이러스들을 탐지해낼 수 있다. 하지만, 날이 갈수록 고급기술에 의한 다형성 바이러스 제작이 늘어가는 현실 속에서 바이러스 창궐 후 분석하고 대처 방안을 세우기에는 시간적으로나 기술적으로 한계가 많다. 그래서, 향후에는 시스템 레벨에서 바이러스의 악성 행위를 감시하고 진단 및 치료할 수 있는 시스템에 대한 연구가 필요하다.

참 고 문 헌

[1] Alan Solomon, "Mechanisms of Stealth," Proc. 5th Int. Comp. Virus and Sec. Conf., New York, pp.232-238, March, 1992.

[2] Andy Nikishin, Mike Pavluschick, "pOLEmorphism," Virus Bulletin, pp.14-15, June, 1999.

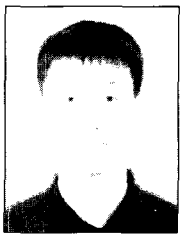
[3] Eugene Kaspersky, Vadim Bogdanov, "Strange-A New Way to Hide," Virus Bulletin, pp.12-13, April, 1993.

[4] Jan Hruska, *Computer Viruses and Anti-virus Warefare*, Ellis Horwood, 1990.

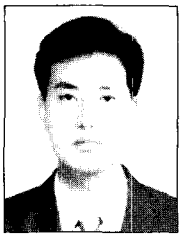
[5] Koray Oner, Luiz A. Barroso, Sasan Iman, Jaeheon Jeong, Krishnan Ramamurthy and Michel Dubois, "The design of RPM : an FPGA-based multiprocessor emulator," Proc. of the third International ACM symposium on Field-programmable gate arrays, pp.60-66, 1995.

[6] Marko Helenius, "Automatic and Controlled Virus Code Execution System," Proc. of eicar Conference'95, EICAR, 1995.

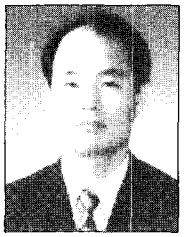
[7] Peter Szor, "Attacks on Win32 - Part II," Proc. of VB2000, pp.111, 2000.
[8] Vesselin Bontchev, "Future Trends in Virus Writing," Anti-Virus Papers Online, Virus Bulletin, 2001.
[9] 권석철, 주영흠, 김판구, "컴퓨터 바이러스 완전소탕", 크라운출판사, 1997.
[10] 안철수, "바이러스 분석과 백신 제작", 정보시대, 1995.



김 두 현
e-mail : mindul@mina.chosun.ac.kr
1999년 조선대학교 전자계산학과(이학사)
1999년~2002년 조선대학교 전자계산학과 (이학석사)
관심분야 : 네트워크 보안, 시스템 보안, 컴퓨터 바이러스



백 동 현
e-mail : dhpaek@hauri.co.kr
1996년 숭실대학교 인공지능학과(공학사)
1998년~현재 (주)하우리 기술연구소 소장
관심분야 : 컴퓨터 바이러스, PC보안



김 판 구
e-mail : pkkim@mina.chosun.ac.kr
1998년 조선대학교 컴퓨터공학과(공학사)
1990년 서울대학교 컴퓨터공학과(공학석사)
1994년 서울대학교 컴퓨터공학과(공학박사)
1995년~현재 조선대학교 컴퓨터공학부 부교수
관심분야 : 시스템 보안, 운영체제, 정보검색, 영상처리