개념 그래프 기반의 효율적인 악성 코드 탐지 기법

김 성 석[†] · 최 준 호[†] · 배 용 근[††] · 김 판 구[†††]

요 약

현재까지 존재하는 무수한 악성 행위에 대응하기 위해서 다양한 기법들이 제안되었다. 그러나 현존하는 악성행위 탐지 기법들은 기존의 행위에 대한 변종들과 새로운 형태의 악성행위에 대해서 적시적절하게 대응하지 못하였고 긍정 오류(false positive)와 틀린 부정(negative false) 등을 해결하지 못한 한계점을 가지고 있다. 위와 같은 문제점을 개선하고자 한다. 여기서는 소스코드의 기본 단위(token)들을 개념화하여 악성행위 탐지에 응용하고자 한다. 악성 코드를 개념 그래프로 정의할 수 있고, 정의된 그래프를 통하여 정규화 표현으로 바꿔서 코드 내 악성행위 유사관계를 비교할 수 있다. 따라서 본 논문에서는, 소스코드를 개념 그래프화하는 방법을 제시하며, 정확한 악성행위 판별을 위한 유사도 측정 방안을 제시한다. 실험결과, 향상된 악성 코드 탐지율을 얻었다.

키워드 : 악성 코드 탐지, 개념 그래프 유사도

A Method for Efficient Malicious Code Detection based on the Conceptual Graphs

Sung-Suk Kim[†] · Jun-Ho Choi[†] · Young-Geon Bae[††] · Pan-Koo Kim[†††]

ABSTRACT

Nowadays, a lot of techniques have been applied for the detection of malicious behavior. However, the current techniques taken into practice are facing with the challenge of much variations of the original malicious behavior, and it is impossible to respond the new forms of behavior appropriately and timely. There are also some limitations can not be solved, such as the error affirmation (positive false) and mistaken obliquity (negative false). With the questions above, we suggest a new method here to improve the current situation. To detect the malicious code, we put forward dealing with the basic source code units through the conceptual graph. Basically, we use conceptual graph to define malicious behavior, and then we are able to compare the similarity relations of the malicious behavior by testing the formalized values which generated by the predefined graphs in the code. In this paper, we show how to make a conceptual graph and propose an efficient method for similarity measure to discern the malicious behavior. As a result of our experiment, we can get more efficient detection rate.

Key Words : Malicious Code Detection, Conceptual Graphs Similarity

1. 서 론

인터넷 사용이 증가함에 따라 인터넷 서비스를 이용한 악성 코드가 확산되고 있다. 악성 코드는 파일 감염을 통해 전파되는 바이러스가 일반적인 형태였지만, 최근에는 인터넷 웜(worm)과 같은 스크립트 형태의 악성 코드가 나타나고 있으며, 복잡하고 지능적으로 진화하고 있다.

비정상적인 동작, 정보 유출, 또는 분산 서비스 공격 등의 악영향을 일으키기 위한 목적으로 작성된 악성 코드 중 특히 VBScript로 작성된 윈도우 운영체제 기반의 악성 스크립트가 증가하고 있다[20].

현재 VBScript의 악성 행위 탐지를 위해 스크립트 소스 코드의 특정 문자열을 이용한 시그니쳐(signature) 기반의 스캐닝 방법과 휴리스틱(heuristic) 분석 기법이 널리 사용되고 있다[20].

시그니쳐 기반의 스캐닝 기법은 기존 악성 코드를 분석하여 악성 패턴에 해당되는 시그니쳐를 추출한 후 이를 악성 코드 탐지에 활용하고 있다. 하지만 알려지지 않은 새로운 악성 코드나 구조적으로 변형된 소스 코드에 대해서는 악성 행위를 탐지할 수 없는 단점을 가지고 있기 때문에 알려지지 않은 악성 행위에 대응하기 위해서 휴리스틱 분석 기법을 이용하는 것이 일반적이다[4].

휴리스틱 분석 기법 중에 정적 휴리스틱 분석은 악성 행위를 위해 자주 사용되는 메소드 또는 내장 함수 호출들을

※ 이 논문은 2004년도 조선대학교 연구 보조비 지원에 의하여 연구되었음.
† 준 회 원 : 조선대학교 대학원 전자계산학과
†† 정 회 원 : 조선대학교 컴퓨터공학부 교수
††† 종신회원 : 조선대학교 컴퓨터공학부 교수, 교신저자
논문접수 : 2005년 8월 18일, 심사완료 : 2006년 1월 26일

데이터베이스화 하여두고 대상 스크립트를 스캔하여 일정 수 이상의 위험한 호출 나타나면 이것을 악성 스크립트로 간주하는 방식이다. 이는 비교적 빠른 속도와 높은 탐지율을 보이긴 하지만 정상 행위를 악성으로 탐지하는 긍정 오류가 상당히 높다는 단점을 가지고 있다[5, 14].

일반적인 탐지 기법들은 소스 코드 형태로 존재하는 악성 스크립트에서 정형화된 코드 블록을 찾아내는 데에는 많은 어려움이 있다. 따라서 소스 코드의 개념적인 분석을 통하여 의미를 파악하고 그 개념들 간의 연관관계를 이용하여 동적인 정보를 이해할 수 있는 방법이 필요하다.

본 논문에서는 스크립트 소스 코드인 VBScript의 구문 분석을 통해 개념(concept)과 관계(relation)를 정의하고, 개념 그래프로 표현한 후, 이를 악성 코드 탐지에 적용하기 위해 개념 그래프들 간의 유사도 측정식을 제안한다.

2. 관련 연구

2.1 기존 악성 코드 탐지 기법

악성 여부를 판별하는 대응 기법에는 크게 코드 분석 방법과 행위 분석 방법으로 나뉠 수 있는데 코드 분석 방법에서는 이진 코드를 위한 기법들을 그대로 이용하거나, 소스 프로그램 형태인 스크립트에 적합하도록 다소간의 변형을 가한 기법을 적용하는 것이 일반적이다.

알려지지 않은 악성 코드를 탐지하기 위한 방법론들은 대부분 행위 분석 방법을 취하고 있다. 일반적인 악성 코드 탐지 기법들을 살펴보면 시그니쳐 탐지 기법, 휴리스틱 분석 기법, 무결성 검사 기법 등이 있는데, 구체적인 내용은 다음과 같다[3].

시그니쳐 탐지 기법은 사람들을 구분하기 위해 지문을 사용하여 구분하듯이 악성 코드가 가지고 있는 독특한 문자열인 '시그니쳐(혹은 패턴)'를 이용하여 악성 코드를 탐지하는 방법이다. 이 기법은 악성 코드의 패턴을 많이 확보할수록 보다 많은 악성 코드를 탐지할 확률이 높아진다.

시그니쳐 탐지 기법을 수행하기 위한 방법은 악성 행위의 실행 코드에 대해 코드 처음부터 끝까지 실행 코드를 따라가면서 진단하는 순차적 문자열 검사법과 파일을 전체 혹은 일부만을 검색해 특정 문자가 있는지 검사하는 특정 문자열 검사 방법으로 구분된다.

먼저, 순차적 문자열 검사법은 검색 속도가 빠르다는 장점이 있다. 하지만 변형 바이러스를 탐지하는데 있어서 특정 문자열 검사법보다 떨어진다는 단점을 가지고 있다.

반면에 특정 문자열 검사법은 대부분의 안티 바이러스 프로그램에서 사용하는 방식으로 검색 속도는 다소 떨어지지만 변형 악성 코드를 쉽게 파악할 수 있는 장점이 있다. 하지만 이 기법은 정상 행위를 악성으로 탐지하는 긍정 오류가 상당히 높다는 단점을 가지고 있다[2, 15].

휴리스틱 분석은 정적 분석과 동적 분석이 있는데, 정적 휴리스틱 분석은 알려지지 않은 악성 코드를 감지하는데 널리 사용되는 기법이다. 특정 시그니쳐만을 찾는 방법 대신에 보통의 응용 프로그램에서 발견할 수 없는 어떤 특별한 명령이나 행위를 찾는다.

정적 휴리스틱 분석 기법을 통하여 각 메소드 호출의 리턴 값과 파라미터 또는 문장 간의 관계를 고려하는 세밀한 분석을 통하여 악성 행위를 하는 방법이다.

그러나 실행 전에 수행되는 정적 분석의 한계로 인해 그 값이 실행 중에 일치함을 결정할 수 없는 데이터가 존재할 경우 틀린 부정을 발생하게 된다[5].

동적 휴리스틱 분석은 가상 기계를 구현한 에뮬레이터 상에서 해당 코드를 수행하면서 프로그램 수행 중에 발생하는 시스템 호출과 시스템 자원들에 발생하는 변화를 감시함으로써 악성 행위를 탐지하는 방법이다[4].

그러나 이를 위해서는 완전한 가상 기계를 구현이 필요하다. 가상 기계는 하드웨어, 운영체제 뿐 아니라 관련된 시스템 객체 및 제반 환경을 모두 포함하여야 하므로 구현이 매우 어렵고, 부하 또한 큰 것으로 알려져 있다[14].

무결성 검사 방식은 로컬 디스크에 존재하는 파일들 전체 또는 일부에 대하여 파일 정보 및 체크섬(checksum), 또는 해쉬 값을 기록하여 두었다가 일정 시간이 지난 후 파일들이 변형되었는가를 기준으로 탐지한다. 하지만 지정된 파일의 변형을 감지하는 방식이므로 적법한 내용의 변화가 예상되는 파일에 사용할 경우 높은 긍정 오류를 발생시킨다는 단점을 가지고 있다[5, 20].

일반적인 악성 코드 대응 기법들을 비교하여 보면 <표 1>과 같은 내용으로 분석된다.

〈표 1〉 악성 코드 대응 기법 비교 분석

		검색 속도	긍정 오류	틀린 부정
시그니쳐 탐지 기법	순차 문자열	빠름	×	○
	특정 문자열	보통	○	×
휴리스틱 분석 기법	정적	보통	×	○
	동적	느림	×	×
무결성		빠름	○	×

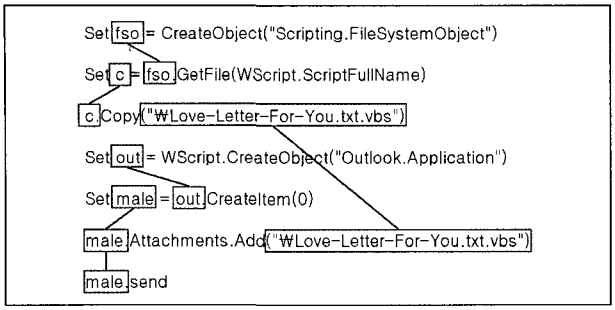
2.2 정적 분석을 이용한 악성 코트 탐지

정적 분석 기법은 스크립트 형태의 특수성을 고려하여 메소드 호출 또는 애트리뷰트와 같은 특정 단어들의 존재나 출현 빈도를 이용하여 탐지하는 방법이다.

그러나 악성 행위에 사용되는 메소드들은 실제 정상적인 스크립트에도 빈번하게 사용되기 때문에, 단순한 존재 유무를 통한 판단만으로는 높은 긍정 오류를 발생시킨다. 이러한 기존의 정적 분석 기법의 단점을 (그림 1)과 방법으로 극복하였다.

각각의 위험한 메소드 호출이 아니라 악성 행위를 구성하는 메소드 시퀀스들을 정의하고 악성 행위를 정확하게 감지함으로써 극복될 수 있다. 그러나 앞에서 상술한 바와 같이 실행 전에 수행되는 정적 분석의 한계로 인하여 틀린 부정의 문제점을 가지고 있다[5].

악성 코드 탐지 기법들은 각각의 특성에 맞는 접근 방법을 통하여 악성 코드를 탐지한다. 하지만 이러한 기법들은



(그림 1) 정적 분석 기법 통한 탐지 예

대량의 패턴을 보유하더라도 약간의 데이터 변형이나 새로운 형태의 악성 코드에 대해서는 탐지할 수 없는 문제점을 가지고 있다.

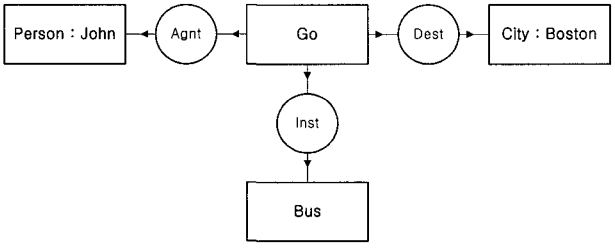
본 논문은 알려지지 않은 악성 행위를 탐지하기 위해 악성 행위를 구성하는 소스 코드에 대한 개념과 관계 정의를 통하여 탐지하는 대상의 데이터가 어떠한 의미인지를 의해하고 악성 행위의 개념을 정확하게 감지함으로써 새로운 행위에 대하여 지능적인 탐지를 할 수 있다.

이를 위해서 스크립트 소스 코드의 기본 단위를 개념적으로 분류하여 각 요소들의 개념과 관계들을 정의하여 개념 그래프로 표현하고 개념 그래프간 유사도 측정식을 이용하여 악성 코드를 탐지 기법을 제시한다.

2.3 개념 그래프

개념 그래프는 여러 의미망(semantic networks)을 통합한 지식 표현 언어로 개념도식을 이용하여 논리적으로 간결하면서 자연어 수준의 표현력을 지니고, 인간이 쉽게 이해할 수 있으며, 컴퓨터에 의한 자연어처리 등에서 쉽게 이용할 수 있는 형태로 의미를 기술할 수 있다[4, 5].

예를 들어, "John is going to Boston by Bus"라는 문장을 개념 그래프로 도식화하면 다음 (그림 1)과 같이 나타낼 수 있다[3, 6, 7, 8].



(그림 2) 개념 그래프

위 (그림 2)에서 직사각형으로 표시된 부분은 개념을 의미하고, 원으로 표시된 부분은 개념간의 관계를 나타낸다. 그리고 각 노드들을 연결하는 지시선이 있다. 'Agnt', 'Dest', 'Inst'는 관계를 의미하고, 'John', 'Boston', 'Bus'는 개념을 나타내고 있다. 'Person : John'의 표현은 'John'이 'Person'이라는 개념의 요소(instance)임을 의미한다. 따라서 위의 그래프는 다음과 같은 의미를 갖는다.

- Go는 사람(Person)인 John을 행위자(Agnt)로 갖는다.
- Go는 도시(City)인 Boston을 목적지(Dest)로 갖는다.
- Go는 버스를 도구(Inst)로 갖는다.

위의 개념 그래프를 선형 표기법으로 표시하면 다음 <표 2>와 같다[8, 17, 18].

<표 2> 개념 그래프의 선형 표기법

```
[Go] -
  (Agnt) → [Person : John]
  (Dest) → [City : Boston]
  (Inst) → [Bus].
```

또한, 개념 그래프는 확장된 BNF 표기법인 CGIF(Conceptual Graph Interchange Format) 형태로 변환할 수 있는데, <표 3>은 (그림 2)의 개념 그래프를 CGIF 형태로 표현한 예이다. 본 논문에서는 개념 그래프로 표현된 악성 코드를 CGIF 형태로 변환하여 유사도를 측정한다[9, 12, 21, 24, 25].

<표 3> 개념 그래프의 CGIF 표현

01:	[City*a:'Boston']	05:	(agent?d?c)
02:	[Bus*b:'']	06:	(dest?d?a)
03:	[Person*c:'John']	07:	(inst?d?b)
04:	[Going*d:'']	08:	

표현된 개념 그래프는 개념간의 유사도 측정이 가능하기 때문에 이를 이용하면 개념 그래프간의 의미적 유사성을 측정 및 비교가 가능하다. 따라서 본 논문에서는 스크립트 소스 코드에 대한 개념 그래프 표현과 개념 유사성을 측정하여 악성 행위를 판별할 수 있는 방법을 제시하고자 한다.

3. 악성 코드의 개념 그래프 표현

3.1 VBScript의 개념과 관계 정의

본 절에서는 VBScript의 구문 및 어휘 분석을 통한 개념화 방법 및 관계 정의에 대해서 기술한다.

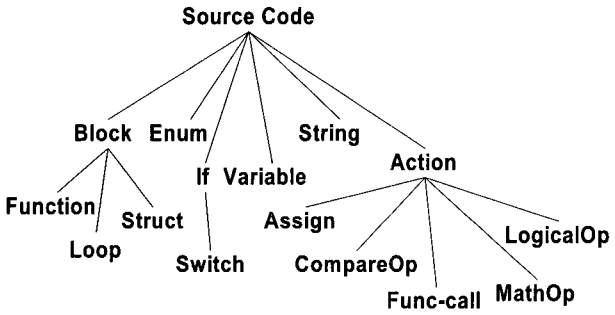
<표 4>는 자기 복제를 수행하고 이를 통하여 다양한 방법으로 복제된 악성 코드를 전파하는 악성 소스 코드의 일부를 보여주고 있다.

<표 4> 자기 복제 악성 VBScript 코드

```
01: On Error Resume Next
02: Set Obj_A = Createobject("scripting.filesystemobject")
03: Obj_A.copyfile wscript.scriptfullname, Obj_A.GetSpecialFolder(0)&
    "\ShakiraPics.jpg.vbs"
04: Set Obj_B = CreateObject("WScript.Shell")
    Obj_B.regwrite "HKLM\SOFTWARE\Microsoft\Windows\Current
    Version\Run\Registry", "wscript.exe"& Obj_A.GetSpecialFolder(0)&
    "\ShakiraPics.jpg.vbs %"
```

VBScript 소스 코드에서 악성 행위를 수행하기 위해 필요한 문법적 요소가 존재하는데, 악성 행위는 특정 메소드와 메소드의 특정 시퀀스로 구성되어짐을 알 수 있다. <표 4>를 살펴보면, '*Createobject*' 함수를 이용하여 자동화 개체에

대한 '*obj_A*' 개체를 생성한 후 '*copyfile*' 메소드를 통하여 지정된 특정 폴더로 자신의 코드 복제를 수행한다. 그리고 지정된 Registry의 시작프로그램 영역에 WSH(Windows Script Host)를 이용하여 쉘 명령어를 실행시킬 수 있도록 기록('*regwrite*')한다. 이와 같은 과정이 실행되면 해당 악성 코드는 E-메일이나 특정 애플리케이션을 통하여 다른 시스템으로 전파된다.



(그림 3) 프로그래밍 언어 구성 요소 계층 구조

VBScript의 개념 그래프 표현을 위해서 우선 소스 코드의 개념과 관계 정의 과정이 필요하다. VBScript와 같은 프로그래밍 소스 코드 내에서의 개념과 관계 정의를 위해서 프로그래밍 언어를 구성하고 있는 요소를 다음 (그림 3)과 같이 계층적으로 분류하였다. 본 논문에서는 각 계층 구조를 이루고 있는 구성 요소를 소스 코드의 개념으로 정의한다.

위에서 분류된 계층 구조를 기반으로 MSDN의 VBScript Language Reference를 참고하여 <표 5>와 같이 개념을 정의하였다. 또한, 소스 코드 내 개념 간의 관계를 정의하면 다음 <표 6>과 같은 관계를 정의할 수 있다[23].

예를 들면, 문법적 개념인 'Procedure'는 {Condition, Argument} 관계와 관련되어 있음을 나타내고 있다[16].

3.2 개념 그래프 표현

VBScript 소스 코드를 개념 그래프로 표현하기 위해서 먼저 원시 소스 내에서 개념과 관계에 해당되는 문법 요소를 추출하기 위한 파싱 과정이 필요하다. 추출된 문법 요소는 정형화된 문법 구문에 따라 개념과 관계로 분류하여 개

〈표 5〉 VBScript 소스 코드의 개념(concept) 정의

개 념		설 명	관련 문법
Procedure		문제를 해결하기 위하여 수행되는 일련의 작업 순서 및 과정	Sub, End Sub 등
Statement	Conditional	주어진 조건에 따라 서로 다른 방향으로 프로그램의 실행을 제어할 수 있도록 사용되는 문장	If…Then..else, Select Case 등
	Loop	주어진 일련의 명령어들을 반복해서 실행할 수 있도록 하는 프로그램의 문장	Do…Loop, While…Wend, For…Next 등
	Error	어떠한 연산이 수행되어야 한다고 예상된 방법으로 동작하지 않고 다른 방법으로 동작하는 것	On Error Resume Next, Error 등
Operator	Comparison	입력으로 전달된 두 개의 자료에 대한 크기를 비교하는 작업	'<', '=', '< >' 등
	Logical	논리 연산자들을 논리 변수에 적용하여 참 또는 거짓이라는 결과를 생성하는 연산	'And', 'Or', 'Xor' 등
	Arithmetic	실수 또는 정수 등과 같은 수치 데이터에 대한 사칙 연산	'+', '-', '/', '*' 등.
	Concatenation	두 문자열을 결합하여 하나로 만드는 연산	'&', '+' 등
Function		명확한 서비스를 수행하도록 지명된 하나의 프로시저	라이브러리 루틴
Assign		프로그램에서 기억 장소에 값을 할당하는 문장	'+=', '=' 등
Procedure-Call		프로시저 호출	
Object		클래스의 인스턴스(Instance)	
Method		클래스에 정의된 함수	
Properties		특정 Object가 지니고 있는 속성 또는 성질	
Arguments		함수를 호출할 때 함수의 작업을 위하여 함수에 전달되는 정보	
String		하나의 자료를 구성하기 위하여 일련의 문자들의 집합으로 구성된 정보	
Variable		프로그램에서 하나의 값을 저장할 수 있는 기억 장소의 이름	

〈표 6〉 VBScript 소스 코드의 관계(relation) 정의

관계	정 의	관계 조건	
		상위 개념	하위 개념
Condition	분기 구문을 위한 조건	Conditional, Loop	Statements, Operator, Assign, Procedure(-Call), String, Variable 등
Contains	다른 개념을 포함하는 개념	*	*
Comment	주석	*	String
Return	반환값을 반환해주는 개념	Function, Method	Function, String, Variable 등

(*: 모든 개념들과 관계 조건을 포함하는 개념들의 집합)

념 그래프로 표현한다. 예를 들면, VBScript의 'IF' 문에 대한 정형화된 구조는 다음 <표 7>과 같이 표현할 수 있다.

〈표 7〉 'IF' 구문 문법 구조

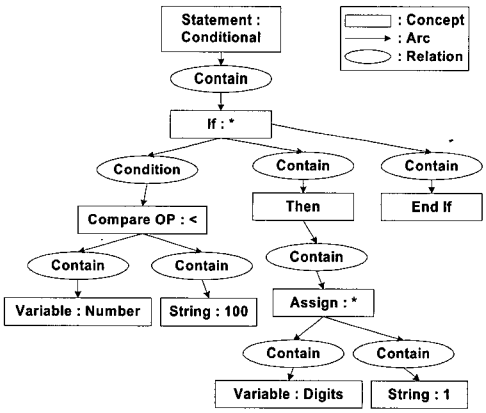
```
statement
→ if "("expr e")" statement s1 then
        concept1 : = create concept(IF)
        connect_concepts(concept1, e, CONDITION)
        connect_concepts(concept1, s1, CONTAINS)
    (elseif statement s2)? then
        connect_concepts(concept1, e, CONDITION)
        connect_concepts(concept1, s2, CONTAINS)
    (else statement s3)?
        connect_concepts(concept1, s3, CONTAINS)
    end if
```

〈표 8〉 'IF' 구문의 예

```
If Number < 100 Then
   Digits = 1
End If
```

<표 8>과 같은 VBS 소스 코드의 'IF' 구문을 본 논문에서 제시한 개념 그래프 표현 방법을 적용하여 도식화하면 다음 (그림 4)와 같다.

(그림 4)에서 소스 코드 내의 조건 분기 개념인 'IF' 구문은 'Parameter', 'Condition', 'Contain' 등의 관계와 연결되어 있음을 알 수 있다.



(그림 4) 'IF' 문의 개념 그래프 표현

3.3 악성 코드 패턴 정의 및 개념 그래프 표현

본 절에서는 악성 행위에 따라 악성 코드의 구문을 추출을 위해 정의된 개념과 관계에 따라 개념 그래프로 표현한다. VBScript로 기술된 악성 코드의 수행은 먼저 객체를 생성하고, 그 객체에 대한 메소드가 호출됨으로써 이뤄진다. 악성 코드 수행과 관련된 윈도우 객체는 다음과 같다.

- Scripting.FileSystemObject
- WScript.Shell
- WScript.Network
- Outlook.Application

이 중에서 'Outlook.Application'은 Outlook이 설치된 시스템에만 존재하고 나머지 객체들은 WSH(Windows Script Host)가 설치되어 있는 시스템에 항상 존재한다.

대표적인 악성 코드인 'Love Letter'의 특징은 윈도우즈 Registry 조작, E-mail을 통한 복제, 특정 파일 삭제, 그리고 악성 HTML 파일 생성 등이다.

이러한 특정 악성 행위를 수행하는 코드를 분석하기 위해 악성 코드에 대한 개념과 관계를 생성하여 이를 개념 그래프로 표현한 후 시스템 적용에 필요한 정규 형식으로 변환한다. 다음 <표 9>의 악성 코드 샘플은 공통적으로 로컬 시스템 폴더에 자기복제를 수행하는 악성 코드들을 나열한 것이다. 이 샘플 코드를 참조하여 악성 코드의 일반적인 패턴을 개념 그래프로 표현한다. (그림 5)는 이를 정형화된 개념 그래프 형식으로 표현한 것이다. 또한, <표 10>은 악성 행위의 개념 그래프 표현에 필요한 개념들과 관계를 분류하기 위해 악성 행위와 관련된 소스 코드를 토큰 단위로 분류하는 과정이다. 'Statements' 개념과 자기 복제를 수행하는 'Method'와 이 메소드를 포함하고 있는 'Arguments'에 대한 개념으로 분류하여 이들 사이의 관계를 이용하면 (그림 5)와 같이 자기 복제 악성 코드를 개념 그래프로 표현할 수 있다.

〈표 9〉 '자기복제' 악성코드 패턴

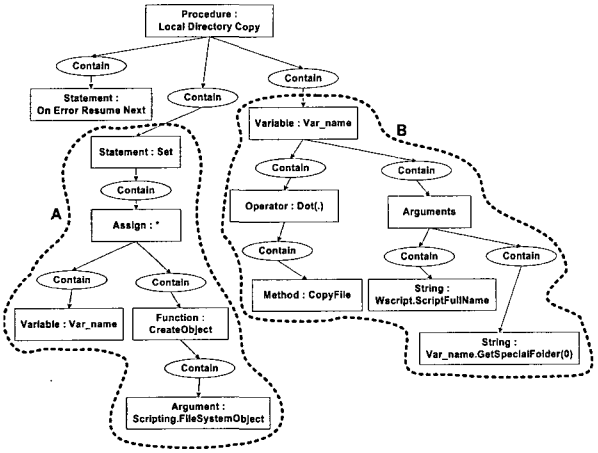
악성코드 Sample 1	On Error Resume Next *Set Obj_A = Createobject("scripting.filesystemobject")* Obj_A.*copyfile wscript.scriptfullname,* Obj_A.*GetSpecialFolder(0)*& -"\xxx.jpg.vbs"
악성코드 Sample 2	main = "c:\www.symantec.com.vbs" *Set maincopy = CreateObject("Scripting.FileSystemObject")* maincopy.*CopyFile WScript.ScriptFullName,* main
악성코드 Sample 3	*Set fso = CreateObject("Scripting.FileSystemObject")* Set dirwin = fso.*GetSpecialFolder(0)* Set dirsystem = fso.*GetSpecialFolder(1)* Set dirtemp = fso.*GetSpecialFolder(2)* Set c = fso.*GetFile(WScript.ScriptFullName)* c.Copy(dirsystem&"\MSKernel32.vbs") c.Copy(dirwin&"\Win32DLL.vbs") c.Copy(dirsystem&"\Very Funny.vbs") …

〈표 10〉 악성 코드의 토큰 분류

Statements	Method Arguments
Set, Obj_A, Createobject, scripting.filesystemobject	ObjA.copyfile, wscript.scriptfullname, Obj_A.GetSpecialFolder(0), \x.jpg.vbs
Set, maincp, Createobject, scripting.filesystemobject	maincp, WScript.ScriptFullName, main, c:\www.symantec.com.vbs
Set, fso, Createobject, scripting.filesystemobject	fso.GetFile, WScript.ScriptFullName, c.copy, dirsystem, \MSKernel32.vbs

(그림 5)는 악성 코드가 자신의 코드를 로컬 시스템에 자기 복제를 수행하는 프로시저를 개념 그래프로 표현한 것으로 악성 행위를 수행하기 위해 개념인 'Statements'(A 영역)와 이를 이용하여 복제를 수행하는 개념인 'Method'(B 영역)을 포함하고 있음을 알 수 있다. 'Statements : Set'을 통하여 악성 행위에 사용할 개념을 생성하고, 이를 'Method :

CopyFile' 개념을 이용하여 악성 코드의 위치와 복제를 수행할 위치를 지정한다. 위와 같은 방법을 이용하여 다양한 형태로 존재하는 악성 코드를 개념 그래프로 표현하면 소스가 변형된 악성 코드나 새로운 악성 코드가 발생하더라도 개념적인 악성 행위의 인식이 가능하다.

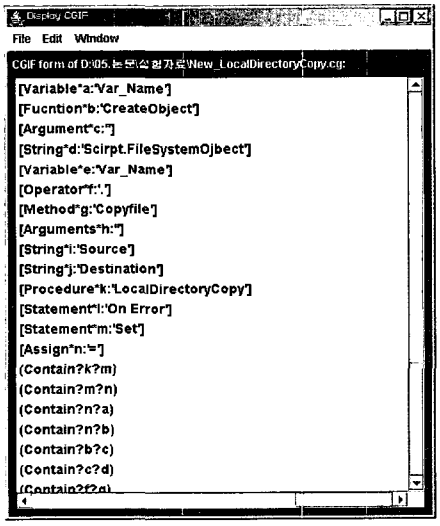


(그림 5) '자기복제' 악성 코드 개념 그래프

4. 악성 코드 탐지를 위한 유사도 측정

4.1 유사도 측정을 위한 악성 코드의 CGIF 변환

개념 그래프로 표현된 악성 코드의 유사도를 측정하기 위해 CGIF로 변환한다. CGIF에서 사용하는 기호에 대한 의미는 다음과 같다. '[ ]' 기호는 개념을 의미하고, 개념형(concept type)은 '*' 기호로 표시하며, 참조대상(referent)은 ' ' 기호로 표시한다. 또한, 개념과 관계의 구분은 ':' 기호를 사용하고, '( )' 기호는 관계를 의미하며, '?' 기호는 관련된 개념 사이의 관계를 표현한다. (그림 6)은 개념 그래프 편집기인 'CharGer v3.4'을 이용하여 (그림 5)의 개념 그래프를 CGIF 형태로 변환한 것이다[10, 11, 21, 22].



(그림 6) '자기 복제' 개념 그래프의 CGIF 표현

4.2 개념 그래프 유사도 측정

VBScript에서 변환된 개념 그래프는 유사도 측정을 통해 악성 코드의 존재 여부 판별과 변형된 악성 코드의 위험률을 수치화할 수 있다. 악성 행위 판별의 정확성을 위해서 개념과 관계 정의 시 악성 행위의 위험 정도에 따른 가중치를 부여하는 것이 바람직하다. 따라서 본 논문에서는 악성 코드의 특징을 고려하여 개념 그래프의 문법적 요소에 대한 가중치를 부여한다. 악성 코드의 특징에 따른 가중치를 고려하지 않고 유사도를 측정하면 정상적인 코드를 악성 코드로 인식하는 긍정 오류가 발생할 가능성이 높다. 또한, 악성 행위에 해당되는 VBScript 코드에서 빈번하게 사용되는 문법 요소 순으로 높은 가중치를 부여하여 악성 코드의 유사도 측정에 활용한다[1, 13, 16].

유사도 측정식을 유도하기 위해서 3단계의 과정을 거친다. 먼저 첫 번째 단계에서는 개념 그래프의 구성 요소인 개념형과 참조대상에 대한 값을 측정하고, 두 번째 단계에서 기본적인 개념 그래프의 유사도를 유도한 후, 세 번째 단계에서는 정의된 개념과 참조대상, 그리고 관계에 적합한 최적의 가중치를 적용하여 악성 행위에 필수적으로 나타나는 개념들의 빈도수에 따라 가중치가 부여된다. 이러한 과정을 토대로 유사도 측정식을 유도하여 탐지에 활용하였다. 이를 단계별로 살펴보면 다음과 같다.

[단계 1] VBScript의 개념 그래프 요소 중 개념형과 참조대상에 대한 관계식을 정의한다. 먼저 개념의 구성 요소인 개념형과 참조대상을 다음과 같은 기호로 정의한다.

$$c_1^t : \text{개념형},\ c_1^r : \text{참조대상} \qquad \text{(정의 1)}$$

하나의 개념인 c_1은 c_1^t, c_1^r의 두 구성요소를 값을 가지게 되는데, 개념형 c_1^t는 <표 11>을 토대로 개념의 중요도에 따라 부여되는 값이고, 참조대상 c_1^r은 개념을 포함하고 있는 노드를 의미한다.

〈표 11〉 개념 중요도에 대한 우선순위

순위	개념	순위	개념
1	Procedure	7	Assign
2	Procedure-Call	8	Operator
3	Function	9	Properties
4	Object	10	Arguments
5	Method	11	Variable
6	Statement	12	String

따라서 개념 c_1은 다음 (식 1)과 같이 개념형과 참조대상을 기반으로 다음과 같이 표현할 수 있다.

$$c_1 = c_1^t \times c_1^r \qquad \text{(식 1)}$$

[단계 2] 개념 c_1와 c_2 사이의 개념형 유사도와 참조대상 유사도는 단계 1에서 제시한 개념 c를 이용하여 다음 (식 2)와 같이 정의하였다.

$$\text{개념형 유사도} : sim(c_1^t, c_2^t) = c_1^t \cdot c_2^t$$
$$\text{참조대상 유사도} : sim(c_1^r, c_2^r) = c_1^r \cdot c_2^r \quad \text{(식 2)}$$

(식 2)를 참조하여 두 개념 그래프 G_1, G_2사이의 전체 유사도는 (식 3)과 같이 정의할 수 있다.

$$sim(G_1, G_2) = \sum_{i=1}^{n} \sum_{j=1}^{m} sim(c_i^t, c_j^t) \cdot sim(c_i^r, c_j^r) \cdot (c_i) \cdot (c_j) \quad \text{(식 3)}$$

개념 그래프 G_1에 포함된 개념 집합을 $\{c_1, c_2, \ldots, c_n\}$, G_2가 포함된 개념 집합을 $\{c_1, c_2, \ldots, c_m\}$이라고 할 때 각 개념들의 요소인 개념형 유사도와 참조대상 유사도를 곱한 값에 각 개념들의 값을 곱해 준다. (식 2)를 토대로 개념 그래프의 개념 간 유사도를 측정해보면 다음과 같은 과정을 통해 값이 유도된다.

〈표 12〉 개념형과 참조대상간의 유사도

G_1		G_2	
개념 : 참조대상	값	개념 : 참조대상	값
[Loop *a : '*']	1	[Loop *a : '*']	1
[Block *b : '*']	0.5	[String *b : 'terminate']	0.9
[String *c : 'kill']	0.45		
$sim(c_i^t, c_j^t)$		$sim(c_i^r, c_j^r)$	
개념형 : 개념형	값	참조대상 : 참조대상	값
sim(Loop, Loop)	1	sim(*, *)	1
sim(Loop, String)	0.1	sim(*, terminate)	0.5
sim(Block, Loop)	0.1	sim(*, *)	1
sim(Block, String)	0.1	sim(*, terminate)	0.5
sim(String, Loop)	0.1	sim(kill, *)	0.5
sim(String, String)	1	sim(kill, terminate)	0.9

따라서 <표 12>에서 계산된 개념형과 참조대상 사이의 유사도 측정값을 기반으로 두 그래프 G_1, G_2 사이의 유사도는 다음과 같이 계산할 수 있다.

sim(G_1, G_2)
= *sim*([Loop:*],[Loop:*]) · [Loop:*] · [Loop:*]
+ *sim*([Loop:*],[String:sleep]) · [Loop:*] · [String:sleep]
+ *sim*([Block:*],[Loop:*]) · [Block:*] · [Loop:*]
+ *sim*([Block:*],[String:sleep]) · [Block:*] · [String:sleep]
+ *sim*([String:wait],[Loop:*]) · [String:wait] · [Loop:*]
+ *sim*([String:wait],[String:sleep]) · [String:wait] · [String:sleep]
= 1.5045

[단계 3] 단계 2에서 유도된 측정값은 일반적인 개념 그래프를 대상으로 하였음으로 본 논문에서 제시한 악성 코드에 대한 유사도 측정은 악성 코드와 관련된 개념의 빈도수를 고려하여 악성 코드에서 가장 빈번하게 나타나는 개념에 대해 가중치를 부여한다. 따라서 악성 코드에서 주요하게 사용되는 개념 사이의 관계를 고려하여 최종적으로 악성 코드에 적용할 수 있는 유사도 측정값을 유도한다.

$$w(r) : \text{관계 가중치} \quad \text{(정의 2)}$$

유사도 측정 시 사용되는 관계 가중치는 악성 코드 에서 악성 행위와 밀접하게 관련된 개념과 관계의 중요도를 나타낸다. 각각의 개념 사이의 관계가 같더라도 관계 가중치에 따라 악성 코드의 측정값은 정상적인 코드에 비해 높게 나타날 확률이 크다. 단계 2에서 유도된 (식 3)은 다음 <표 13>에서 제시된 관계 가중치를 고려하여 악성 코드 개념 그래프 G_1, G_2에 대한 유사도 측정이 이루어진다. 이를 이용한 최종적인 유사도 측정식은 다음 (식 4)와 같다.

$$fsim(G_1, G_2) = \sum_{i=1}^{n} \sum_{j=1}^{m} sim(c_i^t, c_j^t) \cdot sim(c_i^r, c_j^r) \cdot w(r_i) \cdot (c_i) \cdot w(r_j) \cdot (c_j) \quad \text{(식 4)}$$

〈표 13〉 관계 가중치를 이용한 유사도

G_1		G_2	
개념 : 참조대상	가중치	개념 : 참조대상	가중치
[Loop *a : '*']	1	[Loop *a : '*']	1
[Block *b : '*']	0.5	[String *b : 'terminate']	0.9
[String *c : 'kill']	0.45		
관계	가중치	관계	가중치
(Contain?a?b)	0.75	*(Contain?a?b)*	0.95
(Contain?b?c)	0.475		

(식 4)에서 제시된 최종식을 기반으로 두 그래프 G_1, G_2 사이의 악성행위에 대한 유사도 측정값은 다음과 같이 계산할 수 있다.

fsim(G_1, G_2)
= sim([Loop:*],[Loop:*]) · [Loop:*] · [Loop:*] · (Contain?a?b) · *(Contain?a?b)*
+ sim([Loop:*],[String:sleep]) · [Loop:*] · [String:sleep] · (Contain?a?b) · *(Contain?a?b)*
+ sim([Block:*],[Loop:*]) · [Block:*] · [Loop:*] · (Contain?a?b) · (Contain?b?c) · *(Contain?a?b)*
+ sim([Block:*],[String:sleep]) · [Block:*] · [String:sleep] · (Contain?a?b) · (Contain?b?c) · *(Contain?a?b)*
+ sim([String:wait],[Loop:*]) · [String:wait] · [Loop:*] · (Contain?b?c) · *(Contain?a?b)*
+ sim([String:wait],[String:sleep]) · [String:wait] · [String:sleep] · (Contain?b?c) · *(Contain?a?b)*
= 0.9437

두 개념 그래프 G_1, G_2가 동일한 그래프라면 측정값은 '1'로 나타나고, 반대로 유사한 부분이 없는 경우에는 '0' 값이 출력된다. 또한 단계 3에서 제시된 최종식과 같이 관계 가중치를 고려하여 측정하면 악성 코드의 특징이 반영된 악성 코드 유사도 측정이 이루어짐을 알 수 있다.

5. 실험 및 평가

본 장에서는 개념 그래프를 이용한 악성 코드 유사도 측정식을 이용하여 악성 코드를 탐지하는 방법에 대한 실험 및 평가를 기술한다.

실험에 사용된 총 150개의 샘플들은 MS Windows의

'Outlook.Application' 개체를 이용하여 주소록 참조 및 메일 전송을 수행하는 악성 코드를 탐지하기 위해서 제작 및 수집하였다.

현재 인터넷에 유포되고 있는 악성 코드들과 자체적으로 수집한 악성 코드 샘플들 100개, 알려지지 않은 형태의 악성 코드에 대한 샘플 30개는 'VBScript Worm Generator' 프로그램을 이용하여 제작하고 특정한 프로그램이 아닌 자체적으로 제작한 샘플들로 구성되었다. 그리고 일반적인 행위를 하는 샘플들 20개를 실험에 활용하였다.

제안하는 기법과 현재 상용화된 프로그램들과의 기본적인 탐지율을 평가하기 위해서 알려진 악성 행위에 대한 샘플들을 'A 그룹'으로 그룹화 하였다.

틀린 부정 즉, 악성 코드임에도 불구하고 탐지하지 못하는 문제점에 대한 실험을 위하여 알려지지 않은 악성 코드 샘플들을 'B 그룹'으로 그룹화 하였다.

마지막으로 긍정 오류에 대한 문제점을 실험하기 위하여 악성 행위에 빈번하게 사용되는 코드를 이용하여 실제로는 악성 행위를 수행하지 않는 정상적인 주소록 참조 기능이 포함된 샘플들을 'C 그룹'으로 나누어 비교·분석하였다.

〈표 14〉 E-mail을 이용한 자기 복제 악성 코드 CGIF

01: [String*a:'0']	24: [String*x:'i']
02: [Statement*b:'Set']	25: [Statement*y:'Set']
03: [Variable*c:'NoteItem']	26: [Varialbe*z:'ObjApp']
04: [Object*d:'ObjApp']	27: [String*aa:'Outlook']
05: [ReferenceOP*e:'.']	28: [Function*ab:'CreateObject']
06: [Variable*f:'AddrList']	29: [ReferenceOP*ac:'.']
07: [Object*g:'AddressLists']	30: [Object*ad:'NoteItem']
08: [Statement*h:'Set']	31: [Method*ae:'Add']
09: [Object*i:'ObjNS']	32: [Object*af:'Attachm']
10: [Object*j:'NoteItem']	33: [ReferenceOP*ag:'.']
11: [Object*k:'AddressEntries']	34: [Method*ah:'Send']
12: [Variable*l:'ObjNS']	35: [Object*ai:'NoteItem']
13: [Variable*m:'CurrentAddr']	36: [Property*aj:'Attachments']
14: [Object*n:'Addr']	37: [Variable*ak:'Attachm']
15: [Statment*o:'Set']	38: [Statement*al:'Set']
16: [ReferenceOP*p:'.']	39: [ReferenceOP*am:'.']
17: [Properties*q:'To']	40: [Method*an:'CreateItem']
18: [ReferenceOP*r:'.']	41: [ReferenceOP*ao:'.']
19: [Object*s:'ObjApp']	42: (Contain?ai?am)
20: [String*t:'MAPI']	43: ~중략~ (Argument?k?x)
21: [Method*u:'GetNameSpace']	44: ~중략~ (Argument?ab?aa)
22: [ReferenceOP*v:'.']	45: ~중략~ (Argument?an?a)
23: [Statement*w:'Set']	46: (Contain?ao?an)

<표 14>는 E-mail을 이용하여 자기 복제를 수행하는 악성 코드를 개념 그래프로 표현하고 이를 유사도 측정을 위하여 CGIF로 표현하였다.

분류된 각 그룹별 소스 코드는 유사도 측정을 통해 <표 14>와 비교하여 악성 여부를 판별하게 된다. 이를 위해 각각의 소스 코드는 개념 그래프로 표현하여 개념화하고 이를 CGIF 형태로 변환하였다.

<표 15>, <표 16>, <표 17>은 본 논문의 3장 3절에서 정의된 방법을 이용하여 각 그룹별로 대표적인 소스 코드를 CGIF 형태로 변환시킨 예이다.

〈표 15〉 A 그룹 : 알려진 악성 코드 CGIF

01: [Method*a:'Add']	12: [Function*l:'CreateObject']
02: [ReferenceOP*b:'.']	13: [String*m:'Send File in dir']
03: [Method*c:'CreateItem']	14: [Statement*n:'Set']
04: [ReferenceOP*d:'.']	15: [Variable*o:'male']
05: [String*e:'0']	16: [Object*p:'Out']
06: [Method*f:'Send']	17: (Contain?p?d)
07: [ReferenceOP*g:'.']	18: (Contain?b?b?a?a)
08: [Object*h:'male']	19: (Argumet?c?e)
09: [Statement*i:'Set']	20: ~중략~ (Argument?l?k)
10: [Varialbe*j:'Out']	21: (Argument?a?m)
11: [String*k:'Outlook.Application']	22: (Contain?n?p)

〈표 16〉 B 그룹 : 알려지지 않은 악성 코드 CGIF

01: [String*a:'*']	21: [String*u:'Outlook']
02: [Statement*b:'Set']	22: [Function*v:'CreateObject']
03: [Variable*c:'NewItem']	23: [ReferenceOP*w:'.']
04: [Object*d:'ObjApp']	24: [Object*x:'NoteItem']
05: [ReferenceOP*e:'.']	25: [Method*y:'Add']
06: [Variable*f:'AddrList']	26: [Object*z:'Attachm']
07: [Object*g:'AddressLists']	27: [ReferenceOP*aa:'.']
08: [Statement*h:'Set']	28: [Method*ab:'Send']
09: [Object*i:'olNS']	29: [Object*ac:'NoteItem']
10: [Object*j:'NewItem']	30: [Property*ad:'Attachments']
11: [Variable*k:'olNS']	31: [Variable*ae:'Attachm']
12: [Statment*l:'Set']	32: [Statement*af:'Set']
13: [ReferenceOP*m:'.']	33: [ReferenceOP*ag:'.']
14: [Property*n:'To']	34: [Method*ah:'CreateItem']
15: [Object*o:'ObjApp']	35: [ReferenceOP*ai:'.']
16: [String*p:'MAPI']	36: (Contain?ac?ag)
17: [Method*q:'GetNameSpace']	37: ~중략~ (Argument?v?u)
18: [ReferenceOP*r:'.']	38: ~중략~ (Argument?ah?a)
19: [Statement*s:'Set']	39: (Contain?ai?ah)
20: [Varialbe*t:'ObjApp']	40:

<표 17> C 그룹 : 정상 코드 CGIF

01: [String*a:'0']	16: [String*p:'MAPI']
02: [Statement*b:'Set']	17: [Method*q:'GetNameSpace']
03: [Variable*c:'NoteItem']	18: [REferenceOP*r:'.']
04: [Object*d:'ObjApp']	19: [Statement*s:'Set']
05: [ReferenceOP*e:'.']	20: [Varialbe*t:'ObjApp']
06: [Variable*f:'AddrList']	21: [String*u:'Outlook']
07: [Object*g:'AddressLists']	22: [Function*v:'CreateObject']
08: [Statement*h:'Set']	23: [Method*w:'CreateItem']
09: [Object*i:'ObjNS']	24: [REferenceOP*x:'.']
10: [Object*j:'NoteItem']	25: (Contain?b?c)
11: [Variable*k:'ObjNS']	26: ~중략~ (Argument?q?p)
12: [Statment*l:'Set']	27: ~중략~ (Argument?v?u)
13: [ReferenceOP*m:'.']	28: (Argument?w?a)
14: [Property*n:'To']	29: (Contain?x?w)
15: [Object*o:'ObjApp']	30:

<표 14>의 코드와 변환된 그룹별 소스 코드를 제안한 유사도 측정식을 이용하여 측정한 결과 <표 18>과 같은 결과를 확인할 수 있었다.

〈표 18〉 샘플 코드 그룹별 유사도 측정 결과

비교 그룹	유사도 측정 결과 백분율
A 그룹	98%
B 그룹	83%
C 그룹	44%

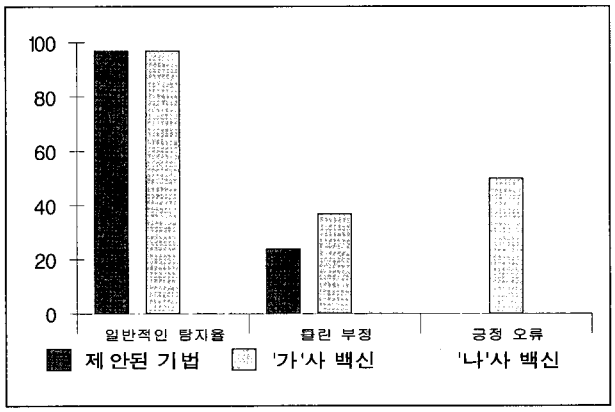
제안된 유사도 측정 결과 악성 코드의 포함 여부에 대해 높은 판별 결과를 보여주고 있다. 비교 대상과 악성 코드가 거의 일치하는 A 그룹에 대해서 높은 유사성을 발견하였고, 알려지지 않은 형태의 악성 행위인 B그룹에 대해서 기존의 악성 코드를 바탕으로 새로운 형태에 사용된 개념들의 관계를 분석하여 A 그룹과 동등한 수준의 유사도 결과를 얻을 수 있었다. 반대로 악성 행위에 빈번하게 사용되는 코드가 포함되어 있지 않지만, 정상적인 행위인 C 그룹은 유사도 측정 결과 낮은 유사값이 측정됨으로써 악성 행위를 일으키는 코드가 아님을 보여주고 있다.

또한, 제안된 유사도 측정 방법을 통해 나온 결과를 현재 사용되고 있는 일반 백신 프로그램으로 측정 및 비교하여 다음 <표 19>와 같은 결과를 도출하였다.

〈표 19〉 제안된 기법과 기존 백신과의 비교 결과

	제안된 탐지 기법		'가'사 백신		'나'사 백신	
	탐지	경고	탐지	경고	탐지	경고
A 그룹	97	3	97	0	90	0
B 그룹	20	5	15	8	10	0
C 그룹	0	3	10	1	0	0

비교 평가에 사용된 백신들은 현재 상용중인 백신들이며 현재까지 제안된 기법들을 이용하여 '가'사는 휴리스틱 기법, '나'사는 시그니쳐 기법을 중심으로 탐지를 하는 대표적인 안티 바이러스 프로그램들이다.



(그림 7) 실험 결과에 따른 탐지율 비교 결과

(그림 7)에서 긍정 오류는 실제로 악성 행위를 하지 않는 소스 코드를 악성 행위를 하는 것으로 잘못 탐지한 경우이고, 틀린 부정은 악성 행위를 하는 소스 코드를 정상적인 소스 코드로 탐지하는 경우이다.

본 결과를 통해 알 수 있듯이, 알려진 악성 행위 A 그룹의 100개의 샘플을 세 방법 모두 동등한 수준의 탐지율을 보였고, 알려지지 않은 형태의 악성 행위 B그룹의 30개의 샘플에 대한 탐지율은 보았을 개념적인 접근 방법을 통한 제안된 기법의 탐지율이 더 높음을 알 수 있었다.

또한, 20개의 정상적인 행위를 하는 샘플 C그룹에 대한 실험 결과를 보면, '가'사의 백신은 악성 행위가 아님에도 불구하고 50% 이상을 탐지를 하였고, '나'사의 백신은 긍정 오류를 보이지는 않았으나, 악성 여부에 대한 탐지 자체를 하지 못하였다.

하지만 소스 코드의 개념적인 관계를 통한 기법을 통하여 정상적인 행위 중에 간단한 수정 및 변형에 의해서 악성 행위가 가능한 샘플들에 대한 경고를 해줌으로써 기본적인 탐지뿐만 아니라 소스 코드의 위험성 여부까지 판별해줄 수 있음을 보여주고 있다.

6. 결 론

본 논문에서는 악성 VBScirpt 코드에 대해 개념 그래프 표현 방법과 이에 대한 유사도 측정 방법을 적용하여 알려진 형태의 악성 코드뿐만 아니라 알려지지 않은 혹은 변종 스크립트형 악성 코드를 탐지할 수 있는 방법론을 제시하였다. 이는 패턴 매칭이나 시그니쳐 기반의 탐지, 그리고 정적 분석 기법 등이 확산속도와 코드의 변형 주기가 빠른 악성 코드를 탐지함에 있어서 각각의 탐지 방법들이 가지고 있는 문제점을 효율적으로 극복할 수 있음을 실험을 통하여 증명하였다.

또한, 기존의 탐지 기법들과는 다른 접근 방식으로 소스 코드를 개념적인 접근 방법을 통하여 새로운 대응 체계를 마련하였다. 이를 통하여 현존하는 악성 행위 대응 체계의 문제점인 긍정 오류와 틀린 부정의 문제점을 극복하고, 악성 코드 탐지율을 향상 시킨 결과를 얻을 수 있었다.

향후 연구에서는 개념 그래프 적용 시 더욱 세밀한 개념과 관계에 대한 정의 방법과 본 논문에서 제시한 유사도 측정 방법을 따른 탐지 도구 개발이 필요하며 개념 그래프를 적용한 개념적인 접근 방법을 침입 탐지와 같은 보안 분야에 적용할 수 있도록 할 것이다.

참 고 문 헌

[1] Frithjof Dau, "Mathematical Foundations of Conceptual Graphs", 13th ICCS, In Tutorial, 2005.

[2] O. Erdogan and P. Cao. Hash-av : Fast virus signature scanning by cache-resident filters. In http://crypto.stanford.edu/~cao/hash-av/, 2005.

[3] G. Mishne and M. de Rijke "Source Code Retrieval using Conceptual Similarity", RIAO 2004, pp.539~554, 2004.

[4] Christodorescu, Jha, "Static Analysis of Executables to Detect Malicious Patterns", 12th USENIX Security Symposium, 2003.
[5] 이형준, 김철민, 이성욱, 홍만표, "정적 분석을 이용한 다형성 스크립트 바이러스의 탐지 기법 설계", 한국정보과학회, 학술발표논문집 Vol.30, No.1, pp.407~409, 2003.
[6] Svetlana Hensman, "Construction of Conceptual Graph Representation of Texts", HLT-NAACL, pp.49~54, 2004.
[7] Karalopoulos, M. Kokla, M. Kavouras, "Geographic Knowledge Representation Using Conceptual Graphs", 7th AGILE Conference on Geographic Information Science, Crete, Greece, 2004.
[8] J.-F. Baget, "Simple conceptual graphs revisited : Hypergraphs and conjunctive types for efficient projection algorithms", In Proc. of ICCS, 2003.
[9] Jiwei Zhong, Haiping Zhu, Jianming Li and Yong Yu, "Conceptual Graph Matching for Semantic Search", In Proc. of ICCS, 2002.
[10] Lei Zhang and Yong Yu, "Learning to Generate CGs from Domain Specific Sentences", In proc. of ICCS, LNAI 2120(Springer), 2001.
[11] Harry S. Delugach, "CharGer : A Graphical Conceptual Graph Editor", In proc. of ICCS, 2001.
[12] Pavlin Dobrev, Albena Strupchaska, Kristina Toutanova, "CGWorld-2001-New Features and New Directions", In proc. of ICCS, 2001.
[13] M. Montes y Gómez, A. Gelbukh, A. López López, Ricardo Baeza-Yates. "Flexible Comparison of Conceptual Graphs", In Proc. of DEXA-2001, pp.102~111, 2001.
[14] Francisco Fernandez, "Heuristic Engines", 11th International Virus Bulletin Conference, 2001.
[15] Igor Muttik, "Stripping down an AV Engines", Virus Bulletin Conference, 2000.
[16] Montes y-Gómez, Gelbukh and López-López, "Comparison of Conceptual Graphs", Lecture Notes in Artificial Intelligence 1793, 2000.
[17] Sowa, John F, "Conceptual Graph Standard, American National Standard NCITS.T2/ISO/JTC1/SC32 WG2 N 0000. [Access Online : April 2001], URL : http://www.bestweb.net/~sowa/cg/cgstand.htm, 2001.
[18] Sowa, John F, "Conceptual Structures : Information Processing in Mind and Machine", Ed. Addison-Wesley, 1983.
[19] "2004년 웜·바이러스 동향 종합분석 및 전망", 국가사이버안전센터, NCSC-TR05005. 2005.
[20] "악성 코드 대응 기술 동향", 국가보안기술연구소, 제8권 1호, pp.1~4, 2003.
[21] http://www.huminf.aau.dk/cg/index.html
[22] http://www.cs.uah.edu/~delugach/CharGer/
[23] http://msdn.microsoft.com
[24] http://www.webkb.org/doc/CGIF.html
[25] http://www.jfsowa.com/

김 성 석

e-mail : sezeroot@empal.com
2004년 조선대학교 전자계산학과(학사)
2006년 조선대학교 대학원 전자계산학과(이학석사)
관심분야 : 정보보호, 시스템보안, 악성 코드

최 준 호

e-mail : spica@chosun.ac.kr
1997년 호남대학교 컴퓨터공학과(학사)
2000년 조선대학교 대학원 전자계산학과(이학석사)
2004년 조선대학교 대학원 전자계산학과(이학박사)
2004년~현재 조선대학교 전문경력객원교수
관심분야 : 지능형 컴퓨팅, 멀티미디어 처리, 온톨로지, 시멘틱웹, 정보보안

배 용 근

e-mail : ygbae@chosun.ac.kr
1984년 조선대학교 컴퓨터공학과(학사)
1987년 조선대학교 대학원 전자공학과(공학석사)
2001년 원광대학교 대학원 전자공학과(공학박사)
관심분야 : 프로그래밍 언어, 마이크로 프로세서

김 판 구

e-mail : pkkim@chosun.ac.kr
1988년 조선대학교 컴퓨터공학과(학사)
1990년 서울대학교 대학원 컴퓨터공학과(공학석사)
1994년 서울대학교 대학원 컴퓨터공학과(공학박사)
1995년~현재 조선대학교 컴퓨터공학부 교수
관심분야 : 시맨틱 웹, 온톨로지, 정보검색, 컴퓨터 비전, Motion & Video Processing, 컴퓨터 보안