정 보 보 호 학 회 지
제24권 제1호, 2014. 2

악성코드 변종 탐지를 위한 코드 재사용 분석 기법

김 태 근*, 임 을 규**

요 약

본 논문은 수년간 급격하게 증가되어 많은 피해를 초래하고 있는 악성코드를 탐지하기 위한 기법을 제안한다. 악성코드 제작자로부터 생산되고 인터넷에 유포되는 대부분의 악성코드는 처음 개발된 제로-데이 악성코드의 코드 일부를 그래도 재사용하는 경우가 많다. 이러한 특징에 의해 악성코드 변종들 사이에는 악의적 행위를 위해 사용되는 함수들 중 공통으로 포함되는 코드들이 존재하게 된다. 논문에 저자는 이점에 착안하여 코드 재사용 검사 여부를 통한 악성코드 변종 탐지 기법을 제안하고 있다. 그리고 변종 샘플을 이용한 변종 탐지의 가능성을 증명하는 실험과 실제 공통으로 존재하는 재사용 코드 일부(함수) 추출 정확성을 알아보는 실험을 수행하여 주장을 뒷받침한다.

Ⅰ. 서 론

악성코드의 꾸준한 증가와 그로 인하여 발생되는 여러 피해가 지속적으로 발생하기 때문에 이를 방어하기 위한 여러 연구가 진행되고 있다. 여러 방면의 연구 가운데 가장 활발히 진행되고 있는 연구는 바로 악성코드 분석을 통해서 알 수 없는 프로그램에 대한 악성 여부를 탐지하는 방법론에 관한 것이다.

악성코드 분석은 크게 정적 분석과 동적 분석으로 분류된다. 정적 분석의 경우, 프로그램을 실행시키지 않고 프로그램의 특성을 잘 나타낼 수 있는 특징을 추출하는 방법이다.[1][2][3] 동적 분석은 이와 반대로 프로그램의 실행을 통해 나온 결과를 이용하여 프로그램의 특징을 추출한다. 본 논문에서 제안되는 기법은 정적 분석이며 별도의 실행 없이 프로그램에 포함된 코드 섹션을 이용한 특징 추출을 수행한다.[4]

특정 시스템을 대상으로 공격하는 형태의 APT 악성코드를 제외하고 일반 사용자를 대상으로 널리 유포되는 악성코드의 경우, 특정 악성코드 제작자에 의해 처음 제로-데이 악성코드가 개발된 후 해당 악성코드를 변형하거나 악성코드로부터 일부 코드를 그대로 다른 악성코드 제작에 활용하는 방식으로 변종을 생성하고 퍼뜨린다. 변종이 이러한 방식으로 제작되는 이유는 다음의 3가지 정도로 요약 된다.

- 안티바이러스 시스템 우회를 위한 목적
- 악성코드 제작비용 및 시간의 절감
- 유사 악성 행위의 반복

첫 번째, 안티바이러스 시스템 우회를 위해 제로-데이 악성코드를 그대로 사용하지 않고 변형하여 유포한다. 많은 안티바이러스 시스템은 파일의 해시 값이나 포함된 스트링을 이용하여 탐지를 수행하기 때문에 간단한 변형을 통해 회피가 쉽다. 따라서 기존의 중요 코드는 포함하면서도 이외 코드를 변형하여 악성코들 배포한다.

두 번째, 악성코드 제작비용과 시간을 절약하기 위해 이미 만들어진 제로-데이 악성코드의 코드를 재사용한다. 악성코드가 등장하기 시작한 초기와 달리, 시간이 지나감에 따라 악성코드 제작자의 성격이 점점 변하기 시작했다. 더 이상 제작의 목적이 악성코드를 만들어 유행시키는 것이 아닌 '금전적 이득'을 목적으로 제작되고 있는 추세이다. 즉, 악성코드 제작자에게 악성코드는 목적이 아닌 수단으로 변화한 것이다. 사이버 블랙마켓이

---

* 한양대학교 컴퓨터소프트웨어학과 (cloudio17@hanyang.ac.kr)
** 한양대학교 컴퓨터공학부 (imeg@hanyang.ac.kr)
이 논문은 2013년도 정부(미래창조과학부)의 재원으로 한국연구재단-차세대정보컴퓨팅기술개발사업의 지원을 받아 수행된 연구임(No. 2011-0029924).

형성되었고 제작된 악성코드는 시장 원리에 따라 거래가 된다. 이러한 이유로 한번 제작된 악성코드를 변형하거나 일부 코드를 재사용함에 따라 다양하게 많은 악성코드를 빠른 시간 내에 생산하는 것이 당연하게 되었다.

세 번째, 일반 사용자를 대상으로 하는 악성코드가 일반적으로 수행하는 행위가 상당히 유사하기 때문에 코드가 재사용될 수 있다. 종류가 매우 다양하지만 결국 하고자 하는 행위는 유사한 형태이며 이것은 즉, 한번 제작된 코드가 재사용될 수 있는 이유가 된다.

본 논문의 저자는 앞서 말한 내용을 바탕으로 임의 프로그램이 특정 악성코드의 변종인지 아닌지 판별하는데 코드 재사용 여부가 큰 기준이 될 것으로 판단하였다. 그리고 이에 대한 연구를 진행하였다.

논문은 다음과 같이 구성된다. 2장에서는 본 연구와 관련된 연구를 설명한다. 3장에서는 저자가 제안하는 악성코드 변종 탐지를 위한 코드 재사용 여부 분석 프레임워크의 전체 구조와 그를 구성하는 세부 모듈들을 차례로 설명한다. 4장에서는 변종 샘플을 이용한 변종 탐지의 가능성을 증명하는 실험과 실제 공통으로 존재하는 코드 일부(함수)를 잘 추출하는지 확인하는 실험을 수행한 결과를 설명한다. 5장에서는 연구 내용을 요약하고 학문적 기여도, 한계 및 추후 연구 방향에 관한 설명을 한다.

Ⅱ. 관련 연구

[5] 의 저자 Qinghua Zhang et al.는 본 논문에서 제안하는 프레임워크와 유사한 방식으로 메타몰픽 악성코드를 탐지하기 위한 기법을 제안하고 있다. 이 논문의 기법은 저자가 제안하는 기법과 두 바이너리로부터 추출된 각각의 코드를 구성하는 일부 코드가 동일한 지 판별하는 전체적인 흐름에서 유사성을 갖는다. 본 저자가 수행한 연구는 변종 관계를 찾기 위한 목적으로 하고 있기 때문에 여러 코드 난독화 기법에 대한 처리가 이루어지지 않지만 향후 이를 보안하여 메타몰픽 악성코드에 대응할 수 있도록 계획하고 있다. [5]에서 제안하는 기법은 본 저자가 주장하는 코드 재사용을 통한 변종 탐지 기법이 유용하다는 주장을 같이 하고 있다. 논문에는 많은 장점 존재하지만 반면에 미처 고려되지 않은 사항도 발견되었다. 코드 재사용 여부를 분석하는데 있어 중요한 요소는 바로 동일하다고 추출된 코드가 동일한 컴파일러로부터 생성된 코드이거나 라이브러리 함수의 코드를 제외시키는 일이다. 개발자가 직접 작성한 코드들 이외에도 예외처리를 위해 컴파일러가 바이너리를 생산할 때 추가적인 코드를 덧붙이기 때문에 설사 다른 프로그램이라도 동일한 컴파일러에 의해 컴파일 되면 동일한 코드 일부가 추출될 수 있다. 그리고 라이브러리 함수의 경우, 하나의 특징이라고도 할 수 있으나 많은 프로그램들에서 동일하게 나타날 수 있으므로 변종 여부를 결정하는데 변별력을 떨어뜨리는 결과를 초래할 수 있다. 그런데 해당 관련 논문에서는 이를 고려하지 않고 있다.

[6] 의 저자 Gil Tahan은 Mal-ID라고 하는 자동 악성코드 탐지 기법을 제안하고 있다. 제안하고 있는 알고리즘은 공통 세그먼트를 정상 소프트웨어와 악성코드로부터 미리 추출하고 이를 탐지 이용한다. 본 논문의 기법과 유사한 흐름을 갖는다. Mal-ID는 바이너리를 세그먼트로 나눌 때 명령어 단위의 코드 세그먼트가 아닌 단순한 바이트 세그먼트를 사용한다. 이를 통해 선 처리 과정이나 언패킹, 디스어셈블 과정이 생략될 수 있지만 이로 인해 발생하는 문제도 존재한다. 언패킹의 경우, 동일한 프로그램일지라도 패킹 알고리즘이 다를 경우, 다른 종류의 바이트 세그먼트가 발생할 수 있고 디스어셈블 명령어를 고려하지 않기 때문에 미리 저장된 공통 세그먼트의 의미나 중요성을 판단할 수 없다. 단순한 바이트 시퀀스이기 때문에 전체 바이너리로부터 어느 길이로 세그먼트로 분할하느냐에 따라 결과가 매우 달라질 수 있다.

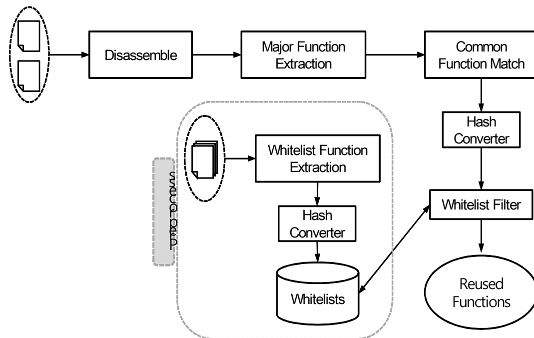
Ⅲ. 코드 재사용 탐지

본 장은 두 바이너리의 변종 관계 여부를 알아보기 위해 공통적으로 나타나는 함수를 추출하는 프레임워크를 설명한다. 함수 단위로 코드를 그대로 사용하는 것이 개발을 할 때 가장 쉽게 다른 코드를 재사용하는 경우이기 때문에 함수 단위로 재사용 여부를 검사한다.

3.1. 코드 재사용 탐지 분석 과정

두 바이너리에서 공통적으로 나타나는 코드(함수)를 찾기 위해서 5가지 과정이 수행된다. 과정 (1) 두 바이너리로부터 디스어셈블 코드를 각각 추출한다. 과정 (2)

두 개의 디스어셈블 코드를 블록 및 함수 단위로 분할하고 함수 가운데 API, 라이브러리 함수, 사용자 정의 함수를 호출하는 루틴이 포함된 함수만을 선택한다. 과정 (3) 선택된 주요 함수들을 비교하여 공통적으로 나타나는 함수를 추출한다. 과정 (4) 각 함수의 해시 값을 배정한다. 과정 (5) 각 함수 해시 값을 비교하여 공통 함수 가운데 컴파일러가 생산한 코드 혹은 라이브러리 함수를 제거한다.



〔그림 1〕 공통 함수 추출 과정

3.2. 디스어셈블

저자는 보다 쉽게 데이터를 가공하고 조작하기 위하여 기존에 잘 알려진 IDA Pro나 Ollydbg와 같은 디스어셈블러를 이용하지 않고, 파일의 코드 섹션을 순차적으로 스캔하며 개별 명령어에 대하여 디스어셈블을 수행하는 “Linear Sweep” 방식으로 디스어셈블러를 구현하였다. 이에 따라 프로그램을 블록 단위로 분할할 때 블록을 구성하는 명령어들을 편리하게 관리할 수 있다. 디스어셈블러는 바이너리 파일의 분석을 위해 사용되는 명령어의 변환 모듈로, 파일의 코드 섹션을 읽어 섹션의 시작 주소와 코드 섹션의 크기를 이용하여 순차적으로 디스어셈블을 수행한다.

3.3. 주요 함수 추출

3.3.1 블록 및 함수 단위 분할

디스어셈블 코드는 함수들의 집합으로 구성되며, 각 함수들 또한 블록의 집합으로 구성된다. 따라서 명령어를 기준으로 함수와 블록을 구분하는 작업을 수행한다. 함수의 경우, RETN 명령어를 기준으로 함수를 구분하는데, 이는 대부분의 함수 호출 규약에서 함수가 끝날 때 RETN 명령어가 사용되는 특징을 이용한 것이다. 블록의 경우는 JMP와 같은 분기 명령어를 이용하여 분할한다. 조건 분기 명령어나 무조건 분기 명령어가 나타날 경우 한 블록이 끝나고 다음 블록이 시작된다.

3.3.2 주요 함수 선택

두 바이너리를 구성하는 함수의 수가 많기 때문에 공통 함수를 추출하기 위해 각 바이너리로부터 추출된 함수들을 모두 비교하는 데에는 많은 시간이 소요된다. 그러므로 두 바이너리에 포함된 함수들을 비교하는데 소요되는 수행시간을 감소시키기 위해 API나 라이브러리 함수 및 사용자 지정 함수 호출 명령어가 포함된 함수를 주요 함수로 선정한다. 함수 호출 명령어가 포함된 함수를 선택하는 이유는 프로그램이 어떠한 기능을 수행하기 위해서는 필수적으로 운영체제나 라이브러리에서 제공하는 함수를 호출이 필수적이기 때문이다. 그리고 이를 포함하지 않는 경우는 해당 프로그램의 주요 행위를 담당한다고 할 수 없기 때문에 제외가 가능하다.

주요 함수를 탐색하기 위해서는 각 함수를 블록 단위로 스캔하여 CALL 명령어가 포함되어 있는지 확인한다. 함수 호출은 대부분 CALL 명령어를 통하여 이루어지기 때문에 CALL 명령어가 포함되어 있을 경우 API, 라이브러리, 사용자 정의 함수를 호출하는 것으로 판단할 수 있다. 일부 컴파일러는 함수 호출에 JMP 명령어를 사용하는 경우도 있다. 따라서 JMP 명령어를 이용하여 함수를 호출하는 루틴을 정의하고 검사하여 호출 여부 검사에 정확성을 향상시켰다.

3.4. 공통 함수 추출

앞서 언급한 바와 같이 함수는 블록들의 연속된 집합으로 구성된다. 따라서 두 함수가 공통 모듈인지 판별하기 위해서는 두 함수를 구성하는 각각의 블록들을 매칭시키고, 그 결과를 통해 두 함수가 실제로 유사한 정도를 분석한다. 매칭은 두 함수 중 자신을 구성하는 블록의 개수가 적은 함수를 기준으로 각 블록이 최대 유사도 값을 갖는 상대 블록을 자신과 맵핑하는 방식으로 이루어진다. 이때 만약 두 함수가 같은 개수의 블록을 갖을

경우, 전체 함수의 명령어 개수를 비교하여 작은 함수를 기준으로 한다 이렇게 블록의 개수나 명령어의 개수를 비교하여 작은 함수를 기준으로 삼는 이유는 큰 함수를 기준으로 할 경우, 두 함수가 포함관계에 있는 상황을 반영 하지 못하기 때문이다. 두 함수의 유사성을 판별하기 위해서는 두 가지 계산 과정이 요구된다. 첫 번째는 실제 두 블록들 간의 유사 여부를 나타내는 유사도 계산 과정이고, 두 번째는 블록 간 유사도 결과 값을 종합하여 나온 두 함수의 최종 유사도 결과를 계산하는 과정이다. 그리고 두 함수의 최종 유사도 결과 값이 임계 값 이상일 경우, 두 함수를 공통 함수로 정의한다.

3.4.1 블록 간 유사도 계산

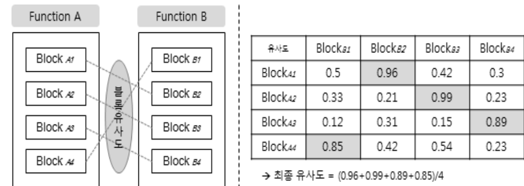
블록 간의 유사도는 각 블록을 구성하고 있는 명령어의 Opcode 종류가 얼마나 공통분모를 갖는지 계산한 결과이다. Jaccard similarity coefficient를 이용하여 유사도를 계산하다. [수식 1]은 블록 간 유사도 계산을 나타낸다. BA, BB는 블록A와 블록B를 의미하고 SAB는 두 블록의 유사도를 의미한다. 각각의 블록은 명령어의 집합으로 구성되며 이 두 집합의 공통 명령어 개수를 최대 집합 크기로 나누어 계산한다.

$$B_A = \{I_{A_1}, I_{A_2}, \cdots, I_{A_n}\}$$
$$B_B = \{I_{B_1}, I_{B_2}, \cdots, I_{B_n}\}$$
$$S_{AB} = \frac{n(B_A \cap B_B)}{\max(n(B_A, B_B)}$$

〔수식 1〕 블록 간 유사도

3.4.2 함수 간 유사도 계산

[그림 2]는 최종 유사도를 계산하는 과정을 나타낸다. 함수 간 유사도를 계산하기 위하여 필수적으로 각 함수를 구성하는 블록들 간의 유사도 계산이 선행된다. 이때 3.4.1에서 설명한 블록 간 유사도 계산 방법이 사용된다. 블록 간 유사도가 모두 계산되면 [그림 2]의 오른쪽에 나타난 블록 대 블록의 유사도 결과가 테이블의 형태로 완성된다. 두 함수가 유사하다는 것은 그 함수를 구성 하는 블록이 얼마나 잘 매칭 되느냐에 따라 결정된다고 할 수 있다. 블록 간의 매칭관계를 찾기 위하여 두 함수 중 블록의 개수가 적은 함수를 기준으로 그 함수를 구성하는 블록이 갖는 최대 유사도 값을 선택하여 그의 평균을 계산하다. [그림 2]의 예에서는 함수 A의 블록을 기준으로 각 블록이 갖는 최대 유사도를 선택하였다. 최대 유사도를 갖는 블록과의 매칭결과는 (BlockA1-BlockB2), (BlockA2-BlockB3, Block-A3-BlockB4), (BlockA4-BlockB1) 로 나타난다. 그리고 각 매칭 쌍의 유사도를 평균 낸 결과가 최종유사도로 계산됨을 알 수 있다.



유사도	BlockB1	BlockB2	BlockB3	BlockB4
BlockA1	0.5	0.96	0.42	0.3
BlockA2	0.33	0.21	0.99	0.23
BlockA3	0.12	0.31	0.15	0.89
BlockA4	0.85	0.42	0.54	0.23

〔그림 2〕 최종 유사도 결과 예시

3.5. 화이트 리스트 생성 및 필터링

추출된 공통 함수는 악성코드 제작자에 의해 재사용된 함수를 이외에도 컴파일러에 의해 생성된 함수나 라이브러리의 함수가 포함되어 있을 수 있다. 이를 제거하기 위해 컴파일러 생성 함수나 라이브러리 함수를 미리 화이트 리스트로 생성하고 필터링하는 과정이 필수적이다. 세부 절에서 이에 대한 방법을 설명한다.

3.5.1 화이트 리스트 생성

컴파일러 생성이나 라이브러리 함수를 화이트 리스트로 미리 저장하기 위해서 동일한 컴파일러로 컴파일된 악성코드, 그 중에서도 공통 함수가 존재하지 않는 다른 패밀리의 악성코드간의 공통 함수를 추출하였다. 다른 패밀리로 분류된 악성코드 즉, 변종 관계에 있지 않은 악성코드 간에서 추출된 공통 함수의 경우, 개발자에 의해 코드(함수)가 재사용되기보다는 컴파일러에 의한 생성이나 라이브러리 함수의 사용에 의한 결과라고 판단할 수 있다.

3.5.2 화이트 리스트 필터링

화이트 리스트를 필터링하는 과정은 두 바이너리로부터 추출된 공통 함수와 미리 저장된 화이트 리스트

함수를 비교하여 제거하는 것을 의미한다. 이때 화이트리스트의 함수 개수가 많아질수록 시간 소모가 상당히 증가되므로 이를 해결하기 위한 방안이 필요하다. 저자는 함수의 명령어 시퀀스를 하나의 스트링으로 간주하고 스트링 해시 값을 비교하는 방식으로 필터링 시간 복잡 도를 줄일수 있었다. 이때 사용된 해시 알고리즘은 Robert Sedgwick가 제안한 RSHash 알고리즘 [7] 이다.

Ⅳ. 실험 결과

실험은 크게 변종 관계에 있는 악성코드 간 공통 함수를 추출하여 제안하는 프레임워크가 변종 구분에 활용될 수 있는지 확인하는 실험과 의도적으로 공통 함수를 삽입하여 작성된 두 프로그램으로부터 제대로 공통 함수가 추출되는지 확인하는 실험으로 구성하였다. 세부 절에서 각 실험 결과를 설명한다.

4.1. 변종 탐지

다른 패밀리간의 악성코드의 경우, 재사용되는 코드(함수)가 나타나지 않음과 동시에 같은 패밀리내의 악성코드 간에서는 재사용되는 코드(함수)가 다수 존재해야 구분에 좋은 척도라고 할 수 있다. 이를 알아보기 위해 타 패밀리 관계의 악성코드를 프레임워크에 적용시켜 공통 함수를 추출하는 실험과 같은 패밀리의 악성코드로부터 공통 함수를 추출하는 실험을 수행했다.

실험은 VxHeaven [8] 라고 하는 웹 페이지에서 제공하는 악성코드를 이용하여 진행되었다.

타 패밀리 악성코드 간의 공통 함수 추출 결과는 [표 1]에 나타나있다. 10 종류의 각기 다른 패밀리에 속하는 악성코드 샘플을 이용하여 실험을 수행하였다. 악성코드 45쌍으로부터 공통 함수를 추출한 결과, 전혀 공통 함수가 추출되지 않은 경우가 29쌍, 1개의 공통 함수가 추출된 경우가 12쌍, 2개 이상의 공통 함수가 추출된 경우가 4쌍이 존재하였다. 공통 함수가 추출된 경우, 확인 결과 모두 컴파일러 생성 함수 혹은 라이브러리 함수가 화이트리스트에 포함되어 있지 않아 제거되지 않은 것이었다. 예를 들어, RedFox 악성코드와 나머지 다른 9가지 악성코드에서 추출된 공통함수는 CPaintDC()와 CWinApp()의 라이브러리 함수이었고 23개의 많은 공통모듈이 추출된 Agent, Darksky, Skyper 악성코드 역시 모두 키보드 입력을 받는 MFC 함수들이 공통적으로 포함되어있는 것을 확인 할 수 있었다. Net-Worm.Win32.Zusha라고 하는 악성코드 패밀리를 구성하는 샘플 5가지로부터 공통 함수를 추출하였다. 그 결과는 [표 2]에 나타났다. 앞서 진행된 실험과 달리 다수의 공통 함수가 추출된 것을 볼 수 있다. 직접 샘플 분석을 시행한 결과, 추출된 함수는 각 변종들에게

〔표 2〕 동종 패밀리 악성코드 간 공통함수 추출

공통 함수 추출 개수		Zusha				
		a	b	c	e	h
Zusha	a		25	16	14	22
	b	25		16	14	19
	c	16	16		21	23
	e	14	14	21		19
	h	22	19	23	19	

〔표 1〕 타 패밀리 악성코드 간 공통 함수 추출 결과

추출 결과	Red-Fox	Skyper	Small	Callbox	Klez	Agent	DFMA	Zusha	Darksky	Spaco-om
RedFox		1	1	0	0	1	1	0	2	0
Skyper	1		1	0	0	23	0	0	23	0
Small	1	1		0	0	1	1	0	1	0
Callbox	0	0	0		0	1	0	1	1	0
Klez	0	0	0	0		0	0	0	0	0
Agent	1	23	1	1	0		1	0	23	0
DFMA	1	0	1	0	0	1		0	0	0
Zusha	0	0	0	1	0	0	0		0	0
Darksky	2	23	1	1	0	23	0	0		0
Spacoom	0	0	0	0	0	0	0	0	0	

서 그래도 재사용되어 악성행위에 이용된 것을 알 수 있었다.

4.2. 공통 모듈 추출 정확성

본 실험은 각기 다른 기능을 하는 임의 프로그램 A, B, C, D에 공통 함수를 삽입하고 각 프로그램들을 제안하는 프레임워크에 적용시켜 삽입된 함수가 공통적으로 프로그램에 포함되어있을 경우 이를 잘 검출 할 수 있는지 알아본다. 삽입된 함수는 직접 작성되었으며 동일한 컴파일러로 프로그램을 컴파일하였다.

삽입된 공통함수 종류는 파일 입출력 관련 함수, 레지스트리 관련 함수, 네트워크 관련 함수로 구성되며 기능에 따라 이름을 명명하였다. 이렇게 비슷한 기능을 하는 함수를 함께 실험에 포함시킨 이유는 동일한 라이브러리를 사용하는 함수가 존재할 때 이를 잘 구분할 수 있을지 확인하기 위함이다.[표 3]을 보면 프로그램 별로 어떤 함수가 삽입되었는지 알 수 있다. 그리고 [표 4]는 한 쌍의 프로그램을 제안하는 프레임워크에 적용하여 검출된 공통 함수를 결과로 나타낸다. [표 4]에 나타난 각 프로그램 쌍의 공통 함수 추출 실험 결과, 공통 함수로 검출된 결과와 예상 결과와 동일함을 알 수 있다. 또한 화이트 리스트 필터링 기법을 적용한 경우와 반대 경우를 비교했을 때 공통 함수로 추출된 양이 확연이 줄어든 것을 볼 수 있다. 즉, 동일한 컴파일러에 의해 생성된 함수와 동일한 라이브러리 함수가 공통 함수 추출 정확성에 많은 영향을 주는 것을 알 수 있다.

이를 단적으로 보여주는 예는 프로그램C와 프로그램 D를 이용한 실험이다. 두 프로그램은 각각 클러스터링

〔표 3〕 프로그램-함수 포함 관계

함수	함수가 삽입된 프로그램
FileCreate()	Program A,B,C
FileCpoy()	Program A,B,C
FileSearch()	Program A
RegWriteStr()	Program A,B
RegReadStr()	Program A,D
RegWriteInt()	Program A
RegReadInt()	Program A,D
TCPconnect()	Program A,B
SentPacket()	Program A,C
SentFile()	Program A,C

과 머신러닝을 수행하는 프로그램이다. 유사한 기능을 수행하기 때문에 비슷한 종류의 라이브러리 함수를 다수 사용하고 있다. 그 결과 화이트 리스트 필터링의 적용 없이 그대로 공통 함수를 추출하게 되면 실제 공통함수가 존재하지 않는 경우에서도 3795개의 함수가 최종 결과로 나타나는 것을 알 수 있다.

추가적으로 추출된 공통 함수가 발생하였는데 이는 직접 작성한 디스어셈블러의 오류에 인한 것으로 확인되었고 제안하는 알고리즘의 문제로 볼 수 없다. 추후 디스어셈블 모듈의 성능 향상이 요구된다.

〔표 4〕 공통 함수 추출 결과

	필터링 적용	필터링 미적용
A,B	추출 결과	추출 결과
공통 함수	FileCreate	FileCreate
	FileCopy	FileCopy
	RegWriteStr	RegWriteStr
	TCPconnect	TCPconnect
	기타 (3개)	기타 (102개)
A,C	추출 결과	추출 결과
공통 함수	FileCreate	FileCreate
	FileCopy	FileCopy
	SendPacket	SendPacket
	SendFile	SendFile
	기타 (0개)	기타 (131개)
A,D	추출 결과	추출 결과
공통 함수	RegWriteStr	RegWriteStr
	RegReadStr	RegReadStr
	RegReadInt	RegReadInt
	TCPconnection	TCPconnect
	기타 (2개)	기타 (135개)
B,C	추출 결과	추출 결과
공통 함수	FileCreate	FileCreate
	FileCopy	FileCopy
	기타 (0개)	기타 (127개)
B,D	추출 결과	추출 결과
공통 함수	RegWriteStr	RegWriteStr
	TCPconnect	TCPconnect
	기타 (0개)	기타 (127개)
C,D	추출 결과	추출 결과
공통 함수	추출 함수 없음	추출 함수 없음
	기타 (14개)	기타 (3795개)

Ⅴ. 결론 및 향후 연구

본 논문에서는 악성코드 제작자로부터 처음 제로-데이 악성코드가 개발되고 변종이 그로부터 만들어질 때 코드가 재사용이 이루어지는 점에 착안하여 알 수 없는

두 프로그램이 변종 관계인지 확인하기 위한 공통 함수 추출 알고리즘을 제안하였다. 바이트 시퀀스가 아닌 디스어셈블된 명령어 시퀀스를 이용하였다. 그리고 주요 행위를 담당할 만한 즉, 기능적으로 중요도가 높은 함수를 추출, 비교하여 공통 함수가 존재하는지 알아내는 방법을 제안하였다. 이때 컴파일러 생성 함수 및 라이브러리 함수에 대한 화이트 필터링 방법도 제안하였다. 그리고 공통 함수 추출 정확성에 있어 화이트 리스트 필터링이 많은 영향을 줄 수 있음을 실험을 통해 증명하기도 하였다. 이외에도 변종 샘플을 이용한 변종 탐지의 가능성을 증명하는 실험과 실제 공통으로 존재하는 코드 일부(함수)를 잘 추출하는지에 대한 증명을 위한 실험을 수행하여 주장을 뒷받침하였다. 실험결과를 통해 동일 패밀리의 악성코드 변종 간에는 공통으로 포함된 함수가 존재하였고 그 외 타종 악성코드 간에는 존재하지 않음을 확인하였다. 따라서 변종 탐지나 패밀리 별 주요 행위 분석에 본 프레임워크를 적용할 수 있을 것으로 판단된다. 그러나 실험 결과, 화이트리스트 데이터베이스가 아직 많은 컴파일러 생성 함수 및 라이브러리 함수를 커버하고 있지 못해 몇몇 오류도 발견되었다. 더 많은 수의 악성코드로부터 화이트 리스트를 생성해야 할 것이며 이는 추후에 지속적으로 진행될 계획이다.

참고문헌

[1] S. Momina Tabish, M. Zubair Shafiq, and Muddassar Farooq, "Malware Detection using Statistical Analysis of Byte-Level File Content", *the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, June 2009.

[2] Bilar, Daniel. "Opcodes as predictor for malware." *International Journal of Electronic Security and Digital Forensics*, pp. 156-168. January 2007.

[3] Santos, I., Penya, Y. K., Devesa, J., & Bringas, P. G. “N-grams-based File Signatures for Malware Detection”, *ICEIS*, pp. 317-320, 2009

[4] Park, Y. J., Zhang, Z., & Chen, S. “Run-time detection of malwares via dynamic control-flow inspection”, *Application-specific Systems, Architectures and Processors*, pp. 223-226, July 2009

[5] Zhang, Qinghua, and Douglas S. Reeves. "Metaaware: Identifying metamorphic malware." *Computer Security Applications Conference*, pp. 411-420, December, 2007.

[6] Tahan, Gil, Lior Rokach, and Yuval Shahar. "Mal-ID: Automatic Malware Detection Using Common Segment Analysis and Meta-Features." *The Journal of Machine Learning Research* pp. 949-979, 2012

[7] Robert Sedgwick, ALGORITHMS. Second Edition, Addison-Wesley Publishing, Company, Inc. 1988.

[8] Vxheaven, http://vx.netlux.org/

〈저자소개〉

김 태 근 (Kim TaeGuen)
정회원
2011년 2월 : 한양대학교 컴퓨터 전공 학사
2013년 2월 : 한양대학교 일반대학원 전자컴퓨터통신공학과 석사
2013년 3월~현재 : 한양대학교 일반대학원 컴퓨터소프트웨어공학과 박사과정
<관심분야> 취약점분석, 악성코드 분석, 정보보호

임 을 규 (Eul Gyu Im)
정회원
1992년 : 서울대학교 컴퓨터공학과 학사
1994년 : 서울대학교 컴퓨터공학과 석사
2002년 : Univ. of Southern California, Computer Science, Ph.D.
2005년~현재 : 한양대학교, 컴퓨터공학부 부교수
<관심분야> 제어시스템 보안, 악성코드, 정보보호, 소프트웨어 취약 점검