*Journal of The Korea Institute of Information Security & Cryptology* ISSN 1598-3986(Print)
VOL.25, NO.2, Apr. 2015 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2015.25.2.321

# 변종 악성코드 유사도 비교를 위한 코드영역의 함수 분할 방법*

박 찬 규,[1†] 김 형 식,[1] 이 태 진,[2] 류 재 철[1‡]
[1]충남대학교, [2]한국인터넷진흥원

# Function partitioning methods for malware variant similarity comparison*

Chan-Kyu Park,[1†] Hyong-Shik Kim,[1] Tae jin Lee,[2] Jae-Cheol Ryou[1‡]
[1]Chungnam National University, [2]Korea Internet & Security Agency

## 요 약

백신 프로그램이 일반화되면서 이를 우회하기 위한 목적으로 기존 악성 프로그램에 포함된 문자열 혹은 코드 일부가 변경된 변종 악성코드가 많이 나타나고 있다. 기존의 백신 프로그램이 시그너처에 기반한 분석을 통하여 악성코드 여부를 판단하기 때문에 이미 알려진 악성코드라고 하더라도 일부만 변경되면 탐지하기 어려운 문제가 있었다. 본 논문에서는 해쉬값을 이용한 코드 비교 방법을 확장하여 일부만 변형된 악성코드를 손쉽게 탐지하기 위한 새로운 방법을 제안한다. 악성코드 전체에 대한 해쉬값 뿐만 아니라 함수 단위와 코드블록 단위로 해쉬값을 생성하여 일부만 일치하는지 판단하고 상수나 주소 등을 제거한 후에 해쉬값을 생성함으로써 상수나 주소 때문에 다르게 판단하는 오류를 제거하였다. 제시된 방법을 이용하여 변형된 악성코드에 숨겨진 유사성을 해쉬값 비교로 탐지할 수 있음을 확인하였다.

## ABSTRACT

There have been found many modified malwares which could avoid detection simply by replacing a sequence of characters or a part of code. Since the existing anti-virus program performs signature-based analysis, it is difficult to detect a malware which is slightly different from the well-known malware. This paper suggests a method of detecting modified malwares by extending a hash-value based code comparison. We generated hash values for individual functions and individual code blocks as well as the whole code, and thus use those values to find whether a pair of codes are similar in a certain degree. We also eliminated some numeric data such as constant and address before generating hash values to avoid incorrectness incurred from them. We found that the suggested method could effectively find inherent similarity between original malware and its derived ones.

**Keywords:** Malware variant, Similarity comparison, Function division

## I. 서 론

악성코드의 수가 급증하고 있는 가운데, 최근 등장하고 있는 악성코드 중에서는 기존 악성코드 내 문자열이나 실행코드 일부를 변경하여 백신 프로그램의 탐지를 우회하는 변종 악성코드가 존재한다. 이러한 변종 악성코드가 등장할 수 있는 이유는 악성코드 제작자가 새로운 악성코드를 제작하는 과정에서 기존의

접수일(2015년 1월 29일), 수정일(2015년 3월 12일), 게재확정일(2015년 3월 12일)
* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 정보통신·방송 연구개발사업의 일환으로 수행하였음. [14-824-06-001, 사이버 공격의 사전 사후 대응을 위한 사이버 블랙박스 및 통합 사이버보안 상황분석 기술 개발]
† 주저자, jasucy@home.cnu.ac.kr
‡ 교신저자, jcryou@home.cnu.ac.kr(Corresponding author)

코드를 재사용하기 때문이다. 하지만 시그니처 기반으로 악성코드를 탐지하고 있는 백신 프로그램에서 시그니처 영역의 데이터가 한 비트만 바뀌어도 악성코드를 탐지할 수 없는 문제점이 발생한다. 이에 백신 프로그램 제조사들은 기존 시그니처 기반 분석 외에도 다양한 기술들을 도입하여 악성코드를 탐지하고 있다. 프로그램 내에 호출되는 함수의 순서나 빈도를 분석하거나[1], 악성코드를 분석용 클라우드 서버로 전송하여 동적 분석[2]을 하는 방법 등이 대표적이다. 하지만 신규 악성코드가 등장할 경우 악성코드의 분석 시간이 많이 소요되며, 이로 인하여 대응이 늦어질 수밖에 없다.

따라서 등장하고 있는 변종 악성코드에 효과적으로 대처하기 위해선 악성코드 간에 유사도 비교를 수행할 필요가 있다. 신규 악성코드가 발견되었을 때, 기존 악성코드와의 유사도 비교를 통하여 유사도가 높을 경우 같은 부류의 악성코드로 추론할 수 있다. 이러한 과정을 통해 악성코드의 분석 과정을 생략할 경우 악성코드임을 판단하는 시간을 줄이고, 신규 악성코드에 대한 빠른 대응을 할 수 있다. 또한 같은 악성코드 제작자가 기존의 악성코드의 일부를 재사용했을 경우 해당 제작자가 제작한 악성코드들과 유사도를 비교하여 악성코드의 제작자 또는 공격경로를 파악할 수 있다. 다만 계속해서 새로운 악성코드가 접수되면 신규 악성코드와 기존의 모든 악성코드들과 유사도를 비교하기에 너무 많은 시간이 걸리기 때문에 이를 해결하기 위한 방법으로 이미 분석을 완료한 유사한 악성코드끼리 그룹[3]을 만들어 그룹 단위로 유사도 비교를 하는 방법이 있다.

따라서 본 연구에서는 악성코드의 유사도 비교를 위해 코드영역에서 함수를 나누는 방법과 나눈 함수에서 함수 정보를 추출하는 방법에 대하여 제안한다. 특히 함수 정보를 추출하는 단계에서 함수 내에 포함된 상수 혹은 주소를 제거한 상태에서 이를 기본 블록(Basic block)으로 나눈 단위를 '코드블록(Code block)'이라 제안하고, 코드블록 단위에서 변종 악성코드의 유사도를 비교하는 방법에 대하여 제안한다.

Fig.1.은 유사도 비교를 위한 전체 시스템 구조이다. 본 연구에서는 Windows 실행파일 형태의 악성코드에서 코드영역을 함수와 코드블록을 나눈 다음 나누어진 부분을 해쉬값으로 생성하여 데이터베이스에 저장하는 '악성코드 추출 모듈(Malware Extraction Module)'에 대하여 설명한다.

Fig.2.는 유사도 비교를 위한 함수 정보들이다. 유사도는 크게 3가지를 정보를 이용하여 비교하는데, 일차적으로 원본 함수들로 가지고 비교를 한다(HASH 1). 하지만 원본 함수만을 가지고 비교를 할 경우 동일한 작업을 수행하는 코드라 할지라도 컴파일러 버전이나 시스템 환경 등에 따라 상수 혹은 주소가 바뀔 수 있다. 이렇게 되면 다른 해쉬값이 생성되어 버리기 때문에 정확한 유사도 비교를 할 수 없다. 이러한 문제를 해결하기 위해 어셈블리 코드 내에 포함된 상수 혹은 주소를 제거한 형태의 함수(이하 '상수제거 함수')를 유사도 비교에 사용한다

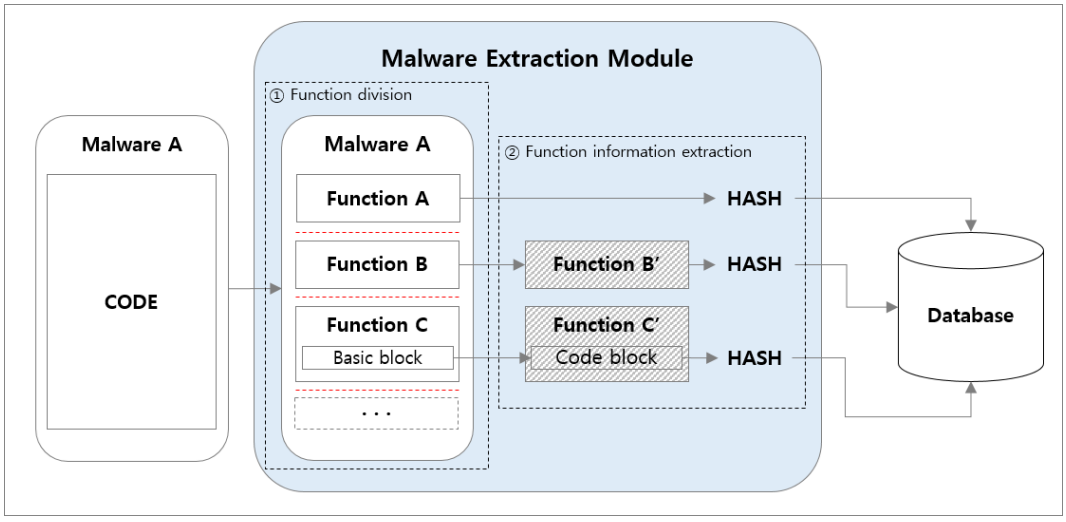



Fig. 1. System overview for code block similarity comparison

(HASH 2). 본 논문에서 제안하는 '상수제거 함수'는 상수뿐만 아니라 주소 등을 모두 제거한 값을 가리킨다. 더 나아가 상수제거 함수를 기본 블록 단위로 나누어(이하 '코드블록') 이를 가지고 유사도를 비교한다(HASH 3).

본 논문은 총 5장으로 구성되어 있으며, 1장에서는 본 연구에서 수행하고자 하는 내용에 대하여 소개하고, 2장에서는 현재 악성코드 유사도 비교와 관련한 국내·외 논문들을 소개한다. 3장에서는 코드영역에서 함수를 나누는 방법과 함수 정보를 추출하기 위한 방법에 대해 설명하고, 앞서 제안한 코드블록을 추출하는 방법에 대해 설명한다. 4장에서는 개발한 엔진의 실험 결과를 설명하고, 5장에서는 추후 연구 방향에 대해 소개한다.

## II. 관련 연구

악성코드 유사도 비교에 관한 연구는 이미 국내·외에 제안된 바가 있다. 발표된 논문에 대해 소개하자면 다음과 같다.

주요 블록 비교를 통한 악성코드 탐지 기법[4]은 악성코드로 의심되는 파일들을 대상으로 기존 악성코드들과 비교하여 유사도가 높은 경우 악성코드임을 판단할 수 있는 방법을 제시하였다. 우선 바이너리 코드를 함수와 블록 단위로 나누게 되는데 함수는 RETN 명령어를 기준으로 나누고, 블록은 분기 명령어를 기준으로 나눈다. 나누어진 블록들 중에서 악성코드임을 결정지을 수 있는 블록들을 선별하여(이하 '주요 블록') 주요 블록들만으로 유사도 비교를 한다. 해당 연구에서 제시한 주요 블록은 API나 라이브러리 함수 혹은 사용자 지정함수의 호출이 포함된 코드이다. 제안된 방법으로 유사도 비교를 할 경우 비교 대상이 되는 블록들 중 불필요한 블록들을 줄일 수 있어 데이터베이스의 성능문제 없이도 유사도 비교를 할 수 있다. 하지만 악성코드를 제작한 환경에 따라 같은 코드를 생성하더라도 프로그램이 달라질 수 있다. 가령 함수에서 스택을 할당하기 위한 크기가 컴파일러 옵션에 따라 배수가 달라진다던가, 함수 간에 이동을 위한 상대 주소가 바뀐 상태에서 프로그램이 생성될 경우 서로 다른 함수로 인식하기 때문에 정확한 분석이 어렵다.

BitShred[5]는 유사도 비교를 통해 새로 접수된 악성코드가 기존 악성코드들 중에서 재사용된 코드가 있는지 찾아내고 유사한 악성코드들끼리 분류할 수 있는 기술을 제안하였다. 유사도 비교를 위해 악성코드를 정적·동적 분석을 통해 악성코드의 특징들을 추출한다. 이 연구에서 정적 분석방법으로는 n-gram 코드 분석을, 동적 분석방법은 행위 분석 기반을 채택하였다. 앞서 추출한 특징들을 Jaccard index[6]라는 계산식으로 유사도를 측정한다. 그 다음 악성코드 군집들 간에 유사도 비교를 통해 신규 악성코드의 군집을 분류한다. 계속 등장하고 있는 신

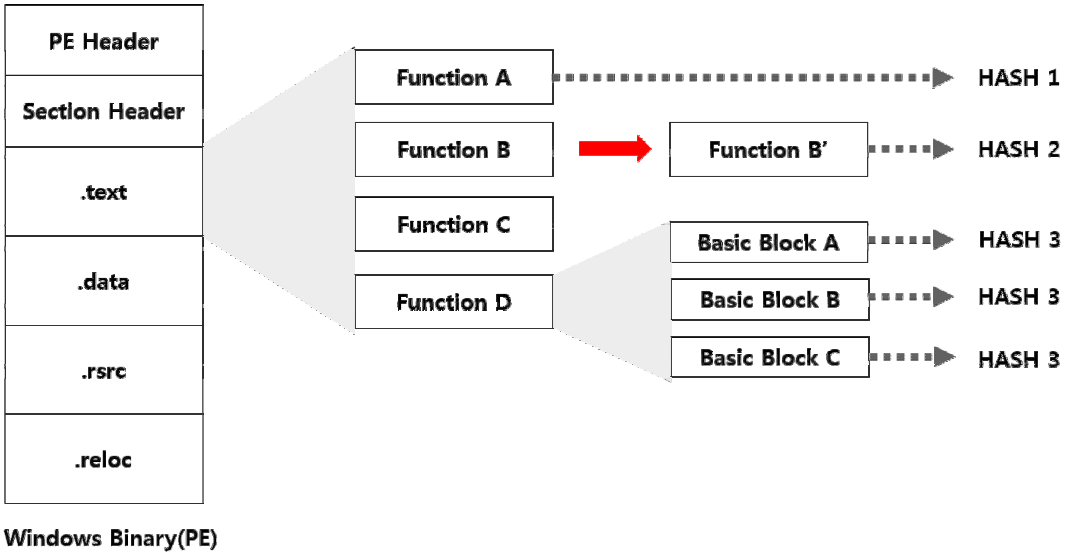



Fig. 2. Generation of hash values for a PE file

규 악성코드를 처리하기 위해선 하루에 약 78만개 정도의 악성코드를 처리해야 한다. 기존 유사도 비교 프로그램인 SimMetrics[7]를 사용할 경우, 10만대 이상의 컴퓨터가 필요하지만, 제안한 방법을 사용할 경우 45대의 컴퓨터로 처리할 수 있다고 소개하였다. 최근 BitShred 기술에 Hadoop[8]을 적용하여 더 빠른 속도로 악성코드 클러스터링을 수행 할 수 있다. 하지만 악성코드의 특징을 추출하는 과정에서 n-gram을 사용할 경우 변수 n에 따라 유사도에 영향을 받을 수 있을 뿐만 아니라[9][10], 동적 분석을 수행하는 과정에서 악성코드가 특정 시간에만 동작할 경우 분석을 위해 특정 시간까지 기다리거나, 아니면 악성 행위를 탐지하지 못하는 문제점이 존재한다.

따라서 본 연구에서는 정적분석 환경에서 유사도 비교를 수행 시 악성코드 제작 환경에 크게 영향을 받지 않는 방법에 대하여 제시하고자 한다.

## III. 유사도 추출 모듈

### 3.1 함수 분할 방법

코드영역에서 함수를 분할하는 방법은 크게 '프롤로그 분석' 단계와 '에필로그 분석' 단계로 나누어진다. 프롤로그 분석 단계는 함수를 시작하기 위해 스택 프레임을 생성하는 과정인 프롤로그 명령어를 기준으로 함수의 시작 위치들을 찾아나가는 과정이고, 에필로그 분석 단계는 함수에서 빠져나가기 위해 스택 프레임을 해제하는 과정인 에필로그 명령어를 기준으로 찾아나가는 과정이다.

#### 3.1.1 프롤로그 분석

프롤로그 분석 단계는 코드영역을 대상으로 함수의 프롤로그 코드들을 찾아나가는 과정이다. 대표적인 프롤로그 명령어는 x86 명령어 기준, 'push ebp, mov ebp, esp'가 있다. 본 단계에서는 코드영역 처음부터 프롤로그 명령어를 기준으로 찾아나간다[11].

우선 Windows 실행파일 내에 실제 명령어가 포함되어 있는 코드영역을 찾는다. 컴파일러에 따라 코드영역의 위치가 달라질 수 있으며, 잘못된 코드영역을 가리킬 경우 함수를 제대로 나눌 수 없기 때문에 정확한 코드영역을 찾는 과정이 필요하다. Windows 실행파일 구조에는 프로그램이 실행되었을 때 맨 처음 명령어를 시작할 주소가 포함되어 있으며, 프로그램을 실행할 때 해당 필드를 참고하여 명령어를 실행한다. 하지만 이 필드는 파일 오프셋이 아닌 메모리에 프로그램이 로드되었을 때 상대주소(Relative Virtual Address)로 표현되어 있으므로, 실제 파일 오프셋 위치로 계산하여 코드영역을 구한다.

그 다음 함수를 벗어나는 RET 명령어 다음에 프롤로그 코드가 나올 경우 프롤로그 시작 주소를 저장한다. 정적 분석환경에서는 Hex 값으로 프롤로그 명령어를 찾게되는데 이 과정에서 코드 내에 상수가 프롤로그 명령어의 Hex 값과 일치할 경우 함수의 시작주소로 잘못 판단할 수 있다. 따라서 단순히 프롤로그 명령어뿐만 아니라 그 전후로 나올 수 있는 명령어를 검색하여 프롤로그를 찾을 때 함수의 시작주소를 잘못 찾을 수 있는 문제점을 해결할 수 있다.

프롤로그 명령어 검색이 끝나면 코드 내 CALL 명령어의 대상 주소를 찾아 저장한다. CALL 명령어는 함수를 호출할 때 사용하는 명령어로 JMP 명령어와 비교했을 때 두 명령어 모두 코드를 건너뛴다는 점에서 공통점을 갖는다. 하지만 CALL 명령어는 함수를 수행한 다음 되돌아갈 주소와 함수를 데이터를 스택에 저장한 다음 대상 주소로 이동한다는 점에서 JMP 명령어와 차이점을 갖는다. 다만 CALL 명령어가 공유 라이브러리 내 함수를 호출하는 경우 실제 코드가 프로그램 내에 존재하지 않고, JMP 명령어를 통해 실제 함수로 호출하는 방식으로 이루어져있기 때문에 제외한다. 앞서 설명한 방식들로 찾은 함수의 시작주소들을 저장하여, 에필로그 분석을 수행할 때 기준 주소로 사용한다.

#### 3.1.2 에필로그 분석

에필로그 분석은 함수의 끝을 찾아 그 크기를 정하고 프롤로그 분석 단계에서 나눠지지 않은 함수들을 추가적으로 나누는 과정이다. 대표적인 에필로그 명령어로 'ret'가 있으며, 프롤로그 분석 단계에서 찾은 함수의 시작주소부터 에필로그 명령어를 찾아나간다. 에필로그 분석을 통해 함수의 크기를 구하고 함수의 크기만큼의 바이트 코드들로 해쉬값을 생성한다. 함수의 크기가 정확하지 않으면 생성되는 해쉬값이 달라져 정확한 비교를 수행할 수 없기 때문에 정확한 함수의 크기를 구해야 한다. 에필로그 분석도

Hex 값으로 에필로그 명령어를 찾기 때문에, 코드 내의 상수가 에필로그 명령어와 동일하면 함수의 끝을 잘못 판단할 수 있는 문제가 발생 할 수 있다. 따라서 프롤로그 단계와 마찬가지로 추가적인 조건을 두어 문제를 해결하여야 한다. 에필로그 명령어 다음 나타나는 코드의 형태는 크게 2가지로 나눌 수 있다.

첫 번째는 '바이트 정렬'이다. 컴파일러가 기계어 코드를 생성할 때 함수의 크기를 일정 배수 단위로 맞추기 위해 함수의 끝에 정렬코드를 삽입한다. 정렬 코드를 삽입하는 이유는 CPU가 메모리에서 데이터를 로드할 때 CPU가 한 번에 처리할 수 있는 단위로 함수의 크기를 맞추면, 더 빠른 처리를 할 수 있기 때문이다. 컴파일러마다 바이트 정렬 코드는 각각 다르지만 Borland Delphi 2.0 기준으로 '0x8d4000', '0x8bc0', '0x90' 등이 있다. 따라서 ret 명령어 다음 정렬 코드가 포함되어 있으면 함수의 끝으로 판단 할 수 있다.

두 번째는 'Callee-saved register'이다. 모든 함수가 프롤로그 코드로 시작하는 것은 아니다. 코드 작성 단계에서 '__declspec( naked ) int func( formal_parameters ) {}'[12]와 같은 형태로 작성하면 스택 프레임 없이 함수를 생성할 수 있다. 이 때 프롤로그 명령어 대신에 Callee-saved register(EBX, EDI, ESI)와 EBP, ESP 레지스터가 스택에 삽입되는데 ret 명령어 다음에 Callee-saved register가 스택에 삽입되면 함수의 끝으로 판단할 수 있다[13]. 따라서 에필로그 분석 단계를 통해 함수에 프롤로그 코드가 없더라도 새로운 함수를 나눌 수 있다.

### 3.2 함수 정보 추출

앞서 프롤로그/에필로그 분석 단계를 통해 나눈 함수에서 해쉬값을 추출한다. 생성된 해쉬값은 데이터베이스에 저장되며, 추후 새로운 악성코드가 접수되었을 때 기존의 악성코드들과 비교하여 유사도를 계산하는데 사용된다. 함수 정보는 한 함수에서 세 가지 정보를 추출하는데 원본 함수 해쉬, 상수제거 함수 해쉬, 코드블록 해쉬로 나뉜다.

#### 3.2.1 원본 함수 해쉬

원본 함수 해쉬는 함수의 원본 바이트 코드로 생성된다. 원본 함수 해쉬값으로 비교하는 것만으로는 정확한 유사도 비교가 이루지지 않을 수 있는데, 가령 악성코드 제작자가 악성코드를 제작할 때 URL을 포함시켰을 경우 이를 대응하는 기관에서는 악성코드를 분석한 뒤 해당 URL을 차단할 것이다. 이에 악성코드 제작자는 기존 URL을 다른 URL로 바꿔서 재유포할 경우, URL이 포함되어 있는 함수의 해쉬값이 달라져 결과적으로 다른 파일로 인식하게 된다. 이처럼 원본함수는 코드의 일부가 바뀌게 되면 다른 파일로 인식되는 문제점이 있다.

#### 3.2.2 상수제거 함수 해쉬

상수제거 함수는 원본 함수에서 발생할 수 있는 문제점을 해결하기 위해 고안한 방법으로 앞서 컴파일러에 따라 값이 변할 수 있는 상수나 주소 등을 제거한 상태로 함수 정보를 추출한 형태이다. 이를 통해 개발환경에 영향을 받지 않고, 함수의 동작 순서만을 가지고 비교할 수 있기 때문에 보다 정확한 유사도 비교를 할 수 있다.

Fig.3.와 Fig.4.는 원본 함수와 상수제거 함수에서 해쉬를 추출하는 과정이다. 왼쪽은 해쉬값을 구하고자 하는 어셈블리 코드이고, 오른쪽은 해쉬값을 구하기 위한 데이터의 모습이다. Fig.3.는 원본 함수에서 해쉬를 구하는 과정이며 원본 함수의 경우 코드의 바이트를 가지고 해쉬값을 생성한다. Fig.4.은 상수제거 함수에서 해쉬를 구하는 과정이며, Fig.4.의 (1)~(5)는 앞서 설명한 바뀔 수 있는 상수 혹은 주소를 가리킨다. 상수제거 함수에서는 이러한 값들은 생략한 상태로 해쉬값을 생성한다.

### 3.3 코드블록 해쉬

코드블록 해쉬는 상수제거 함수에서 기본 블록 단위로 나누어진 형태를 갖는다. 기본 블록은 시작부터 끝까지 순차적으로 수행되는 명령어들의 범위로 분기를 가지지 않는 것이 특징이다. 기본 블록은 원본 함수를 통해 생성되지만, 코드블록은 원본 함수에서 변형이 가해진 상수제거 함수에서 생성되었다는 점에서 차이를 갖는다. 만약 악성코드 제작자가 의도적으로 코드의 일부분을 변경했을 때, 변종 악성코드에 대해 함수 단위에서 비교하면 변경된 부분은 다른 함수로 인식하여 발견하지 못한다. 하지만 코드블록 단위에서 유사도를 비교할 경우 함수보다 작은 단위인 코드블록을 통해 함수의 일부분이라도 유사한 경우를 찾

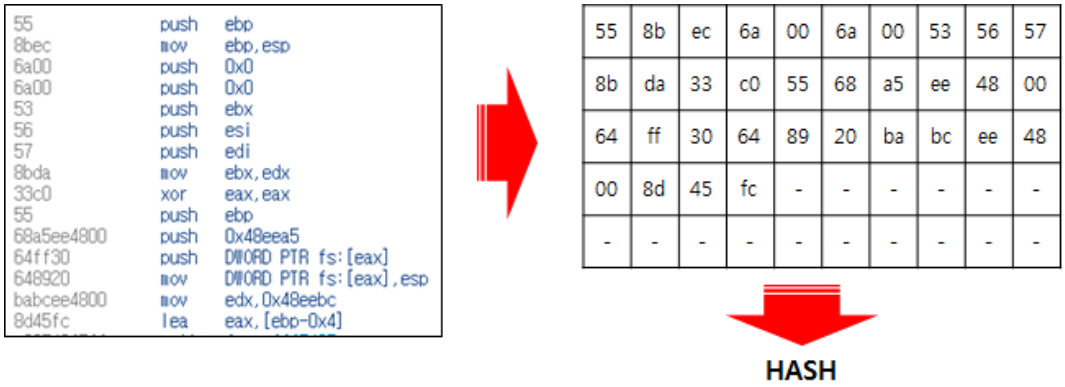



Fig. 3. Computing the hash value for an original function

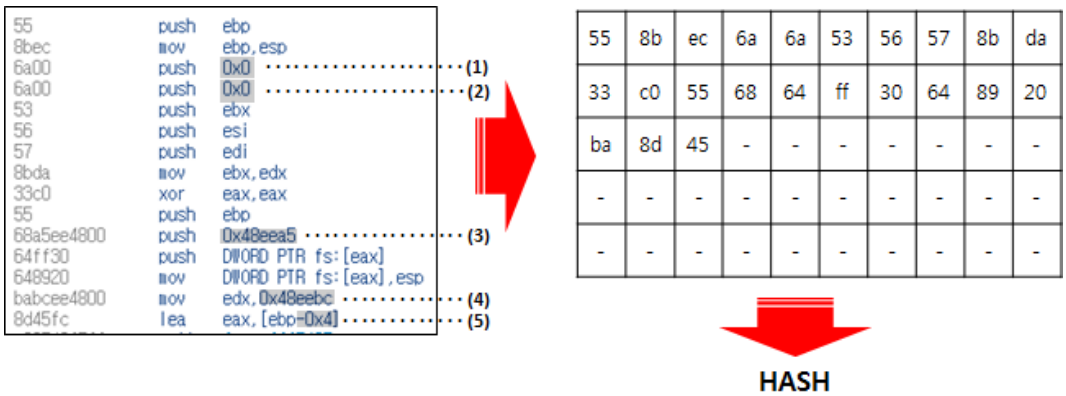



Fig. 4. Computing the hash value for a constant-free function

을 수 있다. 마찬가지로 의도적으로 실행 순서가 바뀐 변종 악성코드의 경우에도 코드블록 단위에서 유사도를 찾을 수 있다. 이를 통해 코드의 일부가 변경되어도 정확한 유사도 비교가 가능하다.

코드블록 해쉬를 생성하기 위한 기본 블록을 구하는 알고리즘을 다음과 같다[14].

1. *함수의 첫 번째 명령어는 기본 블록의 시작주소가 된다.*
2. *goto/jmp 명령어의 타겟 주소가 기본 블록의 시작주소가 된다.*
3. *goto/jmp 명령어 바로 다음에 나오는 명령어가 기본 블록의 시작주소가 된다.*

앞서 제시한 3가지 기준으로 기본 블록을 나눈 뒤 코드블록 해쉬를 생성한다.

## IV. 실험 및 고찰

앞서 제안한 함수 분할 방법과 함수 정보 추출 방법으로 유사도 비교 실험을 수행하였다. 실험은 크게 두 가지로 나누어 진행하였는데, 첫 번째는 원본 함수에서 유사성을 발견하지 못했지만 상수제거 함수에서 유사성을 발견한 경우이고, 두 번째는 상수제거 함수에서 유사성을 발견하지 못했지만 코드블록 단위에서 유사성을 발견한 경우이다. 이 장에서 사용된 해쉬값은 모두 SHA-256이다.

### 4.1 상수제거 함수에 대한 검증

함수 내의 상수 혹은 주소를 제거한 상태에서 함수 간 비교를 수행할 경우 유사성을 찾을 수 있다. 실험은 코드블록 유사도가 높은 두 파일을 실험대상으로 선정하였고, 대상이 되는 파일 A와 파일 B의 정보는 Table 1.에 설명하였다. Table 1.의 'Ori'

는 원본 함수(Original function), 'Con'은 상수제거 함수(Constant-free function), 'Cdb'는 코드블록(Code block)을 가리킨다. 'Function count'는 제안한 엔진으로 함수를 나누었을 때 나누어지는 개수를 가리킨다. 'Identical count'는 해당 파일을 기준으로 유사도 비교를 수행하였을 때 일치하는 원본 함수, 상수제거 함수, 코드블록의 개수이다. 파일 내에는 동일한 함수 정보 해쉬를 가지고 있는 경우가 존재하며, 두 파일의 Identical count는 다르게 나올 수 있다.

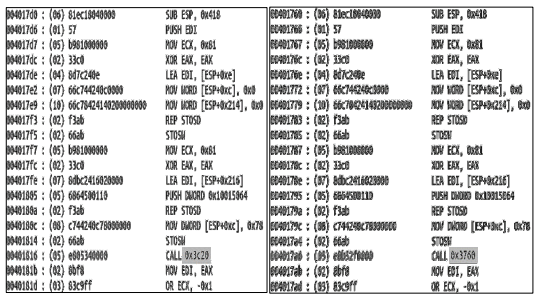
두 파일에 대한 분석 내용 일부를 Table 2.와 Fig.5.에 제시하였다. Table 2.에는 비교하고자 하는 함수들의 해쉬 정보를 제시하였고, Fig.5.는 두 파일의 코드 일부를 제시하였다. Fig.5. 내 표시된 부분이 원본 함수 해쉬에 영향을 줄 수 있으며, 이를 제거한 후 비교했을 때 같은 해쉬값을 갖게 될 경우 두 함수는 유사하다고 할 수 있다. 따라서 악성코드의 기능 위주로 유사도를 비교하기 위해선 상수 혹은 주소의 제거가 필요하다.

Table 1. Function-level comparison of file A and file B

| | | File A | File B |
|---|---|---|---|
| Hash | | 389ee41249... | 2e78d36de1... |
| Detection Name | | Trojan.Win32.Destover.i | UDS:DangerousObject.Multi.Generic |
| Function count | Ori | 310 | 314 |
| | Con | 310 | 314 |
| | Cdb | 4457 | 4411 |
| Identical count | Ori | 71(23%) | 71(23%) |
| | Con | 273(88%) | 273(87%) |
| | Cdb | 4270(96%) | 4264(97%) |

Table 2. A part of hash value file A and file B

| | File A | File B |
|---|---|---|
| Original Function | 219b935571... | 14667c2ad2a... |
| Constant-free Function | 2e2dd0a2ad... | |




(a) File A (b) File B

Fig. 5. A part of code file A and file B

## 4.2 코드블록에 대한 검증

코드의 순서나 코드 내용의 일부가 의도적으로 변경되면 상수제거 함수 단위에서 비교를 하면 유사성을 찾을 수 없지만, 코드블록 단위에서 비교할 경우 기본 블록 단위에서 유사도를 찾을 수 있다. 실험 대상이 되는 파일 C와 파일 D는 Table 3.에 설명하였다. Table 3.의 'Ori'는 원본 함수, 'Con'은 상수제거 함수, 'Cdb'는 코드블록을 가리킨다. Function count는 제안한 엔진으로 함수를 나누었을 때 나누어지는 개수를 가리킨다. Identical count는 각 파일을 기준으로 유사도 비교를 수행하였을 때 일치하는 원본 함수, 상수제거 함수, 코드블록의 개수이다. 파일 내에 동일한 함수 정보 해쉬값을 가지고 있는 경우가 존재하며, 두 파일의 Identical count는 다르게 나올 수 있다.

함수 전체로는 다르더라도 코드블록이 90% 이상 일치하는 함수들도 7개나 찾을 수 있었다. 결과의

Table 3. Code block-level comparison of file C and file D

| | | File C | File D |
|---|---|---|---|
| Hash | | e5d9c8d720... | e1315369d0... |
| Detection Name | | HEUR:Trojan.Win32.Generic | Trojan.Win32.Agent.almso |
| Function count | Ori | 682 | 772 |
| | Con | 682 | 772 |
| | Cdb | 7025 | 7846 |
| Identical count | Ori | 378(55%) | 380(49%) |
| | Con | 661(97%) | 665(86%) |
| | Cdb | 6944(99%) | 7136(91%) |

상세 내용은 Table 4.에 제시하였다.

Table 4.의 Code block은 함수가 가지고 있는 코드블록의 수를 나타낸 것이고, Identical block은 두 상수제거 함수의 코드블록을 비교했을 때 위의 코드블록을 기준으로 일치한 코드블록의 개수이다. Fig.6.은 Table 4.의 4번째 사례의 코드블록을 비교한 결과의 일부이다. 구분한 부분은 이 함수의 코드블록이며, 서로 다른 해쉬값을 가지는 코드블록들은 색으로 표시하였다. 이를 통해 (a)의 0x00409830부터 0x00409850까지, (b)의 0x00409ae0부터 0x00409b00까지 코드블록 일부가 다른 것을 확인하였고, 표시된 부분을 제외한 대부분의 코드블록이 일치하는 것을 확인할 수 있었다. 이를 통해 유사도 관점에서는 일부 코드가 변경되었더라도 대부분의 코드가 동일하다면 유사할 가능성이 있기 때문에 어느 정도의 코드블록이 유사한지 분석하는 것은 의미가 있다.

Table 4. Result of experiment 4.2

| No. | Hash | Code block | Identical block |
|---|---|---|---|
| 1 | a7410a5db0… | 24 | 22 |
| | c1875c9c6d… | 25 | |
| 2 | 92ca1f1e82… | 34 | 33 |
| | 2c97b95af3… | 35 | |
| 3 | 87ce0391a7… | 129 | 124 |
| | 835797ddb9… | 148 | |
| 4 | 9017323b9f… | 229 | 228 |
| | ec4ff0e855… | 229 | |
| 5 | 0eda26bb64… | 11 | 10 |
| | 76a8a87f6e… | 7 | |
| 6 | 1304692c7b… | 21 | 20 |
| | 7663ab38f0… | 21 | |
| 7 | 626ebd13e6… | 185 | 184 |
| | 26897ad8fd… | 185 | |

```
0040980f : (02) 740a                    JZ 0x881b
00409811 : (06) 8b85acfdffff            MOV EAX, [EBP-0x254]
00409817 : (04) 836070fd                AND DWORD [EAX+0x70], -0x3
0040981b : (06) 8b85d8fdffff            MOV EAX, [EBP-0x228]
00409821 : (03) 8b4dfc                  MOV ECX, [EBP-0x4]
00409824 : (01) 5f                      POP EDI
00409825 : (01) 5e                      POP ESI
00409826 : (02) 33cd                    XOR ECX, EBP
00409828 : (01) 5b                      POP EBX
00409829 : (05) e8cbc4ffff              CALL 0x4cf9
0040982e : (01) c9                      LEAVE
0040982f : (01) c3                      RET
00409830 : (01) 90                      NOP
00409831 : (02) 3c90                    CMP AL, 0x90
00409833 : (01) 40                      INC EAX
00409834 : (06) 003d8e40006d            ADD [0x6d00408e], BH
0040983a : (03) 8e4000                  MOV ES, [EAX+0x0]
0040983d : (01) cb                      RETF
0040983e : (03) 8e4000                  MOV ES, [EAX+0x0]
00409841 : (01) 17                      POP SS
00409842 : (03) 8f4000                  POP DWORD [EAX+0x0]
00409845 : (06) 228f4000688f            AND CL, [EDI-0x7097ffc0]
0040984b : (01) 40                      INC EAX
0040984c : (01) 00                      DB 0x0
0040984d : (01) 96                      XCHG ESI, EAX
0040984e : (01) 90                      NOP
0040984f : (01) 40                      INC EAX
00409850 : (01) 00                      DB 0x0
```

(a) File A

```
00409abf : (02) 740a                    JZ 0x8acb
00409ac1 : (06) 8b85acfdffff            MOV EAX, [EBP-0x254]
00409ac7 : (04) 836070fd                AND DWORD [EAX+0x70], -0x3
00409acb : (06) 8b85d8fdffff            MOV EAX, [EBP-0x228]
00409ad1 : (03) 8b4dfc                  MOV ECX, [EBP-0x4]
00409ad4 : (01) 5f                      POP EDI
00409ad5 : (01) 5e                      POP ESI
00409ad6 : (02) 33cd                    XOR ECX, EBP
00409ad8 : (01) 5b                      POP EBX
00409ad9 : (05) e8cbc4ffff              CALL 0x4fa9
00409ade : (01) c9                      LEAVE
00409adf : (01) c3                      RET
00409ae0 : (01) 90                      NOP
00409ae1 : (01) ec                      IN AL, DX
00409ae2 : (01) 92                      XCHG EDX, EAX
00409ae3 : (01) 40                      INC EAX
00409ae4 : (02) 00ed                    ADD CH, CH
00409ae6 : (01) 90                      NOP
00409ae7 : (01) 40                      INC EAX
00409ae8 : (06) 001d9140007b            ADD [0x7b004091], BL
00409aee : (01) 91                      XCHG ECX, EAX
00409aef : (01) 40                      INC EAX
00409af0 : (02) 00c7                    ADD BH, AL
00409af2 : (01) 91                      XCHG ECX, EAX
00409af3 : (01) 40                      INC EAX
00409af4 : (02) 00d2                    ADD DL, DL
00409af6 : (01) 91                      XCHG ECX, EAX
00409af7 : (01) 40                      INC EAX
00409af8 : (02) 0018                    ADD [EAX], BL
00409afa : (01) 92                      XCHG EDX, EAX
00409afb : (01) 40                      INC EAX
00409afc : (03) 004693                  ADD [ESI-0x6d], AL
00409aff : (01) 40                      INC EAX
00409b00 : (01) 00                      DB 0x0
```

(b) File B

Fig. 6. Compared code blocks for File A, B

## V. 결 론

본 논문에서는 변종 악성코드의 유사도 비교를 위하여 Windows 악성코드를 대상으로 함수를 나누는 방법과 나눈 함수들에서 함수 정보를 추출하는 방법에 대해 제안하였다. 본 연구에서 제안한 상수제거 함수와 상수제거 함수를 기본 블록 단위로 나눈 코드블록을 통해 유사도 비교를 할 경우 함수 단위로 비교했을 때 찾을 수 없었던 숨겨진 유사도를 찾을 수 있는 것을 확인하였다.

추후 연구에서는 여러 종류의 악성코드를 수집한 뒤, 수집한 악성코드들을 대상으로 파일들 간에 유사도를 측정할 수 있는 방법에 대해 연구하고, 여러 샘플들을 이용하여 다양한 실험을 하고자 한다. 또한 분석을 수행하는 악성코드의 수가 많아질수록 비교를

하는데 있어 엔진의 성능에 영향을 받을 수 있다. 따라서 엔진의 성능 개선에 관한 연구도 수행할 예정이다.

## References


[1] Kyoung-Soo Han, In-Kyoung Kim, and Eul-Gyu Im, "Malware Family Classification Method using API Sequential Characteristic," Journal of Security Engineering, 8(2), pp. 319-335, Apr. 2011.

[2] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," ACM Computing Surveys (CSUR), vol. 44.2, no. 6, pp. 6:1-6:42, Feb. 2012.

[3] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel and Engin Kirda, "Scalable, Behavior-Based Malware Clustering," Proceedings of the NDSS Symposium 2009, pp.8-11, Feb. 2009.

[4] TaeGuen Kim, In-Kyoung Kim, and Eul-Gyu Im, "Malware Detection Method via Major Block Comparison," Journal of Security Engineering, 9(5), pp. 401-416, Oct. 2012.

[5] Jiyong Jang, David Brumley, and Shobha Venkataraman, "BitShred: feature hashing malware for scalable triage and semantic analysis," Proceedings of the 18th ACM conference on Computer and communication security. ACM, pp. 309-320, Oct. 2011.

[6] Jaccard index, http://en.wikipedia.org/wiki/Jaccard_index

[7] SimMetrics, http://sourceforge.net/projects/simmetrics/

[8] Hadoop, http://hadoop.apache.org/docs/current/

[9] Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo G.Bringas, "N-grams-based File Signatures for Malware Detection," Proceedings of the 11th International Conference on Enterprise Information Systems, pp. 317-320, May. 2009.

[10] Abdurrahman Pektas, Mehmet Eris, and Tankut Acarman. "Proposal of n-gram based algorithm for malware classification," Proceedings of the 5th International Conference on Emerging Security Information, Systems and Technologies. pp. 14-18, Aug, 2011.

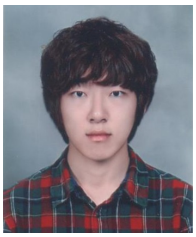
[11] x86 Assembly, http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

[12] Harris Laune C., and Barton P. Miller, "Practical analysis of stripped binary code," ACM SIGARCH Computer Architecture News, vol. 33, no. 5, pp. 63-68, Dec. 2005.

[13] MSDN, http://msdn.microsoft.com/ko-kr/library/dabb5z75.aspx

[14] Basic block, http://en.wikipedia.org/wiki/Basic_block

## 〈저자소개〉



박 찬 규 (Chan-Kyu Park) 학생회원
2014년 2월: 충남대학교 컴퓨터공학과 학사
2014년 3월~현재: 충남대학교 컴퓨터공학과 석사과정
〈관심분야〉 시스템 보안, 악성코드 분석, 디지털 포렌식

김 형 식 (Hyong-Shik Kim) 종신회원
1988년 2월: 서울대학교 컴퓨터공학과 졸업
1990년 2월: 서울대학교 컴퓨터공학과 석사
1997년 8월: 서울대학교 컴퓨터공학과 박사
1999년 9월~현재: 충남대학교 컴퓨터공학과 교수
〈관심분야〉 인터넷보안, 컴퓨터시스템구조

이 태 진 (Tae jin Lee) 정회원
2003년 2월: 포항공과대학교 컴퓨터공학과 졸업
2008년 2월: 연세대학교 컴퓨터공학 석사
2003년~현재: 한국인터넷진흥원 팀장
〈관심분야〉: 네트워크 보안, 시스템 보안, 악성코드 탐지 및 분석

류 재 철 (Jae-Cheol Ryou) 종신회원
1985년 2월: 한양대학교 산업공학과 졸업
1988년 5월: Iowa State University 전산학 석사
1990년 12월: Northwestern University 전산학 박사
1991년 2월~현재: 충남대학교 컴퓨터공학과 교수
〈관심분야〉 정보보호, 네트워크보안, 암호학, 보안프로토콜