메모리 액세스 로그 분석을 통한 프로그램 표절 검출

박 성 윤[†] · 한 상 용[††]

요 약

프로그램 소스코드를 표절하는 것은 소프트웨어의 지적재산권을 침해하는 행위이다. 표절을 감추기 위하여 소스코드의 구조를 일부 바꾸거나 변수 명을 바꾸는 등의 수단을 쓰기도 하기 때문에 표절을 검출하기란 쉽지 않은 일이다. 표절 행위를 막기 위해 이를 검출하기 위한 여러 가지 방법들이 고안되었으며 대부분 프로그램의 소스코드를 다양한 방법으로 분석하여 표절 검출하려고 한다. 본 논문에서는 소스 코드에 기반하지 않고 실행중인 프로그램의 메모리 액세스 로그 분석을 통해 알고리즘 유사도를 측정하여 프로그램의 표절 여부를 검출하는 새로운 방법을 제시한다. 두 프로그램의 메모리 액세스 로그는 일반적인 스트링 비교 알고리즘을 이용하여 분석될수 있으며, 본 논문에서는 일반적인 방법보다 특성화된 방법을 제시한다. 이를 통해 메모리 액세스 로그가 프로그램의 표절 검출도구로서 사용되어질 수 있음을 보여준다.

키워드 : 프로그램 표절, 알고리즘 유사도, 메모리 액세스 로그

Program Plagiarism Detection through Memory Access Log Analysis

Sungyun Park[†] · Sangyong Han[††]

ABSTRACT

Program Plagiarism is an infringement of software copyright. In detecting program plagiarism, many different source program comparison methods has been studied. But, it is not easy to detect plagiarized program that made a few cosmetic changes in program structures and variable names. In this paper, we propose a new ground-breaking technique in detecting plagiarism by Memory Access Log Analysis.

Key Words : Program Plagiarism, Algorithm Similarity, Memory Access Log

1. 서 론

지적재산권의 보호와 관련하여 소프트웨어의 저작권 문제는 여러 가지의 형태를 띄고 있다. 제작된 소프트웨어의 불법적인 사용 및 배포에 관한 문제도 있고, 소프트웨어 제작에 있어서 프로그램 표절(program plagiarism)에 의한 저작권 침해 문제가 있다. 불법 소프트웨어의 사용문제는 그 현상이 뚜렷하여 판단하기 쉽다. 불법 사용자를 찾아내기는 쉬운 일이 아니지만 어떤 상황을 놓고서 소프트웨어의 불법 사용인지 아닌지를 고민해야하는 경우는 없다. 이는 소프트웨어 사용 계약상의 문제이므로 결론이 뚜렷하다. 하지만 프로그램 표절 문제는 표절 여부를 가리는 것 자체가 큰 문제이다. 본 논문에서는 표절에 의한 소프트웨어 개발물을 검출할 수 있도록 두 프로그램의 유사도를 측정하는 새로운 방법을 제시하고자 한다.

지금까지 프로그램 유사도를 측정하기 위한 여러 가지 방법들이 고안되었고 사용되고 있다[1-7]. 이 방법들은 프로그램의 소스코드(source code)를 다양한 방법으로 분석하고, 수치화시켜서 이를 비교하는 방법이다. 소프트웨어 계량법(software metrics)을 이용하거나[1, 2] 소스코드의 토큰(token) 나열을 비교하거나[3], 함수호출 관계를 비교하거나[4], 프로그램 트리(tree) 구조를 분석하기도[5] 한다. 지금까지 연구되어진 방법들의 비교대상은 두 프로그램의 소스코드이다.

본 논문에서는 소스코드에 의한 비교방법에 의해서가 아닌, 실행중인 소프트웨어의 동작으로부터 정보를 추출해 두 소프트웨어의 알고리즘 유사도를 측정해 볼 수 있는 방법을 제안한다.

2. 관련 연구

다양한 방법으로 두 프로그램 소스코드를 비교하여 결과를 보여주는 방법들이 많이 연구되었다. Plague[8], YAP3[3]

※ 이 논문은 IITA(Institute of Information Technology Assessment)에서 후원하는 홈 네트워크 연구 센터(HNRC-ITRC (Home Network Research Center) 산하 중앙대학교 MIC(Ministry of Information and Communication)의 연구비 지원에 의한 것임.
† 준 회 원 : 중앙대학교 컴퓨터 공학과 석사과정
†† 종신회원 : 중앙대학교 컴퓨터 공학과 교수
논문접수 : 2006년 7월 2일, 심사완료 : 2006년 9월 7일

과 같이 두 소스코드의 어휘를 분석하여 토큰을 나열하고, 나열된 두 토큰 스트링(string)을 잘 알려진 스트링 비교 알고리즘을 이용하여 유사도를 판단하는 비교방법이 있다. 스트링 비교 알고리즘으로 GST-RKR[9] 알고리즘을 사용하고 있다. MOSS[8]에서는 문서 지문화(document fingerprinting)[6]를 이용하였고, 프로그램 트리구조를 이용하는 Clonechecker[5]도 있다. 최근에는 다른 언어로 작성된 프로그램간의 비교를 위한 연구가 되고 있으며, 웹 응용 프로그램과 같이 해당 언어의 특징을 이용하여 프로그램을 비교하는 연구도 되어지고 있다.

3. 메모리 사용의 관점에서 소프트웨어의 동작

소스코드로부터 컴파일(compile)되고 링크(link)된 바이너리 실행파일(binary executable file)은 ELF[10], PE[11]등과 같은 형식에 따라 운영체제(operating system)에 의해 메모리에 로드(load)될 때 지정된 메모리 영역이 확정되어있다. 소스코드 혹은 컴파일러(compiler)에 의해 명시된 메모리 사용 명세에 따라 영역이 적절히 나누어진다. 여러 가지 영역이 구분될 수 있으나 CPU(Central Processing Unit)가 구분하는 메모리의 영역은 크게 3가지로 나누어 사용하게 된다. 대표적인 PC(Personal Computer)용 CPU인 인텔계열의 CPU는 일반적인 어플리케이션(application) 동작 레벨(privilege level)인 사용자레벨(level 3)에서 코드(code)/데이터(data)/스택(stack)에 해당하는 3가지 종류의 세그먼트 레지스터(segment registrer)를 가지고 있다[12].

코드영역은 특수한 경우가 아니고는 수정되는 일이 없으며, 명령어 인출(instruction fetch)을 위한 읽기 동작 이외에 메모리에 어떠한 동작도 가해지지 않는 것이 보통이다. 데이터 영역은 코드가 동작하는데 필요한 모든 정보를 유지, 가공하고 코드의 실행에 필요한 스택 영역등을 유지하게 된다. 사실상 코드의 실행에 관련된 대부분의 원인과 결과는 데이터 영역의 내용에 기인한다고 볼 수 있다. 심지어 코드 영역도 바이트(byte) 값의 연속에 불과하다.

코드는 여러 가지 용도로 메모리를 사용하게 되지만, 프로그래머(programmer)가 메모리 영역을 사용할 수 있게 하기 위하여 프로그래밍 언어(programming language)는 몇 가지 메모리 사용 방법을 제공하고 있다. 가장 기본이 되는 개념이 변수(variable)이다. 메모리의 특정부분을 변수이름에 매핑(mapping)시켜 놓고, 변수에 값을 써넣는 동작을 지시함으로써 메모리에 값이 쓰여지기를 기대하게 된다[13].

3.1 메모리의 사용

변수의 종류에 따른 메모리 사용방법은 그 바인딩(binding) 형태와 생명주기(life cycle)에 따라 스택변수/정적변수/힙(heap)할당으로 나누어질 수 있다[13].

정적변수는 위에서 언급한 변수의 개념과 정확히 맞아 떨어지는 변수로 주로 전역변수(global variable)로 사용된다. 실행 시 사용하게 될 메모리의 특정 영역을 컴파일 시에 특정 변수이름에 매핑하는 과정을 거쳐 사실상 메모리주소(memory address)를 이용해 직접 액세스(access) 하는 효과를 갖는다.

이에 반해 지역변수(local variable)로 사용되는 스택변수는 프로그램 수행 중에 스택에 동적으로 변수의 생성과 소멸 및 영역의 확보가 일어난다. 따라서 메모리상에 변수의 위치가 계속 달라질 수 있다.

힙할당은 할당과 해제가 동적으로 일어날 수 있는 힙에 할당된다. 그 외에는 정적변수의 사용과 내용이 일치한다고 볼 수 있다. 단 변수이름에 메모리 영역을 매핑하는 것이 아니라, C언어의 포인터(pointer) 변수 등과 같은 수단을 이용해 할당받은 메모리의 주소값을 직접 이용한 메모리 액세스를 하는 경우이다.

4. 알고리즘 비교를 위한 메모리 사용 내역의 비교

4.1 메모리 사용 내역 분석의 원리

두 프로그램의 소스코드를 비교하는 것은 알고리즘이 얼마나 같은지를 비교하여 그 창의적 측면이 얼마나 도용당했느냐를 알려고 하는 것이다. 알고리즘은 문제를 풀어가기 위한 계산의 절차를 의미한다[14]. 따라서 알고리즘이 같다고 하는 것은, 어떤 답을 찾기 위해 계산해나가는 과정, 즉 연산의 절차가 같다는 것을 의미한다. 연산이라는 것은 구체적으로 컴퓨터 CPU의 동작 하나하나를 의미한다. 연산은 CPU 동작의 가장 원천적인 요소인 것이다. 이 원천적인 동작의 1차적인 결과물은 값이다. 두 개의 연산문을 비교할 때 그 동작이 유사 혹은 같다는 것은 같은 입력에 대한 결과 값이 같은지 다른지를 보면 짐작할 수 있다.

요컨대, 두 프로그램 혹은 알고리즘이 얼마나 유사한지를 알기위해 연산의 최종 결과물인 결과 값의 나열을 비교한다.

4.2 메모리 액세스의 다양한 형태

최신의 프로세서(processor)들은 다양한 형태의 명령어(instruction)를 지원하고 있다. 대부분의 명령어는 연산결과의 처리 방법에 따라 다음과 같이 나누어 볼 수 있다.

- 메모리에 씀 - 연산의 결과가 메모리에 쓰여짐
- 레지스터에 씀 - 연산의 결과가 레지스터에 저장됨
- 제어연산 - 코드 수행의 흐름을 변경함, 결과물의 저장 없음.
- 포트 I/O(port input/output) - port로부터 값을 읽거나 씀.

제어연산은 메모리 연산이나 레지스터 연산 등을 수행하는 절차의 변경을 의미할 뿐이다. 결과 값을 생성하는 연산의 결과 값은 프로세서의 레지스터나 메모리, 혹은 하드웨어(hardware)의 특정 포트에 쓰여진다. 레지스터는 메모리로 가기 위한 경유지이거나 연산의 중간 값을 가지기 위한 저장소, 즉 임시저장소의 역할이 강하다. 포트I/O는 그 자체가 대부분이 해당 하드웨어의 동작을 유발시키는 것으로, 그 결과는 어떤 동작을 의미하는 것이다. 포트I/O는 메모리 액세스와 마찬가지로 하나의 값이 읽혀지거나 쓰여지는 것이다. 따라서 본 논문에서는 포트I/O를 따로 논의하지 않고

실험을 위하여 포트I/O를 발생하지 않는 예제 코드를 이용하여 메모리 액세스만으로 결과를 도출한다. 이제 코드의 수행에 따른 결과를 로그(log)로 남긴다면 다음의 형태가 될 수 있을 것이다.

(R|W , address , value),…

앞서 언급한대로 메모리는 여러 영역으로 분할되어있다. 데이터 영역 중에서도 다양한 형태의 영역이 있으며, 메모리를 액세스 하는 코드의 출처도 다양하다. 메모리 액세스의 종류에 따라 다음과 같이 데이터 영역의 종류를 분류해 볼 수 있겠다.

- 지역 데이터 영역(local data area) - 사용자에 의해 작성되고 컴파일되어 수행하게 될 프로그램이 그 프로세스 내에서 접근 할 수 있는 메모리 영역
- 전역 데이터 영역(global data area) - 운영체제 등의 시스템(system) 차원에서 사용하는 데이터 영역
- 공유 데이터 영역(shared data area) - 프로세스(process) 간 공유가 가능한 데이터 영역

여러 특수한 형태의 메모리 액세스 형태가 가능하겠지만 크게 위의 세 가지 범주로 분류할 수 있다.

이들 모두에 대한 논의가 필요할 것이다. 그러나 수집된 메모리 액세스 정보를 위의 종류대로 구분하기도 수월하지 않을뿐더러 본 논문에서는 소스코드를 컴파일 하여 생성된 사용자 프로세스가 메모리를 사용하는 내역이 관심대상이다. 따라서 본 논문에서는 문제를 단순화하고 실험을 용이하도록 하기 위하여 그 논의 대상을 한정하여 '사용자 프로세스'에 의한 '지역 데이터 영역'의 메모리 액세스 로그(memory access log)를 분석한다.

5. 메모리 액세스 로그의 분석

5.1 정보 추출

실제로 소스코드가 아닌 실행파일을 가지고 메모리의 사용 내역을 확인해보기 위해서 특별한 처리를 해주었다. 메모리의 할당을 표준 라이브러리에 의해 힙에서 할당하지 않고 액세스 트랩(access trap)이 설치된 감시가 가능한 메모리영역에서만 할당이 되도록 메모리 할당 루틴(memory allocation function)을 따로 정의하여, 코드의 메모리 액세스 정보를 추출하였다. 즉, 대상 코드에서 메모리 할당과 해제와 관련된 부분을 일부 수정하여 정보를 추출하였다.

전역변수의 사용은 모두 포인터변수에 의해 동적으로 영역을 할당받도록 하여 컴파일시 바인딩 되지 않고 실행시 정해진 영역에서 할당받도록 하였다. 또 다른 문제점으로 지역변수 문제가 있다. 스택에 할당되는 지역변수의 메모리 액세스는 체크하기가 쉽지 않다. 스택에 지역변수 외에 다른 여러 정보들이 들어가게 되며, 다른 기능에 영향없이 스택변수 영역만 트랩을 설치하기 힘들기 때문이다. 미세한 처리가 필요하겠지만, 실험에서는 지역변수는 모두 포인터 변수로 전환하고 모두 특정 힙에서 공간을 할당받아 사용하도록 하였다. 따라서 모든 메모리 사용은 포인터 변수를 통해서만 이루어지도록 하며, 모든 메모리는 주어진 특정 메모리 할당 함수를 이용해서 트랩이 설치된 힙에서만 메모리를 할당받아 사용하도록 하였다. 이제 사용자 코드가 수행중에 액세스 하게 되는 메모리의 모든 정보를 확인할 수 있게 되었다.

5.2 액세스 로그의 분석

두 프로그램에서 추출한 액세스 로그로부터 얻은 값 v들의 나열을 스트링 S_1, S_2라고 하자.

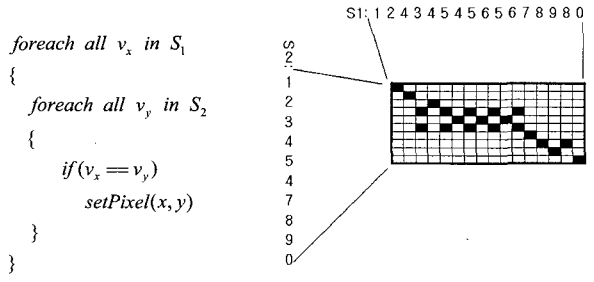
$$S_1 = (v_{1,1}, v_{1,2}, \ldots, v_{1,a})$$
$$S_2 = (v_{2,1}, v_{2,2}, \ldots, v_{2,b})$$

그리고 (그림 1)과 같은 방법으로 두 스트링을 비교하여 시각적으로 확인해 볼 수 있다. 다음은 (그림 1)과 (그림 2)를 구성하기 위한 알고리즘이다.

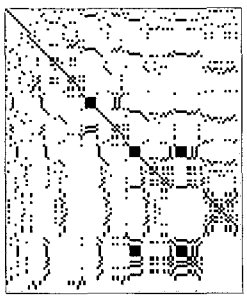
두 값의 나열을 잘 알려진 스트링 비교 알고리즘인 GST 알고리즘[9]에 의해 공통스트링(common string)을 타일링(tiling)하고 검출된 공통스트링 c의 나열을 아래와 같이 기술한다.

$$CS(S_1, S_2) = \{c_1, c_2, c_3, \ldots c_n\}$$
$$c_i = (x_i, y_i, l_i)$$
$$x_i = S_1\text{에서 } c_i\text{의 시작 위치}$$
$$x_i = S_2\text{에서 } c_i\text{의 시작 위치}$$
$$l_i = c_i\text{의 길이}$$

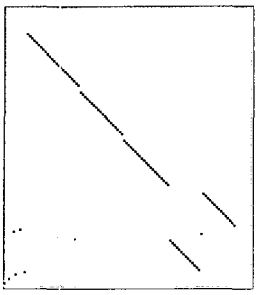
$CS(S_1, S_2)$는 x_i 혹은 y_i에 대해 정렬된 c_i들의 나열이다. GST에 의한 CS의 공통 스트링들로만 이루어진 이미지를



(그림 1) 두 스트링의 공통부분마킹



(그림 2) 실제 데이터의 공통부분 마킹에 의한 이미지



(그림 3) 공통 스트링으로만 이루어진 GST 스트링 타일링 이미지

확인해보면 (그림 3)과 같다.

*CS*로부터 아래와 같이 S_1, S_2의 일치된 정도를 표현하는 m_p값을 계산한다.

$$m_p = \sum_{i=1}^{n} (l_i)^p \quad (1)$$

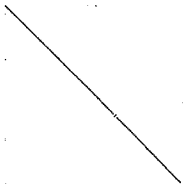
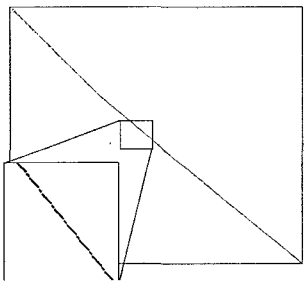
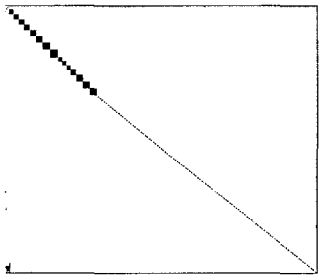
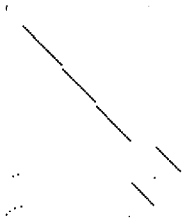
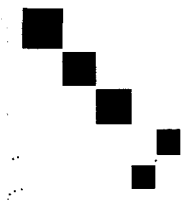
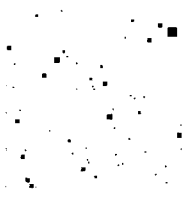
$p=0$일 때, $m_p = n$ 즉, 공통스트링의 개수

$p=1$일 때, m_p = 공통스트링의 총 길이

$p=2$일 때, m_p = 공통스트링 길이들의 제곱합

즉, $p>1$이면 m_p는 공통스트링의 총 길이가 같아도 더 길이가 긴 공통스트링으로 구성된 쪽이 더 큰 값을 갖게 된다. m_p을 정규화 한다면 그 값이 클수록 메모리 액세스 로그는 유사하며, 즉 두 프로그램의 동작은 더욱 유사하다는 의미가 될 수 있다. <표 1>은 실제 네 쌍의 프로그램을 분석한 값과 이미지 이다.

〈표 1〉 실험 결과

실험 대상	$\frac{m_2}{len(S_1)\times len(S_2)}$ / M	GST string tiling 이미지(선)	GST string tiling 이미지(면적)
1	0.514463 0.000344		
2	0.008190 0.005818		
3	0.138358 0.033524		
4	0.014254 0.261336		

1 : 테스트용 표절 소스코드[7]
2 : 테스트용 표절 소스코드(불필요한 메모리 액세스 코드 삽입)
3 : 비슷한 기능의 소스코드(stack이용한 quick sort algorithm, recursive quick sort algorithm)
4 : 전혀 다른 소스코드([7]의 테스트 소스코드중 하나, selection sort algorithm)

5.3 불연속성에 의한 분석

정규화된 m_p를 이용해서 프로그램이 서로 얼마나 유사한지를 추측해 볼 수 있다. 하지만 이 수치는 다음과 같은 문제점이 있다. <표 1>의 첫 번째 그림과 같이 스트링이 거의 같고 단 한군데에서 다른 경우라고 하여도 p=2인 경우 그 값이 반이 되어버리고 p>2인경우는 더 많이 차이난다. 그리고 네 번째 그림과 같이 스트링에서 공통스트링의 위치(프로그램 상에서 실행순서)가 뒤죽박죽이어도 순서대로 공통스트링이 존재하는 경우보다 유사도가 작다는 결론을 내지 못한다. 그리고 두 번째의 경우와 같이 사실상 거의 같은 프로그램인 경우에도 불필요한 메모리 액세스 코드를 군데군데 삽입하여 액세스 로그의 연속성을 일부러 깨트린 경우는 아주 낮은 수치가 나온다.

이 문제는 공통스트링의 양을 계산하지 않고, 공통스트링의 불연속성을 계산하여 극복할 수 있다. 다음은 각 공통스트링들이 서로 얼마나 떨어져 있는지를 계산하는 과정이다. CS는 x 혹은 y에 의해 정렬된 c들의 연속이다. 연이은 두 공통스트링 c_a,c_b에서 c_a의 우측하단을 기준점으로 직각 이동하여 c_b의 좌측상단이 얼마나 떨어져있는지 그 면적거리를 계산한다. CS전체에 걸쳐 이 면적거리를 계산한 후 모두 더하고 이를 전체 면적으로 정규화 하는 과정을 거친다.

$c_{x\max} = (x_{x\max}, y_{x\max}, l_{x\max})$: CS에서 x가 최대가 되는 c

$c_{y\max} = (x_{y\max}, y_{y\max}, l_{y\max})$: CS에서 y가 최대가 되는 c

다음을 $CS'(S_1, S_2)$라고 하고,

$$c_0 = (1,1,0)$$
$$c_{n+1} = (x_{x\max} + l_{x\max} - 1, y_{y\max} + l_{y\max} - 1, 0)$$
$$CS'(S_1, S_2) = CS(S_1, S_2) \cup \{c_0, c_{n+1}\}$$

$CS'(S_1, S_2)$에 대하여, 다음과 같이 불연속 정도를 나타내는 M을 계산한다.

$$M = \frac{\sum_{i=0}^{n} \{|x_{i+1} - (x_i + l_i - 1)| \times |y_{i+1} - (y_i + l_i - 1)|\}}{x_{n+1} \times y_{n+1}} \quad (2)$$

<표 1>에서 계산된 M값을 확인해 볼 수 있다. M은 불연속성에 대한 수치이므로 0에 가까울수록 연속적이라는 의미로서 유사도가 높음을 뜻한다. 완전히 일치하는 두 스트링의 경우 0의 값이 계산되어지고, 공통된 스트링이 전혀 없는 경우는 1의 값이 계산되어 진다. <표 1>에서 보는 바와 같이 공통스트링의 연속성을 계산하기 보다, 반대로 불연속성을 계산하는 것이 더 정확한 결과를 보여주고 있다.

6. 실험 및 결과

실험에서는 표절을 확인하고자하는 한 쌍의 소스코드에 대해서 실행 중 정보추출을 위해 메모리 사용에 관련된 코드에 수정을 가하였다. 프로그램의 흐름이나 소스코드 구조의 변경이 없는 수준의 수정으로 본질적인 실험 결과에 영향이 없도록 하였다. 실험 대상과 그 결과는 <표 1>에 나와 있다.

실험대상 1은 [7]에서 사용되었던 테스트용 표절 소스코드이다. 이는 동작이 완전히 동일한 표절 소스코드로서 사람이 소스코드를 봤을 때 서로 같은 소스코드임을 눈치 채기 힘들게 변형해놓은 예제이다. GST 공통스트링 이미지를 확인했을 때 한 부분을 제외한 메모리 액세스 로그 전체가 완전히 일치했으며, M값을 계산해보면 아주 0에 가까운 값을 가진다.

실험대상 2는 실험대상 1과 마찬가지로 동일한 동작을 하는 표절된 소스코드에 메모리 액세스의 연속성을 저해하는 불필요한 메모리 액세스 코드를 삽입한 예제이다. 메모리 액세스 로그는 거의 연속적이지 않아 m_2를 통해서 표절로 분류하기 힘들지만, M을 통해서 충분히 표절임을 알아낼 수 있다.

실험대상 3은 표절은 아니지만 동작과정이 비슷한 소스코드이다. 이것을 보면 M값이 소스코드의 표절뿐만이 아니라 프로그램 동작의 유사도를 반영해주고 있음을 알 수 있다.

실험대상 4는 전혀 다른 프로그램으로부터 추출한 메모리 액세스 로그를 비교한 것이다. 이미지에서 알 수 있듯이 짧은 공통스트링들이 규칙성 없이 흩어져 있는 것을 알 수 있다.

7. 결론 및 향후 과제

본 논문에서는 표절된 소스코드에서는 메모리 액세스 정보가 서로 비슷한 패턴을 띄게 됨으로서 상호 유사성을 확인할 수 있다는 방법을 제시 하였다. 메모리 액세스의 순서가 바뀌지 않을 수정, 즉 주석문의 삽입, 변수나 함수이름의 변경, 변수나 함수의 선언위치 변경, 매크로의 사용, 함수의 통합 및 분할, 실행되지 않을 불필요코드 삽입 등과 같은 변경[15]을 이용한 표절은 본 논문에서 제안한 방법에 의해 이론상 100% 정확하게 표절을 검출할 수 있다. 이것들 외에 메모리 액세스에 변동이 일어날 수 있는 수정이 가해진다고 해도 그 메모리 액세스 로그가 크게 달라지지는 않는다. 소스코드가 표절된 이상 결국 답을 구하기위해서는 거의 같은 과정을 거칠 것이기 때문이며 본 논문의 방법을 통해서는 불필요한 메모리 액세스의 영향도 어느 정도 견디어낼 수 있다.

실제로 소스코드가 아닌 실행파일을 가지고 메모리의 액세스 로그를 추출하기에는 많은 부분이 더 연구 되어야한다. 본 논문에서 한정한 사항들을 넘어서는 메모리 액세스 정보들을 모두 분석하려고 한다면 더욱 많은 정보가 필요하다. 시스템 전체적으로 볼 때 프로그램이 실행되는 동안의 거의 무작위와 같은 메모리 액세스 정보들은 모으기에 그 양도 많고, 의미있는 정보만을 골라내는 것도 힘들다. 액세스 하는 메모리의 영역과 액세스 하려는 코드들의 출처가 다양하기 때문이다. 의미있으면서 원하는 정보를 골라내기 위해서는 운영체제의 메모리 관리자 도움이 꼭 필요하다.

그럼에도 불구하고 본 논문은 한정된 상황 속에서 의미있는 정보를 추출함으로서 메모리 액세스 정보가 알고리즘 비교의 수단으로서 가치를 지님을 보여주고 있다. 추후 연구가 더 진행된다면 소스코드없이 두 실행파일의 실행을 통해서 정보를 추출하고 이를 통해 두 프로그램의 유사성을 판단할 수 있을 것이다.

참 고 문 헌

[1] Karl J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," ACM SIGCSE Bulletin, Vol.8, Issue 4, pp.30-41, 1976.

[2] Sam Grier, "A Tool that Detects Plagiarism in Pascal Programs," Twelfth SIGCSE Technical Symposium, St. Louis, Missouri, pp.15-20, 1981.

[3] M. J. Wise, "YAP3 : Improved Detectin of Similarities in Computer Programs and Other Texts," SIGCSE'96, pp.130-134, 1996.

[4] 전명제, 이평준, 조환규, "학생 프로그램 과제물 표절 탐색 기법", 한국소프트웨어감정평가학회 춘계학술대회, pp.51-55, 2004.

[5] 서선애, "Clonechecker : 프로그램 표절 검사 도구", 한국소프트웨어감정평가학회, 제3회 추계학술발표대회, pp.27-36, 2004.

[6] S. Schleimer et al. "Winnowing: Local Algorithms for Document Fingerprinting," SIGMOD2003, June 9-12, 2003.

[7] D. Gitchell, N. Tran, "Sim: A Utility for Detecting Similarity in Computer Programs," SIGCSE'99, pp.266-270, 1999.

[8] MOSS: http://www.cs.berkeley.edu/~aiken/mos- s.html

[9] M.J. Wise, "String Similarity via Greedy String Tiling and Running Karp-Rabin Matching," http://www.bio. cam.ac.uk/~mw263/ftp/doc/RKR_GST.ps, Dept. of CS, University of Sydney, 1993.

[10] Tool Interface Standards(TIS), 'Executable and Linkable Format (ELF),' Version1.2, 1995.

[11] Tool Interface Standards(TIS), 'Portable Executable Format Specification for Windows,' Version1.0, 1993.

[12] 'IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture,' Intel Corporation, 2005.

[13] R. W. Sebesta, 'Concepts of Programming Languages,' Fifth Ed., AddisonWesley, 2002.

[14] R. E. Neapolitan, 'Foundations of Algorithms using C++ Pseudocode,' Jones and Bartlett Publishers, 1998.

[15] 이효섭, 임홍태, 도경구, "프로그램 표절 탐지를 위한 프로그램 유사성 측정 방법 조사", 한국소프트웨어감정평가학회 추계학술대회, pp.9-23, 2005.

박 성 윤

e-mail : mouse500@ec.cse.cau.ac.kr

2004년 중앙대학교 컴퓨터공학과 (공학사)

2004년~2005년 휴맥스(주) 개발팀 근무

2005년~현재 중앙대학교 대학원 컴퓨터공학과 석사과정

관심분야 : Web Services, Semantic Web, Information Retrieval, Data Mining

한 상 용

e-mail : hansy@cau.ac.kr

1975년 서울대학교 공과대학(공학사)

1984년 Minnesota 공과대학(공학박사)

1984년~1995년 IBM 책임연구원

1995년~현재 중앙대학교 컴퓨터 공학과 교수

관심분야 : Virtual Prototyping, EC(Electronic Commerce), Internet Application