오토마타를 이용한 소프트웨어 악성코드 식별 기술

장희진*, 박재근**, 소우영*
*한남대학교 컴퓨터공학과
**국방과학연구소연구원
e-mail:zzang@neuro.hannam.ac.kr

On the Technique for Software Malicious Code Detection Using Automata

Huijin Jang*, Jaekeun Park**, Wooyoung Soh*
*Dept of Computer Engineering, HanNam University
**Agency for Defense Development

요 약

프로그래밍 기술과 인터넷 통신의 발달로 인하여 보안성이 검증되지 않은 다양한 프로그램들이 생성되고 쉽게 유포되어 보안 취약성으로 인해 야기되는 다양한 문제의 심각성이 더해가고 있다. 따라서 사용자가 보안상 안전하게 사용할 수 있는 소프트웨어 인증절차가 필수적으로 요구되고 있는데, 이를 위해 정보유출, 파괴 등을 목적으로 하는 악성코드를 소프트웨어의 소스코드레벨에서 사전에 검출할 수 있는 기술이 요구된다. 따라서 본 논문에서는 정규형 오토마타 이론을 이용하여 프로그램의 흐름을 예측 및 분석하고 이에 따른 악성프로그램의 흐름을 규칙으로 정하여 이러한 흐름에 만족하는 경우 악성코드를 식별하는 기법을 제안한다.

1. 서론

프로그래밍 기술과 인터넷 통신의 발달로 인하여 보안성이 검증되지 않은 다양한 프로그램들이 생성되고 쉽게 유포되어 소프트웨어에 대한 보안 취약성으로 인해 야기되는 다양한 문제의 심각성이 더해가고 있다. 따라서 보안상 안전하게 사용할 수 있는 소프트웨어 인증절차가 필수적으로 요구되고 있는데, 이를 위해 정보유출, 파괴 등을 목적으로 하는 악성코드를 사전에 검출할 수 있는 기술이 요구된다. 컴퓨터 시스템에서 사용되고 있는 다양한 형태의 프로그램들 중 불법적인 백도어(Back door), 트로이목마(Trojan horse) 등과 같은 악성코드를 포함한 악성프로그램들은 시스템에 치명적인 위협을 내포하고 있다. 이는 소프트웨어공학이 추구하는 오류방지와 생산성 향상의 두 가지 목표 외에 보안성 확보라는 제3의 목표가 추가됨을 의미한다. 이러한 시급성에도 불구하고 보안성을 고려한 소프트웨어공학의 미비로 안전한 소프트웨어 개발에 많은 어려움이 존재하고 있다. 기존의 안전성 검증 시스템들은 소프트웨어에 대한 검증이 기능적인 측면과 물리적인 피해 측면에서만 이루어지고 있는 실정이다. 이러한 기능적·물리적 피해 측면에서의 안전성 검증은 절차가 복잡하고 숨겨진 위협요소에 대한 탐지가 어려울 뿐더러, 이미 피해 후 탐지, 또는 피해 후 복구가 되기 때문에 사전에 방지할 수 없다는 문제를 가진다. 그러므로 보안상 위협을 줄 수 있는 악성소프트웨어에 대한 소스코드(source code)레벨에서의 안전성 검증 기법이 요구된다. 이를 통해 중요 정보체계의 보안상 문제점을 사전에 검출함으로써 보안 안전성이 보장된 소프트웨어 평가 인증을 위한 시스템 환경을 구축할 수 있을 것이다.

본 논문에서는 오토마타 이론을 이용해서 프로그램 소스코드를 정규형 언어로 나타내 프로그램의 흐름을 예측 및 분석하고 이에 따라 악성프로그램의 악성코드 흐름을 규칙으로 정하여 일반적인 프로그램이 이러한 흐름에 만족하는 경우 악성코드를 식별하고 안정성을 검증하는 기법을 제안한다. 2장에서는 관련연구에 대해 기술하고, 3장에서는 악성코드 식별 기술을 제안하고, 마지막으로 4장에서 결론을 맺는다.

2. 관련연구

프로그램의 흐름을 분석하기 위해서는 소스코드를 컴파일전 구문 분석하기 위해 어휘분석 과정을 통해 토큰화해야 한다. 어휘분석 과정을 통한 예약된 심볼(토큰의 타입과 값)들은 구문 분석에 전달되고 오토마타 정규형 언어로 표현되어 프로그램 각 흐름의 상태전이를 나타낸다[1][2][3][4].

오토마타란 컴퓨터에 대한 추상모델이며, 모든 오토마타들은 몇 가지 필수적인 기능들을 갖는다. 우선 오토마타는 입력을 받아들이는 장치를 갖는다. 입력은 주어진 알파벳에 대한 문자열이고 입력 파일에 저장된다. 입력 파일은 셀 단위로 구분되며, 각 셀은 하나의 심볼을 저장할 수 있다. 입력 장치는 (EOF 조건을 검사함으로써) 입력 문자열의 마지막을 감지할 수 있다. 오토마타는 어떤 형태로든 출력을 생성할 수도 있다. 또한, 오토마타는 임시 기억장소를 가질 수 있다. 이 기억장소는 무한개의 셀들로 구성되어 있으며, 각 셀은 주어진 알파벳(이는 반드시 입력 알파벳과 같을 필요는 없다) 내의 한 심볼을 저장할 수 있다. 오토마타는 제어 장치를 가진다. 이 제어 장치는 유한개의 내부 상태들 중 한 상태에 있을 수 있으며, 미리 정해진 규칙에 따라 상태를 바꿀 수 있다. 임의의 주어진 시간에 제어 장치는 어떤 내부 상태에 있게 되며, 입력 장치는 입력 파일의 특정 심볼을 읽어 들인다. 다음 단계에서의 제어 장치의 내부 상태는 전이 함수에 의하여 결정된다. 전이 함수는 현재 상태, 현재의 입력 심볼, 현재 임시 기억장소에 저장된 내용 등에 따라 다음 상태를 결정한다. 한 단계에서 다음 단계로 전이가 발생하는 동안 출력이 생성되거나 임시 기억장소의 내용이 변화될 수 있다.[1][2]

본 논문에서는 푸쉬다운 오토마타로 입력하여 비교 판단을 하기 위해 프로그램 소스코드를 정규형 언어로 나타내 프로그램의 흐름을 예측 분석한다. 유한상태 오토마타로는 악성프로그램의 악성코드 흐름을 규칙으로 정한다. 각각의 푸쉬다운 오토마타와 유한상태 오토마타는 다음과 같이 정의된다[1][2].

2.1 푸쉬다운(push-down) 오토마타

푸쉬다운 오토마타는 다음 7가지 요소로 정의된다.

$M = (Q, \Sigma, \Gamma, \delta, q0, Z, F)$

Q : 상태들의 유한 집합
Σ : 입력 알파벳의 유한 집합
Γ : 스택 알파벳의 유한 집합
δ : 전이 함수, $Q\times(\Sigma\cup\{\lambda\})\times\Gamma\rightarrow Q\times\Gamma^*$의 부분집합
$q0\in Q$: 초기상태
$Z\in\Gamma$: 스택시작심블
$F\subseteq Q$: 최종상태들의 집합

2.2 유한상태 (finite-state) 오토마타

유한 오토마타는 다음 5가지 요소로 정의된다.

$M = (Q, \Sigma, \Gamma, \delta, q0, Z, F)$

Q : 상태들의 유한집합
Σ : 입력 알파벳의 유한 집합
δ : $Q\times\Sigma\rightarrow Q$인 전이 함수(transition function)
q0: Q의 원소인 시작상태 혹은 최초상태
F : Q의 부분집합인 최종상태 혹은 수락상태

3. 악성코드 식별 기술

3.1 프로그램 소스코드 흐름 분석

프로그램 소스코드를 컴파일러의 어휘분석기를 통해 아래와 <표 1>과 같은 심볼 테이블에 따라 토큰화하고 심볼 테이블에 없는 새로운 타입이 발생할 경우 추가 등록하도록 한다.

<표 1> 심볼 테이블 토큰의 타입과 내용

타입	내용
binary	이항 연산자 지정
break_stmt	break문 지시
case_label	case문 레이블 지시
cast	데이터 자료형 강제 변경 지시자
compound_stmt	함수 블록 지시, 조건문이나 반복문 블록 지시
conditional	조건문 지시
data_decl	변수의 자료형 선언
data_declaration	인자의 자료형
declarator	함수 선언 시 함수명, 함수 인자 선언 시 인자명, 변수명 지정
default_label	switch문의 default 레이블 지시
dowhile_stmt	dowhile문 지시
enumerator	열거문 지시
expression_stmt	C 프로그램 구문(;로 끝나는 프로그램 구문)
for_stmt	for 반복문 지시
function_call	함수 호출 발생 지정
function_decl	함수 선언 구문
goto_stmt	goto문 지시
identifier	변수와 상수 정의 식별자
if_stmt	if 조건문 지시
.....	

토큰들은 푸쉬다운 오토마타에 입력하여 비교 판단을 하기 위해 정규형 언어로 표현된다.

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <pwd.h>
5
6  void drop_priv()
7  {
8    struct passwd *passwd;
9
10   if ((passwd = getpwuid(getuid())) == NULL)
11   {
12     printf("getpwuid() failed");
13     return;
14   }
15   printf("Drop user %s's privilege\n", passwd->pw_name);
16   seteuid(getuid());
17 }
18
19 int main(int argc, char *argv[])
20 {
21   drop_priv();
22   printf("About to exec\n");
23   execv(argv[1], argv + 1);
24 }
```

<그림 1> 프로그램 소스코드

<그림 1>의 프로그램 소스코드를 정규형 언어로 표현하면 다음 <그림 2>와 같다. 아래 <그림 2>의 예에서 1번째 라인 'fb'는 function block이라는 토큰의 타입을 나타내고 'drop_priv'는 이에 해당하는 함수 이름, 즉 토큰의 값을 나타낸다. 2번째 라인에

서 '1074984776'은 스택에 저장된 메모리 주소를 나타내고 'function_decl'는 함수 선언이라는 심볼이 된다.

```
1  fb "drop_priv"
2  { 1074984776 function_decl
3          { 1074984672 declarator "drop_priv" }
4          { 1074945816 compound_stmt
5                  { 1074938000 data_decl
6                          { 1074986816 variable_decl
7                                  { 1074937872 data_declaration }
8                                  { 1074986720 declarator "passwd" }
9                          }
10                 }
11                 { 1074937616 expression_stmt
12                         { 1074937480 binary "="
13                                 { 1074937344 identifier "passwd" *1074937872 }
14                                 { 1074937016 lexical_cst "0" }
15                         }
16                 }
                   ........
                   ........
                   ........
```

<그림 2> 토큰화된 정규형 표현

이러한 토큰들은 위와 같은 방법으로 프로그램의 흐름에 따라 순서대로 열거되며 함수와 우선순위 종속관계에 따라 '{'.. '}' 으로 구분된다.

3.2 악성프로그램 악성코드 규칙 정의

위 <그림 1>의 프로그램은 위에서 프로그램의 흐름분석을 위해 푸쉬다운 오토마타로 입력을 위해 정규형 언어로 표현하기 위한 일반적 프로그램의 예로 사용되었다. 그러나 이것은 Unix/Linux에서 현재 프로세스를 수행한 특정권한을 획득하여 프로그램을 실행시킬 위험 가능성이 존재하는 악성프로그램이다. 이러한 프로그램에서 다음과 같은 악성코드 흐름을 찾아 규칙을 정의할 수 있다. <그림 1>에서 10번째 라인의 'getuid()'는 현재 프로세스의 실제 사용자 ID를 얻어오고, 'getpwuid(uid_t uid)'는 사용자 UID 와 일치하는 엔트리를 '/etc/passwd' 에서 가져오고 각 필드의 내용을 분리해서 'passwd' 구조체에 대한 포인터를 통해서 반환한다.

16번째 라인은 'seteuid(uid_t euid)'는 유효 사용자 ID 설정을 하기 위해서 사용된다. 유효 사용자 ID는 파일 생성과 접근등에 영향을 미치게 된다. 그리고 23번째 라인 'execv(argv[1], argv + 1)'은 실행하게 되면 현재 프로세스 이미지를 두 번째 인자 값의 새로운 프로세스 이미지로 바꾼다. 10번째 라인의 'if' 조건문의 결과가 만족하여 현재 프로세스의 실제 사용자 ID를 획득한 상태에서 16번째 라인이 수행되지 않으면 유효 사용자 ID로 권한이 전환되지 않아 23번째 라인에서 현재 프로세스의 실제 사용자 권한 하에 인자 값으로 주어진 프로그램이 실행되게 된다. 따라서 위에서 열거된 문제의 속성들은 유한상태 오토마타에 의해 다음과 같은 전이 상태로 정의할 수 있다. 't'는 전이(transition)를 의미한다.

```
1  seteuid(FSA)
2  t euid_0 euid_1 { function_call { identifier seteuid }
3                                  { function_call { identifier getuid } } }
4  t euid_0 euid_0 { other }
5  t euid_1 euid_0 { function_call { identifier seteuid } { lexical_cst 0 } }
6  t euid_1 euid_1 { other }
7
8  execv(FSA)
9  t before_exec after_exec { function_call { identifier execv } { ellipsis } }
10 t before_exec before_exec { other }
11 t after_exec after_exec { function_call { identifier execv } { ellipsis } }
12 t after_exec before_exec { other }
13
14 q euid_0 before_exec
15 f euid_0 after_exec
```

<그림 3> 악성코드 전이 규칙

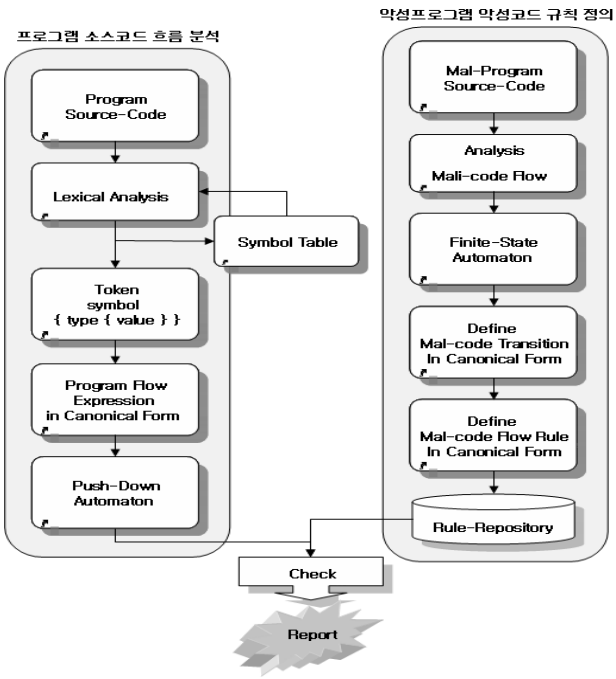
<그림 3>에서 16번째 라인에 관한 'seteuid'에서 'euid_0'은 유효 사용자 ID로 전환되지 않은 상태, 'euid_1'은 유효 사용자 ID로 전환된 상태를 말한다. 2번째 라인의 전이는 심볼 테이블에 따라 함수 호출 'seteuid(getuid())'에 의한 유효 사용자 ID로의 전환에 대한 표현이고, 5번째 라인은 함수 호출 'seteuid(0)'에 의한 'root' ID로의 전환에 대한 표현이다. 4번째와 6번째 라인의 전이는 2번째와 5번째에 정의된 속성들 이외의 어떤 다른 여러 가지 속성에 의해 전이가 나타나기 때문에 와일드카드로써 'other'로 표현된다.

23번째 라인에 관한 'execv'에서 'before_exec'는 'execv'가 실행되지 않은 상태, 'after_execv'는 'execv'가 실행된 상태를 말한다. 9번째 라인과 11번째 라인의 전이는 심볼 테이블에 따라 함수 호출 'execv(argv[1], argv+1)'에 의한 현재 프로세스의 실제 사용자 권한으로 인자 값으로 주어진 프로그램 실행에 대한 표현이고, 'ellipsis'는 'execv(....)' 함수 내에 어떠한 인자가 들어와 프로그램이 실행될지 모르기 때문에 생략의 의미로 사용된 심볼이다. 10번째와 12번째 라인의 전이는 <그림 3>에서와 같이 와일드카드로써 'other'로 표현된다.

위에서와 같이 'seteuid'와 'execv'의 전이가 정규형 언어로 표현되면, 유효 사용자 ID로 권한이 전환되지 않은 상태에서 현재 프로세스의 실제 사용자 권한 하에 인자 값으로 주어진 프로그램이 실행되게 하는 문제의 흐름은 14번째 라인에 초기 상태 'q' 15번째 라인에 최종상태 'f'로 악성코드 전이 규칙으로 정의한다.

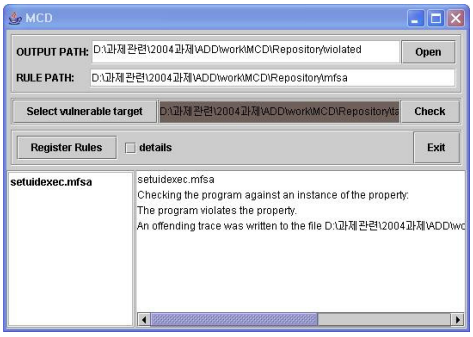
3.3 악성코드 식별

여러 악성코드 전이 규칙이 분석되고 정의되어 특정한 룰 데이터로 저장되면, 푸쉬다운 오토마타에 입력을 위해 소스코드는 <그림 2>와 같이 흐름을 가진 토큰화된 정규형 언어로 표현되고, <그림 3>과 같이 유한상태 오토마타에 의해 앞에서 정의된 규칙과 비교하여 해당되는 전이를 포함하고 있는 경우에 위험 가능성을 내포하고 있는 악성프로그램으로 간주되고 해당되는 속성의 코드를 악성코드로 정의하여 나타내도록 한다. 다음 <그림 4>는 이러한 악성코드를 식별하는 기법의 순서를 설명한다.



<그림 4>악성코드 식별 메커니즘 순서도

<그림 5>는 이러한 기법에 따라 구현된 악서코드 검증 시스템의 프로토타입 실행화면이다.



<그림 5> 악성코드 식별 시스템 GUI

4. 결론

본 논문에서는 프로그램의 소스코드 내부에서 악의의 목적으로 포함될 수 있는 악성코드를 식별하여 실행전에 안전성을 검증할 수 있는 기법을 연구 제안하였다. 이는 기존의 기능적·물리적 피해 측면에서의 단순한 안전성 검증기법의 문제점을 해결하였으며, 또한 프로그램 구조의 흐름을 분석하여 오토마타 이론에 따라 전이 규칙을 생성함으로써 악성코드의 흐름과 일치하는 프로그램의 구조에서 악성코드를 식별하여 안전성을 검증함으로써 오용검증 편차가 큰 단순 패턴매칭으로 이루어지는 기존 기법과 전혀 다른 해결 방안을 제시하게 되었다.

향후 연구 과제로는 제안된 기법으로 설계 및 구현을 통해 기존의 기법들과의 성능비교분석이 이루어져야 하고, 더 다양한 악성프로그램의 코드를 식별하여 안전성을 검증하기 위해 악성코드 규칙 정의에 대한 연구가 지속적으로 수행되어져야 한다.

참고문헌

[1] 김대수, 생능출판사, "오토마타와 계산이론", 1998.

[2] 오세만, 정익사, "컴파일러 입문", 2004.

[3] Karen A. Lemone, "Fundamentals of COMPILERS - an Introduction to computer language translation"

[4] John R. Levine, Tony Mason & Doug Brown, "Lex & Yacc"

[5] Hao Chen, David Wagner and David Schultz at Computer Science Division, UC Berkeley, "MOPS User's Manual"

[6] Hao Chen and David Wagner University of California at Berkeley "MOPS: an Infrastructure for Examining Security Properties of Software"

[7] Hao Chen, David Wagner, and Robert Johnson, "Model Checking Software for Security Violations", Short talk at 2001, IEEE Symposium on Security and Privacy.

[8] Thomas Ball, Sriram K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces", SPIN 2001, Workshop on Model Checking of Software, LNCS 2057, May 2001, pp. 103-122.

[9] Thomas Ball, Sriram K. Rajamani, "Boolean Programs: A Model and Process for Software Analysis", MSR Technical Report 2000-14.