J. Internet Comput. Serv.
ISSN 1598-0170 (Print) / ISSN 2287-1136 (Online)
http://www.jics.or.kr

RNN을 이용한 코드 재사용 공격 탐지 방법 연구

Detecting code reuse attack using RNN

김 진 섭[1] 문 종 섭[1*]
Jin-sub Kim Jong-sub Moon

요 약

코드 재사용 공격은 프로그램 메모리상에 존재하는 실행 가능한 코드 조각을 조합하고, 이를 연속적으로 실행함으로써 스택에 직접 코드를 주입하지 않고도 임의의 코드를 실행시킬 수 있는 공격 기법이다. 코드 재사용 공격의 대표적인 종류로는 ROP(Return - Oriented Programming) 공격이 있으며, ROP 공격에 대응하기 위한 여러 방어기법들이 제시되어왔다. 그러나 기존의 방법들은 특정 규칙을 기반으로 공격을 탐지하는 Rule-base 방식을 사용하기 때문에 사전에 정의한 규칙에 해당되지 않는 ROP 공격은 탐지할 수 없다는 한계점이 존재한다. 본 논문에서는 RNN(Recurrent Neural Network)을 사용하여 ROP 공격 코드에 사용되는 명령어 패턴을 학습하고, 이를 통해 ROP 공격을 탐지하는 방법을 소개한다. 또한 정상 코드와 ROP 공격 코드 판별에 대한 False Positive Ratio, False Negative Ratio, Accuracy를 측정함으로써 제안한 방법이 효과적으로 ROP 공격을 탐지함을 보인다.

☞ 주제어 : 코드 재사용 공격, ROP(Return-Oriented Programming), RNN(Recurrent Neural Network)

ABSTRACT

A code reuse attack is an attack technique that can execute arbitrary code without injecting code directly into the stack by combining executable code fragments existing in program memory and executing them continuously. ROP(Return-Oriented Programming) attack is typical type of code reuse attack and serveral defense techniques have been proposed to deal with this. However, since existing methods use Rule-based method to detect attacks based on specific rules, there is a limitation that ROP attacks that do not correspond to previously defined rules can not be detected. In this paper, we introduce a method to detect ROP attack by learning command pattern used in ROP attack code using RNN(Recurrent Neural Network). We also show that the proposed method effectively detects ROP attacks by measuring False Positive Ratio, False Negative Ratio, and Accuracy for normal code and ROP attack code discrimination.

☞ keyword : code reuse attack, ROP(Return-Oriented Programming), RNN(Recurrent Neural Network)

1. 서 론

컴퓨팅 환경과 네트워크가 발전함에 따라 시스템을 공격하는 악성코드 또한 증가하게 되었다. 그리고 많은 악성코드들은 시스템의 권한을 획득하기 위해 시스템의 커널 또는 애플리케이션 프로그램에 내제된 메모리 취약점을 악용한다. 이에 따라 악성코드의 공격으로부터 메모리를 보호하기 위한 다양한 방어 기법들이 제시되었다. 특히 $W \oplus X$와 DEP(Dada Execution Prevention)[9]는 스택 또는 힙 메모리 영역에서 실행 권한을 제거함으로써, 해당 메모리상에 존재하는 어떠한 코드도 실행되지 못하게 한다. 따라서 버퍼 오버플로우 취약점을 통해 메모리에 직접 쉘 코드를 주입하고 실행하는 전통적인 공격 방법은 더 이상 성공시키기 어려워졌다. 그러나 위와 같은 메모리 보호 기법은 코드 재사용 공격이라는 공격 기법을 통해 우회될 수 있다[6].

코드 재사용 공격이란 프로그램 메모리상에 존재하는 실행 가능한 코드 조각을 조합하고, 이를 연속적으로 실행함으로써 임의의 코드를 실행시킬 수 있는 공격 기법을 말한다. 코드 재사용 공격을 사용하는 공격자는 실행하고자 하는 코드를 메모리에 직접 주입하지 않는다. 대신 공격자는 실행 권한이 부여된 프로그램의 바이너리 코드 또는 공유 라이브 코드의 일부를 재사용함으로써, $W \oplus X$와 DEP를 우회할 수 있다. 이러한 코드 재사용 공격에 해당하는 대표적인 공격으로는 ROP(Return-Oriented Programming)[11] 공격이 있다.

ROP 공격은 RET 명령으로 끝나는 짧은 코드 조각인

1 Graduate School of Information Security, Korea University, Seoul, 02481, Korea
* Corresponding author (jsmoon@korea.ac.kr)
[Received 12 April 2018, Reviewed 19 April 2018, Accepted 5 May 2018]

가젯(gadget)을 이용하여 공격자가 원하는 임의의 코드를 수행할 수 있다. 이러한 특징 때문에 ROP 공격은 $W \oplus X$ 또는 DEP 같은 메모리 보호기법을 우회하여 시스템의 관리자 권한을 획득할 수 있다. 이러한 위험성 때문에 ROP 공격에 대응하기 위한 다양한 방어 기법들이 제시되어왔다[1, 2, 7 & 10]. 그러나 기존의 방어 기법들은 ROP 공격으로부터 발견할 수 있는 특정 규칙을 기반으로 공격을 탐지하는 Rule-base 방식을 사용한다[2]. Rule-base 방식은 사전에 정의한 규칙에 해당하지 않는 공격에 대해서는 탐지가 불가능하다는 한계점이 있다.

본 논문에서는 Rule-base 방식의 한계점을 해결하기 위해 딥 러닝(Deep learning)을 사용하여 학습된 인공 신경망을 통해 ROP 공격을 탐지하는 방법을 제안한다. 연속적으로 실행되는 명령어의 패턴을 학습하기 위해 Recurrnet Neural Network(RNN)이라는 인공 신경망 모델을 사용하였으며, DBI(Dynamic Binary Instrumentation) 도구와 텐서플로우(Tensorflow)를 이용하여 성능 측정을 위한 실험을 진행하였다. 실험 과정에서는 샘플링을 통해 학습 데이터를 재생성함으로써 학습 데이터 부족 현상을 보완하였다.

본 논문은 다음과 같이 구성된다. 2장에서는 ROP 공격과 기존에 제시된 방어 기법, 그리고 RNN에 대해 소개한다. 3장에서는 RNN을 사용하여 동적으로 ROP 공격을 탐지하는 방법에 대해 소개한다. 4장에서는 제안하는 방법을 통한 ROP 공격 탐지의 성능을 측정하기 위한 실험을 수행하고, 마지막 5장에서는 결론과 향후 연구방향을 제시하며 논문을 끝맺는다.

2. 관련 연구

2.1 Return-Oriented Programming

ROP 공격은 적은 개수의 명령어로 이루어진 짧은 코드 조각을 사용하며, 이 때 사용하는 코드 조각을 가젯이라고 한다. ROP 공격에서 사용하는 가젯은 항상 RET 명령어로 끝난다. RET 명령어는 다음에 실행할 명령어의 주소를 스택 최상단으로부터 가져오고, 스택 포인터를 증가시킨다. 따라서 RET 명령과 스택의 데이터에 따라 실행 흐름을 조작할 수 있다. ROP 공격을 사용하는 공격자는 먼저 버퍼 오버플로우 취약점을 이용하여 스택 메모리에 가젯의 주소를 덮어쓴다. 그리고 스택 포인터를 가젯의 주소로 덮어쓴 영역에 위치시켜 가젯의 코드와 RET 명령을 실행하도록 만든다. 각 가젯의 RET 명령어는 다음 가젯으로 실행 흐름을 옮기는 역할을 수행한다. 위와 같은 방법으로 공격자는 원하는 임의의 코드를 실행시킬 수 있다.

2.2 ROP 공격에 대한 기존의 방어 기법

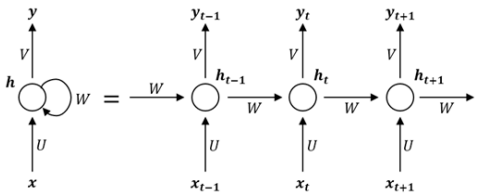
ROP 공격에 대응하기 위한 여러 가지 방어 기법들이 제시되었으며, 방어 기법은 크게 컴파일러 수준의 방어 기법[1. 7 & 10]과 바이너리 수준의 방어 기법[2]으로 구분된다.

컴파일러 수준의 방어 기법은 소스코드로부터 기계어를 생성하는 과정에서 RET 명령에 해당하는 모든 기계어 코드를 제거함으로써 ROP 공격을 원천적으로 방어하는 것이다[1, 7, & 10]. 그러나 컴파일러 수준의 방어 기법을 적용하기 위해선 소스코드를 이용하여 프로그램을 다시 컴파일을 해야 하기 때문에, 소스코드를 구할 수 없는 프로그램에 대해서는 적용하기 어렵다는 한계가 있다[10].

바이너리 수준의 방어 기법은 프로그램이 실행되는 중에 동적으로 ROP 공격을 탐지하는 것이다. 대표적인 바이너리 수준의 방어 기법으로는 Ping Chen 등에 의해 제안된 DROP[2]가 있다. DROP는 ROP 공격에 사용되는 가젯이 적은 개수의 명령어로 구성된다는 사실을 이용한다. DROP는 프로그램이 실행하는 명령어를 감시하여 매 RET 명령어 사이에 실행된 명령어의 수를 계산하고, 이를 가젯의 길이로 정의한다. 만약 가젯의 길이가 사전에 정의된 값(T_0)보다 짧다면 해당 가젯을 공격 의심 가젯으로 판별한다. 공격 의심 가젯이 사전에 정의된 값(T_1) 이상 연속적으로 등장하면 ROP 공격으로 간주한다. DROP는 가젯의 길이와 T_0의 비교를 통해 공격 의심 가젯을 판별한다. 그러나 위와 같은 방법을 사용하면 사전에 정의한 T_0보다 큰 길이의 가젯을 사용하는 경우 절대 공격 의심 가젯으로 판별되지 않는다는 한계를 갖고 있다. 따라서 본 논문에서는 RNN을 이용하여 가젯을 구성하는 명령어의 패턴을 학습하고, 이를 통해 공격 의심 가젯을 판별한다.

2.3 Recurrent Neural Network

명령어들은 CPU에 의해 순서대로 실행되기 때문에 일종의 순차 데이터라고 볼 수 있다. RNN(Recurrent Neural Network)은 이러한 순차 데이터를 학습하는데 특화된 인공 신경망이다. 그림 1은 RNN의 기본 형태를 나타낸 것이다.



(그림 1) RNN의 기본 구조
(Figure 1) Basic structure of RNN

그림 1에서 x_t는 시간 t일 때 RNN의 입력 값을 의미한다. h_t는 시간 t일 때 RNN의 은닉 상태 값을 의미하고, RNN에서 h_t를 나타내는 뉴런을 메모리 셀(memory cell)이라고 부른다. y_t는 시간 t일 때 RNN의 출력 값을 의미한다. 시간 t일 때의 h_t와 y_t는 다음과 같은 수식으로 표현할 수 있다.

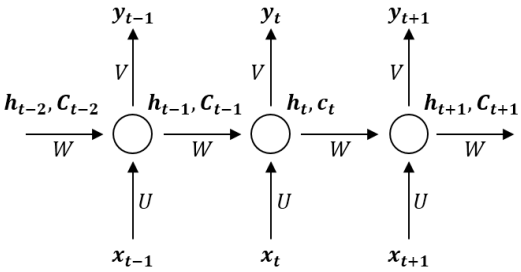
$$h_t = f(Ux_t + Wh_{t-1} + b)$$
$$y_t = Vh_t + c$$

h_t를 계산하기 위해 메모리 셀은 입력으로 x_t와 h_{t-1}을 받는다. f는 h_t 계산에 사용되는 활성화 함수를 의미한다. RNN에서는 활성화 함수로 비선형 함수인 tanh 함수를 주로 사용한다. U는 x_t와 h_t 사이의 가중치 행렬을 의미하고, V는 h_t와 y_t 사이의 가중치 행렬을 의미한다. W는 h_t와 h_{t-1} 사이의 가중치 행렬을 의미하고, b와 c는 상수 값이다.

RNN을 학습하기 위해 Gradient Descent와 BPTT (Backpropagation Through Time)[5] 알고리즘이 사용된다. 기존의 Backpropagation 알고리즘은 시간을 거슬러 올라가는 형태의 학습이 불가능하다는 문제가 있다. RNN의 학습 과정에서는 Backpropagation 알고리즘을 확장한 BPTT 알고리즘을 사용한다. 그러나 RNN의 학습 과정은 에러가 발생한 시점에서 멀리 떨어진 과거 시점일수록 에러의 그래디언트가 전달되지 않는 Vanishing gradient 문제를 내포하고 있다[12, 13]. 이 문제로 인해 기본적인 형태의 RNN은 시퀀스 길이가 긴 데이터를 학습하는데 한계점을 보인다. 이를 보완하기 위한 해결책으로는 ReLU를 활성 함수로 사용하는 방법과 GRU(Gated Recurremt Units)[3], LSTM(Long Short-Term Memory)[4]을 사용하는 방법이 있다. 본 논문에서는 LSTM을 사용함으로써 Vanishing gradient 문제를 보완한다.

2.4 Long Short-Term Memory

LSTM은 길이가 긴 시퀀스 데이터에 대한 학습이 어려운 RNN을 보완하기 위해 등장한 인공 신경망 알고리즘이다. 그림 2는 LSTM의 기본 구조를 나타낸 것이다. LSTM의 기본 구조는 RNN의 기본 구조와 동일하지만, 메모리 셀에서의 상태 값을 더 복잡하게 계산한다. LSTM에서는 메모리 셀에서 은닉 상태 값 h_t와 셀 상태 값 c_t를 계산한다. 은닉 상태 값은 상위 계층인 y_t를 계산하는 것과, 다음 시간의 은닉 상태 값 h_{t+1}을 계산하는데 사용된다. 셀 상태 값은 상위 계층으로 전달되지 않으며 h_{t+1}을 계산하기 위해 시간 $t+1$의 메모리 셀에 전달된다.



(그림 2) LSTM의 기본 구조
(Figure 2) Basic structure of LSTM

LSTM은 셀 상태 값 c_t를 계산하기 위해 삭제 게이트(forget gate)와 입력 게이트(input gate)를 이용한다. 삭제 게이트(f_t)는 이전의 상태 값 중 잊어버릴 정보를 학습하고, 입력 게이트(i_t)는 새롭게 추가되는 현재 정보를 학습한다. f_t와 i_t를 이용하여 h_t와 c_t를 계산하는 수식은 아래와 같다. 이 때 U_*, W_*는 가중치 행렬을 나타내며 아래 첨자에 따라 다른 가중치 행렬 값을 갖는다. b_*는 상수 값을 의미하며 아래 첨자에 따라 다른 값을 갖는다.

$$f_t = c_{t-1} \times sigmoid(U_f x_t + W_f h_{t-1} + b_f)$$
$$i_{sgm} = sigmoid(U_{sgm} x_t + W_{sgm} h_{t-1} + b_{sgm})$$
$$i_{\tanh} = \tanh(U_{\tanh} x_t + W_{\tanh} h_{t-1} + b_{\tanh})$$
$$i_t = i_{sgm} \times i_{\tanh}$$
$$c_t = f_t + i_t$$
$$h_t = \tanh(c_t) \times sigmoid(U_o x_t + W_o h_{t-1} + b_o)$$

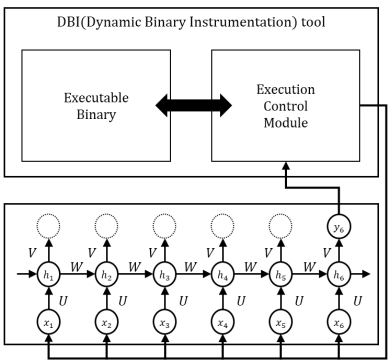
3. 제안하는 방법론

3.1 ROP 공격 탐지 개요

본 논문에서는 ROP 공격 탐지를 위해 RNN을 사용하며, LSTM 메모리 셀을 사용하여 Vanishing gradient 현상을 보완하였다. 가젯을 구성하는 명령어들은 RNN의 입력 데이터로 주어지는데, 각 명령어들은 RNN이 처리하기 용이한 데이터 형태로 인코딩(encoding)된다.

인공 신경망을 효과적으로 학습시키기 위해선 많은 양의 데이터가 필요하다. 그러나 ROP 공격에 사용되는 데이터는 일반적으로 정상 데이터에 비해 많은 양의 데이터를 확보하기 어렵기 때문에, 본 논문에서는 샘플링(sampling)을 이용하여 추가적으로 학습 데이터를 확보하였다.

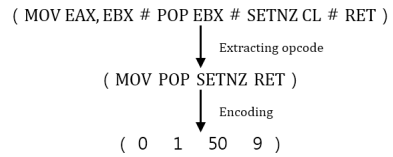
그림 3은 ROP 공격을 탐지하는 전체 시스템을 나타낸 것이다. DBI(Dynamic Binary Instrumentation) 도구를 사용하여 제작된 실행 제어 모듈(Execution Control Module)을 이용하여 바이너리의 실행을 제어한다. 실행 제어 모듈은 바이너리가 실행한 명령어를 동적으로 추출하고, 명령어를 인코딩하여 RNN의 입력으로 전달한다. 또한 RNN의 출력 값을 받고 이를 통해 ROP 공격을 탐지한 경우 실행 중인 바이너리를 중지시키는 역할을 수행한다.



(그림 3) ROP 공격 탐지 시스템 개요
(Figure 3) Concept of ROP attack detection system

3.2 RNN의 입력 데이터

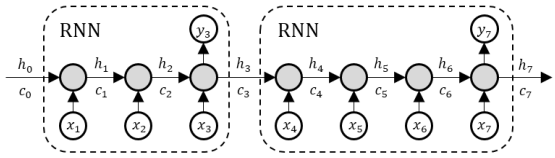
가젯을 구성하는 명령어를 RNN의 입력으로 사용할 수 있게 변환한다. 명령어가 수행하는 연산의 패턴을 학습시키기 위해 명령어에서 opcode를 추출한다. opcode는 “pop”, “mov”와 같은 문자열 형태이므로 RNN이 데이터를 용이하게 처리할 수 있도록 각 opcode를 서로 다른 정수형 데이터로 인코딩한다. 인코딩 과정은 다음과 같다. 먼저 정상 프로그램에서 추출한 가젯에 가장 많이 존재하는 상위 50개의 명령어를 추출하고, 가장 많이 등장하는 순서대로 0부터 49의 숫자를 부여한다. 상위 50개에 속하지 않은 명령어들은 모두 숫자 50으로 인코딩한다. 그림 4는 가젯의 명령어를 RNN의 입력으로 변환하는 과정을 나타낸 것이다.



(그림 4) 명령어 인코딩 과정
(Figure 4) Processing of instruction encoding

3.3 ROP 공격 탐지에 사용되는 RNN 구조

그림 5는 ROP 공격 탐지를 위한 RNN의 전체적인 구조를 나타낸 것이다. Vanishing gradient 현상을 보완하기 위해 LSTM 메모리 셀을 사용하였다. RNN의 시간 스텝은 입력되는 가젯의 길이, 즉 가젯을 구성하는 명령어의 개수와 같다. 가젯 단위로 RNN의 출력 값을 얻기 위해 각 RNN의 마지막 시간 스텝에서만 y_t값을 출력한다. RNN의 출력 y_t는 1×2 벡터 형태이다. 따라서 y_t를 벡터 형태로 표현하면 $y_t = [y_{t1}, y_{t2}]$와 같이 쓸 수 있다. RNN의 학습 과정에서 정상 가젯의 라벨은 $y = [1,0]$으로 부여되고, 공격 가젯의 라벨은 $y = [0,1]$로 부여된다. 임의의 가젯을 입력으로 주었을 때의 RNN의 출력 y_t에 대해 $y_{t1} > y_{t2}$인 경우 해당 가젯은 정상으로 판별되고, $y_{t1} < y_{t2}$인 경우 공격으로 판별된다. 은닉 상태 값 h_t와 셀 상태 값 c_t는 최초 0으로 초기화된 값을 사용하며, 다음 RNN으로 h_t와 c_t를 전달한다.



(그림 5) ROP 공격 탐지를 위한 RNN 구조
(Figure 5) Structure of RNN for detecting ROP attack

3.4 RNN의 학습

ROP 공격에 사용되는 대부분의 가젯의 길이는 11이하의 값을 갖는다[6]. Kayaalp 등에 의해 수행된 연구에 따르면, 표준 C 라이브러리(libc)에서 발견된 가젯들 중 약 85%가 5이하의 길이를 갖고 있으며, 8이상의 길이를 갖는 가젯은 1% 미만이다[6]. 따라서 제한된 길이의 가젯에 대하여 정상 가젯과 공격 가젯을 구분할 수 있도록 RNN을 학습시킬 필요가 있다. 본 논문에서는 RNN 학습에 사용되는 가젯의 길이를 제한하며, 제한 길이는 학습에 사용되는 ROP 공격 가젯의 최대 길이로 한다.

3.5 ROP 공격 탐지

학습이 완료된 RNN을 이용하여 ROP 공격을 탐지한다. DROP의 경우 가젯의 길이를 통해 공격 의심 가젯을 판별하였으나[2], 본 논문에서는 RNN에 가젯의 명령어를 입력하고 그 출력 값을 통해 공격 의심 가젯을 판별한다. ROP 공격 탐지는 DROP와 마찬가지로 공격 의심 가젯이 특정 횟수(T) 이상 연속적으로 등장하는 경우 ROP 공격으로 간주한다. T값을 매우 작게 설정하면 ROP 공격을 탐지할 가능성이 높아지지만 정상적인 코드 수행을 ROP 공격으로 잘못 탐지할 가능성 또한 높아진다. 반대로 T 값을 매우 높게 설정하면 정상 코드를 ROP 공격으로 판단하는 경우는 줄어들지만, ROP 공격 코드의 가젯 개수가 T값 보다 작은 경우 공격을 탐지할 수 없게 된다. 따라서 적절한 T값을 사용해야 하며, 실험을 통해 최대의 성능을 나타내는 T값을 사용한다. Table 1은 ROP 탐지 알고리즘의 의사 코드를 나타낸 것이다.

(Table 1) ROP 공격 탐지 알고리즘 의사코드

ROP 공격 탐지 알고리즘 의사코드

```
1: for t in time step
2:   if y_t2 > y_t1 then
3:     suspicious_gadget = suspicious_gadget + 1
4:     if suspicious_gadget ≥ T then
5:       ROP_Alert()
6:     end if
7:   else
8:     suspicious_gadget ← 0
9:   end if
10: end for
```

4. 실험 및 평가

4.1 실험 환경

실험은 Ubuntu 14.04 amd64, 리눅스 커널 3.13.11 버전에서 수행되었다. 실험에 사용된 하드웨어의 정보는 표 1과 같다.

(표 1) 실험에 사용된 하드웨어의 정보

하드웨어	정보
CPU	2.2GHz Intel Core i7
RAM	16GB 1600MHz DDR3
VGA	Intel Iris pro 1536MB
HDD(SSD)	256GB SSD

4.2 정상 및 공격 데이터

정상 프로그램의 가젯을 확보하기 위해 표 2와 같이 Ubuntu 14.04 amd64에 존재하는 정상 프로그램 30개를 선정하였다. ROP 공격 코드를 얻기 위해 악성코드 수집 웹 사이트인 Exploit-DB를 이용하였고, 총 20개의 ROP 코드를 획득하였다. ROP 공격에 사용된 가젯의 총 개수는 720개이며, 이 중 가장 긴 가젯의 길이는 8이다. 정상 프로그램으로부터 RNN 학습 및 테스트에 필요한 가젯을 추출하기 위해 DBI 도구인 Pin[8]을 사용하였다. Pin을 이용하여 정상 프로그램에서 확보한 가젯의 총 개수는 20,003개이며, 8 이하의 길이를 갖는 가젯의 총 개수는 3,556개이다.

(표 2) 실험에 사용된 정상 프로그램

ls	rmdir	gzip 1.6
cat	readelf 2.24	size 2.24
pwd	python 2.7.6	cp
find	strings 2.24	hexdump
touch	gcc 4.8.4	rm
file 5.14	sha256sum 8.21	bzip2 1.0.6
uname	tar 1.27.1	date 8.21
ps	last	chown
mkdir	zip 3.0	echo
objdump 2.24	whoami	chmod

4.3 샘플링을 통한 추가 학습 데이터 확보

RNN의 학습 성능을 높이기 위해 학습에 앞서 충분한

양의 데이터를 확보해야 한다. 정상 프로그램이 실행하는 코드는 수백 개 이상의 가젯들로 구성되는 반면, ROP 공격 코드를 구성하는 가젯의 수는 정상 프로그램에 비해 현저히 적다. 따라서 ROP 공격 코드의 가젯은 정상 프로그램의 가젯에 비해 확보하는데 많은 어려움이 따른다. 본 논문에서는 샘플링 기법을 이용하여 ROP 공격 가젯 데이터를 추가로 확보하고 이를 학습에 사용한다.

공격 가젯을 샘플링하기 위해 먼저 기존에 확보한 ROP 공격 가젯으로부터 가젯 길이 분포와 가젯의 첫 번째 opcode의 분포, 그리고 임의의 opcode에 대하여 다음에 등장하는 opcode의 분포를 얻었다. 본 논문에서는 위 세 가지 분포를 각각 P_{length}, P_{first}, P_{opcode}라고 정의한다. Table 2는 P_{length}, P_{first}, P_{opcode}를 이용하여 가젯을 샘플링하는 알고리즘의 의사코드이다. 하나의 가젯을 생성하기 위해 먼저 P_{length}로부터 가젯의 길이 정보를 샘플링한다. 가젯을 구성하는 명령어를 생성하기 위해 가젯의 첫 번째 명령어를 P_{first}로부터 샘플링한다. 이후 가젯을 구성하는 명령어의 개수가 P_{length}로부터 샘플링된 가젯의 길이 정보와 같아질 때 까지 P_{opcode}로부터 명령어를 연쇄적으로 샘플링한다. 그림 7과 그림 8은 Table 2의 알고리즘을 사용하여 샘플링된 1000개의 가젯과, 기존에 확보하고 있던 실제 ROP 공격 가젯의 P_{length}와 P_{first} 분포를 나타낸 것이다. 샘플링된 가젯과 기존 가젯의 분포가 거의 일치함을 확인할 수 있다. 본 실험에서는 샘플링을 통해 총 3,200개의 가젯을 추가로 확보하였고 이를 학습 데이터로 사용하였다.

(Table 2) 가젯 샘플링 알고리즘의 의사코드

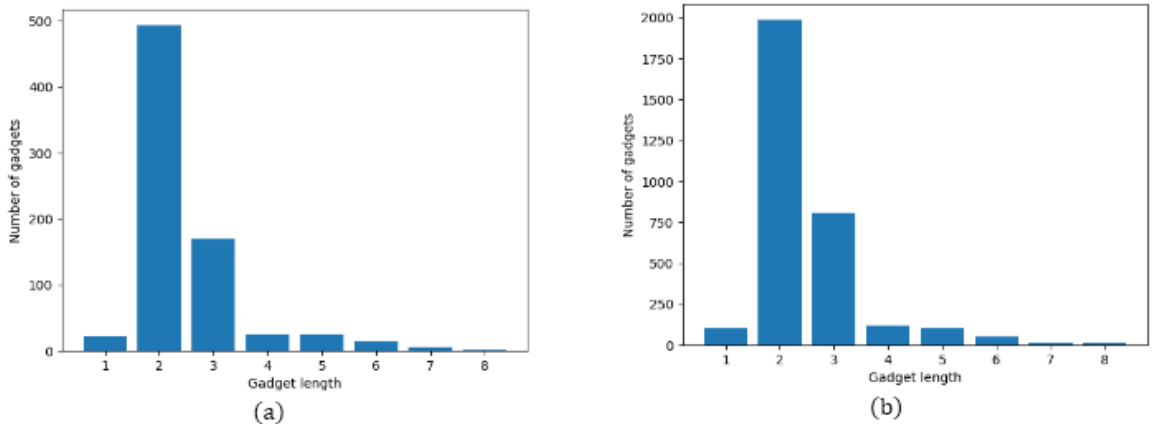
가젯 샘플링 알고리즘의 의사코드

```
gadget = ( )
gadget_length ~ P_length
if gadget_length > 1 then
    first_opcode ~ P_first
    gadget += first_opcode
    while sizeof(gadget) < gadget_length - 1 do
        next_opcode ~ P_opcode
        gadget += next_opcode
    end while
end if
gadget += 'ret'
```

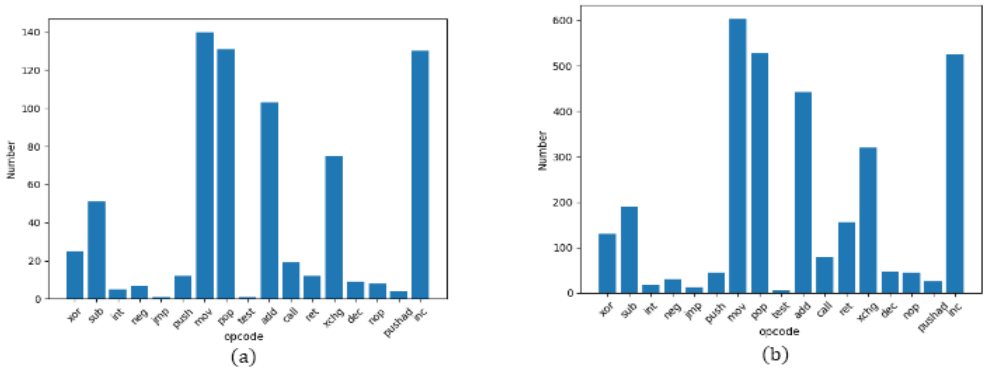
4.4 탐지 성능 실험

RNN 학습을 위한 정상 데이터는 총 30개의 정상 프로그램 중 24개의 프로그램에 존재하는 가젯이 사용되었고, 그 중 8이하의 길이를 갖는 약 2,800개의 가젯이 사용되었다. 테스트용 정상 데이터는 학습에 사용되지 않는 6개의 프로그램의 가젯이 사용되었다. 학습용 공격 데이터는 샘플링을 통해 확보한 3,200개의 가젯이 사용되었다. 테스트용 공격 데이터는 Exploit-DB에서 확보한 20개의 실제 ROP 공격 코드가 사용되었고, 총 720개의 가젯이 사용되었다.

탐지 성능 지표는 FPR(False Positive Ratio), FNR(False Negative Ratio), Accuracy 3가지이다. FPR은 테스트에 사



(그림 7) 가젯 길이 분포
(a) 기존 데이터 (b) 샘플링된 데이터



(그림 8) 가젯의 첫 번째 opcode 분포
(a) 기존 데이터 (b) 샘플링된 데이터

용된 정상 프로그램 중 ROP 공격으로 판별된 비율을 의미하고 표 3에서 $\frac{FP}{FP+TN}$에 해당한다. FNR은 테스트에 사용된 ROP 공격 코드 중 정상으로 판별된 비율을 의미한고 표 3에서 $\frac{TP}{TP+FN}$에 해당한다. Accuracy는 전체 테스트 데이터 중 올바르게 판별된 데이터의 비율을 의미하고 표 3에서 $\frac{TP+TN}{TP+FP+FN+TN}$을 의미한다. 각 T에 대하여 총 10번의 탐지 성능을 측정 실험을 진행하였다. 1회 실험에 사용된 ROP 공격 코드의 개수는 20개이고, 정상 프로그램의 개수는 6개이다. 따라서 10회 실험에 대한 FPR, FNR, Accuracy는 다음과 같이 계산된다.

$$FPR=\sum_{k=1}^{10}\frac{FP_k}{FP_k+TN_k}=\sum_{k=1}^{10}\frac{FP_k}{60}$$

$$FNR=\sum_{k=1}^{10}\frac{TP_k}{TP_k+FN_k}=\sum_{k=1}^{10}\frac{TP_k}{200}$$

$$Accuracy=\sum_{k=1}^{10}\frac{TP_k+TN_k}{TP_k+FP_k+FN_K+TN_k}=\sum_{k=1}^{10}\frac{TP_k+TN_k}{260}$$

(표 3) ROP 공격 판별의 분할표

Real / Prediction	True (ROP attack)	False (Normal)
True (ROP attack)	TP	FP
False (Normal)	FN	TN

표 4는 T값의 변화에 따른 ROP 공격 탐지의 성능을 나타낸 것이다. 성능 실험 결과 T값이 커질수록 FPR은 줄어들고, FNR은 높아지는 경향을 보인다. $T=1$인 경우 공격 의심 가젯이 한번이라도 발견되면 ROP 공격으로 간주한다. 따라서 모든 프로그램을 ROP 공격으로 탐지하여 FPR 100%, FNR 0%의 성능을 보였다. $T=10$인 경우 공격 의심 가젯이 10번 연속으로 발견되면 ROP 공격으로 간주한다. 이 경우 False Positive는 발생하진 않았지만, 전체 ROP 공격 코드의 25%를 정상으로 탐지하여 25%의 FNR을 보였다. $T=6$인 경우 1.7%의 FPR과 3.0%의 FNR을 보였지만 Accuracy의 경우 98.46%의 탐지율을 보임으로써 실험적으로 6이 최적의 T값임을 확인하였다.

(표 4) ROP 공격 탐지 성능 실험 결과

T	FPR(%)	FNR(%)	Accuracy(%)
1	100.0	0.0	76.92
2	73.3	0.0	83.08
3	25.0	0.0	94.23
4	13.3	4.0	95.38
5	16.7	6.0	93.84
6	**1.7**	**3.0**	**98.46**
7	0.0	9.0	96.54
8	8.3	13.0	93.08
9	0.0	23.0	91.15
10	0.0	25.0	90.38

5. 결 론

본 논문에서는 시퀀스 데이터 학습에 적합한 딥 러닝 모델인 RNN을 사용하여 ROP 공격을 탐지하는 방법을 제시하였다. 기존에 ROP 공격을 탐지하는 방법은 가젯의 길이를 통해 가젯의 공격 여부를 판단하는 Rule-base 방식을 사용하기 때문에 쉽게 우회가 가능하다. 그러나 제안된 방법에서는 딥 러닝을 사용하여 가젯의 명령어 패턴을 학습시키고 학습 결과를 기반으로 가젯의 공격 여부를 판단하기 때문에 Rule-base 방식에 비해 상대적으로 탐지 알고리즘을 우회하기 어렵다.

그러나 본 논문에서 제안한 방법은 최종적으로 ROP 공격을 판별하기 위한 변수 T값을 실험적으로 결정해야 한다는 한계점이 존재한다. 따라서 RNN을 학습하는 과정에서 최적의 T값 또한 학습할 수 있도록 향후 연구가 이루어질 필요가 있다.

참고문헌(Reference)

[1] P.Bania, “Security Mitigations for Return-Oriented Programming Attacks”, http://piotrbania.com/all/articles/pbania_rop_mitigations2010.pdf, 2010.

[2] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, “DROP: Detecting return-oriented programming malicious code”, 5th Internationak Conference on Information System Security, LNCS Vol 5905, pp. 163-177, 2009. https://link.springer.com/chapter/10.1007%2F978-3-642-10772-6_13

[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation”, Empirical Methods in Natural Language Processing, pp. 1724-1734, 2014.

[4] S. Hochreiter, and J. Schmidhuber, “Long Short-Term Memory”, Neural computation, 1997.

[5] P. J. Werbos, “Backpropagation through time: what it does and how to do it”, Proceedings of the IEEE, 1990.

[6] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, “Scrap: Architecture for signature-based protection from code reuse attacks”, Proceedings of the 2013 IEEE conference on High Performance Computer Architecture, 2013.

[7] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, “Defeating return-oriented rootkits with return-less kernels”. 5th ACM SIGOPS EuroSys conference, 2010.

[8] CK. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation”, PLDI ‘05 Proceedings of the ACM SIGPLAN conference on Programming language design and inplementaion, pp. 190-200, 2005.

[9] Microsoft, Data Execution Prevention(DEP), http://support.microsoft.com/kb/875352/EN-US/, 2006.

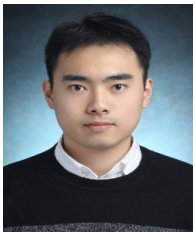
[10] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-free: Defeating return-oriented programming through gadget-less binaries”, ACSAC ‘10 Proceedings of the 26th Annual Computer Security Applications Conference, pp. 49-58, 2010.

[11] H.Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, CCS ‘07 Proceedings of the 14th ACM conference on Computer and Communicatopns Security, pp. 552-56, 2007.

[12] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, “Spoken Language Understanding Using Long Short-Term Memory Neural Network”, IEEE - Institute of Electrical and Electronics Engineers, 2014.

[13] K. Yao, B. Peng, G. Zweig, D. Yu, X. Li, and F. Gao, “Recurrent Conditional Random Field For Language Understanding”, IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2014. https://doi.org/10.1109/ICASSP.2014.6854368

◐ 저 자 소 개 ◑



김 진 섭(Jin-sub Kim)
2016년 연세대학교 의용전자공학과
2016년~현재 고려대학교 정보보호대학원 석사과정
관심분야 : 시스템 보안, 기계학습.
E-mail : less8430@naver.com

문 종 섭(Jong-sub Moon)
1981년 서울대학교 계산통계학과
1983년 서울대학교 대학원 계산통계학과
1991년 Illinois Institute of Technology 전산학 박사
1993년~현재 고려대학교 전자 및 정보공학부 교수
관심분야 : 생체인식, 운영체제, 침입탐지
E-mail : jsmoon@korea.ac.kr