Kỷ yếu Hội nghị Quốc gia lần thứ X về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR), Đà Nẵng, ngày 17-18/08/2017
DOI: 10.15625/vap.2017.000100

SỬ DỤNG PHƯƠNG PHÁP HỌC SÂU TRONG BÀI TOÁN PHÁT HIỆN MÃ ĐỘC

Nguyễn Minh Hải, Quản Thành Thơ

Khoa Khoa học và Kỹ thuật máy tính, Trường Đại học Bách khoa Thành phố Hồ Chí Minh

hainmmt@cse.hcmut.edu.vn, qttho@cse.hcmut.edu.vn

TÓM TẮT: *Hiện nay, mã độc là một vấn đề vô cùng quan trọng, thu hút sự quan tâm rất lớn và đang dần trở thành mối đe doạ thực sự đối với nền kinh tế mỗi quốc gia. Năm 2015, theo một thống kê bởi Cybersecyrity Market Report, khoảng 77 tỉ đô la đã được tiêu tốn để khắc phục những hậu quả của mã độc. Con số này ngày càng gia tăng do mức độ độc hại và tinh vi của mã độc ngày càng lớn. Để phân tích và phát hiện mã độc, hầu hết những phần mềm công nghiệp sử dụng kỹ thuật nhận dạng chữ ký. Trong kỹ thuật này, mỗi mã độc sẽ được biểu diễn dưới dạng một chuỗi bit nhị phân duy nhất và đặc trưng. Tuy nhiên, những mã độc đa hình phức tạp có thể sử dụng các phần mềm đóng gói để thay đổi chữ ký bản thân và khiến cho kỹ thuật nhận dạng chữ ký trở nên không hiệu quả. Để giải quyết những bài toán thực tế đó, bài báo này trình bày hướng tiếp cận mới sử dụng phương pháp học sâu kết hợp với công cụ BE-PUM để nhận dạng mã độc. Công cụ BE-PUM được phát triển với mục tiêu xây dựng đồ thị luồng điều khiển chính xác của một chương trình mã độc và xử lý được những kỹ thuật làm rối rắm đặc trưng của các phần mềm đóng gói như lệnh nhảy không trực tiếp, mã tự thay đổi... Đồ thị luồng điều khiển này sẽ được chuyển đổi thành hình ảnh tương ứng và được học trong mô hình học sâu để tiến hành huấn luyện. Việc sử dụng phương pháp học sâu giúp hệ thống có thể học và phân loại hiệu quả trên không gian trạng thái phức tạp được sinh ra bởi đồ thị luồng điều khiển của các mã độc. Chúng tôi đã tiến hành thử nghiệm trên một tập mã độc thực tế được thu thập từ VirusShare và đạt được kết quả nhận diện tốt hơn so với các phương pháp truyền thống.*

Từ khóa: *phân tích mã độc, phương pháp hình thức, học sâu, trí tuệ nhân tạo.*

I. GIỚI THIỆU

Sự phát triển bùng nổ của ngành công nghệ thông tin đã mang lại những lợi ích to lớn về nhiều mặt như kinh tế, văn hóa, giáo dục, y tế, thương mại, ngoại giao… Tuy nhiên, lĩnh vực này cũng tồn tại những mối hiểm họa như mất dữ liệu, rò rỉ thông tin và đặc biệt là mã độc gây tổn thất rất lớn cho các công ty, doanh nghiệp,... Mã độc (malware - malicious software) là những chương trình máy tính độc hại xâm nhập vào hệ thống một cách trái phép với mục tiêu là ăn cắp thông tin, phá hủy hay làm hư hỏng hệ thống [1]. Malware được chia thành nhiều loại như virus, worm, spyware, trojan... Malware có thể sống kí sinh trên các tập tin thực thi trong hệ thống, trên các tập tin ứng dụng con người thường tải về hay do các hacker lan truyền thông qua tập tin trên mạng internet [2].

Để phát hiện mã độc, thông thường có 2 phương pháp chính, có thể kể đến phương pháp nhận diện chữ ký (signature recognition) [3] và mô phỏng (emulation) [4] quá trình thực thi chương trình trong môi trường có kiểm soát, thường là hộp cát (sandbox). Chữ ký là những cấu trúc mẫu bit đặc trưng của từng loại malware. Các phần mềm phát hiện malware trong công nghiệp sẽ sử dụng kĩ thuật so khớp mẫu (signature matching), từ đó đối chiếu thông tin cấu trúc bit của đối tượng với thông tin mã độc có trong cơ sở dữ liệu (database). Tuy nhiên, vấn đề cơ sở dữ liệu mã độc của phần mềm chưa được cập nhật kịp thời sẽ dẫn đến việc so khớp mẫu chưa đạt được hiệu quả tốt nhất. Hơn thế nữa, signature detection gặp phải khó khăn lớn do những virus thế hệ mới áp dụng những kỹ thuật rắc rối hóa (obfuscation technique) để làm thay đổi chữ ký đặc trưng [5]. Ví dụ trong đoạn mã dưới đây trình bày những khó khăn trong vấn đề phân tích malware bằng phương pháp signature matching. Trong Code 1a là đoạn mã của virus Avron nổi tiếng. Code 1b cho thấy một biến thể khác của Avron, trong đó phép XOR được áp dụng để đưa giá trị 0 lên đỉnh stack. Trong Code 1c là một biến thể khác của Avron. Thay vì sử dụng thanh ghi *ebx* như hình 1a, biến thể này sử dụng thanh ghi *eax* để đưa giá trị 0 lên đỉnh stack. Như vậy, cùng 1 hành vi, Avron có thể sử dụng nhiều câu lệnh khác nhau, và dẫn đến nhiều biến thể với các chữ ký khác nhau. Điều này đã gây ra rất nhiều khó khăn trong việc nhận dạng mã độc Avron của phương pháp signature matching truyền thống.

0: mov ebx, 0 1: push ebx 2: call ds:GetModuleHandleA (a) Mã độc Avron nguyên bản	0: xor ebx, ebx 1: push ebx 2: call ds:GetModuleHandleA (b) Biến thể của Avron
start: 0: cmp ebx, 0 1: jl then lelse: 2: mov ebx, offset start + 1 3: jmp lcont lhalt: 4: halt (c) Một biến thể khác của Avron	lthen: 5: mov ebx, offset l1 + 4 6: mov eax, 0 7: push eax 8: call ds:GetModuleHandleA l1: 9: sub ebx, 3 lcont: 10: sub ebx, 1 11: jmp ebx

Code 1. Code mã độc

Các phương pháp mô phỏng thực thi (emulation) được đề xuất để xử lý các trường hợp này. Ý tưởng chính của emulation là xây dựng một sandbox để khám phá hành vi của malware, bằng cách mô phỏng toàn bộ quá trình thực thi của hệ thống và đặc biệt là hoạt động của APIs [4, 5]. Các tập tin mục tiêu có thể được hiện thực trong một môi trường ảo, hoặc sandbox, để khám phá các hành vi của phần mềm độc hại. Tuy nhiên, vấn đề của cách tiếp cận này là một số mã độc có thể sử dụng các kỹ thuật chống mô phỏng, bao gồm chống lại gỡ lỗi và chống tháo gỡ, gây phiền nhiễu cho việc phát hiện phần mềm độc hại trong môi trường mô phỏng. Hơn thế nữa, đây là một kỹ thuật rất phức tạp, tốn thời gian và đòi hỏi chi phí lớn. Khi lây nhiễm vào các tập tin nạn nhân, không gian xử lý của bài toán trở nên quá lớn và quá phức tạp. Thậm chí hãng diệt virus nổi tiếng Symantec còn tuyên bố rằng các kỹ thuật antivirus truyền thống đang không theo kịp với sự phát triển của của các kỹ thuật làm rối rắm phức tạp của các mã độc đa hình.

Trong bài báo này, chúng tôi đề xuất một hướng tiếp cận mới, sử dụng phương pháp học sâu (deep learning) kết hợp với công cụ BE-PUM để nhận dạng malware. Chúng tôi đã phát triển công cụ BE-PUM (Binary Emulator for PU shdown Model generation) với mục tiêu xây dựng đồ thị luồng điều khiển (CFG - Control Flow Graph) chính xác của một chương trình mã nhị phân và xử lý được những kỹ thuật làm rối rắm (obfuscation) đặc trưng của mã độc như lệnh nhảy không trực tiếp, code tự thay đổi... CFG này sẽ được chuyển đổi thành hình ảnh tương ứng và được học trong mô hình học sâu (deep learning) để tiến hành huấn luyện. Để kiểm chứng tính hiệu quả của hướng tiếp cận này, chúng tôi đã tiến hành thí nghiệm trên tập 63771 malware thực từ VirusShare.

Cấu trúc bài báo được mô tả như sau. Phần II giới thiệu ngắn gọn những kiến thức nền tảng về phương pháp học sâu và công cụ BE-PUM. Phần III trình bày phương pháp đề xuất để nhận dạng mã độc. Trong phần IV, chúng tôi giới thiệu về thí nghiệm. Phần V trình bày kết luận và một số hướng nghiên cứu trong tương lai.

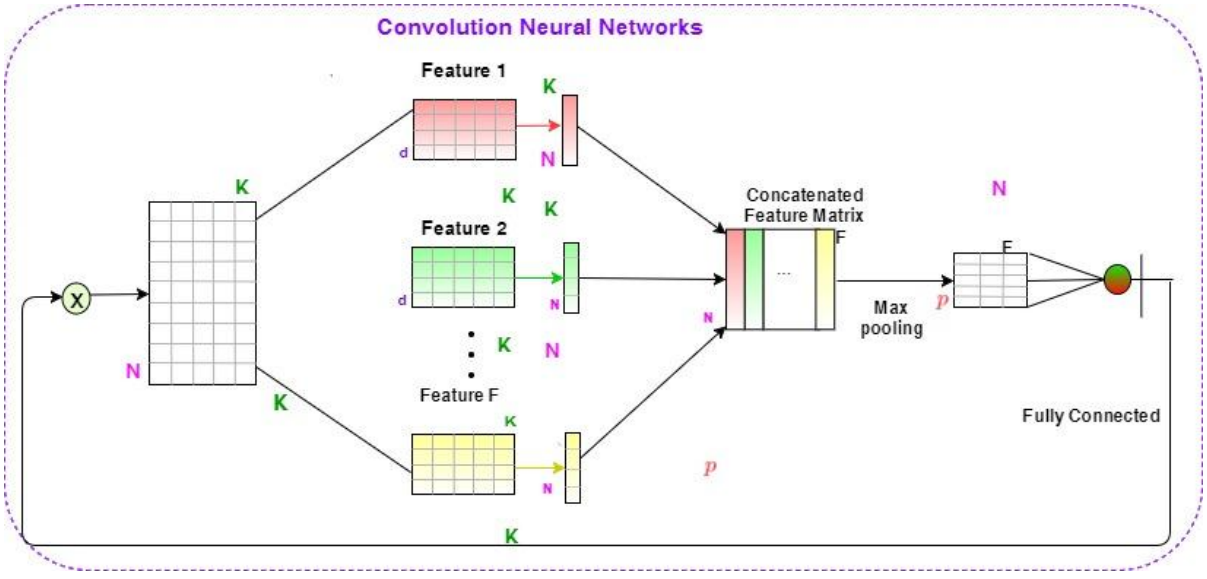
II. KIẾN THỨC NỀN TẢNG

A. Phương pháp học sâu (deep learning)

Phương pháp học sâu (deep learning) [6] là một nhánh trong lĩnh vực học máy và đang dần trở nên phổ biến trong những năm gần đây. Deep learning tập trung xử lý các bài toán trong mạng thần kinh nhân tạo và được ứng dụng trong nhiều lĩnh vực như nhận diện giọng nói, thị giác máy tính và xử lý ngôn ngữ tự nhiên. Deep learning gồm nhiều mô hình như Artificial Neural Network (ANN), Recurrent Neural Networks (RNN), Graph Neural Networks (GNN), Convolutional Neural Network (CNN)… Trong đó, Convolutional Neural Network là kĩ thuật nhận dạng hình ảnh đạt hiệu quả cao. CNN là mô hình được lựa chọn trong bài báo này để nhận dạng mã độc.

1. Mạng tích chập (Convolution Neural Networks - CNN)

CNN (Convolutional Neural Network) [7, 8] là một trong những mô hình deep learning tiên tiến, được ứng dụng để xây dựng những hệ thống thông minh với độ chính xác cao. CNN được sử dụng đầu tiên trong lĩnh vực xử lý tín hiệu số (signal processing). Dựa trên nguyên lý biến đổi thông tin, các nhà khoa học đã áp dụng kỹ thuật này vào xử lý ảnh, video số và chữ viết. Mô hình cơ bản của CNN bao gồm một số lớp (layers) của tích chập (convolution) kết hợp với hàm kích hoạt phi tuyến (nonlinear activation function) như ReLU, Sigmoid hay Tanh. Từ đó, CNN tạo ra những thông tin trừu tượng hơn (abstract/higher-level) cho các layer tiếp theo. Hình 1 trình bày mô hình CNN chuẩn, bao gồm các lớp tích chập (convolutional layer), pooling và fully connected [9]. Đầu vào của mô hình CNN là ảnh và đầu ra là lớp tương ứng.



Hình 1. Kiến trúc của mạng tích chập

Convolution layer: sử dụng phép toán tích chập (convolution), layer này có vai trò xử lý thông tin bằng cách quét các bộ lọc (filter) có kích thước cố định trên dữ liệu đầu vào và sinh ra những dữ liệu tinh hơn dựa trên tính bất biến, kết hợp cục bộ, chia sẻ trọng số.

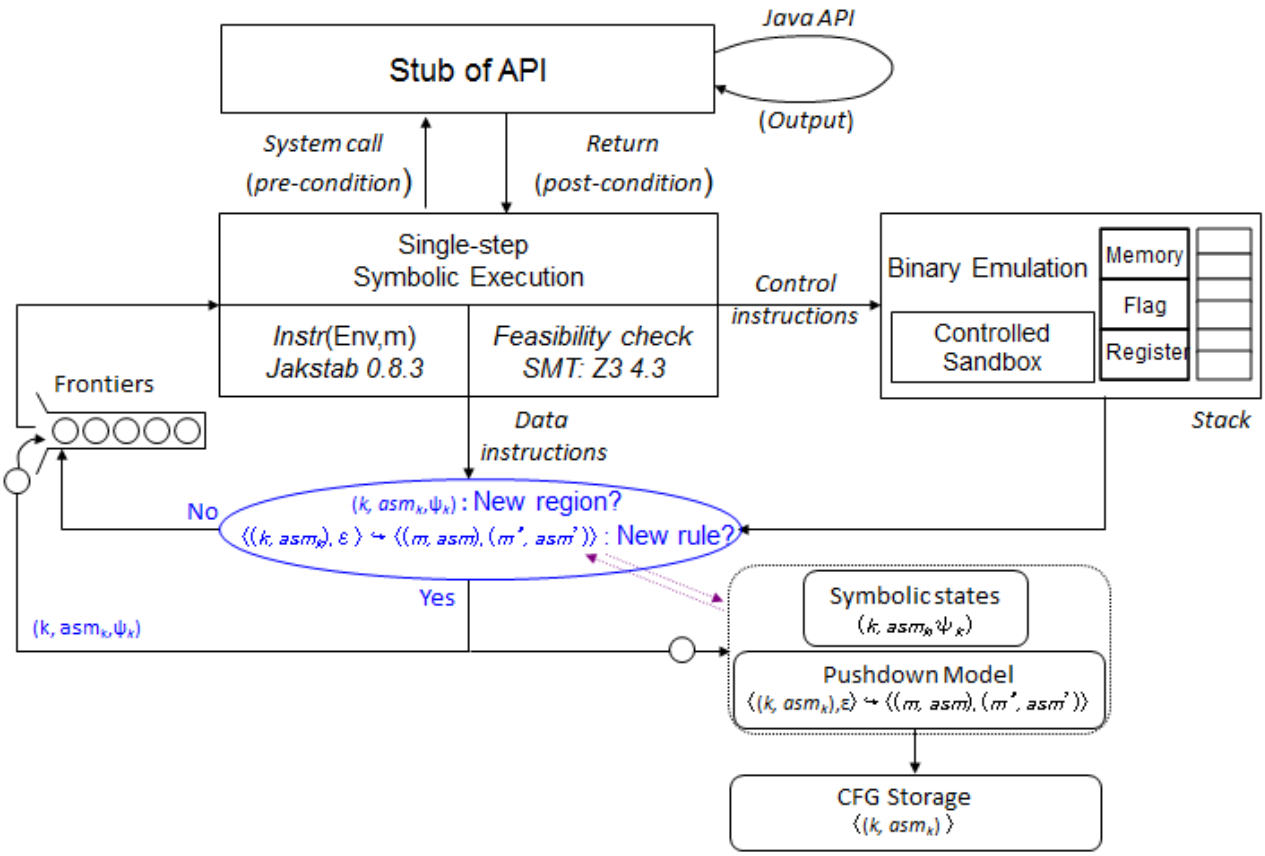
Pooling layer: Chuẩn hóa đầu ra các vector để bảo đảm các vector có số chiều là như nhau. Ngoài ra, vai trò của layer này là để loại bỏ các thông tin nhiễu.

Fully connected layer: Các neuron trong lớp kết nối đầy đủ (fully connected) kết nối với tất cả neuron của lớp trước đó thông qua trọng số *w* (weighted vector). Mục tiêu của lớp này là để học và điều chỉnh trọng số *w* thông qua cơ chế backtracking.

B. BE-PUM

1. *Giới thiệu BE-PUM*

BE-PUM (Binary Emulation for Pushdown Model generation of Malware) [10] là một công cụ xây dựng luồng điều khiển của một tập tin thực thi dựa trên kỹ thuật thực thi ký hiệu động [11]. BE-PUM sử dụng thư viện mã nguồn mở Jackstab [12, 13] để dịch ngược mã nhị phân cho từng câu lệnh hợp ngữ tương ứng với từng câu lệnh thực thi của tập tin và chương trình SMT Z3.4.4 để giải các điều kiện, từ đó tìm ra đường đi có tính khả thi.

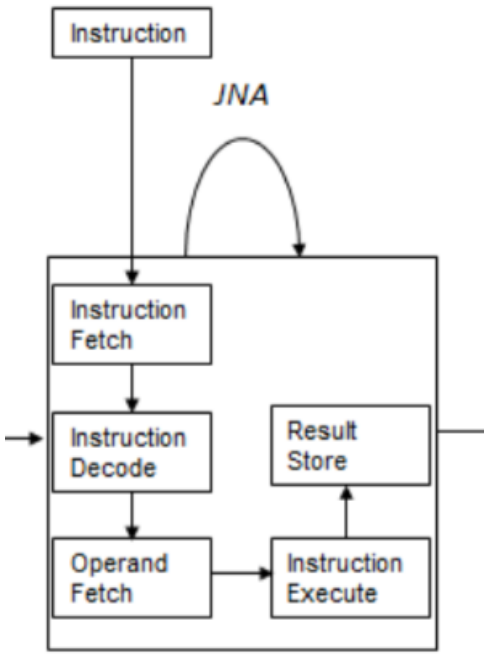


Hình 2. Kiến trúc của BE-PUM

Kiến trúc của hệ thống BE-PUM bao gồm 3 thành phần chính: symbolic execution, binary emulation và CFG storage. Hình 2 mô tả kiến trúc của hệ thống BE-PUM. Trong đó, vai trò của các thành phần này được mô tả như sau.

- Thành phần symbolic execution sử dụng Jackstab để dịch ngược và chuyển đổi thành mã nhị phân thành câu lệnh hợp ngữ tương ứng. Trong trường hợp câu lệnh hợp ngữ là câu lệnh chỉ tác động tới môi trường thực thi, cụ thể là các câu lệnh tính toán, câu lệnh ghi hoặc đọc trong bộ nhớ, bao gồm cả stack, môi trường thực thi được thay đổi tương ứng. Ngoài ra, vị trí của câu lệnh tiếp theo được xác định bằng vị trí của câu lệnh hiện tại cộng kích thước câu lệnh. Nếu câu lệnh tiếp theo là câu lệnh điều khiển, cụ thể là các câu lệnh gọi hàm, câu lệnh trả về từ hàm, câu lệnh nhảy và câu lệnh nhảy có điều kiện, thì câu lệnh tiếp theo sẽ được tính toán thông qua quá trình thực thi ký hiệu.
- Thành phần binary emulation là thành phần quan trọng trong tổng thể kiến trúc của BE-PUM. Để có thể xử lý một câu lệnh hợp ngữ, hệ thống BE-PUM sẽ giả lập các thành phần của hệ thống, cụ thể là mô hình bộ nhớ

của BE-PUM. Mô hình này bao gồm tập 9 cờ được sử dụng trong hệ thống (AF, CF, DF, IF, OF, PF, SF, TF, và ZF), tập 20 thanh ghi (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, CS, DS, ES, FS, GS, SS, EIP, EFLAGS và 8 thanh ghi debug DRO, DR1, DR2, DR3, DR4, DR5, DR6, DR7, DR8), tập các giá trị bộ nhớ được lưu trữ trong đó một phần của bộ nhớ cho stack được giới hạn bởi thanh ghi esp và thanh ghi ebp. Bộ nhớ được giả lập trong BE-PUM được lưu trữ dưới dạng từng byte. Ngoài việc giả lập câu lệnh hợp ngữ, thì API cũng được giả lập thông qua sử dụng JNA (Java Native Access) cho phép gọi trực tiếp từ các thư viện liên kết động, trong đó giá trị trả về của các API sẽ được lưu trữ trong các thanh ghi tương ứng. Đối với các API đặc biệt tác động trực tiếp đến môi trường thực, một sandbox sẽ được xây dựng cho quá trình xử lý các API này. Hình 3 mô tả các kiến trúc xử lý một câu lệnh thực thi trong hệ thống BE-PUM. Sau khi được nạp vào bộ nhớ, câu lệnh sẽ được giải mã, lấy các toán hạng và tiến hành mô phỏng quá trình thực thi. Kết quả của câu lệnh sẽ được ghi lại vào bộ nhớ.



Hình 3. Kiến trúc xử lý API trong BE-PUM

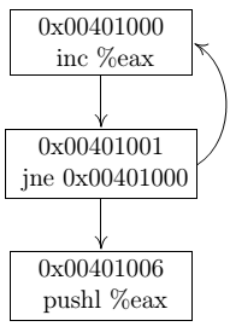
- Thành phần CFG storage được sử dụng để lưu trữ một CFG node và CFG edge sau khi được tính toán chính xác. CFG storage sẽ được sử dụng để xây dựng mô hình quá trình thực thi cuối cùng của tập tin được phân tích.

2. *Ví dụ hoạt động của BE-PUM*

Kết quả của quá trình phân tích một tập tin thực thi là một mô hình quá trình thực thi. Mô hình được sinh ra bởi hệ thống BE-PUM được biểu diễn dưới dạng Control Flow Graph hay CFG của chương trình. Một CFG là một tập của các đỉnh và cạnh. Trong đó một đỉnh của CFG bao gồm địa chỉ của câu lệnh và câu lệnh hợp ngữ tại địa chỉ đó, một cạnh của CFG là cạnh nối giữa 2 node của CFG. Xem xét đoạn code sau.

```
00401000  pushl 0x00401007
00401005  xor eax, eax
00401007  jne 0x00401005
0040100B  call ss:[esp]
```

Code 2. Code ví dụ của BE-PUM



Hình 4. CFG được xây dựng bởi BE-PUM

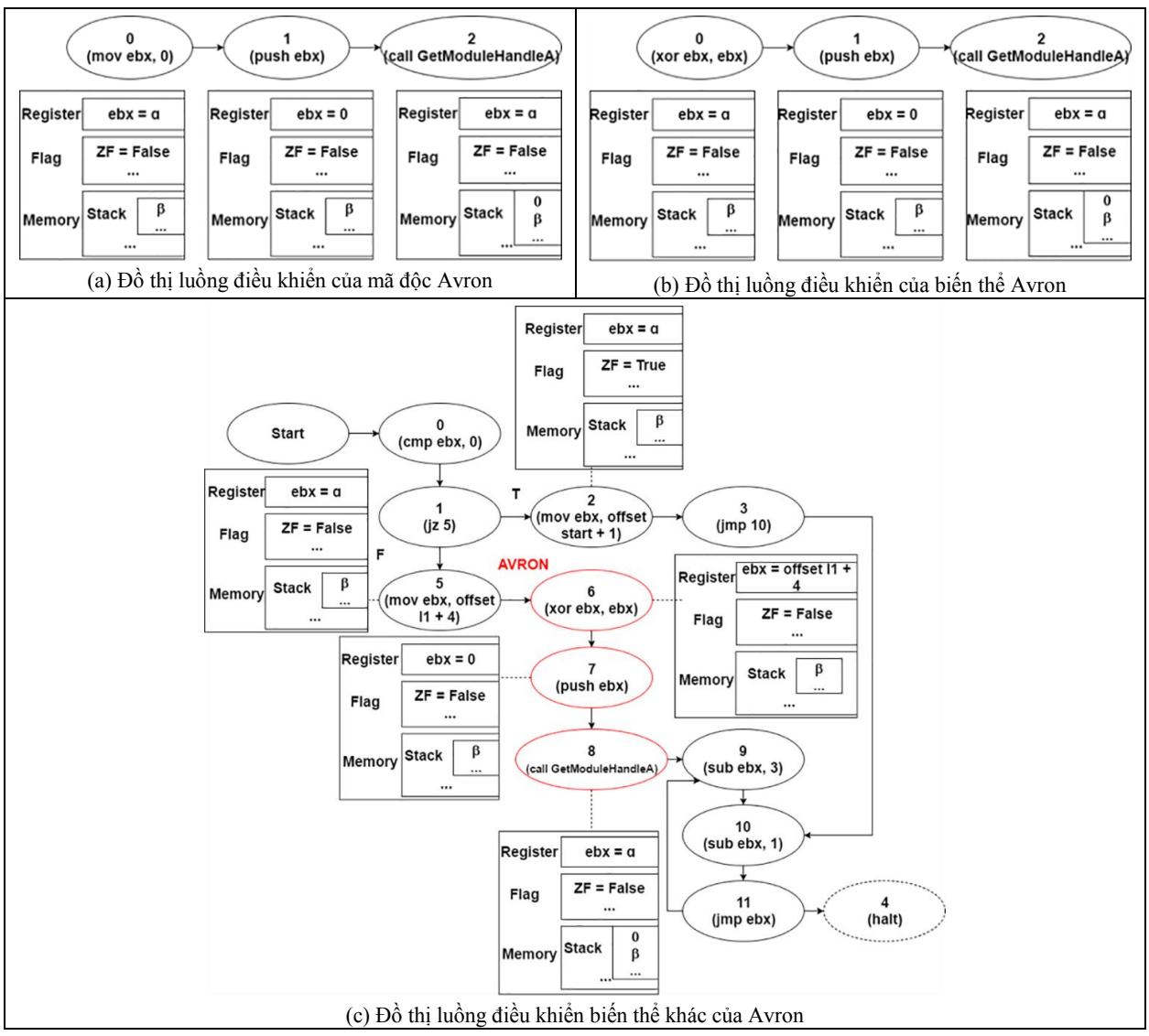
Kết quả quá trình phân tích trên hệ thống BE-PUM sẽ sinh ra được CFG của chương trình thực thi và được mô tả trong Hình 4.

III. NHẬN DẠNG MÃ ĐỘC SỬ DỤNG PHƯƠNG PHÁP HỌC SÂU

A. Trong bài báo này, chúng tôi đưa ra một hướng tiếp cận mới sử dụng phương pháp học sâu để phát hiện mã độc. Trước tiên chúng tôi sử dụng công cụ BE-PUM để xây dựng một đồ thị luồng điều khiển (CFG - Control flow graph) từ mã nhị phân cần phân tích. Tiếp đó, chúng tôi biến đổi CFG thành dạng ma trận kề sao cho các tính chất ban đầu của chương trình gốc vẫn được giữ nguyên trong ma trận kề này. Ma trận kề này sẽ được chuyển thành các ảnh. Các biến thể của cùng một mẫu mã độc sẽ được biểu diễn như các đối tượng tương tự nhau trong ảnh sinh ra. Nhờ vậy, chúng tôi có thể sử dụng kỹ thuật deep learning để nhận dạng các đối tượng này một cách hiệu quả.

B. Xây dựng CFG từ mã nhị phân của chương trình

Có rất nhiều công cụ cho phép xây dựng CFG từ mã nhị phân của chương trình, như IDA Pro [14], Jackstab và BE-PUM. Hình 5 biểu diễn CFG sinh ra bởi đoạn mã Code 1 từ công cụ BE-PUM. Một CFG của chương trình là một đồ thị có hướng, trong đó một đỉnh tương ứng với một câu lệnh trong chương trình gốc. Cạnh nối giữa các đỉnh biểu diễn luồng thực thi của chương trình. Mỗi đỉnh được kết nối với một trạng thái môi trường (environment state), phản ánh thông tin của các thanh ghi, bộ nhớ và stack của hệ thống trước khi câu lệnh tương ứng với đỉnh này được thực thi. Ví dụ như ở Hình 5a, trước khi lệnh (0) được thực thi, giá trị của thanh ghi *ebx* chưa được xác định nên sẽ được biểu diễn bằng một giá trị trừu tượng (symbolic value) α. Sau khi lệnh (0) được thực thi, giá trị của thanh ghi *ebx* sẽ được xác định bằng một giá trị cụ thể 0. Tiếp đó, sau khi giá trị của ebx được đẩy vào *stack* bởi lệnh (1), giá trị trên đỉnh stack sẽ được biểu diễn bằng giá trị cụ thể 0, thay vì là giá trị trừu tượng β.



(a) Đồ thị luồng điều khiển của mã độc Avron

(b) Đồ thị luồng điều khiển của biến thể Avron

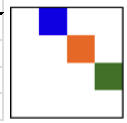
(c) Đồ thị luồng điều khiển biến thể khác của Avron

Hình 5. Đồ thị luồng điều khiển của mã độc

C. Chuyển đổi CFG thành hình ảnh

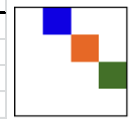
Trong phần này, chúng tôi trình bày quá trình chuyển đổi CFG thành một ma trận ảnh (pixel matrix). Trước tiên, mỗi đỉnh trên CFG sẽ được xem là một trạng thái *(state)*. Mỗi state được biểu diễn như bộ ba <*Register*, *Flag*, *Memory* >.

	0	1	2	End
0		0		
1			18	
2				Δ
End				



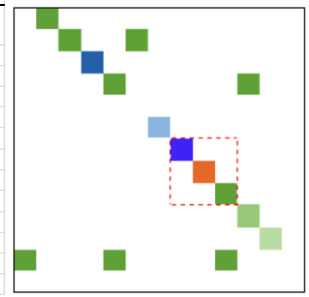
(a) Ma trận kề và hình ảnh tương ứng của Avron virus

	0	1	2	End
0		0		
1			18	
2				Δ
End				



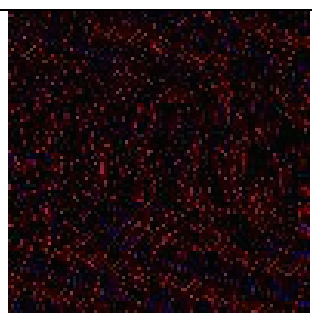
(b) Ma trận kề và hình ảnh tương ứng của biến thể Avron

	0	1	2	3	4	5	6	7	8	9	10	11	End
0		Δ											
1			Δ	75		Δ							
2				82									
3					Δ						Δ		
4													
5							58						
6								0					
7									18				
8										Δ			
9											156		
10												212	
11	Δ				Δ					Δ			
End													

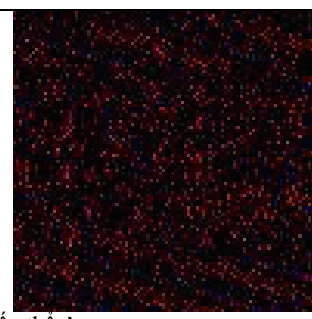


(c) Ma trận kề và hình ảnh tương ứng của biến thể Avron

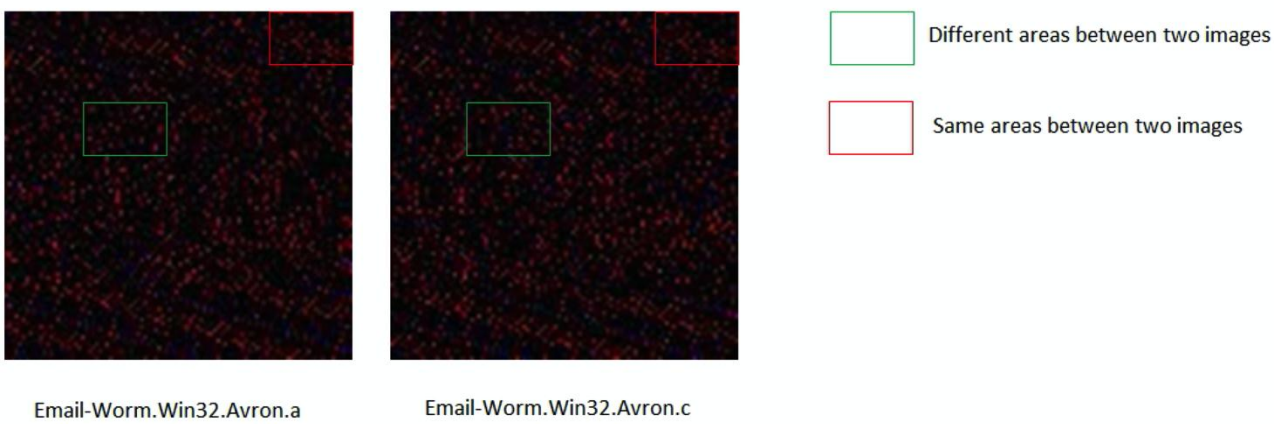
Hình 6. Ma trận kề và hình ảnh tương ứng



(a) Hình ảnh Avron



(b) Hình ảnh biến thể Avron



(c) So sánh ảnh

Hình 7. Hình ảnh của Avron và các biến thể

Nếu tồn tại một cạnh nối từ đỉnh *i* đến đỉnh *j* của CFG, giá trị *(i,j)* của ma trận được xác định bằng công thức |*state* (*j*) - *state* (*i*) |. Chúng ta có giá trị của |*state* (*j*) - *state* (*i*)| là bộ ba <*Red, Green, Blue*> trong đó.

Red = |*Register*(*j*) - *Register* (*i*)|
Green = |*Flag*(*j*) - *Flag*(*i*)|
Blue = |*Stack* (*j*) - *Stack*(*i*)|

Do trong hệ thống có 8 thanh ghi (Eax, Ebx, Ecx, Edx, Esi, Edi, Esp, Ebp) nên

$$|Register(j)\text{-}Register(i)|= \frac{(|Eax(j) - Eax(i)| + |Ebx(j) - Ebx(i)| + |Ecx(j) - Ecx(i)| + |Edx(j) - Edx(i)| + |Esi(j) - Esi(i)| + |Edi(j) - Edi(i)| + |Ebp(j) - Ebp(i)| + |Esp(j) - Esp(i)|)}{8} \quad (1)$$

Tương tự, có 9 cờ, *(CF, PF, AF, ZF, SF, TF, IF, DF, OF)*, nên:

$$|Flag(j) - Flag(i)| = \frac{(|CF(j) - CF(i)| + |PF(j) - PF(i)| + |AF(j) - AF(i)| + |ZF(j) - ZF(i)| + |SF(j) - SF(i)| + |TF(j) - TF(i)| + |IF(j) - IF(i)| + |DF(j) - DF(i)| + |OF(j) - OF(i)|)}{9} \quad (2)$$

Bộ nhớ CFG trong BE-PUM được biểu diễn, *Memory* = {*(Loc(k), Val(k))* | *0≤k≤n*}, nên:

$$|Memory(j) - Memory(i)| = \frac{\sum_{k=1}^{n} |Val_j(\mathrm{k}) - Val_i(\mathrm{k})| \text{ if } Loc_j(\mathrm{k}) == Loc_i(\mathrm{k})}{n} \quad (3)$$

Ngoài ra, khi tính toán giá trị |X-Y| trong các công thức trên, do X và Y có thể là giá trị trừu tượng hay cụ thể, giá trị của |X-Y| được tính toán như sau:

$$|X - Y| = \begin{cases} |X - Y| \text{ nếu kết quả là giá trị cụ thể} \\ 255 \text{ nếu kết quả là giá trị trừu tượng} \end{cases}$$

Bộ ba <Red, Green, Blue> được kết hợp thành giá trị hexa và chuyển thành biểu diễn màu trong ảnh. Hình 6 biểu diễn giá trị ma trận kề và ảnh sinh ra bởi mã độc Avron. Hình 7 trình bày những ảnh được sinh ra bởi toàn bộ chương trình mã độc. Trong Hình 7, hình ảnh được sinh ra bởi những biến thể khác nhau của Avron có những vùng giống nhau. Đặc tính này cho phép chúng tôi sử dụng phương pháp học sâu trong bài toán nhận diện và phân tích mã độc.

D. Sử dụng phương pháp học sâu để nhận dạng mã độc

Chúng tôi sử dụng kỹ thuật deep learning và mô hình CNN để phát hiện mã độc. Kiến trúc tổng quan mạng deep learning được trình bày trong Hình 1. Trước tiên, một cửa sổ *kernel,* gọi là ma trận tích chập (convolution matrix), sẽ được sử dụng để rút ra các đặc trưng (features) từ ma trận ban đầu. Tiếp theo đó, một lớp pooling sẽ được sử dụng để giữ lại các giá trị quan trọng từ các đặc trưng này. Quá trình thay đổi giữa lớp convolution và lớp pooling có thể lặp lại nhiều lần. Cuối cùng, một lớp kết nối đầy đủ (fully connected layer) sẽ kết nối các kết quả học được, so sánh với kết quả mong đợi và điều chỉnh các trọng số trên toàn matrix.

Như vậy, để giải quyết được bài toán xác định các vùng giống nhau trong ảnh, chúng ta cần phải xác định (i) chiều dài của kernel để mạng học sâu có thể học được toàn bộ (entire) mẫu virus trong một cửa sổ kernel và (ii) xác định phép pooling phù hợp. Để giải quyết bài toán (i), cửa sổ kernel sẽ có chiều dài (*m x n)* với m là đường đi dài nhất trên tất cả CFG của các mẫu được học và n là số điểm mục tiêu đối đa (maximal targets) của một lệnh nhảy động trên các mẫu virus. Hơn thế nữa, phép max pooling truyền thống là được sử dụng để giữ lại các giá trị đặc trưng quan trọng. Ngoài ra, hệ thống của chúng tôi chỉ cần một lớp convolution và pooling để có thể phát hiện virus hiệu quả.

IV. THÍ NGHIỆM

A. Mô tả thí nghiệm

Chúng tôi đã tiến hành thí nghiệm nhận diện mã độc trên hệ thống WinXP SP3, AMD Athlon II X4 635 2.9GHz, 8GB RAM, JDK 1.8. Thí nghiệm được tiến hành trên mẫu mã độc thực tế được lấy từ 63771 mẫu mã độc từ nguồn VirusShare. Để huấn luyện và phát hiện mã độc, chúng tôi sử dụng công cụ BE-PUM để sinh đồ thị luồng điều khiển từ mã nhị phân của mã độc và chuyển đổi thành ảnh tương ứng. Sau đó, ảnh kết quả này được nạp vào mô hình CNN để tiến hành huấn luyện. Để đánh giá, chúng tôi sử dụng kỹ thuật kiểm chứng chéo (cross validation) *k*-fold với *k* =10. Chúng tôi tiến hành so sánh phương pháp đề xuất với những kỹ thuật nổi tiếng như SVM và Artificial Immune System (AIS)[15].

B. Kết quả thí nghiệm

Bảng 1 trình bày kết quả thí nghiệm với những tập mẫu có kích thước khác nhau. Chúng tôi so sánh các thông số về độ chính xác (precision), độ đo tính toàn vẹn (recall) và trung bình điều hòa (F-measure) để đánh giá các phương pháp phân loại. Phương pháp AIS và phương pháp của chúng tôi cho ra một kết quả tốt hơn so với SVM do những phương pháp này hoạt động hiệu quả trên những mã độc tự thay đổi (self-modified malware). Ngoài ra, khi cung cấp đủ số lượng tập huấn luyện, phương pháp học sâu có kết quả với độ chính xác cao nhất. Tuy nhiên, nhược điểm của phương pháp này là thời gian huấn luyện lâu so với các phương pháp khác.

Bảng 1. Kết quả thí nghiệm

Size of sample		1500	2000	2500	5000	10000	20000	40000	50000	63771
SVM	Precision	83.44%	83.07%	82.19%	79.89%	77.98%	77.16%	75.18%	75.90%	74.93%
	Recall	79.17%	78.97%	77.44%	77.16%	76.54%	75.88%	75.07%	74.36%	72.77%
	F-measure	81.25%	80.97%	79.74%	78.5%	77.25%	76.51%	75.12%	75.12%	73.83%
AIS	Precision	90.36%	90.17%	88.68%	88.11%	87.63%	85.79%	85.31%	84.86%	84.06%
	Recall	92.44%	91.83%	89.38%	89.07%	88.48%	87.29%	86.48%	85.88%	85.47%
	F-measure	91.39%	90.99%	89.03%	88.59%	88.05%	86.53%	85.89%	85.37%	84.76%
CNN	Precision	98.87%	98.16%	97.06%	96.95%	96.16%	95.88%	95.52%	93.98%	93.28%
	Recall	91.34%	90.57%	88.68%	88.18%	87.64%	86.39%	85.49%	84.86%	84.40%
	F-measure	94.96%	94.21%	92.68%	92.36%	91.70%	90.89%	90.23%	89.19%	88.62%

V. KẾT LUẬN

Bài báo này đã đề xuất hướng tiếp cận mới để nhận diện mã độc bằng phương pháp học sâu. Phương pháp này kết hợp công cụ BE-PUM để xây dựng mô hình và phương pháp học sâu để huấn luyện và nhận dạng mã độc. Kết quả thí nghiệm trên tập malware đã cho thấy hướng tiếp cận này có kết quả chính xác hơn so với hướng tiếp cận truyền thống. Tuy nhiên, mặt hạn chế của phương pháp này là thời gian xử lý khá lâu. Để giải quyết vấn đề này, chúng tôi sẽ áp dụng giải thuật song song hóa để giảm thời gian thực thi. Đây là một hướng nghiên cứu trong tương lai.

TÀI LIỆU THAM KHẢO

[1] BitDefender, “Anti-virus Technology Whitepaper”, Technical report, Washington, DC. USA, 2007.

[2] Shih Dong-Her, Chiang Hsiu-Sen, "E-mail Viruses: How Organizations Can Protect Their E-mails", Online Information Review, Vol. 28, Issue: 5, pp.356-366, doi:10.1108/14684520410564280.

[3] Filiol E., “Malware Pattern Scanning Schemes Secure against Black-box Analysis”, Journal in Computer Virology, Vol. 2, pp. 35-50, 2006.

[4] Izumida T., Futatsugi K. and Mori A., “A Generic Binary Analysis Method for Malware”. International Workshop on Security, pp. 199-216, 2010.

[5] A. Mori, T. Izumida, T. Sawada, and T. Inoue, “A tool for analyzing and detecting malicious mobile code,” in ICSE 2006, pp. 831-834, 2006.

[6] Deng, L., Yu, D. (2014). "Deep Learning: Methods and Applications". In Foundations and Trends in Signal Processing. **7** (3-4): 199-200. 2014.

[7] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S., McClosky, D.: “The stanford corenlp natural language processing toolkit”. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations. pp. 55-60, 2014.

[8] Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., Potts, C., et al.: “Recursive deep models for semantic compositionality over a sentimenttreebank.” In Proceedings of the conference on empirical methods in natural languageprocessing (EMNLP). vol. 1631, p. 1642. Citeseer (2013).

[9] Wang, X., Jiang, W., Luo, Z.: “Combination of convolutional and recurrent neural network for sentiment analysis of short texts”. In: COLING 2016, 26th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, December 11-16, 2016, Osaka, Japan. pp. 2428-2437 (2016).

[10] Nguyen, M. H., Nguyen, T. B., Quan, T. T., & Ogawa, M. (2013), “A Hybrid Approach for Control Flow Graph Construction from Binary Code”. In Proceeding of Software Engineering Conference (APSEC), 2013 20th Asia-Pacific, Vol. 2, pp. 159-164, 2013

[11] Nguyen, M. H., Ogawa, M., & Tho, Q. T., “Obfuscation Code Localization Based on CFG Generation of Malware”. In Proceeding of International Symposium on Foundations and Practice of Security, pp. 229-247, LNCS 9482, 2015.

[12] Kinder, J., Zuleger, F., & Veith, H. (2009), “An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries”. In Proceeding of International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 214-228, 2009.

[13] J. Kinder and D. Kravchenko. Alternating control flow reconstruction. In Proceeding of International Workshop on Verification, Model Checking, and Abstract Interpretation,, pages 267-282, 2012. LNCS 7148.

[14] IDA Pro, https://www.hex-rays.com/products/ida/ (accessed 15th June 2017).

[15] Al-Enezi, J., Abbod, M., AI-Sharhan, S. “Artificial Immune Systems - Models, Algorithms and Applications”. International Journal of Research and Reviews in Applied Sciences, Paper 3(2), 118-131, 2010.

MALWARE DETECTION USING DEEP LEARNING

Nguyen Minh Hai, Quan Thanh Tho

ABSTRACT: *Nowadays, malware has been becoming a real threat to each nation. In 2015, according to a report by Cybersecyrity Market Report, more than 77 billion dollars has been spent on the war against malware and its destruction. For malware detection, most of industrial anti-virus software tends to identify signature patterns characterizing malwares. However, since most of the advanced polymorphic malwares are either packed or obfuscated, they can easily change their signature to evade this technique. This paper proposes a novel approach for malware detection using a combination of BE-PUM tool and deep learning. First, BE-PUM tool is applied for generating a precise control flow graph (CFG) from binary code of malware. The CFG is then converted to an image. Finally, deep learning is applied to effectively identify the similar objects in the generated images. As deep learning is very effective for learning and classifying in complicated state space generated from CFG of real-world malware, our proposed technique can achieve very high accuracy. We have performed the experiments on real-world malwares taken from VirusShare for checking the effectiveness of our approach and the result is very promising.*

.