ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

NGUYỄN MINH HẢI

KẾT HỢP PHÂN TÍCH TĨNH VÀ KIỂM TRA ĐỘNG TRONG VIỆC XÂY DỰNG ĐỒ THỊ LUỒNG ĐIỀU KHIỂN PHỤC VỤ PHÂN TÍCH MÃ NHỊ PHÂN

Chuyên ngành: KHOA HỌC MÁY TÍNH
Mã số chuyên ngành: 62.48.01.01

TÓM TẮT LUẬN ÁN TIẾN SĨ KỸ THUẬT

TP. HỒ CHÍ MINH NĂM 2018

Công trình được hoàn thành tại **Trường Đại học Bách Khoa – ĐHQG-HCM**

Người hướng dẫn khoa học 1: PGS. TS. QUẢN THÀNH THƠ
Người hướng dẫn khoa học 2:

Phản biện độc lập 1:
Phản biện độc lập 2:

Phản biện 1:
Phản biện 2:
Phản biện 3:

Luận án sẽ được bảo vệ trước Hội đồng chấm luận án họp tại
....
....
vào lúc giờ ngày tháng năm

Có thể tìm hiểu luận án tại thư viện:

- Thư viện Khoa học Tổng hợp Tp. HCM
- Thư viện Trường Đại học Bách Khoa – ĐHQG-HCM

DANH MỤC CÔNG TRÌNH ĐÃ CÔNG BỐ

Tạp chí chuyên ngành quốc tế

[CT1] Nguyen Minh Hai, Ha Minh Ngoc, Nguyen Thien Binh and Quan Thanh Tho,"Toward an Approach on Probability Distribution for Polymorphic Malware Analysis", in *GSTF Journal on Computing (JOC)*, Volume 5 (1), pp. 61-68, 2016, ISSN:2251 – 3043 (selected from *7th Annual International Conference on ICT: Big Data, Cloud and Security (ICT-BDCS 2016)*, Singapore - **Best Paper Award**).

[CT2] Nguyen Minh Hai, Le Nguyen Dung, Nguyen Xuan Mao and Quan Thanh Tho, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning", *Computers & Security Journal,* Volume 7, pp. 128-155, July 2017 ***(SCI-E)***.

[CT3] Pham Phuoc Hung, Md. Golam Rabiul Alam, Nguyen Minh Hai, Quan Thanh Tho, Eui-Nam Huh, "A Dynamic Scheduling Method for Collaborated Cloud with Thick Clients", *The International Arab Journal of Information Technology*, 16(4), 2019 (***SCI-E***).

Tạp chí chuyên ngành trong nước

[CT4] Nguyen Minh Hai, Quan Thanh Tho, A Statistical Approach for Packer Identification, In *Journal of Science and Technology*, Vietnam Academy of Science and Technology, vol. 54 (3A), Special issue of Intelligent System and its Applications, pp. 129-139, 2016 (selected papers from *Proceedings of International Symposium Intelligent Systems and Applications 2016 (ISA2016)*, Ho Chi Minh city, Vietnam).

Hội thảo chuyên ngành trong nước và quốc tế

[CT5] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan and Mizuhito Ogawa (2013), A Hybrid Aproach for Control Flow Graph Construction from Binary Code, In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC 2013), Postgrad Symposium*, Thailand.

[CT6] Nguyen Minh Hai, Mizuhito Ogawa, Quan Thanh Tho, "Obfuscation code localization based on CFG generation of malware", *The 8th International Symposium on Foundations & Practice of Security*, Springer, Clermont-Ferrand, France, 2015.

[CT7] Nguyen Minh Hai, Do Duy Phong, Quan Thanh Tho, Le Duc Anh, "Precise Packer Detection Using Model Checking", in *The 10th SOUTH EAST ASIAN TECHNICAL UNIVERSITY CONSORTIUM SYMPOSIUM (SEATUC2016)*, Tokyo, Japan, 2016.

[CT8] Nguyen Minh Hai, Quan Thanh Tho, "An Experimental Study on Identifying Obfuscation Techniques in Packer", *5th World Conference on Applied Sciences, Engineering & Technology (WCSET),* 02-04 June 2016, HCMUT, Vietnam, ISBN 978-81-930222-2-1.

[CT9] Nguyen Minh Hai, Quan Thanh Tho and Le Duc Anh, "Multi-Threaded On-the-fly Model Generation of Malware with Hash Compaction *18th International Conference on Formal Engineering Methods (ICFEM 20)*, 14-18 November 2016, TKP Conference Centre, Tokyo, Japan.

[CT10] Nguyen Minh Hai, Do Duy Phong, Quan Thanh Tho, "Formal Methods for Packer Detection", *9th National Conference on Fundamental and Applied IT Research (FAIR'9)*, 04-05 August 2016, Can Tho University (CTU), Vietnam (in Vietnamese).

[CT11] Nguyen Minh Hai, Quan Thanh Tho, "Applying Deep Learning for Malware Analysis", *10th National Conference on Fundamental and Applied IT Research (FAIR'10)*, 17-18 August 2017, The University of Da Nang - University of Education (UED), Vietnam (in Vietnamese).

[CT12] Nguyen Minh Hai, Quan Thanh Tho, "Packer Identification using Hidden Markov Model", *The 11th Multi-disciplinary International Workshop on Artificial Intelligence (MIWAI 2017)*, 2017, Gadong – Brunei

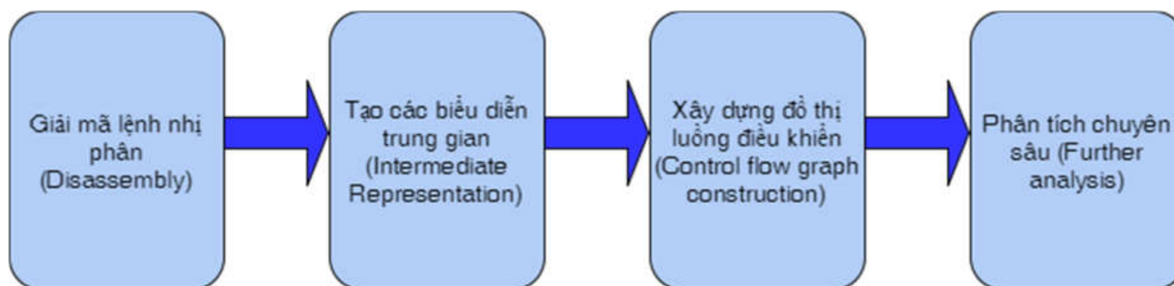
[CT13] Nguyen Minh Hai, Mizuhito Ogawa and Quan Thanh Tho, "Packer Identification Based on Metadata Signature", *The 7th Software Security, Protection, and Reverse Engineering Workshop (SSPREW-7)*, San Juan, Puerto Rico, USA, 2017.

[CT14] Nguyen Minh Hai, Le Nguyen Dung and Quan Thanh Tho, "Applying Symbolic Execution for Malware Analysis", *The 2nd Symposium on Information Security (SOIS 2017)*, 02-03 December 2017, University of Information Technology Ho Chi Minh City, Vietnam (in Vietnamese).

CHƯƠNG 1. GIỚI THIỆU

1.1 Giới thiệu

Trong ngành công nghiệp phần mềm, xu hướng kiểm tra phần mềm dựa trên mã nhị phân đang phát triển một cách mạnh mẽ. Hình 1-1 mô tả bốn bước chính trong quá trình phân tích mã nhị phân. Trong bước đầu tiên, chương trình tiến hành dịch ngược mã nhị phân. Bước thứ hai là quá trình xây dựng *biểu diễn trung gian* (intermediate representation) dựa trên kết quả bước 1. Trong bước 3, *đồ thị luồng điều khiển* (control flow graph) được xây dựng dựa trên kết quả của bước 2. Trong đó, các đỉnh đại diện cho các lệnh và các cạnh đại diện cho bước nhảy trong luồng điều khiển. Dựa trên đồ thị luồng điều khiển được xây dựng, các chương trình phân tích sẽ kiểm tra tính độc hại của chương trình hoặc các tính năng chuyên sâu khác trong bước 4. Trong Hình 1-1, quá trình xây dựng đồ thị luồng điều khiển đóng một vai trò thiết yếu. Tuy nhiên, vấn đề xây dựng đồ thị luồng điều khiển ở cấp độ mã nhị phân vẫn còn là một nhiệm vụ đầy thách thức do những trở ngại đã phân tích ở trên và đặc biệt là vấn đề lệnh nhảy không trực tiếp.



Hình 1-1 Tổng quan các bước trong phân tích chương trình

1.2 Phát biểu vấn đề

Như đã phân tích trong phần trên, luận án này nêu lên bài toán cần phải giải quyết là phát triển một phương pháp kết hợp giữa phân tích tĩnh và kiểm tra động trong phân tích mã nhị phân, mà cụ thể là bài toán xây dựng đồ thị luồng điều khiển.

1.3 Câu hỏi nghiên cứu

Để thực hiện nghiên cứu này, luận án cần phải xem xét giải quyết và trả lời các câu hỏi sau:

[RQ1]. Đã có một khung thức hay một công cụ tổng quát cho việc kết hợp giữa phương pháp phân tích
tĩnh và kiểm tra động để xây dựng đồ thị luồng điều khiển trong mã nhị phân hay chưa?

[RQ2]. Làm sao để rút ngắn thời gian thực thi của chương trình trong bài toán xây dựng đồ thị luồng điều khiển ?

[RQ3]. Làm sao để khai thác tri thức dựa trên đồ thị luồng điều khiển của chương trình được sinh ra từ mã nhị phân ?

1.4 Mục tiêu nghiên cứu

Mục tiêu của luận án là giải quyết và trả lời ba câu hỏi nghiên cứu mà luận án đã đề ra. Trong đó, các mục tiêu cụ thể là:

[OB1]. Đưa ra một khung thức tổng quát cho việc kết hợp hai kỹ thuật phân tích tĩnh và kiểm tra động nhằm xây dựng đồ thị luồng điều khiển trong phân tích mã nhị phân.

[OB2]. Đưa ra một phương pháp tăng tốc thời gian thực thi của chương trình phân tích với cơ chế *đa luồng* (multithreading) và loại bỏ những *đường đi dư thừa* (redundant path) đã được xử lý trong chương trình.

[OB3]. Khám phá tri thức dựa trên đồ thị luồng điều khiển của chương trình được sinh ra từ mã nhị phân. Nghiên cứu này tập trung vào chủ đề phân tích mã nhị phân của mã độc. Luận án thực hiện đồng thời hai việc, (i) nhận diện *chương trình đóng gói* (packer) được sử dụng trong mã độc bằng cách áp dụng kỹ thuật *kiểm định Chi bình phương* (Chi square test) và *mô hình Markov ẩn* (Hidden Markov Model); (ii) nhận diện mã độc sử dụng *phương pháp học sâu* (deep learning).

1.5 Những đóng góp chính của nghiên cứu

Các đóng góp chính của luận án được tóm tắt như sau:

i. Luận án đề xuất một khung thức tổng quát cho bài toán xây dựng đồ thị luồng điều khiển từ mã nhị phân của chương trình. Các công bố liên quan đến đóng góp này là [CT5] và [CT6].

ii. Luận án đưa ra giải pháp để tăng tốc quá trình thực thi của chương trình. Đó là áp dụng giải thuật song song hóa với tính toán đa luồng để tăng tốc độ xử lý các nút. Công bố có liên quan đến đóng góp này là [CT3] và [CT9].

iii. Luận án đề xuất cách khai thác tri thức dựa trên đồ thị luồng điều khiển của mã nhị phân. Luận án tập trung vào vấn đề nhận diện chương trình đóng gói trên mã độc với hai hướng

tiếp cận: (i) sử dụng kiểm định Chi bình phương; (ii) sử dụng mô hình Markov ẩn. Các công bố có liên quan đến đóng góp này là [CT1], [CT4], [CT7], [CT8], [CT10], [CT12] và [CT13].

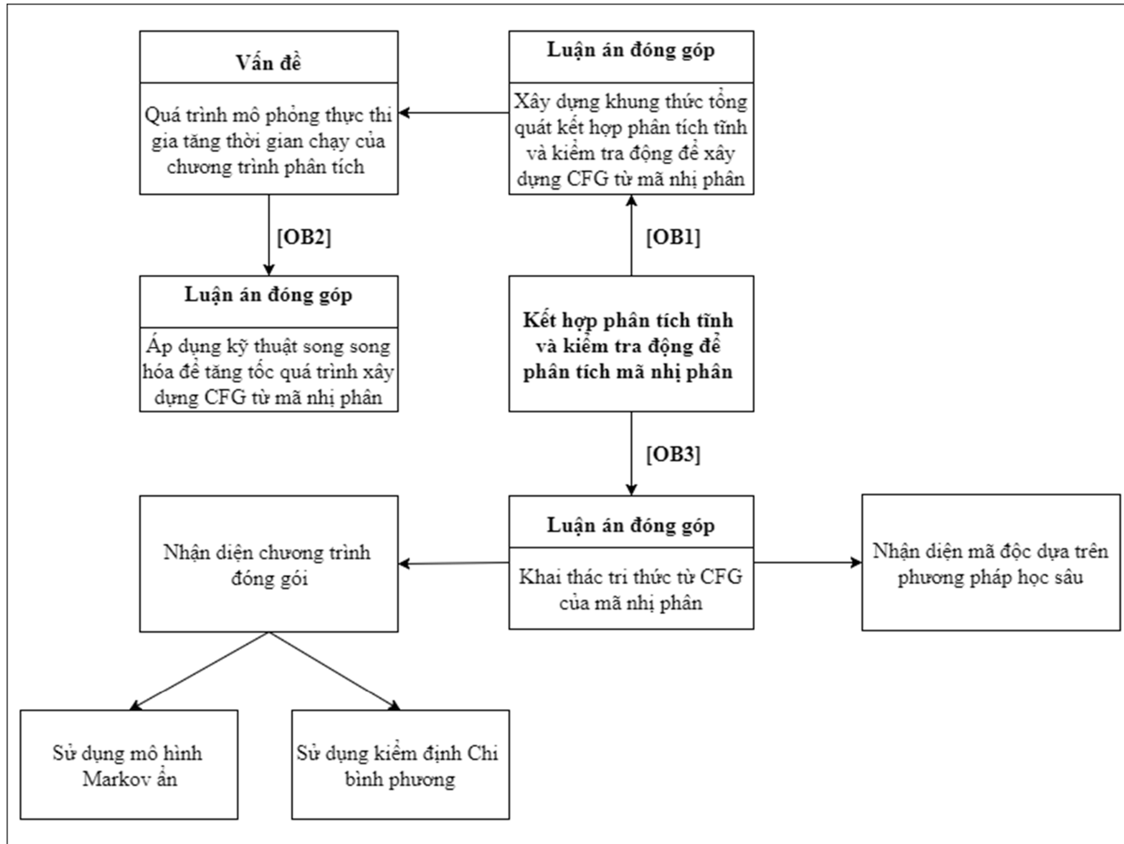
iv. Luận án này đã đề xuất cách nhận diện mã độc dựa trên phương pháp học sâu. Các công bố có liên quan đến đóng góp này là [CT2] và [CT11].

v. Cuối cùng, luận án cũng đã xây dựng một công cụ hoàn chỉnh có tên là BE-PUM cho việc xây dựng đồ thị luồng điều khiển từ mã nhị phân. Công bố liên quan đến công cụ BE-PUM là [CT6] và [CT14].

1.6 Bố cục của luận án

Trong Chương 1 này, chúng tôi đã trình bày về bối cảnh nghiên cứu cũng như đặt vấn đề nghiên cứu. Cấu trúc tổng quan các phần của luận án được trình bày trong Hình 1-2. Chương 2 trình bày tổng quan về các kiến thức nền tảng, định hướng cho nghiên cứu của luận án. Chương 3 trình bày về khung thức tổng quát cho bài toán xây dựng đồ thị luồng điều khiển từ mã nhị phân một cách tự động. Chương 4 trình bày về đóng góp thứ hai của luận án, đó là phương pháp tăng tốc quá trình thực thi của chương trình với giải thuật tính toán đa luồng kết hợp với bảng băm.

Chương 5 trình bày về các phương pháp nhận diện chương trình đóng gói sử dụng kiểm định Chi bình phương. Chương 6 trình bày giải thuật áp dụng mô hình Markov ẩn vào việc phát hiện và phân loại chương trình đóng gói. Trong chương 7, chúng tôi đưa ra một phương pháp nhận diện mã độc sử dụng phương pháp học sâu. Chương 8 trình bày về công cụ BE-PUM. Đây là công cụ hiện thực hóa tất cả các lý thuyết đã được trình bày trong luận án. Các kết luận của luận án và các định hướng nghiên cứu tiếp theo trong tương lai sẽ được chúng tôi trình bày trong Chương 9.



Luận án đóng góp

Xây dựng một công cụ hoàn chỉnh, BE-PUM cho việc xây dựng CFG từ mã nhị phân

Hình 1-2 Cấu trúc luận án

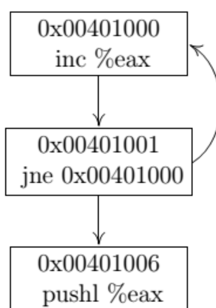
CHƯƠNG 2. KIẾN THỨC NỀN TẢNG

2.1 Đồ thị luồng điều khiển

Đồ thị luồng điều khiển (Control Flow Graph - CFG) là một đồ thị có hướng và dùng để biểu diễn chương trình. Trong đó, đỉnh của đồ thị bao gồm địa chỉ của câu lệnh và câu lệnh hợp ngữ tại địa chỉ đó. Cạnh của đồ thị là thể hiện luồng thực thi của chương trình. Hình 2-2 mô tả ví dụ về đồ thị luồng điều khiển tương ứng với đoạn mã trong Hình 2-1.

```
00401000  inc %eax
00401001  jne 0x00401001
00401006  pushl %eax
```

Hình 2-1 Ví dụ minh họa đồ thị luồng điều khiển



Hình 2-2 Đồ thị luồng điều khiển tương ứng với Hình 2.1

2.2 Kỹ thuật kiểm thử thực thi ký hiệu động

Kỹ thuật *kiểm thử thực thi ký hiệu động* (concolic testing) là một kỹ thuật kiểm chứng phần mềm lai ghép kết hợp hai phương pháp *thực thi cụ thể* (concrete execution) và thực thi ký hiệu. Kỹ thuật này được sử dụng trong một số công cụ kiểm thử phần mềm như PathCrawler, jCUTE và SAGE. So sánh với phương pháp *kiểm thử hộp trắng* (whitebox testing) truyền thống, kỹ thuật kiểm thử thực thi ký hiệu động có ưu điểm là cho phép giảm số đường thực thi cần phải kiểm tra.

2.3 Chương trình đóng gói

Phần mềm đóng gói là một chương trình chuyển đổi mã nhị phân của chương trình gốc thành một chương trình thực thi khác. Chương trình thực thi mới này vẫn gìn giữ những tính năng nguyên bản nhưng có nội dung hoàn toàn khác với chương trình gốc khi được lưu trữ.

Chính vì điều này đã làm cho kỹ thuật quét chữ ký không thể liên kết giữa hai phiên bản này. Hơn 80% mã độc sử dụng rất nhiều loại phần mềm đóng gói khác nhau.

2.4 Kiểm định Chi bình phương

Phương pháp *kiểm định Chi bình phương* (Chi-square test) là một trong những phương pháp tiêu chuẩn để phân loại dựa trên các thuộc tính. Chúng tôi sử dụng phương pháp kiểm định chi bình phương trong bài toán phân loại. Trong đó, chúng tôi xác định *bậc tự do* (degree of freedom) là 1, giá trị *mất mát* (loss) là 0,05 (thường được dùng làm tiêu chuẩn) và giá trị hệ số tương quan tương ứng $\varepsilon = 3{,}84$.

CHƯƠNG 3. KHUNG THỨC TỔNG QUÁT XÂY DỰNG ĐỒ THỊ LUỒNG ĐIỀU KHIỂN

3.1 Giới thiệu

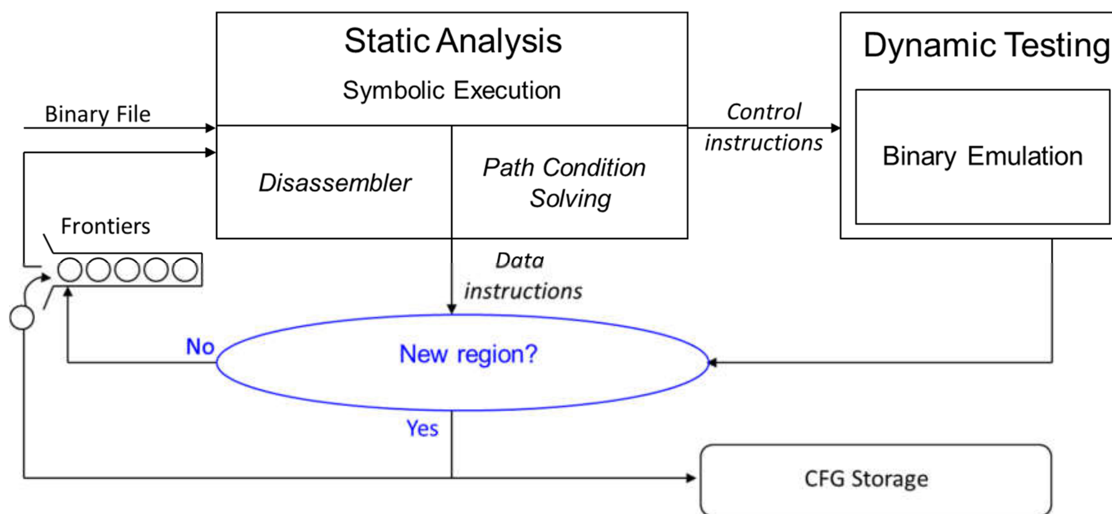
Chúng tôi đề xuất phương pháp kiểm thử thực thi ký hiệu động, kết hợp kỹ thuật *phân tích tĩnh* (static analysis) và *kiểm tra động* (dynamic testing) để xây dựng đồ thị luồng điều khiển từ mã nhị phân. Ý tưởng chính của phương pháp này là áp dụng phân tích tĩnh để xây dựng đồ thị luồng điều khiển nội bộ thủ tục cho đến khi gặp lệnh nhảy gián tiếp hay các lời gọi hàm. Khi đó, kỹ thuật kiểm tra động được áp dụng bằng cách sinh ra các *dữ liệu thử nghiệm* (test-case) để xác định điểm đến chính xác của câu lệnh nhảy. Chúng tôi áp dụng kỹ thuật thực thi kí hiệu để tạo ra dữ liệu thử nghiệm thích hợp. Phương pháp này cung cấp cho một đồ thị luồng điều khiển thực tế chính xác hơn (ngay cả với trường hợp lệnh nhảy động) so với phương pháp *suy diễn trừu tượng* (abstract interpretation) dựa trên phân tích tĩnh.

Hình 3-1 mô tả khung thức tổng quát của phương pháp. Trong đó, vai trò của các thành phần này như sau.

- Khối Static Analysis đảm nhận quá trình phân tích tĩnh với kỹ thuật thực thi ký hiệu. Thành phần Disassembler dịch ngược *mã thực thi* (opcode) thành câu lệnh hợp ngữ. Thành phần Path Conditon Solving giải các điều kiện đường đi và sinh ra dữ liệu thử nghiệm thích hợp.
- Khối Dynamic Testing đảm nhận quá trình kiểm tra động. Đây là một thành phần quan trọng trong tổng thể kiến trúc của khung thức nhằm giả lập và mô phỏng hoạt động các thành phần của hệ thống với khối Binary Emulation.
- Khối CFG storage lưu trữ đồ thị luồng điều khiển 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.
- Thành phần Frontier lưu trữ thông tin về đường đi trong quá trình phân tích.

Trong khung thức này, chương trình được chia thành các *khu vực* (area). Mỗi khu vực bao gồm một khối các câu *lệnh hợp ngữ* (assembly code) được *dịch ngược* (disassemble) từ mã thực thi thông qua khối Disassembler. Trong giai đoạn phân tích tĩnh, chúng tôi áp dụng kỹ thuật thực thi kí hiệu để xây dựng đường thực thi trong một khu vực và tạo ra các đồ thị luồng điều khiển phụ tương ứng. Quá trình thực thi ký hiệu này được thực hiện cho đến khi gặp phải một câu lệnh nhảy động. Khi gặp phải một bước nhảy động, chúng tôi giải *điều kiện đường đi* (path

condition) tương ứng với đường thực thi trong khu vực này thông qua khối Path Condition Solving. Sau đó, các trường hợp kiểm thử được sinh ra để bao phủ tất cả các đường thực thi. Trong lúc đó, đồ thị luồng điều khiển của khu vực sẽ được cập nhật. Sau đó, giai đoạn phân tích động sẽ được thực hiện. Trong giai đoạn này, chương trình sẽ tiến hành thực thi chương trình sử dụng các trường hợp kiểm thử được sinh ra ở bước trên thông qua khối Binary Emulation.



Hình 3-1 Kiến trúc tổng quát của khung thức

3.2 Các nghiên cứu liên quan

Có rất nhiều các công cụ xây dựng mô hình cho phân tích mã nhị phân. Để giải quyết vấn đề lệnh nhảy động, các công cụ này có thể đi theo hướng, phân tích tĩnh hay kiểm tra động. Các công cụ CodeSurfer/x86, McVeto, và JakStab sử dụng kỹ thuật phân tích tĩnh. Trong khi đó OSMOSE, BIRD, Renovo, Syman, Codiasm và SAGE sử dụng phân tích động. Một cách tổng quát, kiểm tra động hiệu quả hơn phân tích tĩnh trong vấn đề phân tích mã độc. Khung thức của chúng tôi sử dụng phương pháp kết hợp cả 2 cách trên.

CHƯƠNG 4. ÁP DỤNG KỸ THUẬT SONG SONG HÓA KẾT HỢP VỚI BẢNG BĂM VÀ GIẢI THUẬT DI TRUYỀN ĐỂ GIẢM THỜI GIAN THỰC THI CỦA CHƯƠNG TRÌNH

4.1 Giới thiệu

Mục tiêu của chương này sẽ tập trung vào những công việc sau đây. Chúng tôi đề xuất sử dụng giải thuật đa luồng để tăng tốc độ xử lý các nút trong BE-PUM. Giải thuật của chúng tôi yêu cầu rất nhỏ về vấn đề đồng bộ và giao tiếp giữa các luồng xử lý. Bên cạnh đó, chúng tôi kết hợp giải thuật xử lý song song với kỹ thuật bảng băm để giảm lượng bộ nhớ sử dụng cho việc lưu trữ thông tin về các nút đã được xử lý. Hơn thế nữa, chúng tôi sử dụng phương pháp phát hiện trùng lặp và giải thuật di truyền để ngăn chặn sự phân tích trùng lặp giữa các luồng xử lý.

4.2 Những nghiên cứu liên quan

Có rất nhiều công cụ được sử dụng trong phân tích mã nhị phân, ví dụ như CodeSurfer/x86, McVeto, JakStab, BIRD và BINCOA. Tuy nhiên, theo quan sát của chúng tôi, không có công cụ nào thực thi mô hình xử lý đa luồng.

4.3 Nén sử dụng bảng băm

Nén sử dụng bảng băm (hash compactation) là một phương pháp được giới thiệu bởi Holzmann với mục đích tối thiểu lượng bộ nhớ sử dụng để lưu trữ các nút. Ý tưởng chính của phương pháp này là sử dụng một hàm hash *H* để ánh xạ từ vectơ *V* sang một chuỗi bit có độ dài cố định *B*. Độ dài của *B* có thể là 32 hoặc 64 bit. *V* là một cấu trúc dữ liệu không nhập nhằng dùng để biểu diễn trạng thái của nút. Mỗi một nút sau khi được phân tích xong sẽ được lưu vào danh sách. Tuy nhiên, thay vì lưu toàn bộ trạng thái của nút, chúng ta chỉ cần lưu giá trị băm của nút đó và làm giảm kích thước bộ nhớ cần lưu trữ.

4.4 Mô tả giải thuật xử lý đa luồng

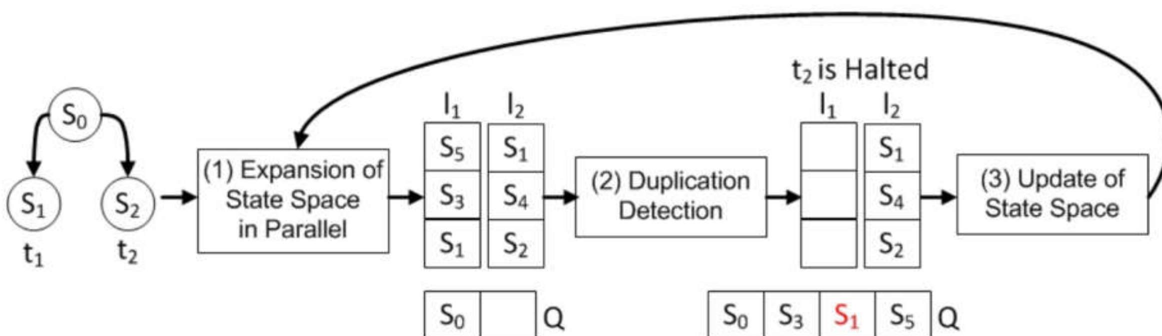
4.4.1 Tổng quan giải thuật

Trạng thái S của một nút được mô tả, $S = (k, asm, H_1(Env), H_2(Env))$

- k là địa chỉ câu lệnh,
- asm là câu lệnh hợp ngữ.

- *Env* là môi trường bao gồm giá trị của *thanh ghi* (registers), *cờ* (flags), và *trạng thái bộ nhớ* (memory status) trong đó bao gồm trạng thái của *ngăn xếp* (stack).
- $H_1(Env)$ và $H_2(Env)$ là hàm *Hash* sẽ ánh xạ biến môi trường *Env*. Biến môi trường có độ dài cố định 32 bit. Chúng tôi sử dụng hàm Băm như đã mô tả ở phần 4.3.

Tiếp theo, *tập* $T = (t_1, t_2, \dots, t_N)$ là tập hợp các luồng xử lý được sử dụng để phân tích. Tập $L = (l_1, l_2, \dots, l_N)$ là tập hợp các danh sách cục bộ dùng để lưu trạng thái của các nút đã xử lý của các luồng xử lý tương ứng.



Hình 4-1 Tổng quan giải thuật xử lý đa luồng

Cấu trúc dữ liệu chính trong mục tiêu song song hóa của chúng tôi bao gồm:

- Một danh sách toàn cục Q để lưu trữ tất cả trạng thái của tất cả các luồng xử lý.
- Những danh sách cục bộ l_i sẽ lưu trữ tất cả những trạng thái đã tìm kiếm của từng luồng.

Như mô tả ở Hình 4-1, mỗi luồng sẽ thực hiện 3 bước. Ở bước *mở rộng không gian trạng thái* (extension of state space), luồng t_i sẽ tiến hành tìm kiếm các trạng thái từ các nút theo *chiều sâu* (depth-first) và cập nhật những trạng thái đó vào danh sách cục bộ l_i. Khi danh sách cục bộ đã đầy, ở bước *phát hiện sự trùng lập* (duplicate detection), luồng sẽ tiến hành kiểm tra vấn đề trùng lặp. Nếu phát hiện những trạng thái trong danh sách l_i đã tồn tại trong Q, luồng t_i sẽ dừng lại. Ngược lại, nếu không có trạng thái nào trong l_i bị trùng với các trạng thái trong Q, luồng t_i sẽ tiến hành bước *cập nhật không gian trạng thái* (update of state space). Khi đó luồng t_i sẽ cập nhật tất cả những trạng thái trong danh sách l_i vào Q.

Trong mỗi bước thực hiện, tất cả các luồng đều hoạt động độc lập và không cần đến sự đồng bộ hay giao tiếp với nhau. Đây là tính năng chính của giải thuật mà chúng tôi đưa ra.

4.4.2 Song song hóa kết hợp giải thuật di truyền

Giải thuật di truyền được phát triển bởi John Holland nhằm giải quyết bài toán tối ưu hóa. Tổng quan các bước trong giải thuật di truyền của chúng tôi được mô tả trong Bảng 4-1. Giải thuật di truyền của chúng tôi sử dụng phương pháp khởi tạo các cá thể theo hướng ngẫu nhiên. Các cá thể được khởi tạo với các tham số đầu bao gồm thông tin số lượng luồng thực thi còn trống tại thời điểm gọi và danh sách các đỉnh tương ứng với đường đi đang cần xử lý. Trong quá trình lựa chọn cá thể, chúng tôi sử dụng phương pháp *Roulette Wheel Selection*. Với phương pháp này, một quần thể có *n* cá thể sẽ được chia nhỏ vào một hình tròn có *n* phần nhỏ. Cá thể nào có giá trị thích nghi tốt hơn thì sẽ có một phần lớn hơn trong hình tròn và khả năng cá thể đó được chọn cũng sẽ cao hơn. Trong giải thuật di truyền của chúng tôi, quá trình lai ghép được sử dụng phương pháp *Single Point Crossover*. Phương pháp gây đột biến mà chúng tôi sử dụng trong giải thuật di truyền của mình là *Scramble Mutation*. Phương pháp này sẽ chọn một đoạn ngẫu nhiên trong danh sách *listTask* của cá thể và đảo lộn thứ tự các phần tử trong đoạn đó một cách ngẫu nhiên. Điều kiện dừng được thiết lập bằng một ngưỡng thời gian để giới hạn thời gian chạy của giải thuật 60 giây.

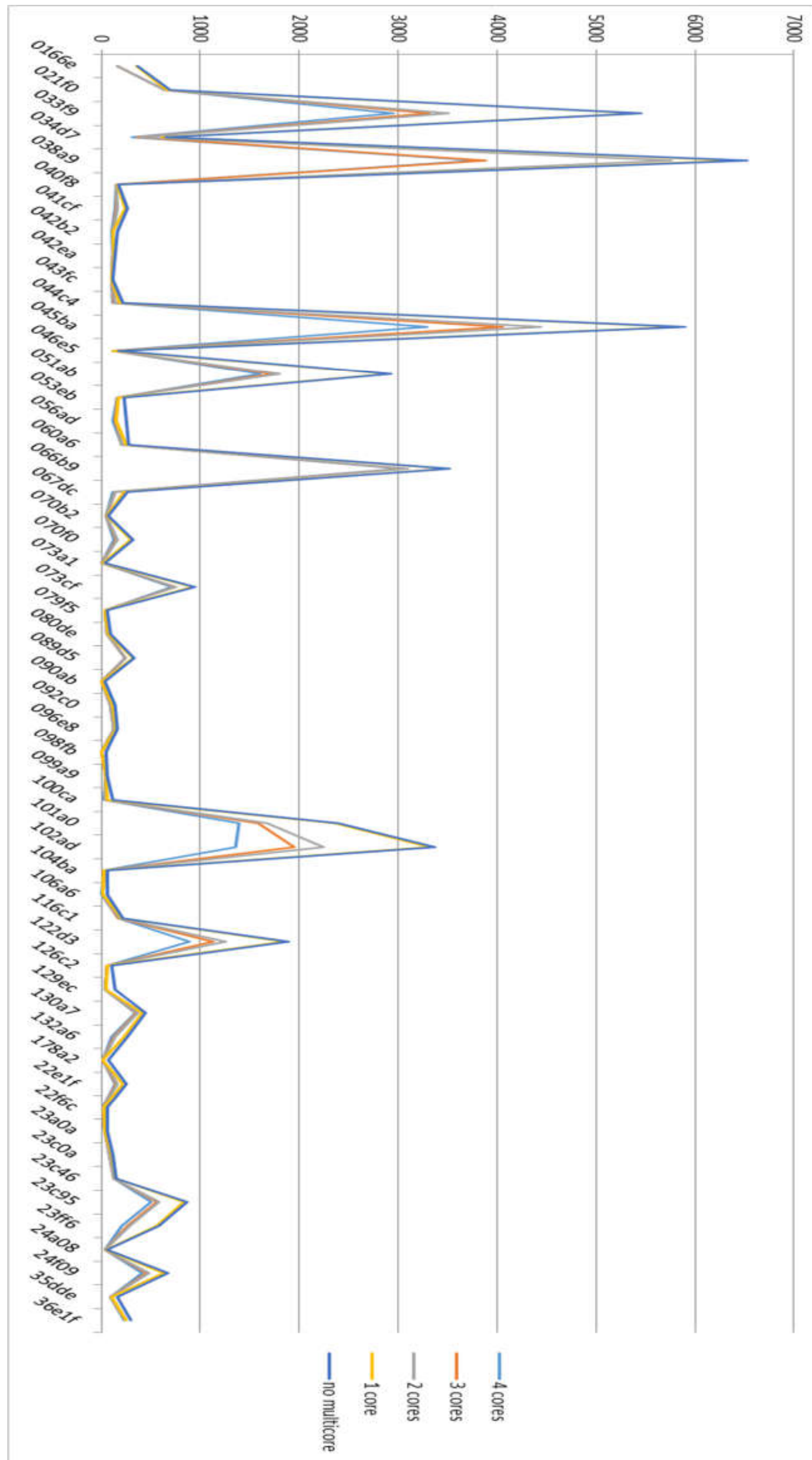
4.5 Thí nghiệm về hiệu năng giải thuật song song hóa trong phân tích mã độc

Chúng tôi đã thực hiện so sánh hiệu năng trên hơn 21920 mã độc thật được thu thập từ VirusTotal [62]. Kích thước của những tập tin này vào khoảng vài trăm kilobyte. Thí nghiệm được thực hiện trên một máy tính 4 nhân, chạy hệ điều hành Windows XP với tốc độ 2.9GHz và bộ nhớ 8GB. Chúng tôi đã thực hiện giải thuật của chúng tôi trên các mã độc với số luồng từ 1 đến 4 và không thực hiện đa luồng (giải thuật gốc ban đầu).

Hình 4-2 trình bày những kết quả thực hiện của chúng tôi. Mã độc được định danh bởi giá trị băm được mô tả theo chiều ngang. Chiều dọc cho thấy thời gian thực thi khi sử dụng giải thuật của chúng tôi với số luồng khác nhau với mỗi mã độc. Hướng tiếp cận của chúng tôi là tìm ra những kết quả tốt hơn về thời gian xử lý với số luồng thực thi tăng lên.

Bảng 4-1 Tổng quan các bước trong giải thuật di truyền

Bước	Công việc
1	Khởi tạo ngẫu nhiên quần thể cho giải thuật di truyền
2	Thực hiện tính toán giá trị hàm mục tiêu và tìm ra cá thể tốt nhất
3	Kiểm tra điều kiện dừng • Nếu thỏa điều kiện dừng thì ngừng giải thuật và trả ra cá thể tốt nhất • Nếu không thì tiếp tục thực hiện từ bước 4 đến bước 7
4	Lựa chọn các cá thể cha mẹ bằng phương pháp Roulette Wheel Selection để tiến hành lai tạo
5	Thực hiện quá trình lai ghép với phương pháp Single Point Crossover để tạo ra các cá thể con
6	Thực hiện quá trình đột biến với phương pháp Scramble Mutation
7	Thay thế các cá thể thích nghi kém trong quần thể
8	Lặp lại từ bước 3 đến bước 7 cho đến khi thoả điều kiện dừng và trả ra cá thể tốt nhất



Hình 4-2 Kết quả thí nghiệm tính toán đa luồng

CHƯƠNG 5. NHẬN DIỆN CHƯƠNG TRÌNH ĐÓNG GÓI SỬ DỤNG KỸ THUẬT CHI BÌNH PHƯƠNG

5.1 Giới thiệu

Hơn 80% các mã độc đã sử dụng chương trình đóng gói với rất nhiều kỹ thuật làm rối để tránh việc bị phát hiện. Các chương trình đóng gói thông dụng nhất có thể kể đến UPX, PECOMPACT và ASPACK. Trong chương này, chúng tôi đề xuất hướng tiếp cận mới để nhận diện các chương trình đóng gói.

- Chúng tôi đề xuất phương pháp nhận diện *chữ ký siêu dữ liệu* (metadata signature) của chương trình đóng gói, thay thế cho phương pháp *nhận diện chữ ký* truyền thống. Đầu tiên, chúng tôi mở rộng công cụ BE-PUM cho phép phát hiện các kỹ thuật làm rối mã. Tiếp theo, kỹ thuật làm rối được sử dụng trong những chương trình đóng gói được phân loại dựa theo khảo sát và được thống kê tự động. Cuối cùng, chúng tôi sử dụng phương pháp kiểm định Chi bình phương để xác định chương trình đóng gói dựa trên chữ ký siêu dữ liệu.
- Chúng tôi thực hiện thí nghiệm để tính toán sự chính xác của hướng tiếp cận của chúng tôi trên 5374 mã độc từ VX Heaven và 7440 mã độc từ Virusshare, trong đó 608 mẫu sinh ra kết quả khác biệt với những công cụ phát hiện khác như PeiD, CFF Explorer và VirusTotal.
- Do bản chất mô phỏng quá trình thực thi, chúng tôi thiết kế BE-PUM như một công cụ giải nén tổng quát. BE-PUM có thể đồng thời giải nén và phát hiện các chương trình đóng gói tự xây dựng dựa trên sự xuất hiện của kỹ thuật nén/giải nén và kỹ thuật hai APIs đặc biệt.

5.2 Phương pháp thực hiện

5.2.1 Mô tả giải thuật

Đầu tiên, chúng tôi tiến hành nhận diện các kỹ thuật làm rối. Chúng tôi thiết lập danh sách các tiêu chuẩn cho mỗi kỹ thuật làm rối. Do đó BE-PUM sẽ tự động nhận dạng các kỹ thuật này trong quá trình dịch ngược.

Chữ ký siêu dữ liệu (metadata signature) của chương trình đóng gói là vectơ tần số xuất hiện của kỹ thuật làm rối trong chương trình đóng gói này.

Chúng tôi chọn 14 kỹ thuật làm rối như được liệt kê trong bảng 5.1 và 5.2. Tập huấn luyện T_r và tập kiểm tra T_e với $T_r \cap T_e = \emptyset$ được chọn từ những mã nhị phân sử dụng những chương

trình đóng gói đã được nhận diện. Trong quá trình xử lý theo kỹ thuật *on-the-fly* để tạo mô hình, BE-PUM nhận diện và thống kê những kỹ thuật làm rối trong những chương trình đóng gói trên.

Xét tập kỹ thuật làm rối $T = \{T_1, T_2, \ldots, T_n\}$, với tập những packer mục tiêu $M = \{M_1, M_2, \ldots, M_m\}$, tập vectơ trung bình $E^i = \{E_1, E_2, \ldots, E_n\}$ và những giá trị ngưỡng $\overline{\lambda_i}$ (được trình bày trong phần 2.6) cho mỗi packer M_i. *O(B)* là vectơ tần số của kỹ thuật làm rối trong quá trình xây dựng đồ thị luồng điều khiển hiện tại của B.

Hàm *On_the_fly_Model_Generation*(*B*) mở rộng đồ thị luồng điều khiển của *B* theo kỹ thuật thực thi ký hiệu động. Hàm *Model_Generation_Stop(B)* sẽ quyết định việc dừng quá trình sinh đồ thị đồ thị luồng điều khiển (nguyên nhân có thể là do gặp câu lệnh không hỗ trợ, gặp API không hỗ trợ hoặc hết thời gian phân tích).

Hàm *Calculate_Membership_Degree(O(B), E^i)* tính toán bậc của thành phần của *O(B)* dựa trên trung bình siêu dữ liệu E^i của chương trình đóng gói M_i bởi kiểm định chi bình phương. Giá trị ngưỡng λ_i của mỗi packer M_i là tập trung bình của bậc trong tập kiểm tra T_e như mô tả ở phần 2.6.

Giải thuật nhận diện chương trình đóng gói được tóm tắt như trong Giải thuật 5-1.

5.3 Thí nghiệm

Tất cả những kết quả được thực hiện trên nền tảng Windows XP với công cụ VMware worktation phiên bản 10.0. Máy chủ dùng hệ điều hành Windows 8 Pro với AMD Athlon II X4 635, 2.9GHz và 8GB bộ nhớ.

Chúng tôi tập trung vào 12 packer, cụ thể là ASPACK v2, CEXE v1.0b, KKRUNCHY v0.23a4, MPRESS v2.19, FSG v2.0, NPACK v1.0, PECOMPACT v2.0, PETITE v2.1, TELOC v0.99, UPX v3.0, YODA v1.3 và UPACK v037-0.39. Kết quả được tổng hợp từ 15031 tập tin được phân chia làm hai kiểu dữ liệu, tập tin bình thường và mã độc.

5.3.1 Nhận diện chương trình đóng gói trong phân tích mã độc

Chúng tôi thu thập 12814 mẫu mã độc thực tế. Để so sánh, mỗi tập tin được quét bởi ba phần mềm nhận diện chương trình đóng gói thông dụng, PeiD, CFF Explorer, và VirusTotal. PeiD là chương trình phổ biến trong việc nhận diện các tập tin bị đóng gói. VirusTotal là một công cụ quét mã độc miễn phí online, kết hợp kết quả nhiều nguồn chống mã độc khác, như Kaspersky, Microsoft, và AVG… CFF Explorer cũng là một công cụ phổ biến trong nhận diện chương trình đóng gói.

Với 12814 mẫu, BE-PUM đã cho kết quả như sau:

- 499 trường hợp với 296 từ VX Heaven và 203 từ Virusshare bị quá thời gian.
- 5923 mẫu với 1419 mẫu từ VX Heaven và 4504 mẫu từ Virusshare được phát hiện không được đóng gói, giống với kết quả của PeiD, CFF Explorer và VirusTotal.
- 6392 mẫu được phát hiện đóng gói
  Chi tiết của 6392 mẫu được đóng gói sẽ được trình bày dưới đây.
- 5459 mẫu với 3270 mẫu từ VX Heaven và 2189 mẫu từ Virusshare được phát hiện đóng gói bởi một trong 12 packer, giống kết quả với PeiD, CFF Explorer và VirusTotal.

Giải thuật 5-1 Giải thuật sử dụng Chi bình phương

Input: Chương trình nhị phân B.

Output: M_i nếu packer được sử dụng để đóng gói là M_i;

NONE nếu không tìm thấy

Algorithm:

$O(B) = (O_1, O_2, \dots, O_n) \coloneqq (0, 0, \dots, 0);$

while *TRUE* **do**

  On_the_fly_Model_Generation(B);

  if *Found_New_Obfuscation_Technique()* = T_j **then**

    $O(B) \coloneqq \left(O_1, \dots, O_j + 1, \dots, O_n\right);$

    foreach i := 1 to m **do**

      $\overline{\lambda_T} = Calculate_Membership_Degree(O(B), E^i)$

      if $\overline{\lambda_T} \geq \overline{\lambda_i}$ **then**

        Return M_i;

      end

    end

  end

  if *Modern_Generation_Stop(B)* **then**

    return NONE;

  end

end

- 402 mẫu với 137 mẫu từ VX Heaven và 265 mẫu từ Virusshare được phát hiện đóng gói bằng một trong 12 packer, các kết quả không được thống nhất giữa PEiD, CFF Explorer, VirusTotal và BE-PUM.
- 325 mẫu với 216 mẫu từ VX Heaven và 109 mẫu từ Virusshare được phân loại được đóng gói với những chương trình đóng gói BE-PUM chưa hỗ trợ.
- 206 mẫu với 36 mẫu từ VX Heaven và 170 mẫu từ Virusshare được phát hiện bởi BE-PUM là những gói tự xây dựng, trong khi đó PeiD, CFF Explorer, VirusTotal không phát hiện được.

CHƯƠNG 6. NHẬN DIỆN CHƯƠNG TRÌNH ĐÓNG GÓI SỬ DỤNG MÔ HÌNH MARKOV ẨN

6.1 Giới thiệu

Một cách tổng quát, chúng tôi biểu diễn một chương trình đóng gói P bằng một chuỗi các kỹ thuật làm rối $O = \{o_1, o_2, \dots, o_n\}$ với o_i đại diện cho kỹ thuật làm rối. Bảng 6-2 mô tả chuỗi các kỹ thuật làm rối trong các chương trình đóng gói.

Bảng 6-1 Đánh số thứ tự các kỹ thuật làm rối

0	Overlapping function	1	Overlapping block	2	Code chunking
3	Overwriting	4	Packing/unpacking	5	Indirect jump
6	SEH	7	2API	8	Obfuscated constant
9	Checksumming	10	Timing check	11	Anti-debugging
12	Stolen bytes	13	Hardware breakpoint		

Bảng 6-2 Chuỗi các kỹ thuật làm rối trong chương trình đóng gói

ASPack v2.12	8_3_3_3_3_7_3_3_5_12_3_4_5_12_3_4_4_8_4_4_4_4_8_8_8_4_4_4_4_4_4_4_4_4_8_8_8_8_4_8_8_4_8_4_4_3_4_3_5_3_3_3_7_3_3_5_3_7
CEXE	5_4_4_5_4_4_4_4_4_6_8_4_4_6_8_4_4_4_8_5_5_7_4_4_4_4_4_4_4_8_3_3_4_3_3
FSG v2.0	3_3_5_3_5_4_3_5_3_5_3_4_3_5_12
KKRUNCHY	4_5_8_8_4_4_4_5_4_4_4_5_5_4_4_5_5_7_4_8_4_4_4_4_4_2_4_8_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4
MPRESS	4_4_4_8_8_8_8_8_4_4_4_4_4_4_2_4_8_3_3_7_5_5_3_4_3_4_3_3_3_3_3_4_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3
nPack v1.0	3_12_5_7_12_4_4_12_3_4_3_3_4_8_3_12

Từ đó, chúng tôi đề xuất ý tưởng sử dụng mô hình Markov ẩn để nhận diện chương trình đóng gói. Chúng tôi cũng đã tiến hành thí nghiệm trên tập 2126 mẫu mã độc từ Virusshare để chứng minh tính hiệu quả của phương pháp đề xuất.

6.2 Phương pháp thực hiện

6.2.1 Mô hình Markov ẩn

Mô hình Markov ẩn $\lambda = \{A, B, \pi\}$ bao gồm các thành phần sau.

- Tập hợp S tất cả các trạng thái: $S = \{s_1, s_2, \dots, s_n\}$ với n là số trạng thái.
- Tập hợp O tất cả các ký hiệu quan sát được $O = \{o_1, o_2, \dots, o_m\}$ với m là số ký hiệu quan sát
- Ma trận xác suất chuyển trạng thái A

$$A = \{a_{ij} | \; a_{ij} = P(q_{t+1} = s_j | q_t = s_i)\}$$

với a_{ij} là xác suất chuyển đổi từ trạng thái s_i sang trạng thái s_j.

- Ma trân xác suất quan sát B

$$B = \{b_i(k) | \; b_i(k) = P(o_k | q_t = s_i)\}$$

Với $b_i(k)$ là xác suất quan sát được ký hiệu o_k tại trạng thái s_i

- Một tập các xác suất khởi đầu, $\pi = \{\pi_i | \; \pi_i = P(q_1 = s_i)\}$

Trong HMM, có hai vấn đề chính là vấn đề huấn luyện và vấn đề nhận dạng.

- Bài toán nhận dạng: Cho trước HMM λ và chuỗi quan sát $O = \{o_1, o_2, \dots, o_t\}$, chúng tôi cần tính toán chuỗi trạng thái $s_1, s_2, \dots, s_t$ sinh ra HMM λ. Bài toán này thường được giải quyết bằng giải thuật Viterbi.
- Bài toán huấn luyện: Chúng tôi xác định các thông số trên mô hình Markov ẩn *A*, *B* và *π* dựa vào tập các chuỗi quan sát trong chương trình đóng gói. Quá trình xây dựng mô hình Markov ẩn được mô tả trong phần tiếp theo.

6.2.2 Xây dựng mô hình Markov ẩn

Trong hướng tiếp cận này, chúng tôi tạo một *trạng thái* (state) cho từng chương trình đóng gói. Để xây dựng mô hình Markov ẩn, chúng tôi tiến hành qua 3 bước.

Bước 1: Xác định xác suất chuyển ban đầu π_i

Mỗi trạng thái tương ứng với một chương trình đóng gói. Giả sử ban đầu có *n* loại chương trình, thì xác suất chuyển trạng thái ban đầu sẽ là bằng nhau $\pi_i = \frac{1}{n}$.

Bước 2: Tính ma trận xác suất quan sát B

Với mỗi chương trình đóng gói T_{s_i} tương ứng với trạng thái s_i, chúng tôi qui định $P = \{P_1, P_2, \dots, P_o\}$ là danh sách tập huấn luyện các mẫu được đóng gói bởi T_{s_i} và *o* là số lượng mẫu. Xác suất quan sát ký hiệu o_k tại trạng thái s_i là tần suất xuất hiện của kỹ thuật o_k trong T_{s_i}.

$$b_i(k) = \frac{\sum_{j=1}^{0} f(O_k, P_j)}{\sum_{l=1}^{m} \sum_{j=1}^{o} f(O_l, P_j)} \qquad (6.1)$$

Với $f(O_k, P_j)$ là tổng số số lần xuất hiện của kỹ thuật O_k trong P_j

Lưu ý là tổng xác suất quan sát các chuỗi trong một trạng thái luôn có giá trị là 1.

$$\sum_{k=1}^{m} b_i(k) = \sum_{k=1}^{m} \frac{\sum_{j=1}^{0} f(O_k, P_j)}{\sum_{l=1}^{m} \sum_{j=1}^{o} f(O_l, P_j)} = 1$$

Bước 3: Tính ma trận xác suất chuyển trang thái A

Xây dựng vector V_{s_i}cho từng trạng thái s_i

$$V_{s_i} = \left\{ \frac{\sum_{j=1}^{o} f(O_1, P_j)}{o}, \frac{\sum_{j=1}^{o} f(O_2, P_j)}{o}, \dots, \frac{\sum_{j=1}^{o} f(O_m, P_j)}{o} \right\} \quad (6.2)$$

Với hai trạng thái s_i và s_i, chúng ta sử dụng khoảng cách cosin để tính sự tương đồng giữa chúng.

$$\cos(s_i, s_j) = \cos\left(V_{s_i}, V_{s_j}\right) = \frac{\sum_{k=1}^{m} \frac{\sum_{i=1}^{o} f\left(O_k, P_l^{s_i}\right) \sum_{l=1}^{0} f\left(O_k, P_l^{s_j}\right)}{o \quad o}}{\sqrt{\sum_{k=1}^{m} \left(\frac{\sum_{l=1}^{o} f(O_k, P_l^{s_i}}{o}\right)^2} \sqrt{\sum_{k=1}^{m} \left(\frac{\sum_{l=1}^{o} f(O_k, P_l^{s_j}}{o}\right)^2}} \quad (6.3)$$

$$\cos(s_i, s_i) = \cos\left(V_{s_i}, V_{s_i}\right) = 1$$

Khi đó, xác suất a_{ij} chuyển từ trạng thái s_i đến trạng thái s_j được tính theo công thức sau.

$$a_{ij} = \frac{\cos(s_i, s_j)}{\sum_{k=1}^{n} \cos(s_i, s_k)} \quad (6.4)$$

Lưu ý rằng, $\sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} \frac{\cos(s_i, s_j)}{\sum_{k=1}^{n} \cos(s_i, s_k)} = 1$ với mọi s_i

6.3 Thí nghiệm

Chúng tôi tiến hành thí nghiệm trên hệ điều hành Windows XP sử dụng VMware workstation 10. Máy chủ dùng hệ điều hành Windows 8 Pro với AMD Athlon II X4 635, 2.9GHz và 8GB bộ nhớ.

Chúng tôi thu thập 2126 mã độc từ Virusshare. Để so sánh hiệu quả, từng tập tin sẽ được quét qua 3 phần mềm phát hiện chương trình đóng gói phổ biến là PEiD, CFF Explorer, và VirusTotal. Trong đó PEiD là một trong những chương trình phát hiện tốt nhất. VirusTotal là chương trình quét và phát hiện mã độc online thông qua việc kết hợp so sánh kết quả nhận diện mã độc từ nhiều nguồn như Kaspersky, Microsoft, và AVG. CFF Explorer được xem như là một công cụ khá phổ biến cho việc phân tích chương trình đóng gói. Có thể thấy, hướng tiếp cận sử dụng HMM cho kết quả tốt hơn so với phương pháp nhận diện chữ ký truyền thống khi sử dụng chương trình PEid, CFF Explorer, và VirusTotal.

Bảng 6-3 Kết quả thí nghiệm

	CFF Explorer	PEid	VirusTotal	HMM
ASPACK v2	183	183	183	219
FSG v2.0	384	384	384	410
NPACK v1.0	77	77	77	115
PECOM- PACT v2.0	92	92	92	112
PETITE v2.1	115	115	115	177
TELOCK v0.99	150	150	150	168
UPX v3.94	360	360	360	430
YODA v1.3	150	150	150	150
UPACK v0.37-.39	310	310	310	345

CHƯƠNG 7. NHẬN DIỆN MÃ ĐỘC SỬ DỤNG PHƯƠNG PHÁP HỌC SÂU

7.1 Giới thiệu

Trong chương này, chúng tôi đề xuất một cách tiếp cận mới để xử lý các mã độc đa hình. Trước hết, chúng tôi đề xuất tạo ra CFG từ mã nhị phân. Sau đó, chúng tôi chuyển đổi đồ thị này sang một ma trận kề tương ứng. Do biểu diễn các ma trận kề như hình ảnh, các biến thể của cùng một mẫu mã độc sau đó được mô tả như các đối tượng tương tự nhau trong các hình ảnh được xây dựng. Sau đó, chúng tôi sử dụng một hệ thống *mạng nơ-ron* (neural network) như *mạng nơ-ron tích chập* (convolution neuron network - CNN), để xác định các đối tượng này một cách hiệu quả. Chúng tôi đánh giá cách tiếp cận của chúng tôi với các bộ dữ liệu thu thập được từ một số kho mã độc trực tuyến.

7.2 Phương pháp thực hiện

Chúng tôi tiến hành quá trình nhận dạng qua 3 bước. Đầu tiên, chúng tôi xây dựng đồ thị luồng điều khiển từ mã nhị phân của chương trình. Trong bước 2, chúng tôi tiến hành chuyển đổi từ đồ thị luồng điều khiển thành ma trận kề và tiến hành xây dựng ảnh. Cuối cùng, ảnh sẽ được học trong mô hình học sâu để tiến hành phân loại trong bước 3.

7.2.1 Xây dựng đồ thị luồng điều khiển từ mã nhị phân chương trình

Hiện nay, cộng đồng nghiên cứu có xu hướng phát hiện phần mềm độc hại dựa trên các hành vi bằng cách tạo ra một mô hình cho phép nắm bắt được luồng thực thi của một chương trình. Mô hình phổ biến nhất được sử dụng trong hướng nghiên cứu này là *đồ thị luồng điều khiển* (CFG). Trong CFG, mỗi đỉnh tương ứng với câu lệnh của tập tin gốc. Sự chuyển tiếp giữa các đỉnh thể hiện luồng thực thi của chương trình khi các lệnh tương ứng được thực hiện. CFG cho phép chúng tôi phân tích các hành vi của chương trình và phát hiện các hoạt động đáng ngờ do phần mềm độc hại gây ra.

7.2.2 Chuyển CFG thành ảnh

Phần này trình bày cách chuyển từ CFG sang ảnh để được tiếp tục xử lý bằng hệ thống học sâu. Thứ nhất, chúng tôi chuyển đổi CFG thành một ma trận kề. Như đã thảo luận, mỗi đỉnh của CFG được coi là một trạng thái. Mỗi trạng thái được mã hoá dưới dạng một chuỗi các bit, bao gồm 3 đoạn, tương ứng với mô tả các giá trị của thanh ghi, cờ và bộ nhớ. Vì kích thước bộ

nhớ quá lớn nên chúng tôi sử dụng hàm băm MD5 để giảm việc sử dụng tài nguyên. Đối với mỗi trạng thái truy cập, thay vì lưu trữ toàn bộ bộ nhớ, chỉ có giá trị băm của bộ nhớ này được lưu trữ. Mô tả này cho phép chúng tôi trình bày sự giống nhau giữa hai trạng thái của chương trình.

Mỗi đỉnh i được thể hiện là dòng i và cột i trong ma trận kết quả. Do đó nếu có đường đi từ đỉnh i qua đỉnh j trên CFG, thành phần *(i, j)* của ma trận được tính theo công thức |*state*(*j*) - *state*(*i*)|. Đặc biệt, mỗi state là bộ gồm <*Register*, *Flag*, *Memory*>, kết quả | *state*(*j*) - *stat* (*i*)| thể hiện cho bộ màu <*red*, *green*, *blue*>.

$$red = |Register(j) - Register(i)|$$

$$green = |Flag(j) - Flag(i)|$$

$$blue = |Memory\,(j) - Memory(i)|$$

Thanh ghi có 8 loại khác nhau, bao gồm *Eax, Ebx, Ecx, Edx, Esi, Edi, Esp, Ebp* và do đó.

$$|Register(j) - Register(i)| = \frac{(|Eax(j) - Eax(i)| + |Ebx(j) - Ebx(i)| + |Ecx(j) - Ecx(i)| + |Edx(j) - Edx(i)| + |Esi(j) - Esi(i)| + |Edi(j) - Edi(i)| + |Ebp(j) - Ebp(i)| + |Esp(j) - Esp(i)|)}{8} \quad (7.1)$$

Có 9 loại cờ bao gồm *CF, PF, AF, ZF, SF, TF, IF, DF, OF,* do đó

$$|Flag(j) - Flag(i)| = \frac{(|CF(j) - CF(i)| + |PF(j) - PF(i)| + |AF(j) - AF(i)| + |ZF(j) - ZF(i)| + |SF(j) - SF(i)| + |TF(j) - TF(i)| + |IF(j) - IF(i)| + |DF(j) - DF(i)| + |OF(j) - OF(i)|)}{9} \quad (7.2)$$

Bộ nhớ *Memory* = {*(Loc(k), Val(k))* | *0≤k≤n*}, với *n* là kích thước bộ nhớ, *loc(k)* là vị trí của bộ nhớ K và *val(k)* là giá trị bộ nhớ. Phép tính giữa 2 trạng thái bộ nhớ được tính như sau.

$$|Memory(j) - Memory(i)| = \frac{\sum_{k=1}^{n} |Val_j(\mathrm{k}) - Val_i(\mathrm{k})| \text{ if } Loc_j(\mathrm{k}) == Loc_i(\mathrm{k})}{n} \quad (7.3)$$

Đặc biệt, khi tính phép trừ |*X* - *Y*| trong các công thức này, vì X và Y có thể là giá trị cụ thể hoặc giá trị tượng trưng, giá trị của |*X* - *Y*| có thể được tính như sau.

$$|X - Y| = \begin{cases} |X - Y| \text{ Nếu kết quả là giá trị cụ thể } (a) \\ X \text{ nếu } X \text{ là giá trị cụ thể và } Y \text{ là giá trị tượng trưng } (b) \\ 255 \text{ trường hợp ngược lại } (c) \end{cases} \quad (7.4)$$

Các bộ <red, green, blue> được nối với một giá trị hex.

7.2.3 Sử dụng phương pháp học sâu để phát hiện mã độc

CNN đạt được kết quả tốt trong vấn đề phân loại. Tuy nhiên, trong cách nhận dạng mã độc từ hình ảnh, chúng tôi áp dụng kỹ thuật của hệ thống Yolo. Ý tưởng chính của công trình này là hệ thống học sâu được phát triển để nghiên cứu hai tính năng cùng một lúc: (i) hộp bao quanh, tức là một hình chữ nhật xung quanh đối tượng và (ii) đối tượng nhận diện trong hộp xung quanh.

7.3 Thí nghiệm

Chúng tôi thực hiện các thí nghiệm nhận dạng mã độc trên CPU Xeon v3 với bộ nhớ 96 Gb. Các mẫu mã độc được thu thập từ nhiều nguồn bao gồm VXHeaven, Virusshare.com và MALICIA [111]. Số lượng mẫu từ VXHeaven, Virusshare và MALICIA tương ứng là 20713, 31609 và 11368. Ngoài ra, chúng tôi cũng thu thập 13752 chương trình bình thường để phục vụ cho các thí nghiệm.

7.3.1 Kết quả thí nghiệm

Bảng 7-1 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%
Simple-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%
Yolo-based CNN	Precision	99.16%	99.08%	98.24%	98.05%	97.37%	96.98%	96.32%	95.87%	95.12%
	Recall	96.27%	95.59%	94.73%	94.11%	92.69%	92.13%	90.89%	90.26%	90.07%
	F-measure	97.69%	97.3%	96.45%	96.04%	94.97%	94.49%	93.53%	92.98%	92.53%

Từ tập dữ liệu mã độc thu thập được, chúng tôi tạo ra nhiều tập huấn luyện cho CNN để thực hiện việc phát hiện mã độc. Để đánh giá, chúng tôi sử dụng kỹ thuật kiểm chứng chéo k-fold với $k = 10$. Chúng tôi so sánh phương pháp đề xuất của chúng tôi với kỹ thuật nổi tiếng Support Vector Machine (SVM) [112] và AIS [113]. Phương pháp AIS hiệu quả để phát hiện các biến thể khác nhau của cùng một phần mềm độc hại. Bảng 7-1 là trình bày kết quả quá trình thí nghiệm. Chúng tôi so sánh các thông số về *độ chính xác* (precision), *độ đo tính toàn vẹn* (recall) và *độ trung bình điều hòa* (F-measure) để đánh giá các phương pháp phân loại. Có thể thấy, AIS và phương pháp học sâu đạt được kết quả tốt hơn so với SVM vì hai phương pháp này có thể xử lý tốt phần mềm độc hại tự sửa đổi. Hơn nữa, khi cung cấp đủ số lượng tập huấn luyện, các phương pháp học sâu cho thấy kết quả với độ chính xác tốt nhất.

CHƯƠNG 8. CÔNG CỤ XÂY DỰNG ĐỒ THỊ LUỒNG ĐIỀU KHIỂN BE-PUM

Như đã trình bày ở các chương trên, công cụ BE-PUM mà chúng tôi xây dựng cung cấp các tính năng khác biệt và độc đáo so với các công cụ tương tự hiện có như sau.

BE-PUM thực hiện mô hình thực thi đa luồng được kết hợp với phương pháp nén sử dụng bảng băm, kỹ thuật phát hiện trùng lặp và giải thuật di truyền để giảm lượng tài nguyên sử dụng. Thông tin chi tiết về giải thuật được chúng tôi trình bày trong các đóng góp [CT3], [CT9] và trong Chương 4 của luận án này.

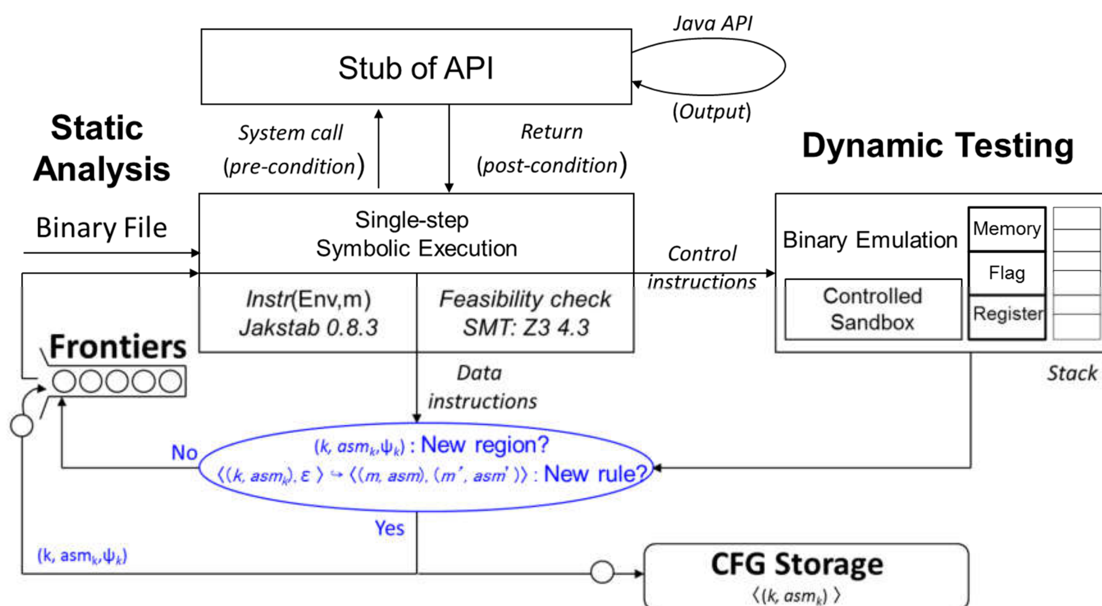
BE-PUM ứng dụng kỹ thuật nhận diện chương trình đóng gói sử dụng phương pháp kiểm định Chi bình phương và mô hình Markov ẩn. Chi tiết cũng như cơ sở lý thuyết về tính năng này được chúng tôi trình bày trong Chương 5, Chương 6 và trong các đóng góp [CT1], [CT4], [CT7], [CT8], [CT10], [CT12] và [CT13].

BE-PUM thực hiện việc nhận diện mã độc sử dụng phương pháp học sâu. Các chi tiết về phương pháp nhận diện này được chúng tôi trình bày trong Chương 7 và trong các đóng góp [CT2], [CT11].

Chúng tôi đã phát triển công cụ BE-PUM dưới hai dạng ứng dụng là *ứng dụng chạy cục bộ trên máy tính* (desktop application) và *ứng dụng trực tuyến* (web application).

8.1 Kiến trúc của BE-PUM

BE-PUM được hiện thực bằng ngôn ngữ Java, sử dụng bộ *dịch ngược* (disassembler) được xây dựng trong JakStab và bộ SMT Solver Z3. Hình 8-1 trình bày kiến trúc của BE-PUM.



Hình 8-1 Kiến trúc của BE-PUM

Kiến trúc của hệ thống BE-PUM bao gồm các thành phần chính, Frontiers, Static Analysis, Dynamic Testing và CFG Storage. Trong đó, vai trò của các thành phần này được mô tả như sau.

- Thành phần Frontiers chứa các vị trí các khu vực cần phân tích trong BE-PUM. Lưu ý là khi bắt đầu phân tích chương trình, Frontier sẽ chứa vị trí của *điểm vào gốc* (original entry point) được lưu trong phần *tiêu đề* (header) của chương trình thực thi.
- Thành phần CFG Storage lưu trữ các đỉnh và cạnh của đồ thị luồng điều khiển sau khi được tính toán chính xác. CFG storage sẽ được sử dụng để xây dựng mô hình quá trình thực thi của tập tin được phân tích.
- Thành phần Static Anlaysis nhận *đầu vào* (input) là *chương trình nhị phân* (binary file) và sử dụng chương trình disassembler cung cấp bởi *Jackstab 0.8.3* 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 (Data instructions), 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 vị trí tiếp theo chưa được xử lý và đây là một khu vực mới (New Region), CFG được cập nhật trong CFG Storage và vị trí này được cập nhật trong thành phần Frontiers. Nếu vị trí tiếp theo đã được xử lý và đây không phải là vùng mới, CFG

được cập nhật trong CFG Storage và một vị trí mới được lấy ra trong thành phần Fontiers để phân tích. Trong trường hợp câu lệnh hợp ngữ là câu lệnh điều khiển (control instructions), cụ thể là các câu lệnh gọi hàm, câu lệnh trả về từ hàm, câu lệnh nhảy và câu lệnh nhảy có điều kiện, điều kiện đường đi sẽ được tính toán thông qua thành phần *thực thi ký hiệu* (symbolic execution) và được kiểm tra tính khả thi với thành phần *SMT Z3.4.3*. Giá trị kiểm thử được sinh ra và được thực thi trong thành phần Dynamic Testing.

- Thành phần Dynamic Testing là thành phần quan trọng trong tổng thể kiến trúc của BE-PUM. Thành phần này đóng vai trò mô phỏng *quá trình thực thi* (binary emulation), bao gồm các lệnh hợp ngữ và Windows API trong chương trình x86. 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 các *cờ* (Flag) được sử dụng trong hệ thống (AF, CF, DF, IF, OF, PF, SF, TF, và ZF), tập các *thanh ghi* (Register) (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* (Memory), các thành phần stack được lưu trữ như một phần bộ nhớ. Do BE-PUM hỗ trợ phân tích mã độc và có thể tác động trực tiếp đến môi trường thực, một môi trường hộp cát có kiểm soát (controlled sandbox) sẽ được xây dựng cho quá trình mô phỏng này.

CHƯƠNG 9. KẾT LUẬN

9.1 Kết luận

Luận án đã hoàn thành việc trả lời 3 câu hỏi nghiên cứu và 3 mục tiêu mà luận án đã đề ra thông qua các đóng góp sau:

i. Luận án đề xuất một khung thức tổng quát cho bài toán xây dựng đồ thị luồng điều khiển từ mã nhị phân của chương trình.

ii. Luận án đưa ra giải pháp để tăng tốc quá trình thực thi của chương trình.

iii. Luận án đã đề xuất cách tiếp cận khai thác tri thức dựa trên đồ thị luồng điều khiển của mã nhị phân.

   a. Luận án tập trung vào vấn đề nhận diện chương trình đóng gói trên mã độc với hai hướng tiếp cận: (i) sử dụng kiểm định Chi bình phương; (ii) sử dụng mô hình Markov ẩn.

   b. Luận án này đã đưa ra giải pháp cách nhận diện mã độc dựa trên phương pháp học sâu.

iv. Cuối cùng, luận án cũng đã xây dựng một công cụ hoàn chỉnh có tên là BE-PUM cho việc xây dựng đồ thị luồng điều khiển từ mã nhị phân.

9.2 Hướng nghiên cứu trong tương lai

Đối với công việc tương lai, nghiên cứu này còn nhiều vấn đề để xem xét. Cụ thể các công việc trong tương lai của luận án là.

- Mở rộng hỗ trợ tập lệnh x86 và Windows API.
- Mở rộng quá trình nhận diện các chương trình đóng gói.
- Nhận diện mã độc với các phương pháp học máy khác.
- Xây dựng mô hình weighted pushdown trên BE-PUM nhằm mục tiêu giảm số trạng thái cần xử lý trong bài toán kiểm tra mô hình.
- Do mã độc sử dụng vòng lặp với bộ đếm rất lớn, có thể lên đến hàng tỷ, thách thức kế tiếp là giải quyết vấn đề *sinh ra điều kiện bất biến cho vòng lặp* (infer loop invariant) khi phân tích tĩnh.