ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA CÔNG NGHỆ PHẦN MỀM

PHẠM HỮU DANH

KHÓA LUẬN TỐT NGHIỆP

PHÁT HIỆN MÃ ĐỘC BẰNG PHƯƠNG PHÁP MÁY HỌC TRÊN HỆ ĐIỀU HÀNH WINDOWS

MALWARE DETECTION USING MACHINE LEARNING IN WINDOWS OPERATING SYSTEMS

KỸ SƯ NGÀNH KỸ THUẬT PHẦN MỀM

TP. HỒ CHÍ MINH, 2018

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA CÔNG NGHỆ PHẦN MỀM

PHẠM HỮU DANH - 14520134

KHÓA LUẬN TỐT NGHIỆP

PHÁT HIỆN MÃ ĐỘC BẰNG PHƯƠNG PHÁP MÁY HỌC TRÊN HỆ ĐIỀU HÀNH WINDOWS

MALWARE DETECTION USING MACHINE LEARNING IN WINDOWS OPERATING SYSTEMS

KỸ SƯ NGÀNH KỸ THUẬT PHẦN MỀM

GIẢNG VIÊN HƯỚNG DẪN
PGS. TS. VŨ THANH NGUYÊN

TP. HỒ CHÍ MINH, 2018

Lời Cảm Ơn

Đầu tiên và trên hết, tôi xin gửi lời cảm ơn chân thành nhất đến giảng viên, PGS. TS. Vũ Thanh Nguyên. Tôi chân thành biết ơn và cảm ơn thầy vì những chỉ dẫn vô giá, những hướng dẫn giá trị và những động lực từ tinh thần nghiên cứu liên tục của thầy. Những kinh nghiệm và sự giảng dạy của thầy trong lĩnh vực khóa học máy tính nói chung, và công nghệ phần mềm nói riêng, tôi có được những kiến thức mới, đầy giá trị và niềm đam mê trong lĩnh vực máy học và nhận dạng mã độc. Thầy là nguồn cảm hứng của tôi qua những trở ngại trong thời gian làm khóa luận tốt nghiệp. Nếu không có sự giúp đỡ của thầy, tôi sẽ không thể hoàn thành khóa luận này.

Tôi cũng gửi lời cảm ơn đến thầy Huỳnh Nguyễn Khắc Huy để về sự hướng dẫn của thầy về học máy và bảo mật máy tính.

Tôi cũng biết ơn tất cả các giảng viên của khoa Kỹ thuật Phần mềm tại trường Đại học Công nghệ Thông tin. Nhờ sự cống hiến của họ, tôi có một nền tảng vững chắc về khoa học máy tính nói chung và kiến thức về công nghệ phần mềm nói riêng. Nhờ các thầy cô, tôi học được khả năng học hỏi và áp dụng các công nghệ mới một cách hiệu quả hơn.

Cuối cùng nhưng không kém phần quan trọng, tôi xin cảm ơn gia đình đã khuyến khích và hỗ trợ tài chính lúc tôi cần. Tôi mãi mãi biết ơn tình yêu vô điều kiện của họ.

Tóm Tắt Khoá Luận

Phát hiện mã độc tĩnh là một lớp cần thiết trong các hệ thống bảo mật, nó cố gắng phân loại các mẫu là độc hại hoặc lành tính trước khi thực thi chúng. Tuy nhiên, hầu hết các nghiên cứu liên quan gặp các vấn đề về khả năng mở rộng, ví dụ, các phương pháp sử dụng neural networks thường mất rất nhiều thời gian đào tạo [41] hoặc sử dụng các tập dữ liệu mất cân bằng [45, 54], làm cho các số liệu đánh giá bị gây hiểu nhầm trong thực tế.

Trong nghiên cứu này, chúng tôi tìm hiểu về cả hai phương pháp nhận diện mã độc trên hệ điều hành Windows: phát hiện mã độc tính (static malware detection) và phát hiện mã độc động (dynamic malware detection). Chúng tôi tiến hành các thí nghiệm để áp dụng các phương pháp học máy trong phát hiện mã độc tĩnh. Hơn nữa, chúng tôi đề xuất phương pháp phát hiện phần mềm độc hại tĩnh sử dụng Portable Executable analysis và thuật toán học máy Gradient Boosting Decision Tree. Chúng tôi tập trung vào việc giảm thời gian đào tạo bằng cách giảm số lượng tính năng một cách thích hợp. Các kết quả thử nghiệm cho thấy phương pháp đề xuất của chúng tôi có thể đạt được tỷ lệ phát hiện lên tới 99,394 % với tỷ lệ cảnh báo sai là 1%, và khi giới hạn tỷ lệ báo động giả dưới 0,1% thì tỷ lệ phát hiện là 97,572 %, dựa trên hơn 600.000 mẫu đào tạo và 200.000 mẫu kiểm thử từ bộ dữ liệu Endgame Malware BEnchmark for Research (EMBER) [3].

Mục lục

Danh sách hình vẽ

Danh sách bảng

Chương 1

Đặt Vấn Đề

Tóm tắt chương

Chương 1 đặt vấn đề cho đề tài, bao gồm sự nguy hiểm của phần mềm độc hại, tổng quan về tình hình nghiên cứu, ứng dụng các phương pháp máy học vào nhận diện mã độc, và đặc biệt là trình bày những hạn chế của các phương pháp đã có, cũng chính là lí do của việc chọn đề tài.

Malware thường được sử dụng như một thuật ngữ chung để chỉ bất kỳ phần mềm nào được thiết kế nhằm mục đích gây thiệt hại cho một máy tính, máy chủ hoặc mạng máy tính [37]. Một sự cố từ malware có thể gây thiệt hại hàng triệu đô la, cụ thể, mã độc zero-day ransomware WannaCry đã gây ra thảm họa trên toàn thế giới từ việc đánh sập hệ thống của các Bệnh viện Dịch vụ Y tế Quốc gia Vương quốc Anh, cho đến việc tắt toàn bộ hệ thống sản xuất của công ty Honda tại Nhật Bản [11]. Hơn nữa, phần mềm độc hại ngày càng trở nên tinh vi và đa dạng hơn mỗi ngày [51]. Theo đó, bài toán phát hiện phần mềm độc hại là một vấn đề quan trọng trong an ninh mạng, đặc biệt khi xã hội trở nên phụ thuộc nhiều vào các hệ thống máy tính.

Các sản phẩm nhận diện malware trước đó thường sử dụng các phương pháp dựa trên quy tắc (rule-based) hoặc dựa trên chữ ký (signature-based), yêu cầu các nhà phân tích xử lý các quy tắc thủ công (handcraft rules) có liên quan để phát hiện mã độc. Cách tiếp cận này có độ chính xác cao. Tuy nhiên, các quy tắc này thường là cụ thể và không thể nhận ra phần mềm độc hại mới, ngay cả khi nó sử dụng cùng chức năng. Vì lý do này, ý tưởng phát hiện mã độc dựa trên các thuật toán học máy được phát sinh. Thuật toán học máy học các mẫu cơ bản (pattern) từ một tập huấn luyện nhất định, bao gồm cả các mẫu độc hại và lành tính. Những pattern phân biệt các phần mềm độc hại từ phần mềm lành tính. Kể từ khi Schultz và các cộng sự chó thấy sự hiệu quả của việc ứng dụng các thuật toán máy học vào nhận diện mã độc [47], máy học đang trở thành một trong những công cụ phổ biến và có nhiều ảnh hưởng trong việc đảm bảo an toàn cho hệ thống.

Một số phương pháp sử dụng học máy đã mang lại các mô hình quá tích cực, thể hiện độ chính xác dự báo đáng kể, nhưng đã dẫn đến nhiều những kết quả dương tính giả (false positives). Các kết quả dương tính giả làm tiêu cực trải nghiệm của người dùng, ngăn không cho triển khai các phần mềm mới. Theo khảo sát của các IT administrator năm 2017 [15], 42% các công ty cho rằng người dùng của họ bị mất năng suất là do liên quan đến những kết quả dương tính giả, tạo ra một điểm nghẹt cho các quản trị viên CNTT trong môi trường doanh nghiệp. Các kỹ sư bảo mật cũng thông báo các báo động giả này thường gây rối khi họ đang làm việc để phát hiện và loại bỏ phần mềm độc hại. Một báo cáo được công bố vào năm 2015 cũng cho thấy rằng nhiều tổ chức ở Hoa Kỳ đã tiêu thụ một lượng tiền khổng lồ để xử lý các cảnh báo phần mềm độc hại không chính xác [29]. Do đó, ngay cả khi một giải pháp có tỷ lệ phát hiện cao nhất, nếu nó có một số lượng lớn các kết quả dương tính giả, nó được xem là vô dụng hơn so với một giải pháp với các kết quả dương tính giả thấp và một tỷ lệ phát hiện vừa phải.

Chúng tôi chọn đề tài "Phát hiện Mã độc nằng phương pháp Máy học trên hệ điều hành Windows"với mong muốn đóng góp một phương pháp mới để giải quyết vấn đề xác định phần mềm độc hại, đạt được tỷ lệ phát hiện cao và tỷ lệ dương giả thấp.

Chương 2

Giới Thiệu Tổng Quan

Tóm tắt chương

Chương 2 trình bày tổng quan về việc áp dụng Học máy trong Phát hiện mã độc tĩnh trên các hệ điều hành Windows; động lực và mục tiêu của nghiên cứu này; liệt kê các nghiên cứu có liên quan của tác giả ở Việt Nam hoặc trên toàn thế giới.

2.1 Tổng quan

Phần mềm độc hại là phần mềm được thiết kế để xâm nhập hoặc gây hại cho hệ thống máy tính mà không có sự đồng ý của chủ sở hữu. Các phân loại đơn giản phần mềm độc hại là nhận dạng tệp phần mềm độc hại và tệp sạch. Phát hiện phần mềm độc hại tĩnh đang phân loại các mẫu là độc hại hoặc lành tính mà không cần thực thi chúng. Ngược lại, phát hiện phần mềm độc hại động phát hiện phần mềm độc hại dựa trên hành vi thời gian chạy của phần mềm độc hại [4, 16]. Mặc dù phát hiện phần mềm độc hại tĩnh được biết đến là không thể giải quyết được toàn bộ bài toán nhận dạng mã độc [13], nó là một lớp quan trọng trong một bộ bảo mật bởi vì khi thành công, nó cho phép xác định các tập tin độc hại trước khi thực thi.

Bên cạnh đó, máy học là một công cụ hấp dẫn cho một khả năng phát hiện các mẫu mới và khả năng phát hiện heuristic dựa trên mẫu đã có (phần 3.3.1). Các mô hình học tập được giám sát (Supervised learning models) sẽ tự động tìm ra mối quan hệ phức tạp giữa các thuộc tính tệp trong dữ liệu huấn luyện và phân biệt giữa các mẫu độc hại và lành tính (phần 3.3.2). Hơn nữa, các mô hình học máy sẽ khái quát hóa với tập dữ liệu mới có các tính năng và nhãn theo một mẫu tương tự với dữ liệu đào tạo.

Ngoài ra, trong các hệ điều hành Windows, định dạng chung cho phần mềm độc hại là định dạng Portable Executable (PE) (phần 3.2), đó là định dạng tệp cho các tệp thi hành, mã đối tượng, tệp DLL, tệp phông chữ FON và các tệp khác được sử dụng trong cả phiên bản 32 bit và 64 bit. Định dạng PE đóng gói thông tin cần thiết cho trình tải hệ điều hành Windows để quản lý mã thực thi được bao bọc.

Do đó, nhiều phương pháp phát hiện phần mềm độc hại PE dựa vào các phương pháp học máy đã được đề xuất bắt đầu từ năm 1995 [27, 47, 28, 49, 45]. Năm 2001, Schultz và cộng sự biểu diễn các tệp PE theo các tính năng bao gồm imported functions, strings và byte sequences [47]. Các mô hình bao gồm các quy tắc được tạo ra từ RIPPER [14], Naive Bayes và một ensemble classifier. Phương pháp này đã được mở rộng bởi Kolter et al. vào năm 2006 [28] by cách sử dụng byte-level N-grams và những kỹ thuât từ xử lý ngôn ngữ tự nhiên, bao gồm trọng số TFIDF cho các strings. Vào năm 2009, Shafiq và các cộng sự đề xuất chỉ sử dụng bảy tính năng từ PE header, do thực tế là hầu hết các ứng dụng phần mềm độc hại trong nghiên cứu của họ thường trình bày các yếu tố đó [49]. Năm 2015, Saxe và Berlin sử dụng two-dimensional byte entropy histograms và một multi-layer neural network cho việc nhận dạng mã độc [45].

2.2 Động lực

Mặc dù nhiều mô hình đã thực hiện độ chính xác dự báo nổi bật, chúng được huấn luyện và xác thực trên tập dữ liệu mất cân bằng vì bài toán phát hiện phần mềm độc hại không nhận được sự chú ý tương tự như các bài toán khác trong cộng đồng nghiên cứu mở. Những hạn chế pháp lý là thách thức chính cho việc công bố tập dữ liệu điểm chuẩn để phát hiện phần mềm độc hại, cụ thể, Saxe và Berlin không thể công bố dữ liệu hoặc mã nguồn cho dự án của họ do tính chất pháp lý và độc quyền của nó [45]. Ngoài ra, không giống như hình ảnh, văn bản và lời nói có thể được gắn nhãn gần như ngay lập tức và trong nhiều trường hợp không cần chuyên gia, việc phân loại tệp độc hại hay lành tính là quá trình tốn thời gian cho cả những chuyên viên được đào tạo tốt. Công việc ghi nhãn có thể được tự động thông qua phần mềm anti-malware, nhưng kết quả có thể là độc quyền hoặc được bảo vệ pháp lý khác. Do đó, các mô hình này có thể không gây ấn tượng với dữ liệu cân bằng [10].

Hơn nữa, một phương pháp hiệu quả, có độ chính xác ấn tượng và tỷ lệ dương tính giả rất thấp, sẽ ngăn chặn những tổn thất lớn từ phần mềm độc hại, cũng như mang lại trải nghiệm tốt cho người dùng và tiết kiệm tài nguyên cho nhiều tổ chức.

2.3 Mục tiêu

Như đã đề cập trong phần 2.2, kết quả từ nhiều phương pháp được đề xuất có thể không ấn tượng khi đánh giá với dữ liệu cân bằng. Vì vậy mục tiêu chính của luận án này là **áp dụng các phương pháp học máy trong phát hiện phần mềm độc hại với tập dữ liệu cân bằng**. Ngoài ra, mục tiêu dự kiến là đề xuất một phương pháp với **tỷ lệ phát hiện cao** và **tỷ lệ báo động giả rất thấp**. Cuối cùng, chúng tôi hướng đến kiến thức về học máy và cách áp dụng chúng để tăng cường bảo mật thông tin người dùng trong việc giải quyết vấn đề phát hiện phần mềm độc hại.

Công việc chi tiết mà chúng tôi đã thực hiện trong luận án này bao gồm:

- Nghiên cứu và xây dựng nền tảng về bảo mật thông tin, đặc biệt là phần mềm độc hại.
- Nghiên cứu và hiểu các kiến thức cơ bản trong Học máy bao gồm phân loại, thuật toán cây quyết định, thuật toán Random Forest, thuật toán Support Vector Machine, Neural Networks và Gradient-Boosting Decision Tree.
- Nghiên cứu và hiểu rõ hơn về thuật toán cây Gradient-Boosting Decision Tree.
- Áp dụng thuật toán Gradient-Boosting Decision Tree vào bài toán phát hiện phần mềm độc hại. Tiến hành các thí nghiệm để đánh giá hiệu suất của mô hình và tối ưu hóa các tham số.

2.4 Những nghiên cứu liên quan

Phát hiện phần mềm độc hại đã phát triển trong vài năm qua, do mối đe dọa ngày càng tăng gây ra bởi phần mềm độc hại cho các doanh nghiệp lớn và các cơ quan chính phủ. Tại Việt Nam, PGS.TS. Vũ Thanh Nguyên và cộng sự đề xuất một phương pháp kết hợp của thuật toán lựa chọn tiêu cực và mạng miễn dịch nhân tạo để phát hiện virus [39], và phương pháp phát hiện phần mềm độc hại biến chất bằng phân tích Portable Executable (PE) cùng Longest Common Sequence (LCS) [54]. Nguyen Van Nhuong và các cộng sự đã đề xuất phương pháp semantic để phát hiện phần mềm độc hại biến chất một cách hiệu quả [52].

Các phương pháp phát hiện phần mềm độc hại có thể được phân loại trong phát hiện phần mềm độc hại tĩnh hoặc phát hiện phần mềm độc hại động [18]. Về lý thuyết, tính năng phát hiện động cung cấp chế độ xem trực tiếp về hành động của phần mềm độc hại, ít bị ảnh hưởng hơn bởi việc làm xáo trộn mã thực thi và làm cho việc sử dụng lại phần mềm độc hại trở nên khó khăn hơn [38]. Tuy nhiên, trên thực tế, phần mềm độc hại có thể xác định liệu nó có đang chạy trong một hộp cát, và ngăn chính nó thực hiện hành vi nguy hiểm [53]. Điều này dẫn đến một cuộc chạy đua vũ trang giữa các phương pháp phát hiện phần mềm độc hại động và phần mềm độc hại. Hơn nữa, trong nhiều trường hợp, phần mềm độc hại không hoạt động một cách chính xác do thiếu dependency hoặc cấu hình hệ thống không mong muốn. Những vấn đề này gây khó khăn cho việc thu thập tập dữ liệu về hành vi phần mềm độc hại.

Ngược lại, phân tích tĩnh không yêu cầu các thiết lập phức tạp, tốn kém để thu thập, nó có các tập dữ liệu khổng lồ có thể được tạo bằng cách tổng hợp các tệp nhị phân. Điều này làm cho phát hiện phần mềm độc hại tĩnh rất phổ biến với phương pháp học máy, có xu hướng hoạt động tốt hơn khi tăng kích thước dữ liệu [5]. Một số chương trình phát hiện phần mềm độc hại dựa trên học máy đã được giới thiệu từ ít nhất năm 1995 [27, 47, 28, 49, 45]. Nhiều tính năng tĩnh đã được đề xuất để trích xuất các tính năng từ tệp nhị phân: printable strings [47], import tables, opcodes, informational entropy, [55], byte n-grams [1], two dimensional byte entropy histograms [45]. Nhiều tính năng khác cũng đã được đề xuất trong Microsoft Malware Classification Challenge trên Kaggle [43], như là opcode images, various decompiled assembly features, và các thống kê tổng hợp. Ngoài ra, lấy cảm hứng từ sự thành công của các mô hình học tập sâu rộng đến đầu cuối trong xử lý hình ảnh và xử lý ngôn ngữ tự nhiên, Raff et al. giới thiệu phát hiện phần mềm độc hại từ raw byte sequences [41]. Có lẽ các mô hình state-of-the-art sẽ thay đổi thành các mô hình học sâu end-to-end trong những tháng hoặc năm tiếp theo, nhưng các hand-crafted features có thể tiếp tục có liên quan do định dạng có cấu trúc của phần mềm độc hại.

Không gian tính năng có thể trở nên lớn, trong trường hợp đó các phương pháp như locality-sensitive hashing [6], feature hashing [22] hoặc random projections [19]đã được áp dụng trong phát hiện phần mềm độc hại. Tuy nhiên, ngay cả sau khi áp dụng việc giảm kích thước, vẫn còn một số lượng lớn các tính năng, có thể gây ra các vấn đề về khả năng mở rộng cho một số thuật toán học máy. Neural networks đã nổi lên như một giải pháp thay thế có thể mở rộng do những tiến bộ đáng kể trong thuật toán đào tạo [2]. Nhiều phương pháp sử dụng neural networks đã được giới thiệu [45, 16, 27, 7] mặc dù không có cách so sánh kết quả rõ ràng vì các tập dữ liệu khác nhau.

Một lựa chọn phổ biến khác là ensemble of trees, có thể mở rộng hợp lý một cách hợp lý bằng cách lấy mẫu không gian tính năng trong mỗi lần lặp [9]. Cây quyết định có thể thích ứng tốt với các loại dữ liệu khác nhau và linh hoạt với nhiều tỷ lệ giá trị trong vectơ đặc trưng, vì vậy chúng mang lại hiệu suất tốt ngay cả khi không có tiêu chuẩn hóa dữ liệu.

Chương 3

Background

Tóm tắt chương

Firstly, chapter 3 presents the theoretical background used in the thesis including malware types, PE file format and academic background in machine learning. Then, machine learning frameworks and tools are introduced to reveal our implementation in practice.

3.1 Malware Types

Classifying is the excellent way to have a better understanding of malware. There are several of the most common types: adware, bots, rootkits, spyware, Trojan horses, viruses, and worms [17].

3.1.1 Virus

A virus is a form of malware that is capable of replicating itself and spreading to other computers by attaching themselves to various applications and executing when a user launches one of those. Viruses can also spread through documents, script files, and cross-site scripting vulnerabilities in web apps. Some famous examples of viruses over the years are the Concept virus, the Chernobyl virus (also known as CIH), the Anna Kournikova virus, Brain, and RavMonE.exe.

3.1.2 Worm

A worm is a standalone software that replicates without targeting and infecting specific files. Think of worms as small programs that replicate themselves and destroy the data. They usually aim the operating system files and work until the drive they are in becomes empty. Some examples include Melissa, Morris, Mydoom, Sasser, and Blaster.

3.1.3 Trojan

A Trojan is a malicious application that misrepresents itself to look useful to fool users into downloading and installing. A trojan can give remote access to an infected computer which is possible for the attacker to steal data, install more malware, monitor user activity, etc. Notable examples also include Trojan horses developed by United States government agencies like the FBI and NSA. Names like Magic Lantern, FinFisher, Netbus, Beast, Gh0st RAT, Clickbot.A, and Zeus have been the reason of horror. While an Android malware discovered in 2015, called Shedun, is one of the many that target mobile devices.

3.1.4 Ransomware

Ransomware, which is one of the most advanced and continuously on the rise these days, is a kind of malware that typically holds a computer system captive while requiring a ransom, e.g., blocking the access to the data of a victim or threating to either publish it. Worse yet, there is no guarantee that paying will get access to the data, or prevent it from publishing. Major ransomware like Reveton, CryptoLocker, CryptoWall, and more recently, the 2017 WannaCry attack, have caused no small amount of destruction [11].

3.1.5 Rootkit

A rootkit is a collection of software specially designed to allow malware that gathers information, into your system. These work in the background so that a user may not notice anything different. But in the environment, a rootkit will permit several types of malware to get into the system. The first rootkit to gain reputation on Windows was NTRootkit in 1999, but the most popular is the Sony BMG copy protection rootkit scandal that shook the company in the year 2005 [46].

3.1.6 Adware

Although the advertising-supported software (adware) is now much more common and known as adware in some circles, this word has been linked to malware for quite some time. While adware can refer to any application that is supported by advertising, malicious adware usually displays ads in the form of popups and windows that cannot be closed. It is the perhaps the most productive and least dangerous malware, designed with the specific purpose of promoting ads on your computer.

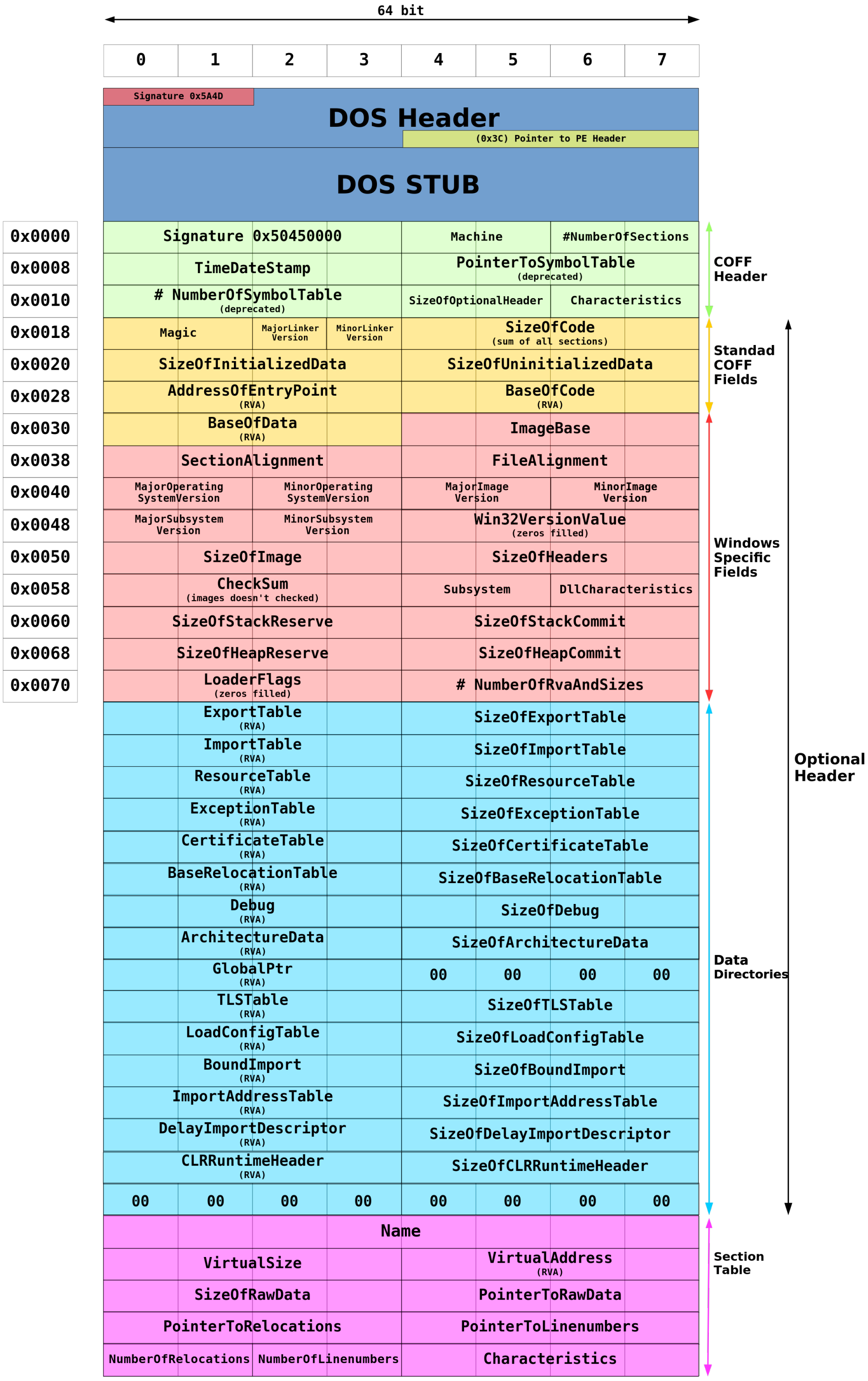
3.1.7 Bot

Bots are software programs created to perform specific operations automatically. While some bots are created for approximately innocent purposes, it is becoming increasingly common to see bots being used maliciously. Bots can be used in botnets (collections of machines to be controlled by third parties) to perform distributed denial of service attacks, send spam, and steal data.

3.2 PE File Format

The Portable Executable (PE) file format describes the predominant executable format for Microsoft Windows operating systems and includes executables, dynamically-linked libraries (DLLs), and FON font files. The format is currently supported on Intel, AMD, and variants of ARM instruction set architectures.

A Portable Executable file consists of some headers and sections that tell the dynamic linker how to map the file into memory. An executable image includes of several different regions, each of which requires different memory protection; so the start of each section must be aligned to a page frame. Typically, headers include the Common Object File Format (COFF) file header that contains essential information such as the type of machine for which the file is intended, the nature of the file (DLL, EXE, OBJ), the number of sections, the number of symbols, etc. The optional header identifies the linker version, the size of the code, the size of initialized and uninitialized data, the address of the entry point, etc. Data directories within the optional header provide pointers to the sections that follow it. These sections include tables for exports, imports, resources, exceptions, debug information, certificate information, and relocation tables. So, it provides a useful summary of the contents of an executable [50]. Finally, the section table outlines the name, offset and size of each section in the PE file.



Hình 3.1 Structure of a Portable Executable 32 bit [60]

PE sections contain code and initialized data that the Windows loader is to map into executable or readable/writeable memory pages, individually, as well as imports, exports, and resources defined by the file. Each section contains a header that specifies the size and address. An import address table instructs the loader which functions to import statically. A resources section may contain resources such as required for user interfaces: cursors, fonts, bitmaps, icons, menus, etc. A basic PE file would commonly contain a .text code section and one or more data sections (.data, .rdata or .bss). Relocation tables are typically stored in a .reloc section, used by the Windows loader to reassign a base address from the executable's preferred base. A .tls section contains special thread local storage (TLS) structure for storing thread-specific local variables. Section names are random from the perspective of the Windows loader, but specific names have been adopted by precedent and are overwhelmingly common.

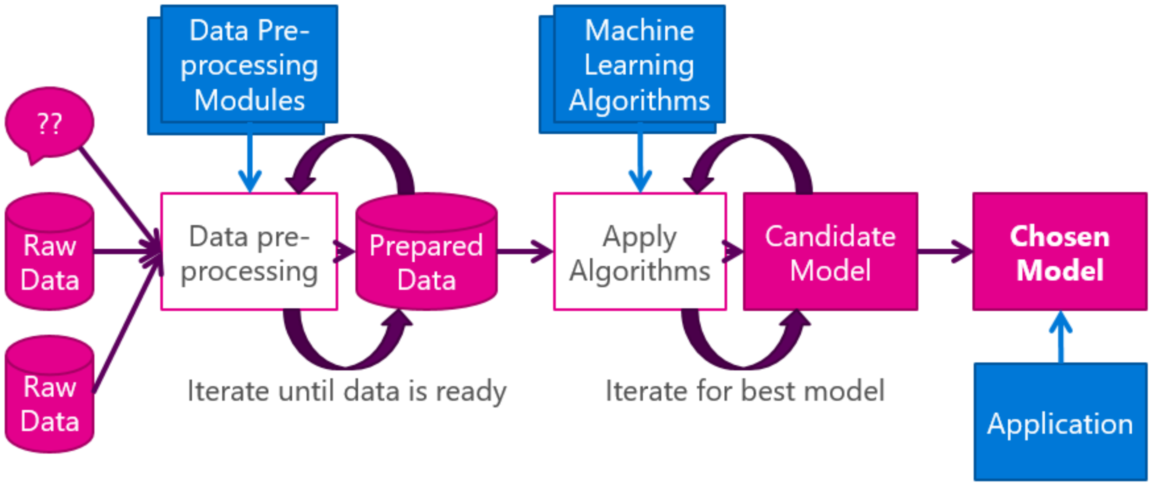
3.3 Machine Learning

3.3.1 Introduction

In recent years, most of the productive research and advancements have come from the sub-discipline of Artificial Intelligence named Machine Learning. The principle of Machine Learning is straightforward; Machine Learning is a method by which computers find patterns in data and makes those patterns available to applications. The application can then gain insights on new data based on similarity to the identified patterns [26].

It is worth going through the general workflow (Figure 3.2) of the machine learning process to have a deeper understanding:

- The primary goal of the process is to identify a model. The model is the main thing that applications can submit requests to gain insight on new data.



Hình 3.2 Machine Learning process [26]

- Before moving into Machine Learning experiments, we must identify the purpose and how to evaluate the result.

- The process starts with preparing data. Prepared data is one or more data sets that have been preprocessed (formatted, cleaned and sampled) in readiness to apply Machine Learning algorithms. Preparing the data means that the data is in the best shape to draw scientific conclusions.

- The next step is applying one or more Machine Learning algorithms intending to producing a Model, which is an iterative process and we may loop around testing various algorithms until we achieve a model that sufficiently reach the purpose.

Not all problems are candidates for a machine learning solution. The problem must be one that can be solved with data, and a sufficient quantity of relevant data must exist and be acquirable. As we shall see, many security problems fit this profile exceedingly well.

3.3.2 Supervised Learning

Supervised learning algorithms make predictions based on a set of samples, e.g., historical stock prices can be used to try to guess at future prices. A supervised learning algorithm looks for patterns in labeled data. It can use any information that might be relevant, and each algorithm looks for different types of patterns. After the algorithm has detected the best pattern it can, it uses that pattern to make predictions for unlabeled data.

When the data are being used to predict a category, supervised learning is also called classification, e.g., assigning an image as a picture of either a cat or a dog. When there are only two choices, it is called two-class, binomial or binary classification. When there are more categories, this problem is known as multi-class classification.

3.3.3 Feature Extraction

As mentioned in section 3.3.1, we should extract the attributes from the input data so that we can feed it into the algorithm. For example, in the image cases, data can be represented as an RGB value of each pixel.

Such attributes are referred to **features**, and the matrix is referred to as feature vector. The process of extracting data from the files is called feature extraction. The goal of feature extraction is to obtain a set of informative and non-redundant data.

It is essential to understand that features should represent the necessary and relevant information about our dataset since we cannot make an accurate prediction without it. That is why feature extraction is often a non-obvious and domain-specific task, which requires a lot of testing and research.

Another critical requirement for a suitable feature set is non-redundancy. Having redundant features, i.e., elements that outline the same information, as well as irrelevant information attributes, that are strictly dependent on each other, can make the algorithm biased and, accordingly, provide an inaccurate result.

Furthermore, if the input data has too many features, it will take more training time. Hence, we may need to reduce the vector dimensions, which is known as the feature selection.

3.3.4 Classification, Regression and Thresholding

Classification is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y). The output variables are often called labels or categories. The mapping function predicts the class or group for a given observation. For example, an email can be classified as belonging to one of two categories: "spam"and "not spam".

Regression is the task of approximating a mapping function (f) from input variables (X) to a continuous output variable (y). An output variable is a real-value, such as an integer or floating point value. These are often quantities, such as amounts and sizes. For example, a house may be predicted to sell for a specific dollar value, perhaps in the range of $100,000 to $200,000.

Classification problems are different from regression problems. Classification is the task of predicting a discrete class label, and regression is the task of predicting a continuous quantity. However, there is some overlap between the algorithms for classification and regression. For example, we can use the probability, which is returned from logistic regression, "as is"or convert it to a binary value. To map a logistic regression value to a binary category, we must define a **classification threshold** (also called the decision threshold). It is tempting to assume that the classification threshold should always be 0.5, but thresholds are dependent on the specific problems, and therefore values that we must tune.

3.3.5 Ensemble, Bagging and Boosting

When we try to predict the target variable using any machine learning method, the leading causes of difference in original and predicted values are noise, variance, and bias. The ensemble helps to reduce these two last factors.

An ensemble is just a collection of predictors which come together (e.g., mean of all predictions) to give a final prediction. The principal reason is that many different predictors trying to predict same target variable will perform a better job than any single one alone. Ensembling techniques are further classified into Bagging and Boosting.

Bagging is a simple ensembling technique in which we build many independent models and combine them using some model averaging methods (e.g., weighted average, majority vote or normal average). We typically use random sub-sample of data for each model, so that all the models are little different from each other. Each observation has the same probability to appear in all the models. Because this technique takes many uncorrelated predictors to make a final model, it reduces error by reducing variance. One example of bagging ensemble is Random Forest (which is mentioned in section 3.4.2).

Boosting is an ensemble technique in which the models are not made sequentially. This technique applies the logic in which the following predictors learn from the mistakes of the previous ones. Accordingly, the observations have an unequal probability of appearing in subsequent models. The predictors can be chosen from a range of models like decision trees, regressors, classifiers, etc. Because new predictors are learning from mistakes committed by previous predictors, it takes fewer iterations to reach close to actual predictions. But we have to choose the stopping rules carefully, or it could lead to overfitting on training data. Gradient Boosting Decision Tree, in section 3.4.3, is an example of boosting algorithm.

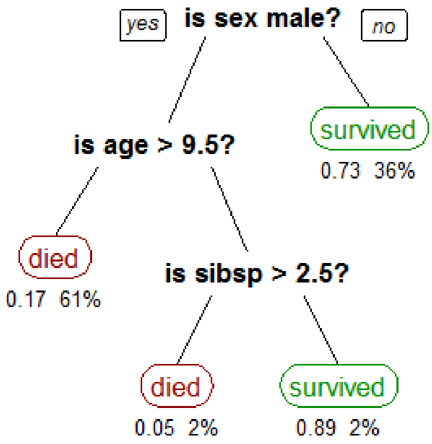
3.4 Machine Learning Methods

3.4.1 Decision Tree

As it assumes from the name, decision trees are data structures that have a structure of the tree. The training dataset is used for the creation of the tree, that is consequently used for making predictions on the test data. In this algorithm, the goal is to achieve the most accurate result with the least number of the decisions that must be made. We can use decision trees can for both classification and regression problems. There is an example in Table 3.1.

Bảng 3.1 An example of decesion tree

Id	Name	Sex	Age	SibSp	Survived
1	Braund, Mr. Owen Harris	male	22.0	1	0
2	Cumings, Mrs. John Bradley	female	38.0	1	1
3	Heikkinen, Miss. Laina	female	26.0	0	1
...	...	...	...	...	...



Hình 3.3 An example of decesion tree [59]

As you can see in Figure 3.3, the model was trained based on the dataset and can now classify the passenger in Titanic is survived or not. The tree consists of the decision nodes and leaf nodes, and decision nodes may have several branches leading to leaf nodes. Leaf nodes represent the decisions or classifications. The first initial node is referred to as root node.

Decision tree method gained its popularity because of its simplicity. It can deal well with large datasets and can handle the noise in the datasets very well. Another advantage is that unlike other algorithms, such as SVM or KNN, decision trees operate in a white box, meaning that we can see how the outcome is obtained and which decisions led to it.

3.4.2 Random Forest

Random Forest is one of the most common machine learning algorithms. It requires almost no data preparation and modeling but usually ends in inaccurate results. Random Forests are based on the decision trees described in the previous section 3.4.1. More specifically, Random Forests are the collections of decision trees, producing a better prediction accuracy. That is why it is called a forest – it is a set of decision trees.

The essential idea is to grow multiple decision trees based on the independent subsets of the dataset. At each node, *n variables* out of the feature set are selected randomly, and the best split on these variables is found.

We can describe the algorithm as follows [8]:

1. Multiple trees are built approximately on the two third of the training data randomly.

2. Several variables are randomly selected out of all the predictor variables. Then, the best split on these is used to split the node. By default, the amount of the selected variables is the square root of the total number of all predictors, and it is constant for all trees.

3. With the rest of the data, the misclassification rate is calculated. The total error rate is calculated as the overall out-of-bag error rate.

4. Each trained tree gives its classification result and the class that received the highest score is chosen as the result.

Since we are using multiple decision trees, this algorithm removes the need for feature selection for removing unnecessary features – they will not be taken into account in any case. The only need for any feature selection with the random forest algorithms arises when there is a demand for dimensionality reduction. Moreover, the out-of-bag error rate, which was mentioned earlier can be considered the algorithm's cross-validation method. This removes the need for tedious cross-validation measures, that would have to be taken otherwise [36].

Random forests inherit many of the advantages of the decision trees algorithms. They are suitable for both regression and classification problems; they are easy to compute and quick to fit. They also regularly result in the better accuracy. However, unlike decision trees, it is not very easy to interpret the results. In decision trees, by examining the resulting tree, we can gain valuable information about which variables are relevant and how they affect the result. Random forests can also be described as a more firm algorithm than the decision trees since it is the combination of many decision trees, the random forest will remain stable [33].

3.4.3 Gradient Boosting Decision Trees

Gradient Boosting Decision Tree (GDBT) is an ensemble model of decision trees, which are trained in sequence [20]. In each iteration, GBDT learns the decision trees by fitting the negative gradients (also known as residual errors). The main cost in GBDT lies in learning the decision trees, and the most time-consuming part of learning a decision tree is to find the best split points. One of the most common algorithms to find split points is the pre-sorted algorithm [34, 48], which lists all possible split points on the pre-sorted feature values. This algorithm is simple and can find the optimal split points. However, it is wasteful in both training speed and memory consumption. Another famous algorithm is the histogram-based algorithm [42, 23, 32].

Instead of finding the split points on the sorted feature values, histogram-based algorithm buckets continuous feature values into discrete bins and uses these bins to construct feature histograms during training.

```
Data: $I$: training data, $d$: max depth
Data: $m$: feature dimension
$nodeSet \leftarrow \{0\}$ ▷ tree nodes in current level
$rowSet \leftarrow \{\{0,1,2,...\}\}$ ▷ data indices in tree nodes
for $i = 1$ to $d$ do
    for $node$ in $nodeSet$ do
        $usedRows \leftarrow rowSet[node]$
        for $k = 1$ to $m$ do
            $H \leftarrow$ new Histogram()
            ▷ Build histogram
            for $j$ in $usedRows$ do
                bin ← I.f[k][j].bin
                H[bin].y ← H[bin].y + I.y[j]
                H[bin].n ← H[bin].n + 1
            end
        end
    end
    Update $rowSet$ and $nodeSet$ according to the best split points
end
```

Algorithm 1: Histogram-based Algorithm

As shown in Algorithm 1, the histogram-based algorithm finds the best split points based on the feature histograms. It costs $O(\#data \times \#feature)$ for building histogram and $O(\#bin \times \#feature)$ for finding the split point. Since $\#bin$ is usually much smaller than $\#data$, histogram building will control the computational complexity. If we can reduce $\#data$ or $\#feature$, we will be able to speed up the training of GBDT extensively.

3.4.4 Support Vector Machine

Support Vector Machines (SVM) is another machine learning algorithm that is commonly used for classification problems. The main idea relies on finding such a hyperplane, that would separate the classes in the best way. The term "support vectors"refers to the points lying closest to the hyperplane, that would change the hyperplane position if removed. The distance between the support vector and the hyperplane is referred to as margin.

Intuitively, we know that the further from the hyperplane our groups lie, the more accurate predictions we can get. That is why, although multiple hyperplanes can be found, the goal of the SVM algorithm is to find such a hyperplane that would result in the maximum margins.

SVMs are usually able to produce good accuracy, particularly on clean datasets. Further, it is good at working with the high-dimensional datasets, also when the number of dimensions is higher than the number of the samples. Additionally, for large datasets with a lot of noise or overlapping classes, it can be more effective. However, with more massive datasets training time can be extended [24].

3.4.5 K-Nearest Neighbors

The k-Nearest Neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. The k-NN does not make any assumptions about the data structure, which makes it become a good solution in the real world where most of the data does not follow the typical theoretical assumptions. K-NN is also a lazy algorithm, which means there is no specific training phase or it is very insignificant. Also, lack of generalization means that k-NN keeps all the training data, i.e., the most the training data is required during the testing phase.

The algorithm is based on feature similarity. How closely out-of-sample features match the training set determines how k-NN classify a given data point. The Euclidean Distance, which is defined by the formula below, is the most used method for continuous variables in k-NN.

$$EuclideanDistance = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

The drawback of the k-NN algorithm is the lousy performance on the unevenly distributed datasets. Hence, if one class hugely overshadows the other ones, it is more likely to have more neighbors of that class due to their large number, and therefore, make incorrect predictions [30].

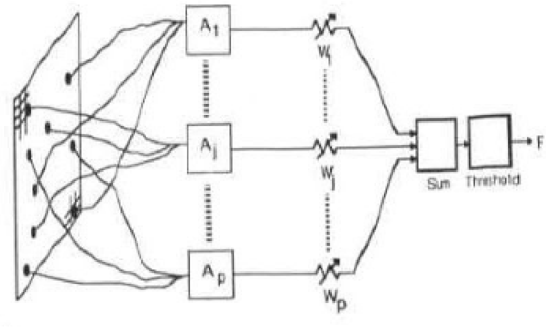
3.4.6 Neural Networks

Overview

The idea behind neural networks, i.e., having computational units that produce “intelligent” results only through interactions with each other, was inspired by the brain. For instance, the Neocognitron system, proposed by Kunihiko Fukushima in 1980, took inspiration from the mammalian visual system and laid the foundation for modern convolutional networks [21]. Hence, the artificial neurons in these networks mimic the structure of biological ones.

The very first artificial model of the biological neuron is in fact perceptron, introduced by Frank Rosenblatt in 1958 [44]. Perceptron is an algorithm for supervised learning in machine learning. Perceptron, in essence, is a simple function that turns inputs (usually a real-valued vector) into one binary output.

Following Figure 3.4, we can see the perceptron receives p inputs, $A_1, A_2, ..., A_p$ (or x_1, x_2, ..., x_p, depending on the source. The inputs are then respectively weighted by w_1, w_2, ..., w_p, which are real numbers indicating the importance of each of the input values. The output F will then be calculated using the sum of those weighted inputs. Additionally, because the output is a binary value, a threshold is used to achieve the desired result. To be more specific, a perceptron is written as follows:



Hình 3.4 Basic structure of a perceptron [35]

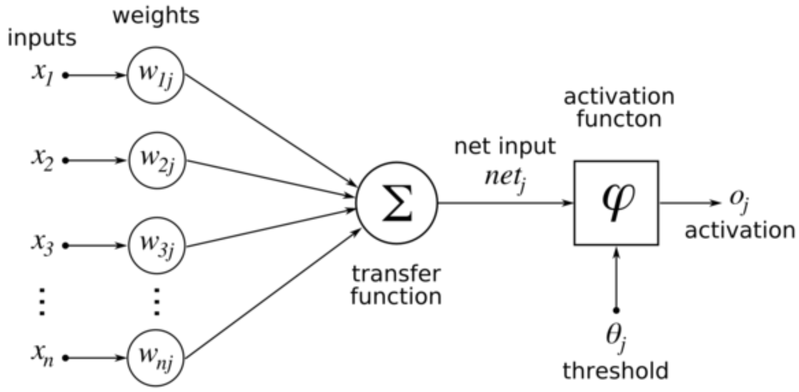
$$F = \begin{cases} 1 & if \sum_i^p A_i w_i \geq Threshold \\ 0 & if \sum_i^p A_i w_i < Threshold \end{cases}$$

Consequently, the output of a perceptron is controlled by two things: the weights w_1, w_2, ..., w_p and the Threshold. In modern neuron networks, however, the equation changed a bit by bringing the Threshold to the other side of the inequalities. The additive inverse of Threshold is then known as Bias, and the perception will be rewritten as:

$$F = \begin{cases} 1 & if \sum_i^p A_i w_i + Bias \geq 0 \\ 0 & if \sum_i^p A_i w_i + Bias < 0 \end{cases}$$

Activation function

Perceptron is the precursor to modern artificial neurons. Instead of returning only a binary output, an artificial neuron now produces values that are anywhere in the range [0,1]. The reason is to help in the learning process of a network. Neural networks can learn, i.e., finding the appropriate weights and biases given sufficient input data. The learning process, however, needs to be progressive, which means weights and biases get increasingly close to the "good"values. Having a binary output for each of the neurons makes it hard for this process to be done. For training a neural network, we use an error function to see how far or close the network is to the optimal results. Because of the binary output of perception, a small change in the network's parameters can lead to a stark difference for the output, making it hard to tune the parameters to achieve a good result. Therefore, modifications must be made to the original perceptron model. Instead of using 0 as the threshold at which signals are allowed to be fired, we can use an activation function to map the output to the range we need appropriately.



Hình 3.5 Structure of an artificial neuron [57]

The activation function takes as input the weighted sum of the input values fed to the perceptron and returns a value in the range [0,1]. Historically, the Sigmoid function was used for this purpose as it can squash the sum to the desired range [31]. Modern networks, however, use different functions as well, such as hyperbolic tan function (tanh) or rectified linear unit (ReLU), to achieve better performance [31].

Feedforward and Backpropagation

Two operations are particularly important in neural networks: Feedforward and Backpropagation.

Feedforward is used in both the training and testing stage of a network. The task we need to do in feedforward is straightforward: Passing output of a layer as the input of the next layer. Since all we are doing is unidirectionally putting values through the network from the input layer to the output layer, the operation is called feedforward.

Backpropagation is generally used in the training stage to help neural networks learn their parameters, and it is considered an optimization. Different from feedforward, the job of this operation is to propagate error of the output values back to the network to update the network parameters. Backpropagation works by first do the normal feedforward operation on the network with the given input. After the output is obtained, we compare the output to the desired output, using a loss function to generate an error term for each of the neuron in the output layer. The error values are then propagated backward from the output layer until every neuron receive their respective error term. The error terms will be used to calculate the gradient, with which we can update the weights of the network to minimize the loss function as the process repeats. To find the most fitting parameters, gradient descend algorithm is usually applied.

3.5 Classification Metrics

In this section, we review how to use several common metrics that are used to evaluate predictions for classification problems.

3.5.1 Logarithmic Loss

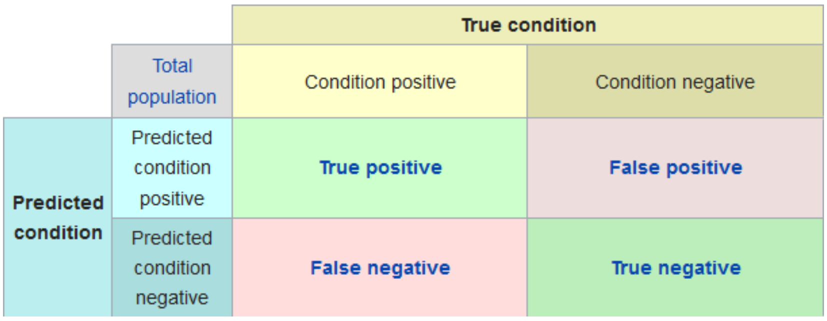
Logarithmic loss, or log-loss for short, is a performance metric for evaluating the predictions of probabilities of membership to a given class.

Log-loss takes into account the uncertainty of your prediction based on how much it varies from the actual label, which gives us a more nuanced view of the performance of our model. In binary classification, with y is a binary indicator (0 or 1) of whether class label $c$is the correct classification and p is the model predicted probability, the formula equals:

$$LogLoss = -(y\log(p) + (1-y)\log(1-p))$$

The scalar probability between 0 and 1 can be seen as a measure of confidence for a prediction by an algorithm. Smaller log-loss is better with 0 representing a perfect log-loss.

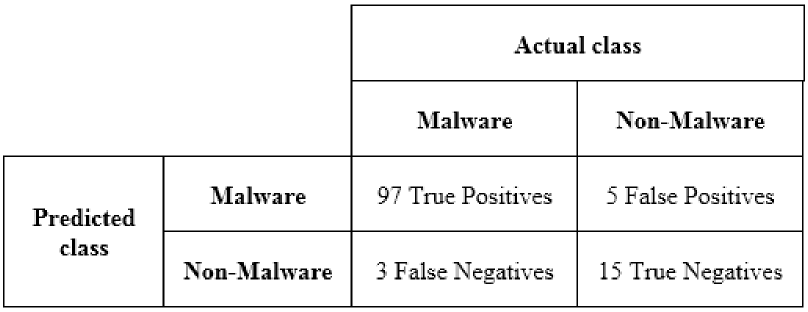
3.5.2 Confusion Matrix



		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	True positive	False positive
	Predicted condition negative	False negative	True negative

Hình 3.6 Confusion matrix [58]

A clean and unambiguous way to show the prediction results of a classifier is to use a confusion matrix (also called a contingency table). For a binary classification problem, the table has two rows and two columns (which is shown in Figure 3.6). Across the top is the actual class labels and down the side are the predicted class labels. Each cell carries the number of predictions made by the classifier that fall into that cell.



		Actual class	
		Malware	Non-Malware
Predicted class	Malware	97 True Positives	5 False Positives
	Non-Malware	3 False Negatives	15 True Negatives

Hình 3.7 An example of confusion matrix

Figure 3.7 is an example of binary classification in malware detection. Some of the input files are malware, and our test correctly says they are positive. They are called true positives (TP). In contrast, some are malware, but the test incorrectly claims they are not. They are called false negatives (FN). Some are clean files, and the test says they are not malware – true negatives (TN). Finally, there might be clean files have a positive test result – false positives (FP).

There are many derived ratios from confusion matrix, and the most common ones are listed below:

- True Positive Rate (TPR), eqv. with hit rate, recall: $TPR = TP/P = TP/(TP + FN)$
- True Negative Rate (TNR): $SPC = TN/N = TN/(TP + FN)$
- Precision or Positive Predictive Value (PPV): $PPV = TP/(TP + FP)$
- Negative Predictive Value (NPV): $NPV = TN/(TN + FN)$
- Fall-out or False Positive Rate (FPR): $FPR = FP/N = FP/(TP + FN) = 1 - TNR$
- False Discovery Rate (FDR): $FDR = FN/(FN + TP) = 1 - PPV$
- Miss Rate or False Negative Rate (FNR): $FNR = FN/(FN + TP) = 1 - TPR$

3.5.3 Overall Accuracy

Overall accuracy is the number of correct predictions made as a ratio of all predictions made.

$$Accuracy = \frac{True\ positive + True\ negative}{Condition\ positive + Condition\ negative}$$

Overall Accuracy essentially tells us out of all of the reference sites what proportion were mapped correctly. The overall accuracy is usually expressed as a percent, with 100% accuracy being a perfect classification where all reference site were classified correctly. Overall accuracy is the easiest to calculate and understand but ultimately only provides the map user and producer with necessary accuracy information.

This is the most common evaluation metric for classification problems, it is also the most misused. It is only suitable when there is an equal number of observations in each class (which is rarely the case) and that all predictions and prediction errors are equally important (which is often not the case).

3.5.4 Precision and Recall

Precision can be thought of as a measure of a classifiers exactness. Precision attempts to answer the question "What proportion of positive identifications was actually correct?". A low precision can also indicate a large number of False Positives.

$$Precision = \frac{True\ positive}{True\ positive + False\ positive}$$

Recall is the number of True Positives divided by the number of True Positives and the number of False Negatives. Computing in another way is the number of positive predictions divided by the number of positive class values in the test data. Recall attempts to answer "What proportion of actual positives was identified correctly?". It is also called Sensitivity or the True Positive Rate.

$$Recall = \frac{\sum True\ positive}{\sum Condition\ positive}$$

3.5.5 Area Under ROC curve

Area Under the Receiver Operating Characteristic curve, AUROC or AUC for short, is a performance metric for binary classification problems. The AUROC has several equivalent interpretations:

- The expectation that a uniformly drawn random positive is ranked before a uniformly drawn random negative.
- The expected proportion of positives ranked before a uniformly drawn random negative.
- The expected true positive rate if the ranking is split just before a uniformly drawn random negative.
- The expected proportion of negatives ranked after a uniformly drawn random positive.

- The expected false positive rate if the ranking is split just after a uniformly drawn random positive.

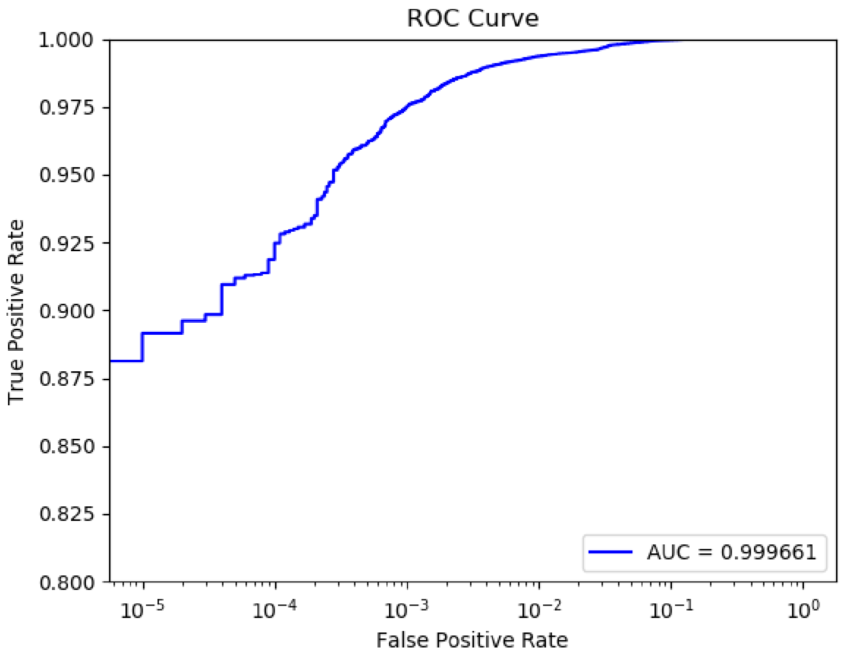
An area of 1.0 represents a model that made all predictions perfectly. A rough guide for classifying the accuracy of a classification test is the typical academic point system:

- 0.9 - 1.0 = Excellent
- 0.8 - 0.9 = Good
- 0.7 - 0.8 = Fair
- 0.6 - 0.7 = Poor
- 0.5 - 0.6 = Fail

Compute the AUROC

Assume we have a binary classifier such as logistic regression. First, we compute two metrics from the confusion matrix (their formula are mentioned in section 3.5.2), which will be later combined into one:

1. **True positive rate (TPR).** Intuitively this metric agrees to the proportion of positive data points that are correctly considered as positive, with respect to all positive data points. In other words, the higher TPR, the fewer positive data points we will miss.
2. **False positive rate (FPR).** This metric corresponds to the proportion of negative data points that are mistakenly considered as positive, concerning all negative data points. In other words, the higher FPR, the more negative data points will be misclassified.



Hình 3.8 An example of Receiver Operating Characteristic curve

Then, we combine the FPR and the TPR into one single metric by computing the two former metrics with many different threshold (for example, 0.00, 10^-5, 10^-4, 10^-3, ..., 1.00, as shown as in Figure 3.8) for the logistic regression, then plot them on a single graph, with the FPR values on the x-axis and the TPR values on the y-axis. The resulting curve is called Receiver Operating Characteristic curve, and the metric we consider is the area under this curve.

3.6 LightGBM - A Gradient Boosting Framework

LightGBM, which means Light Gradient Boosting Machine, is a gradient boosting framework that uses tree-based learning algorithm [25]. This framework is obtaining an extreme reputation due to its following advantages:

- Faster training speed and higher efficiency
- Lower-memory usage

- Better accuracy
- Parallel and GPU learning supported
- Capable of handling the large-scale data

The framework uses two following techniques to solve problems when the feature dimension is high, and data size is considerable: Gradient-based One-Side Sampling and Exclusive Feature Bundling.

3.6.1 Gradient-based One-Side Sampling

Base on the notice that, while there is no weight for data instance in the gradient-boosting decision tree, data instances with different gradients play different roles in the computation of information gain. In particular, according to the definition of information gain, instances with larger gradients (i.e., under-trained instances) will contribute more to the information gain. Since, when subsampling the data instances, to retain the accuracy of information gain estimation, LightGBM tends to keep instances with large gradients (e.g., larger than a pre-defined threshold, or among the top percentiles), and only randomly drops instances with small gradients. They proved that such a treatment could lead to a more accurate gain estimation than uniformly random sampling, with the same target sampling rate, especially when the value of information gain has a vast range.

3.6.2 Exclusive Feature Bundling

Regularly in real applications, although there are a large number of features, the feature space is quite sparse, which provides the LightGBM a possibility of using a nearly lossless approach to reduce the number of active features. In fact, in a sparse feature space, many features are almost exclusive, i.e., they rarely take nonzero values together, e.g., the one-hot encoding features, therefore, the framework could safely bundle such unique features. LightGBM uses an efficient algorithm named Exclusive Feature Bundling, which is a greedy algorithm with a constant approximation ratio. Specifically, they reduce the optimal bundling problem to a graph coloring problem by taking features as vertices and add edges for every two features if they are not together exclusive.

Chương 4

Proposed Method

Tóm tắt chương

This chapter describes the issues of using imbalanced Dataset, feature extraction methods, the proposed Gradient Boosting Decision Trees model, and the reasoning behind.

4.1 The Issues of using Imbalanced Dataset

Most of the related work use the imbalanced dataset [45, 54]. For examples, Saxe and Berlin used the dataset of 431,926 binaries which consists 350,016 malicious files [45], Vu Thanh Nguyen et al. used the dataset of 9690 files which only has 300 benign files [54]. In fact, the number of malicious files is often much more massive than the number of benign files because almost benign binaries are often protected by the copyright laws which do not allow for sharing. This makes malware identification problem become different from other machine learning classification problems, which commonly have fewer samples in important classes. Further, the size of the dataset is usually not large enough because the malware analysis and data labeling are consuming processes that required well-trained security engineers. There are also many risks in publishing a large dataset that includes malicious binaries.

Using imbalanced datasets can make validation metrics misleading. For examples, with 96.9% of data is malicious files, a model that labels all samples as malware achieves 96.9% accuracy, 96.9% precision (P), 100% recall (R) and 0.9843 F-score ($F = 2PR/(P+R)$ [12]). It also gives way to false positives, which cause negative user experiences. According to a survey of IT administrators in 2017, 42 percent of companies assume that their users lost productivity as an issue of false-positive results, which creates a choke point for IT administrators in the business life cycle [15].

4.2 Feature Extraction

By using the simple feature extraction methods inspired by the EMBER dataset owners rather than raw binary files, collecting data is not affected by privacy policies and it is much easier to get a balanced dataset. By conducting many experiments, we decrease the feature dimension by 30 percent (1711 instead of 2351) to reduce the training time but still manage to achieve a better evaluation result. In detail, we extracted each Portable Executable file into eight feature groups which can be classified into two types: format-agnostic features and parsed PE features. File-format agnostic feature groups decrease privacy concerns while parsed PE feature groups encapsulates the information related to executable code.

4.2.1 Format-agnostic Features

We use three groups of features to model the contents of input files in a file-format agnostic way, meaning that it has no dependencies on its format.

Byte-Entropy Histogram

Based on the work published by Joshua Saxe and Konstantin Berlin [45], they show that, in practice, the effect of representing byte values in the entropy context in which they occur separates byte values that arise in the context of, for example, x86 instruction data from, for example, byte values occurring in compressed data.

To compute the byte-entropy histogram, we slide a 2048-length window over all the input bytes with a step size of 1024 bytes. Use a simple trick to calculate the entropy H faster, i.e., reducing the information by half, and pairing it with each byte within the window. Then, we compute a two-dimensional histogram with 16×16 bins that quantize entropy and the byte value. Finally, we concatenate each row vector in the matrix and normalize the final 256-value vector.

Byte Histogram

The byte histogram is a 256-value vector which represents the distribution of each byte value within the file.

String Information

The final format-agnostic group of features is string information. These features are derived from the printable sequences of characters in the range `0x20` to `0x7f`, that have at least five characters long. We use the number of strings, the average length of these strings, the amounts of lines that may sequentially indicate a path (begin with `C:\`), an URL(start with `http://` or `https://`), a registry key (the occurrences of `HKEY_`) and a bundled executable (the short string `MZ`). Also, we use a histogram of the printable characters within these strings.

4.2.2 Parsed Features

In addition to using three format-agnostic groups of features, we extract five other groups from parsing the Portable Executable file by using LIEF - Library to Instrument Executable Formats [40].

General Information

The set of features includes the file size and necessary information collected from the PE header: the virtual size of the file, the number of imported and exported functions, the number of symbols, whether the data has a debug section, thread local storage, resources, relocations, or a signature.

Header Information

We use the information from the Common Object File Format (COFF) header including the timestamp in the header, the target machine and a list of image characteristics. And from the optional header, we use the target subsystem, DLL characteristics, the file magic as a string, major and minor image versions, linker versions, system versions and subsystem versions, and the code size, header size and commit size. We use hashing trick with 10 bins for string features [56].

Imported Functions

Parsing the import address table gives us a report about the imported functions by libraries. We use the set of unique libraries with 128-bin hashing trick. Similarly, we apply the 512-bin hashing trick to capture individual functions, by representing it in string format `library:function`, for example, `kernel32.dll:CreateFileMappingA`.

Exported Functions

Similar to extracting imported functions, we summarize a list of the exported functions into a 128-value vector by hashing.

Section Information

Properties of each section are used: the name, size, entropy, virtual size, and a list of strings representing section characteristics. We still use the hashing trick on `(section name, va` pairs to create 50-value vectors containing section size, section entropy, virtual size, and information about entry point characteristics.

4.3 Classification

For classification, in this study, we use the Gradient Boosting Decision Trees algorithm with 400 iterations and 64 leaves in one tree. We configure that there must be at least 200 samples in one child, and set learning rate at 5 percent. We elaborate on the reasoning behind these choices below.

Firstly, the massive number of features causes scalability issues for many machine learning algorithms. For example, non-linear SVM kernels require $O(N^2)$ multiplication during each iteration, and k-Nearest Neighbors (k-NN) requires significant computation and storage of all label samples during prediction. Accordingly, we target to use neural networks and ensemble decision trees, which are scalable alternatives.

Secondly, our resources, primarily financial support, are lacking. But the cost for training neural networks is extremely computationally expensive. We tried with some complex models, and these take many hours and require costly more GPUs for speeding. Also, neural networks are the black-box that not much can be gleaned from and required much experience to optimize.

Besides, another scalable alternative, tree ensemble algorithms handle very well high dimensional feature spaces as well as a large number of training examples. The two most popular algorithms are Random Forests and Gradient Boosting Decision Trees (GDBT). GBDT training usually takes longer because trees are built sequentially. However, benchmark results have shown GBDT are better learners than Random Forests.

Chương 5

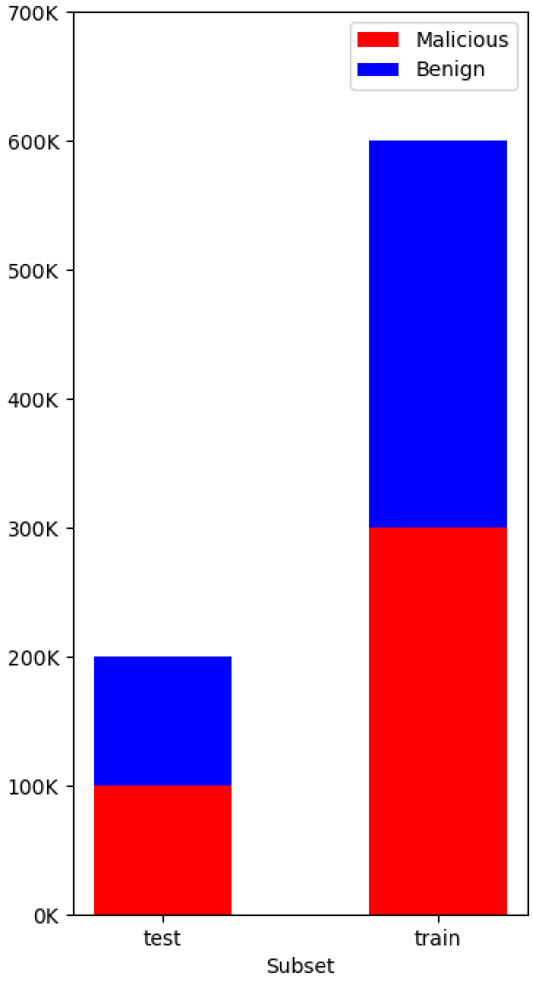
Experiment and Evaluation

Tóm tắt chương

Firstly, we introduce the dataset used for training and evaluating in our experiments. We also present the evaluation metrics and explain the reason for choosing those criteria. Then, the experiment results are shown in comparing to other proposed models. Finally, we present how to set up the environments for experiments.

5.1 Dataset

In our experiment, we use 600,000 labeled training samples and 200,000 testing samples from Endgame Malware BEnchmark for Research (EMBER) dataset [3].



Hình 5.1 Distribution of samples in the dataset.

The EMBER dataset is the sizeable public dataset for malware detection which must include benign files and have an ideal ratio of malicious and benign files for machine learning tasks. This will solve the common problem of predictive accuracy, i.e., it will be misleading when the data is imbalanced.

5.2 Evaluation Criteria

As discussed in section 2.3, the goal of the thesis is to find a machine learning-based malware detection method that operates at a low false positive rate while tries to achieve a high detection rate.

5.2.1 False Alarm Rate

False positives, or false alarms, happen when a detector mistakes a malicious label for a benign file. We intend to make the false positive rate as low as possible, which is untypical for machine learning application. It is important because even one false alarm in a thousand benign files can create severe consequences for users. This problem is complicated by the fact that there are lots of clean files in the world, they keep appearing, and it is more challenging to collect these files. We evaluate the accuracy of our method at two specific false alarm rate values: at less than 0.1%, and at less than 1%.

$$False\ alarm\ rate = \frac{\sum False\ positive}{\sum Condition\ negative}$$

5.2.2 Detection Rate

The detection rate, (eqv. with recall or true positive rate), measures the ratio of malicious programs detected out of the malware files used for testing. With higher recall, fewer actual cases of malware go undetected. In other words, the true positive rate shows the potential of how unseen binaries that were detected.

$$Detection\ rate = \frac{\sum True\ positive}{\sum Condition\ positive}$$

5.2.3 Area Under the ROC curve

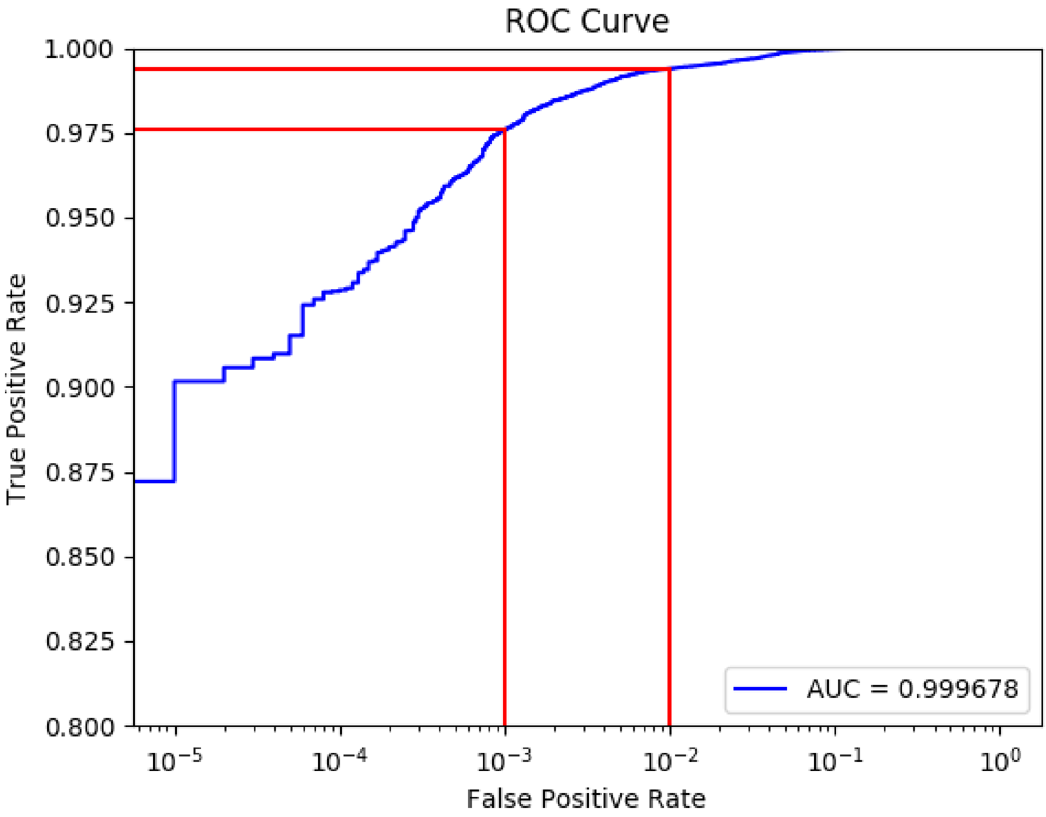
As introduced in section 3.5.5, the Area Under the ROC curve, AUROC or AUC for short, provides an aggregate measure of performance across all possible classification thresholds. AUC is scale-invariant and measures how well predictions are ranked, rather than their absolute values. Besides, AUC is classification-threshold-invariant, so that it can measure the quality of the predictions irrespective of what threshold is chosen. A model whose predictions are 100% wrong has an AUC of 0.0, and the one whose predictions are 100% correct has an AUC of 1.0.

A rough guide for classifying the accuracy of a classification test is the typical academic point system:

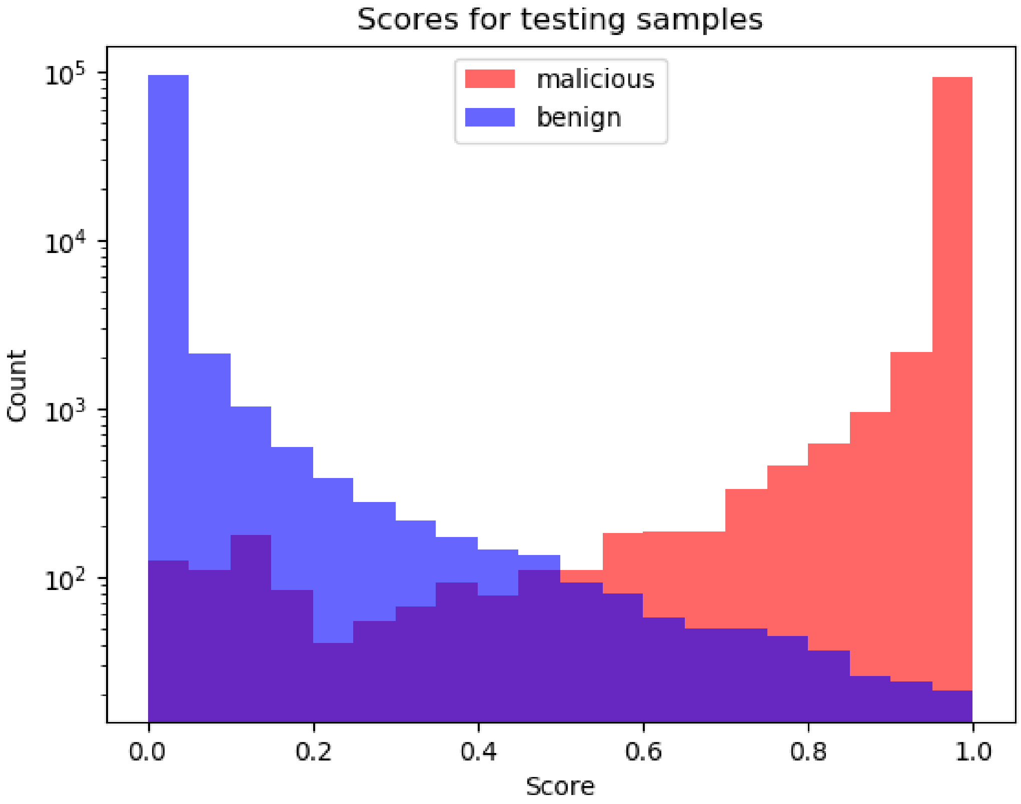
- 0.9 - 1.0 = Excellent
- 0.8 - 0.9 = Good
- 0.7 - 0.8 = Fair
- 0.6 - 0.7 = Poor
- 0.5 - 0.6 = Fail

5.3 Experimental Results

The proposed GBDT-based malware detection method is implemented with LightGBM framework [25], and the input feature vectors have dimension of 1711. All our experiments were run on an instance which has 24 vCPUs and 32 GB memory. Using parallel programming, it took about 10 minutes to vectorize the raw features and about 5 minutes to train the model. The ROC curve of the final model is shown in Figure 5.2 and the distribution of scores for testing samples is shown in Figure 5.3.



Hình 5.2 The ROC curve of proposed model



Hình 5.3 The distribution of scores for testing samples

The area under ROC curve exceeds 0.999678, which means that almost all the predictions are correct. With a threshold of 0.828987, the model score results in less than 0.1% false alarm rate at a detection rate 97.5720%. At less than 1% false positive rate, the model exceeds 99.3940% detection rate with a threshold of 0.307897.

The baseline model has only the area under the ROC curve of 0.99911, score results in less than 0.1% FPR at TPR exceeding 92.99%, and at less than 1% FPR, it exceeds 98.2% TPR. Our model has better performance because of hyper-parameter tuning, and it also takes less time for training as a result of reducing feature space. Evidently, the model has better performance than the MalConv model trained on the raw binaries [3], which has ROC AUC is 0.99821, corresponding to a 92.2% TPR at a FPR less than 0.1%, and a 97.3% TPR at a less than 1% FPR. Table 5.1 and Table 5.2

show the training time and evaluation results of our proposed model in comparison with MalConv model and the dataset owners' baseline model.

Bảng 5.1 The training time of our proposed model in comparison with MalConv model and the dataset owners' baseline model

Model	Input	Specifications	Training time
MalConv	Raw binaries	2 NVDIA TITAN X (Pascal) GPUs	10 days (25 hours/epoch)
EMBER	2351-value vectors	8 vCPUs (2015 MacBook Pro i7)	20 hours
Our model	**1711**-value vectors	24 vCPUs (Google Compute Engine)	**10 minutes**

Bảng 5.2 The evaluation results of our proposed model in comparison with MalConv model and the dataset owners' baseline model

Model	False Alarm Rate (FPR)	Detection Rate (TPR)	Area Under the ROC curve (AUC)
MalConv	0.1 %	92.200 %	0.998210
	1.0 %	97.300 %	
EMBER	0.1 %	92.990 %	0.999110
	1.0 %	98.200 %	
Our model	0.1 %	97.572 %	0.999678
	1.0 %	99.394 %	

5.4 Environment Setup Guide

We mainly use the cross-platform tools in research and development for easily switching between operating systems. We use a Windows 10 Pro virtual machine for static malware analysis, an Ubuntu 16.04 LTS cloud instance for training and testing machine learning models, and use PyCharm Professional as mainly Integrated development environment (IDE).

5.4.1 Windows environment for static analysis

We use a virtual machine to build a background about malware:

- OS: Microsoft Windows 10 Pro
- Version: 10.0.17134
- Architecture: 64-bit

With following tools, we can easily gather malware basic information:

- **CFF Explorer**: PE header parser.
- **PE Explorer** (from Heaventools Software): PE inspection tool.
- **BinText** (from McAfee): extract string from a binary.
- **HxD Hex Editor**: support for viewing file in binary format.

5.4.2 Ubuntu environment for machine learning tasks

Google Cloud Platform

We use an virtual machine for research, that can be deployed with the image from *Cloud Launcher - Canonical - Ubuntu Xenial*. The cloud instance has 24 virtual CPUs, 32 GB for memory, and is located in `asia-southeast1-b` zone, i.e., Jurong West, Singapore.

After deployment, we add two optional firewall rules (*VPC network - Firewall rules - Create a firewall rule*), which allows all in and out connections for the virtual machine, to use Python Interactive Console features in PyCharm IDE.

Anaconda

We choose Anaconda, a free and open source distribution of the Python, to manage package and deploy. The content of `environment.yml` used to deploy is shown below.

```
name: lab
channels:
  - conda-forge
dependencies:
  - python==3.6
  - matplotlib
  - numpy
  - scikit-learn
  - pip:
    - lief
    - git+https://github.com/onnx/onnxmltools
    - lightgbm
```

The environment is created with **Python 3.6** and packages for machine learning:

- **NumPy**: the fundamental package for scientific computing with Python.
- **Matplotlib**: a Python 2D plotting library.
- **Scikit-learn**: a machine learning library.
- **Lief**: library to instrument executable formats.
- **LightGBM**: a gradient boosting framework based on decision tree algorithms.
- **ONNXMLTools**: a tool to convert models to ONNX format.

5.4.3 PyCharm Professional IDE

JetBrains provides *free individual licenses for students* to use PyCharm Professional IDE. This is the powerful Python IDE, which gives us remote development capabilities and supports many scientific tools (e.g., Anaconda, Matplotlib and NumPy).

Following the guide *Configuring Remote Interpreters via SSH* published by JetBrains, we can run, debug remotely from a cloud instance, which gives a great performance and is easy to scale.

Chương 6

Conclusion

Tóm tắt chương

Chapter 6 presents the results of this thesis, including what we have learned and achieved through the experiments. The chapter closes with our proposal for future work.

6.1 Results

Over the course of doing this thesis, we have spent a good amount of time studying about Malware Detection and Machine Learning, including Neural Networks and Gradient Boosting Decision Trees, to acquire essential malware knowledge.

We have learned and distinguished static and dynamic malware detetion. We have also analyzed the technical detail and meaning of features used in proposed static malware detectors. Understanding them is important both for understanding the state-of-the-art methods and for building and optimizing classifiers.

We present and optimize a static malware detection method using hand-crafted features derived from parsing the PE files and Gradient Boosting Decision Trees (GBDT), a widely-used powerful machine learning algorithm. We manage to reduce the training time by appropriately reducing the feature dimension. In detail, rather than using raw binary files, our proposed method uses the statistical summaries to decrease the privacy concerns of various benign files and makes it easy to request the balanced dataset.

The experiment results show that our proposed method can achieve up to 99.394% detection rate at 1% false alarm rate, and score results in less than 0.1% false alarm rate at a detection rate 97.572%, based on more than 600,000 training and 200,000 testing samples from EMBER dataset [3].

6.2 Future Works

The study conducted in this project was a proof-of-concept, and we can identify some future developments related to the practical implementation:

1. **Reduce the feature space.** There is possible to reduce the dimension of feature vectors. Input vectors with smaller size boost the model and take less training time.

2. **Use other datasets.** Although the EMBER dataset is broad, covering most of the malware species, it does not include all possible kinds. Collecting a dataset is a task that requires a lot of time and efforts, especially in malware detection domain. With using format-agnostic features, we can receive more samples from security organizations in future.

3. **Implement the approach in local computer.** We tried to implement the model with ONNX format and Windows ML format but it was not success because the preview version of Windows ML is changed rapidly and has many limits. We plan to build a demonstration application to propose that machine learning-based malware detectors can run smoothly in personal computer.

Tài liệu tham khảo

[1] Abou-Assaleh, T., Cercone, N., Keselj, V., and Sweidan, R. (2004). N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE.

[2] Almási, A.-D., Woźniak, S., Cristea, V., Leblebici, Y., and Engbersen, T. (2016). Review of advances in neural networks: Neural design technology stack. *Neurocomputing*, 174:31–41.

[3] Anderson, H. S. and Roth, P. (2018). Ember: An open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*.

[4] Athiwaratkun, B. and Stokes, J. W. (2017). Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486.

[5] Banko, M. and Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting on association for computational linguistics*, pages 26–33. Association for Computational Linguistics.

[6] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer.

[7] Benchea, R. and Gavriluţ, D. T. (2014). Combining restricted boltzmann machine and one side perceptron for malware detection. In *International Conference on Conceptual Structures*, pages 93–103. Springer.

[8] Biau, G. (2012). Analysis of a random forests model. *Journal of Machine Learning Research*, 13(Apr):1063–1095.

[9] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[10] Chawla, N. V. (2009). Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 875–886. Springer.

[11] Chen, Q. and Bridges, R. A. (2017). Automated behavioral analysis of malware A case study of wannacry ransomware. *CoRR*.

[12] Chinchor, N. (1992). Muc-4 evaluation metrics, in proceedings of the fourth message understanding conference (muc-4). *Morgan Kaufman Publishers*, page 22.

[13] Cohen, F. (1987). Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35.

[14] Cohen, W. W. (1995). Fast effective rule induction. In *Machine Learning Proceedings 1995*, pages 115–123. Elsevier.

[15] Crowe, J. (2017). Security false positives cost companies $1.37 million a year on average.

[16] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426.

[17] DuPaul, N. (2012). Common malware types: Cybersecurity 101.

[18] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6.

[19] Fradkin, D. and Madigan, D. (2003). Experiments with random projections for machine learning. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 517–522. ACM.

[20] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

[21] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

[22] Jang, J., Brumley, D., and Venkataraman, S. (2011). Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM.

[23] Jin, R. and Agrawal, G. (2003). Communication and memory efficient parallel decision tree construction. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 119–129. SIAM.

[24] Jing, R. and Zhang, Y. (2010). A view of support vector machines algorithm on classification problems. In *Multimedia Communications (Mediacom), 2010 International Conference on*, pages 13–16. IEEE.

[25] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157.

[26] Kearn, M. (2016). Machine learning is for muggles too!

[27] Kephart, J. O., Sorkin, G. B., Arnold, W. C., Chess, D. M., Tesauro, G. J., White, S. R., and Watson, T. (1995). Biologically inspired defenses against computer viruses. In *IJCAI (1)*, pages 985–996.

[28] Kolter, J. Z. and Maloof, M. A. (2006). Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744.

[29] Kovacs, E. (2015). False positive alerts cost organizations $1.3 million per year: Report.

[30] Laaksonen, J. and Oja, E. (1996). Classification with learning k-nearest neighbors. In *Neural Networks, 1996., IEEE International Conference on*, volume 3, pages 1480–1483. IEEE.

[31] Li, F.-F., Karpathy, A., and Johnson, J. (2015). Cs231n: Convolutional neural networks for visual recognition. *University Lecture*.

[32] Li, P., Wu, Q., and Burges, C. J. (2008). Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in neural information processing systems*, pages 897–904.

[33] Louppe, G. (2014). Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502*.

[34] Mehta, M., Agrawal, R., and Rissanen, J. (1996). Sliq: A fast scalable classifier for data mining. In *International Conference on Extending Database Technology*, pages 18–32. Springer.

[35] Minsky, M. and Papert, S. (1969). Perceptron (expanded edition).

[36] Mitchell, T. M. et al. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

[37] Moir, R. (2003). Defining malware: Faq. *Microsoft Windows Server*.

[38] Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE.

[39] Nguyen, V. T., Nguyen, T. T., Mai, K. T., and Le, T. D. (2014). A combination of negative selection algorithm and artificial immune network for virus detection. In *Future data and security engineering*, pages 97–106. Springer.

[40] Quarkslab, R. T. (2017). Library to instrument executable formats.

[41] Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. (2017). Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*.

[42] Ranka, S. and Singh, V. (1998). Clouds: A decision tree classifier for large datasets. In *Proceedings of the 4th Knowledge Discovery and Data Mining Conference*.

[43] Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., and Ahmadi, M. (2018). Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*.

[44] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

[45] Saxe, J. and Berlin, K. (2015). Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20.

[46] Schneier, B. (2005). Sony's drm rootkit: The real story.

[47] Schultz, M. G., Eskin, E., Zadok, E., and Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 38–.

[48] Shafer, J., Agrawal, R., and Mehta, M. (1996). Sprint: A scalable parallel classifier for data mining. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 544–555. Citeseer.

[49] Shafiq, M. Z., Tabish, S. M., Mirza, F., and Farooq, M. (2009a). A framework for efficient mining of structural information to detect zero-day malicious portable executables. Technical report, Technical Report, TR-nexGINRC-2009-21.

[50] Shafiq, M. Z., Tabish, S. M., Mirza, F., and Farooq, M. (2009b). Pe-miner: Mining structural information to detect malicious executables in realtime. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141.

[51] Shahi, G., Pang, E., and Fong, P. (2009). *Technology in a Changing World*. lulu.com.

[52] Van Nhuong, N., Nhi, V. T. Y., Cam, N. T., Phu, M. X., and Tan, C. D. (2014). Semantic set analysis for malware detection. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 688–700. Springer.

[53] Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM.

[54] Vu, T. N., Nguyen, T. T., Trung, H. P., Do Duy, T., Van, K. H., and Le, T. D. (2017). Metamorphic malware detection by pe analysis with the longest common sequence. In *International Conference on Future Data and Security Engineering*, pages 262–272. Springer.

[55] Weber, M., Schmid, M., Schatz, M., and Geyer, D. (2002). A toolkit for detecting and analyzing malicious software. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 423–431. IEEE.

[56] Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM.

[57] Wikipedia (2018a). Artificial neural networks.

[58] Wikipedia (2018b). Confusion matrix.

[59] Wikipedia (2018c). Decision tree learning.

[60] Wikipedia (2018d). Portable executable.