Network-based Protocol Implementation Engineering and A Study on Machine Learning based Binary Code Analysis

by Shaowen Sun

B.S. in Electronic and Infomatic Engineering, June 2017, North China University of Technology

A Thesis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Master of Science

May 19, 2019

Thesis directed by

Guru Venkataramani
Associate Professor of Electrical and Computer Engineering
© Copyright 2019 by Shaowen Sun
All rights reserved
Acknowledgments

Foremost, I would like to thank my advisors, Professor Tian Lan, and Professor Guru Venkataramani who are always patient to me and support this thesis. They are not only my thesis & academic advisor but also my mentors. I am deeply grateful to be able to research under their guidance. I also want to thank Professor Payman Dehghanian for serving as one of my thesis dissertation committee.

Besides, I thank all staff in the Lab for Intelligent Networking and Computing (LINC), especially Hongfa Xue, Yurong Cheng, and Yimeng Wang. Thanks for their encourage and help, I benefit a lot throughout the past three semesters.

Last, I much appreciate the support from my parents, Ling Wan and Hao Sun. They have always been tolerating my stubborn and supporting me in the past 20 years.
Abstract

Network-based Protocol Implementation Engineering and A Study on Machine Learning based Binary Code Analysis

Network-based protocol implementations play a significant role in the communication area, but the feature creep and increasing vulnerabilities have become serious problems. To deal with these problems, we proposed TOSS, an online server systems tailoring tool through binary rewriting. TOSS minify binary programs with just-enough features to reduce the potential attack surface and deal with the debloating problem. Besides, this paper also gives a comprehensive study of machine learning based binary code analysis to find the relationship between binary code and machine learning algorithm. For network-based protocol implementations whose source code is available, we proposed a packet generation and analysis functions identification tool through slicing, and test it with several network-based protocol applications.
Table of Contents

Acknowledgments ......................................................... iv
Abstract ................................................................. v
List of Figures .......................................................... viii
List of Tables ............................................................. ix

1 Introduction ............................................................ 1

2 Background ............................................................. 3
  2.1 Network Protocols .................................................. 3
  2.1.1 Transport Layer Security / Secure Sockets Layer .......... 3
  2.1.2 Message Queuing Telemetry Transport .................... 5

3 TOSS: Tailoring Online Server Systems through Binary Feature Customization ...................................................... 10
  3.1 Motivation ........................................................... 10
  3.2 System Design ....................................................... 12
    3.2.1 Feature Identification ........................................ 11
    3.2.2 Feature Customization ....................................... 12
  3.3 Implementation ................................................... 12
  3.4 Evaluation .......................................................... 12
  3.5 Discussion ......................................................... 13

4 Machine Learning based Binary Code Analysis: A Comprehensive Study ......................................................... 14
  4.1 The General Framework .......................................... 14
  4.2 Feature Extraction ................................................ 15
    4.2.1 Graph based Features ...................................... 15
    4.2.2 Code based Features ....................................... 17
  4.3 Feature Embedding ................................................ 19
    4.3.1 Graph embedding ........................................... 19
    4.3.2 Code Embedding .......................................... 19
  4.4 Machine Learning Algorithm .................................... 20
    4.4.1 Supervised Learning ....................................... 20
    4.4.2 Unsupervised Learning .................................... 21
    4.4.3 Deep Learning ............................................. 23
  4.5 Application ........................................................ 23
    4.5.1 Code Clone detection ....................................... 23
    4.5.2 Function Recognition ...................................... 24
    4.5.3 Malware Detection ......................................... 25
    4.5.4 Vulnerability Discovery ................................... 25
    4.5.5 Authorship Recognition ................................... 26
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Transport Layer Security (TLS) / Secure Sockets Layer Handshake (SSL)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of record layer</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Various Packets of TLS / SSL</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>A Message Queuing Telemetry Transport (MQTT) Network</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>Subscribe and Publish with different quality of services</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>The general structure of MQTT packet</td>
<td>7</td>
</tr>
<tr>
<td>2.7</td>
<td>Various packets of MQTT protocol</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>The general framework of TOSS</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>The general framework of machine learning based binary code analysis</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>The Overview of Framework</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>An illustration of RNN</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>An illustration of RAE</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Number of instructions discovered by tracing and Tainting guided symbolic execution on Mosquitto Publisher and Computation Offloading Server . . . . 13

4.1 List of Attributes used in Attributed Control Flow Graph (ACFG). . . . . . . 16
4.2 Supervised Learning in Binary Code Analysis. . . . . . . . . . . . . . . . . 20
4.3 Unsupervised Learning in Binary Code Analysis. . . . . . . . . . . . . . . . 22
4.4 Deep Learning in Binary Code Analysis. . . . . . . . . . . . . . . . . . . . 22
4.5 Binary Code Clone Detection . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.6 Function Recognition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.7 Malware detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.8 Vulnerabilities discovery . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.9 Authorship Recognition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.10 Other Types of Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.1 Training and Testing Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.2 Preliminary Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
Chapter 1: Introduction

With the increasing demand for various user requirements, network-based protocols make the size of their implementations larger, and the increased need also leads to more additional features, which known as feature creep [22]. Meanwhile, the larger size of programs and more features induces larger potential attack surface and more vulnerabilities. For instance, the Heartbleed Bug (CVE-2014-0160) of OpenSSL and trap communication (CVE-2014-3565) of SNMP are all vulnerabilities of protocol applications resulting in serious security problems. Practical approaches are needed to fix these problems.

An effective method to fix these vulnerabilities is to directly customize the program into a new one with just-enough features. For most case, the source code is not available, while binary code analysis allows software engineers to access the program directly. Recently, many binary code analysis techniques [4, 24, 53, 7] have tried to fix these security problems. However, these researches mainly focus on code reused and are with a limited scope of user inputs.

To reduce the potential attack surface and enlarge the scope of users, we present TOSS, a method to reduce the size of programs through customizing online servers and software systems. In binary code, a feature can be represented as a set of basic blocks which corresponding to independent, well-contained and stateless functionalities of programs. Our framework first uses dynamic tainting trace basic blocks of desired features, then use tainting guide symbolic execution to discover more code related to desired features. With the merging result from dynamic tainting and symbolic execution, TOSS outputs a customized program which has desired functionalities.

The binary code analysis can be a good excellent method in vulnerabilities detection, but the size of binary code limit its performance. In 1996, Tesauro et al. [51] first introduced a neural network to detect virus which changes the binary code analysis. Since Tesauro et
al. [51] first introduced a machine learning to binary code analysis, machine learning based binary code analysis has become an import research field. To learn the possible of adapting this technique in network-based protocol implementation, a comprehensive study has to make. Thus, this paper gives a study of machine learning based binary code analysis in different aspects, and machine learning algorithms will be listed and discuss. At last, we give our thought of future direction.

For network protocol implementations whose source code is open, the source code analysis technique is available. However, identifying and modification of source code can be considerable work, so tools are required to identify and modify them automatically. In the source code, a feature can be represented as a few functions. Among functions of network based protocol implementations, the functions of packet generation and analysis play an important role in communication security, but there are many vulnerabilities in these functions. To locate these functions, we are proposed a framework to automatically identify functions of packet generation and analysis via domain specific program slicing. In this framework, the domain specific program slicing is adapted and generated a Control Flow Graph of desired functions. Then, we use random work and sub-path merging to generate paths of desired functions. At last, we use machine learning to create an identification model.
Chapter 2: Background

Before introduce our research in network based protocol, we will give a brief overview of the background needed to understand new framework presented in this work.

2.1 Network Protocols

In our research and study, we adopted two kinds of network based protocol: Transport Layer Security / Secure Sockets Layer and Message Queuing Telemetry Transport.

2.1.1 Transport Layer Security / Secure Sockets Layer

Transport Layer Security (TLS) / Secure Socket Layer (SSL) is a protocol aiming at communication privacy and reliability between applications [17]. This network protocol has been widely adopted in various communication implementation, web browsers, and websites for cryptography. To establish a connection between different applications, there is a handshake process between client and server shown in figure 2.1. During handshake, the client will first send a ClientHello packet to the server. After receiving the ClientHello, the server will send back a ServerHello as the feedback for ClientHello, Certificate & ServerKeyExchange & CertificateRequest to ask the client for authentication and required key exchange algorithms and ServerHelloDone to indicate that the ServerHello message finished. Then, the client will send back ClientKeyExchange, Certificate & CertificateVerify, ChangeCipherSpec, and Finished. Once receives these packet, the server will send back ChangeCipherSpec and the handshake process finishes. After the handshake, both servers and clients can send application data to each side.

In general, The TLS / SSL protocol has four types of packets: changecipherspec, handshake, application data and alert packet. All of them is formed by two sections which are the record layer and corresponding layer (handshake layer, application data layer, alert
layer, and change cipher spec alyer). The record layer, also Known as the record header, is consisted of content type, protocol version, and length of remaining fields. For example, handshake packets may have a record layer shown in figure 2.2. The hex number 16 corresponds to decimal 22 which indicates the content type (handshake), and the content type can also be ChangeCipherSpec (20), Alter (21), and ApplicationData (23). The second 2 bytes (03 01) are the protocol version (TLS 1.0), and the protocol version can also be SSL 3.0 (0x0300), TLS 1.1 (0x0302), and TLS 1.2 (0x0303). The last 2 bytes (01 2e) is the length of the handshake layer (302 bytes). The corresponding layer will be different for various packets. As shown in figure 2.3, the Client Hello and Server Hello packet has a same general structure and both of them has a handshake layer formed by type, length, version, random, session id length, session id, cipher suit, compression, and extensions fields. The only different between them is that there can be multiple CipherSuites and Compression methods in Client Hello, while Server Hello can only have one CipherSuite and one Compression method. The Certificate protocol can be divided into type, length, length of certificate field, and certificate. Similarly, ServerKeyExchange and ClientKeyExchange protocol has type, length, and Algorithm parameters (such as the length of key and key itself). The ServerHelloDone packet is a special one which only has type and length and length will
equal to 0 all the time. Besides, Change Cipher Spec layer has a structure formed by type, length, TLS/SSL version and Change Cipher Message and its length & change cipher message will always be 1. At last, the application data is the only packet used to translate the information which is defined content type, length, and data fields.

### 2.1.2 Message Queuing Telemetry Transport

Besides the TLS/SSL protocol, this thesis paper also involved in another network protocol—The Message Queuing Telemetry Transport [49]. Message Queuing Telemetry Transport (MQTT) is a lightweight machine to machine protocol which is usually used in the Internet of Things (IoT). Due to its lightweight characteristic, MQTT always used in small code footprint or limited bandwidth network. This protocol transmits messages through publishing
and subscribing between remote devices, so the MQTT network can be divided into three sections, which are the publisher, the broker, and the subscriber as shown in figure 2.4 according to the publish and subscribe principle. The publisher in MQTT can be viewed as a server which takes charge of publishing message and topic control. The broker just like a courier who receives messages from publishers and delivery it to receivers. It also has other functions such as elementary identifying the topic words, save the will message, etc. And for subscribers, it just signs up a specific topic and receives messages of this topic from the broker.

Here, this paper will introduce how MQTT messages are translated and packets are formatted. Compared with the TLS/SSL protocol, the handshake of MQTT is much easier. To establish a connection, a client will send a connect command to broker and it will send back a connect acknowledgment. Then, the client will send a subscribe request of a specific topic and receive the acknowledgment from the broker. When a client tries to publish a message, the broker will analysis the publish message from the publisher and deliver this message to all clients who subscribe to this topic. This publishing process shows in figure 2.5a. Under different quality of services (QoS), the publishing process will be different. As shown in figure 2.5b when the QoS of clients is set as 1, there will be a Publish Acknowledgment as feedback after receiving publish messages. As QoS is set as 2, the publish process become more complicated. After clients or brokers receive messages, the receiver side will send a publish receive back to sender sides. Once the message transmission
finishes, the receive side will send out the publish release and the receive side will send out a publish complete to prove that the publishing process completes.

For MQTT packets, the packet of MQTT can be viewed as figure 2.6 in general. The fixed header can be divided into control byte and packet length and control byte can be further divided into control packet type and flags specific. For MQTT version 3.1.1 [49], there are 14 different kinds of control packets which are Connect, Connect Ack, Publish, Publish Ack, Publish Receive, Publish Release, Publish Complete, Subscribe, Subscribe Ack, Unsubscribe, Unsubscribe Ack, ping request, and ping response. For flag bits in MQTT version 3.1.1, it mainly used in publish packets to tell receiver side the duplicate delivery, the quality of service, and Retain. Besides, the variable header change depending on packet type which will be elaborated later. The minimum MQTT packet only contains a fixed header, such as Disconnect, Ping request, and Ping response.

Among these control packets, the Connect packet play an import role in MQTT communication and this paper will introduce it in details. The simplest process to of Connect packet is shown in figure 2.7a. It is formed by five fields which are header flag, message length, protocol name and version, connect flag, keep alive time, and client ID. The header flag corresponds to the fixed header in general structure which contains the type (0001 for
connect) and reserved flag bits (flag bits is not active in connect so the flag bits is 0000). The protocol name & version and keep alive time is the variable header. The protocol name & version is formed by protocol name length, protocol name, and protocol version. For instance, the protocol name and version of MQTT v3.1.1 in Mosquitto [27] are MQIsdp (MQ Integrator SCADA Device Protocol) and MQTT v3.1. The connect flag is a byte to specify the behavior of connection and decide whether fields will exist in payloads. This 8 bits represent user name, password, will-retain, will-qos (2 bits), will-flag, clean-session, and reserved bit. The clean-session bit decides how broker handle the Session state and the details can be viewed in MQTT standard [49]. Besides, the user name, password, will-topic, and will-flag will decide whether user name/ password/ will-topic/ will-message field will present or not. Here, this paper is focusing on will-message which is unique field in MQTT protocol. The will message just like the normal one which has its topic and message, but will message does not directly send to the receiver side. It will save by broker and does not send to receiver side until the sender side disconnect unexpectedly.

The connect acknowledgment and subscribe acknowledgment have a similar structure which can be seen in figure 2.7b: both of them have header flags, message length, and reserved fields. The connect acknowledgment has a return code to indicate connection accepted or refuse, while the subscribe acknowledgment has a return QoS to indicate subscription success/ fail and QoS value. For different QoS, there will be some special packets, such as publish ack, publish receive, publish release, and publish complete. All of them have a same structure formed by header flag, message length, and message identifier.

The subscribe request and publish packet form the core function of MQTT protocol. As shown in figure 2.7c, subscribe request is formed by header flag, message length, message identifier, topic length, topic, and request QoS. On the other hand, the publish packet shown in figure 2.7d formed by header flag, message length, message identifier, topic length, topic, message length, and message.
a. Connect Command

<table>
<thead>
<tr>
<th>Header flag</th>
<th>Message Length</th>
<th>Protocol Name and Version</th>
<th>Connect Flag</th>
<th>Keep Alive time</th>
<th>Client ID</th>
</tr>
</thead>
</table>

b. Connect/subscribe Ack

<table>
<thead>
<tr>
<th>Header flag</th>
<th>Message Length</th>
<th>Reserve</th>
<th>Return Code / Request QoS</th>
</tr>
</thead>
</table>

c. Subscribe Request

<table>
<thead>
<tr>
<th>Header flag</th>
<th>Message Length</th>
<th>Message Identifier</th>
<th>Topic Length</th>
<th>Topic</th>
<th>Request QoS</th>
</tr>
</thead>
</table>

d. Publish Message

Figure 2.7: Various packets of MQTT protocol
Chapter 3: TOSS: Tailoring Online Server Systems through Binary Feature Customization

In this chapter, we proposed TOSS [8], our solution to reduce the potential attack surface and debloated framework. Besides, we use an implementation of MQTT, Mosquitto, to evaluate this framework.

3.1 Motivation

As the program size increases, the amount of vulnerabilities increases significantly. Feature customization can be a suitable method to solve this problem, so we proposed this framework to minify program and the customized program will only contain just-enough features. At the same time, desired features are still available in the customized program. The main challenge of TOSS is that there is only binary code available for online server and software system. Besides, binary reconstruction or binary reuse techniques will produce new binaries that can only process the limited scope of inputs. In our framework, we first identify features via dynamic tracing & symbolic executions and rewrite the binary to create a new self-contained binary program.

Before this paper introduced the framework, features have to be defined first. Given a set of basic blocks $f^n_i$, binary program features can be represented as $F_i = \{f^1_i, f^2_i, f^3_i, ..., f^n_i\}$. The set of all program features is denoted by $F = F^1, F^2, ..., F^m$, and the customized program contains the feature set $A$ which is a subset of $F$.

3.2 System Design

As shown in figure [5.1], TOSS is formed by two models which are feature identification and feature rewriting. Before binary customization, the user’s requirement should be given first, such as lists of required features which can be provided by test-cases. With binary program
and user’s demand, TOSS can generate a light-weight customized binary which only formed by desired features.

### 3.2.1 Feature Identification

Feature identification module will execute the original program inside a whole system emulator, while the feature-related instructions are tainted according to the target packet or certain packet fields. The tainting information is then utilized to guide the program binary symbolic execution as tainting-based symbolic execution (TSE) to discover more code related to the target feature but not executed during the program run. The intuition here is that due to the limited scope of test inputs, there are branches in the binary that will never be taken. However, future inputs could possibly activate those branches. Hence, we cannot remove such branches from the program binary for better input coverage. After this, the discovered runtime instructions will be located in the static program binary and rewriting will be performed. At last, we verify the effectiveness of the customized binary by different sets of inputs.
3.2.2 Feature Customization

After the desired basic blocks are identified, we want to keep them in the static program and remove other basic blocks. The feature customization module will perform the modification to the original binary. First, we convert the addresses of identified runtime instructions to static addresses in the binary then mark those instructions as desired instructions. Then, we remove unmarked instructions in the binary.

3.3 Implementation

In this section, we will introduce the tool used in this prototype of TOSS. For tracing and tainting, we used finer-grained tainting through tracecap plugin and setting the taint bitmap in TEMU [47]. For symbolic execution, we used Angr [46] to execute symbolic execution. Moreover, the PatchAPI in Dyninst [48] is adapted to customize binary programs. The PatchAPI first analyze the program and generate corresponding CFG which has been discussed in section 4.2.1. Then, it achieves customization by removing undesired basic blocks in CFGs and replacing them with NOPs, while original basic blocks will be redirected to exit points.

3.4 Evaluation

**Experiment Setup:** The evaluation of TOSS is on 2.80GHz Intel Xeon(R) CPU E5-2680 20-core server with 16 GB of main memory and the operating system is Ubuntu 14.04 LTS. The testing software of the MQTT client & broker is Mosquitto [28].

Besides, we also craft a Computation Offloading protocol for testing. In a computation offloading communication, a client will send parameter and operation type to server, and sever will compute and send back the result to client. The packet from client contains a header to specify the request type and packet length, and the payload to indicate the parameter of operators and corresponding operands.
Table 3.1: Number of instructions discovered by tracing and Tainting guided symbolic execution on Mosquitto Publisher and Computation Offloading Server

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Mosquito_pub</th>
<th>CO_server</th>
</tr>
</thead>
<tbody>
<tr>
<td>original binary</td>
<td>3305</td>
<td>509</td>
</tr>
<tr>
<td>from tracing</td>
<td>1124</td>
<td>332</td>
</tr>
<tr>
<td>from TSE</td>
<td>188</td>
<td>37</td>
</tr>
<tr>
<td>customized binary</td>
<td>1312</td>
<td>369</td>
</tr>
<tr>
<td>Features removed</td>
<td>Insecure, Publish file, Will, etc</td>
<td>Computation except addition</td>
</tr>
</tbody>
</table>

In Mosquitto, we only keep the basic features to maintain communication between two clients (Subscriber and Publisher). In particular, the hostname, port number, topic, and message are set as necessary features for publishers, while the hostname, port number, and topic as necessary features for subscribers. In mosquito, some security related features can be removed after customization, such as insecure, will message which has been discussed in section 2.1.2, and publish files. Figure 3.1 shows the preliminary experiment results of the customization. The customized 1312 instructions of the publisher can achieve the basic publish-subscribe communication, while the original program has 3305 instructions.

In computation offloading server, we only keep the addition operation. As we can see in figure 2.1.2 the customize computation offload server has 369 instructions, while the original binary has 509 instructions.

3.5 Discussion

Our framework TOSS only has a forward tainting model so that it cannot deal with backward tainting. For instance, given instructions which process not only inbound packets but also outbound packets, the instructions of outbound packets will not be tainted. Thus, the backward tainting is needed for TOSS to increase the tainting capability. Besides, because the binary rewriting model is basing on static binary and corresponding instruction addresses, the obfuscated binary cannot be rewritten.
Chapter 4: Machine Learning based Binary Code Analysis: A Comprehensive Study

Binary code analysis is a technique enables engineers to directly access the binary program without the source code, and it has been widely adopted during past decades. However, since the large scale of binary code and difficulty for the human to read or modify, Machine learning has been introduced into the binary code analysis field. In 1996, Tesauro et al. [51] first introduced a neural network based method on virus recognition which is the first research in Machine Learning based binary code analysis area. In past a few years, machine learning based binary code analysis has been success in various areas, such as Malware detection [36, 26, 44, 42], authorship recognition [39, 5], function recognize [3, 54, 40], code clone detection [14, 58, 41], Recovering Toolchain Provenance [37], etc. In this chapter, we give a study on machine learning based binary code analysis.

4.1 The General Framework

To combine binary code analysis and Machine learning, a general framework of binary code analysis basing on machine learning has been introduced. As shown in figure 4.1, the whole framework can be divided into two parts: the training phase and analysis phase. In training phase, each data in the training set will be analyzed and features are extracted from data. Then, features will be embedded into numeric vectors which can be identified by the machine learning algorithm. During training, the machine learning model will learn these features vectors and construct a machine learning model for further analysis. Besides, binary targets are analyzed in the same way and feature vectors corresponding to each target is extracted & embedded in the analysis phase. Given the machine learning model and target feature vectors, the machine learning model can identify such target feature vectors to achieve certain analyzed goals.
4.2 Feature Extraction

In this section, we list different features extracted from binary file. In general, features can be grouped into graph-based and code-based.

4.2.1 Graph-based Features

**Control Flow Graph-based Features** The Control Flow Graph (CFG) [2] is a graph used to represent all possible paths of programs via nodes and edges. In binary code, nodes correspond to a sequence of consecutively executing instructions, and edges represent control flow transfers between nodes (such as return, jump, and calls). A binary level control flow directly reflects the relationship between instructions and control flow transfers, so binary CFG is a useful method to reflect features between basic blocks. For instance, discovRE [14] used CFG to reflect structural similarity with basic block distance, while Caliskan et al. [5] extracted flow feature via disassembly through CFG.

To reflect the code behavior at instruction-level, Pržulj et al. [34] introduced Graphlets, a small, non-isomorphic subgraphs of the CFG. Graphlets was introduced in this area by...
Rosenblum et al. [37] to represent features in small structure (as additional idiom features which will be discuss later). Besides, Rosenblu et al. [39] adopted supergraphlets and call graphlets (graphlet-like features) to reflect long-range program structure and inter-procedural control flow & libraries communication.

Table 4.1: List of Attributes used in Attributed Control Flow Graph (ACFG).

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block-level attributes</td>
<td>String Constants</td>
</tr>
<tr>
<td></td>
<td>Numeric Constants</td>
</tr>
<tr>
<td></td>
<td># of Arithmetic Instructions</td>
</tr>
<tr>
<td></td>
<td># of Calls</td>
</tr>
<tr>
<td></td>
<td># of Transfer Instructions</td>
</tr>
<tr>
<td></td>
<td># of Instructions</td>
</tr>
<tr>
<td>Inter-block attributes</td>
<td># of Offsprings</td>
</tr>
<tr>
<td></td>
<td>Betweenness</td>
</tr>
</tbody>
</table>

Besides control flow graph and its subgraph, there is a similar graph called as Attributed Control Flow Graph (ACFG) used to represent a static binary function in a different basic-block level. In an ACFG, each node will be represented as an attributes-vertex, and the attributes are list in table 4.1. For instance, Genius [15] took ACFG to calculate the similarity between binary functions, while Genius [57] also adopts ACFG for embedding which will be discussed later.

**Abstract Syntax Tree** Abstract Syntax Tree (AST) is a tree-like graph used to reflect dependencies between tokens at the syntax level. This tree graph has been widely used in source code analysis at syntax level, but it can also but in binary code. For instance, FID [54] first translated each basic block into the corresponding assignment formula as the code semantics, and each assignment formula could be further parsed into AST.
4.2.2 Code based Features

**Tokens**  Token of binary file can be good features because they can be easily extracted from the original binary files. For instance, Katz et al. [23] tokenized binary code sequences into several tokens basing on the appropriate level of granularity as features for binary code for de-compilation. Saebjørnsen et al. [41], Xue et al. [58,59] and Byteweight [3] also adopted the similar method which tokenized binary code sequences into small pieces.

**N-Gram**  N-gram method extracts contiguous sequences from raw bytes. In binary code analysis, Shijo et al. [44] adopted Cuckoo Malware analyzer to generate API call sequence and used n-gram based method to gain n-gram sequences. Rosenblum et al. [39] also introduced n-grams to extract instructions and memory exchanging for authorship identification. Besides, Liangboonprakong et al. [26] proposed n-gram to directly extract sequences from the binary for Malware detection.

**Portable Executable (PE) Feature**  Portable Executable (PE) format is a kind of file format in Windows operating system. In a PE binary file, the PE header contains important information, such as the dynamic libraries, API exported, etc, and thus it can be featured to represent the binary file. For instance, Schult et al. [43] extracted the file size, the name of DLLs, the names of function within a DLL, and relocation table via libBFD to detect new malicious executables. Saxe et al. [42] also extracted the address table into a 256-integrate-array and numerical PE file into a 256-length-array for Malware detection.

**Printable String**  Printable Strings are un-encode strings after compilation and it can be a feature for binary files. For instance, Schultz et al. [43] extracted printable string which are consecutive printable characters from PE header via GUN strings program for Malware detection. Moreover, Shijo et al. [44] proposed printable string information (PSI) as the static feature for Malware detection.


**Feature Entry Points Features**  For a binary function in a binary program, there will be one entry point and multiply exit points, and the entry point of the function is named as Feature Entry Points (FEP). FEP can be a unique features to represent a function. In binary code analysis area, Rosenblum et al. [37, 38, 40] adopted FEP features in the form of entry idioms which will be discussed in paragraph 4.2.2.

**Entropy Histogram Feature**  To deal with binary files without PE format, Saxe et al. [42] creates a 1024 byte window sliding 256 bytes each time and a two-dimensional histogram will be computed over the pair list.

**Idiom feature**  To represent the order of instructions, idiom features are introduced into feature extraction. In general, there are two types of idiom features which are the prefix idiom and the entry idiom. An entry idiom reflects the instruction and its offset immediately (equation 4.1), while an prefix idiom place the offset at begin (equation 4.2). In binary code analysis, Rosenblum et al. [37, 38, 39, 40] all introduced idiom as one of their features.

\[ u_1 = ( push \ ebp \ | \ * \ | mov \ esp, \ ebp ) \]  

(4.1)

where \( \ast \) can be any instruction.

\[ u_2 = ( PRE : ret | int3 ) \]  

(4.2)

**The Malware Instruction Set (MIST)**  Rieck et al. [36] used the Malware Instruction Set (MIST) [52] to search numeric identifiers representing system call and arguments for
malware detection. Equation 4.3 shows the structure of a MIST instruction.

\[
\text{CATEGORY OPERATION}|\text{ARGBLOCK1}|\text{ARGBLOCK2}|...|\text{ARGBLOCKN}
\]  

(4.3)

\[
\begin{array}{c}
\text{level 1} \\
\text{level 2} \\
\text{level 3}
\end{array}
\]

4.3 Feature Embedding

The raw feature extracted from binary files can not use for training directly, and have to embedding into numeric vectors.

4.3.1 Graph embedding

A control flow graph can describe the flow of data but it cannot be read as input for a machine learning model, so graph embedding is adapted for graph based features encoded them into an embedding. discovRE et al. [14] introduces a concept called Basic Block Distance to calculate structural similarity. Besides, Genius [15] embed raw features through calculating the similarity of two ACFGs.

4.3.2 Code Embedding

On the other hand, code based features have its code embedding.

**Instruction Q-grams Embedding**  Rieck et al. [36] proposed a unique method which named as Q-grams embedding. The instruction embeds through a window which will slide over the sequence and split it to a q-length sequence.

**Byte Embedding**  During embedding, tokens has to be translated into vectors for learning. For instance, Shin et al. [45] introduced a method called "one-hot encoding" which will encode bytes into \( \mathbb{R}^{256} \) vectors.
Table 4.2: Supervised Learning in Binary Code Analysis.

<table>
<thead>
<tr>
<th>Supervised Learning</th>
<th>Title</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support Vector Machine</td>
<td>Liangboonprakong et al. [26]; Shijo et al. [44]; Hosfelt et al. [21]; Rosenblum et al. [37]</td>
<td>Classify String Tokens; Classify Malware and Benign; Classify Cryptographic Algorithms; Create Model for Toolchain Provenance</td>
</tr>
<tr>
<td>Bayes Classifier</td>
<td>Hosfelt et al. [21]; Schultz et al. [43]</td>
<td>Classify Cryptographic Algorithms; Detect Malware</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>Hosfelt et al. [21]; Liangboonprakong et al. [26]</td>
<td>Classify Cryptographic Algorithms; Classify Malware</td>
</tr>
<tr>
<td>Random Forest</td>
<td>Caliskan et al. [5]; Shijo et al. [44]</td>
<td>Learn De-anonymizing Pattern; Classify Malware</td>
</tr>
<tr>
<td>Boosting</td>
<td>FID [54]</td>
<td>Recognize Function in Binary Code</td>
</tr>
</tbody>
</table>

**Frequent Embedding** The frequency of tokens varies, so we can represent a token with a unique tag according to frequency. Katz et al. [23] translated sequences of tokens into a sequence of integers through popularity ranking. Additionally, Liangboonprakong et al. [26] proposed Sequential Pattern Extraction [1] and Pattern Statistic to find patterns in sequence.

### 4.4 Machine Learning Algorithm

After discussed feature extraction and embedding, we will discuss machine learning techniques.

#### 4.4.1 Supervised Learning

**Support Vector Machine Learning** Support Vector Machine (SVM) is a supervised learning used as a classification generator. Given inputs of a set of vector, there will be a weight vector generated during training. The weight vector decides the boundary of different classes in the form of margin which is defined as the kernel function. In binary code analysis area, many previous research adopted SVM as their learning model, such as radial-basis kernel [21], linear kernel [21, 37, 39], polynomial kernel [21, 6], and sigmoid kernel [21].
**K-Nearest Neighbours** The Nearest Neighbours algorithm is implemented by saving each training observation and make predictions by finding a similar training observation. discovRE [14] employed k-NN based on k-d trees which is a binary search tree whose nodes are k-dimensional points for vulnerabilities detection.

**Bayes Classier** Bayes model classifies the group by computing the probability of a certain feature belonging to a certain class. In binary code analysis, Muti-Naive Bayes [43], Native Bayes [43], and Gaussian Bayes [43] has been adopted.

**Decision Tree** The decision tree is a kind of supervised learning which does not use any parameter, but some decision rules. A decision tree is formed by three sections which are nodes, arcs, and leaves to reflect feature attribution, feature value, and category. In binary code analysis, Liangboonprakong et al. [26] adopted C4.5 decision tree [35] as one of classification models.

**Random Forest** Random Forest is a classifier with multi-decision trees which is a kind of implementation of the decision tree. Caliskan et al. [5] randomly selected $(logM) + 1$ features from $M$ total features and each of $(logM) + 1$ features will be inputted to a tree for de-anonymizing, and Shijo et al. [44] adopted random forest as one of classifiers for Malware detection.

**Boosting** Boosting is an additional learning model to strengthen the accuracy of main machine learning. For instance, FID [54] combined linear SVM, AdaBoost, and Gradient-Boosting for function identification.

**4.4.2 Unsupervised Learning**

**Clustering** Clustering is unsupervised learning. It first splits training samples into several clusters during the training stage and identifies test samples through calculating the
Table 4.3: Unsupervised Learning in Binary Code Analysis.

<table>
<thead>
<tr>
<th>Unsupervised Learning</th>
<th>Title</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering</td>
<td>K-means</td>
<td>Hosfelt et al. [21], Rosenblum et al. [39]</td>
</tr>
<tr>
<td></td>
<td>Affinity Propagation</td>
<td>Clone-Hunter [58]</td>
</tr>
<tr>
<td></td>
<td>Hierarchical Clustering</td>
<td>Rieck et al. [36]</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>Genius [15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identify Instruction Features;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classify Authorship</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Detect Code Clones in Binaries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classify Behavior</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generate Cluster of ACFGs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identify Malicious Executables Features</td>
</tr>
<tr>
<td></td>
<td>Weighted Prefix Tree</td>
<td>Byteweight [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identify Function Entry Points in Binary Program</td>
</tr>
</tbody>
</table>

Table 4.4: Deep Learning in Binary Code Analysis.

<table>
<thead>
<tr>
<th>Deep Learning</th>
<th>Title</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recurrent neural network</td>
<td>Clone-Slicer [59]; Shin et al. [45]</td>
<td>Detect Code Clone;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Recognize Function in Binary Program</td>
</tr>
<tr>
<td>Deep Neural Network</td>
<td>Xu et al. [57]</td>
<td>Embed ACFGs</td>
</tr>
<tr>
<td>Multiple Layer Perceptron</td>
<td>Liangboonprakong et al. [26]</td>
<td>Classify Malware Families</td>
</tr>
<tr>
<td>One-Sided Perceptron</td>
<td>Gavriliuț et al. [18]</td>
<td>Detect Vulnerabilities</td>
</tr>
</tbody>
</table>

distant (pattern similarity) between test samples and clusters. Some previous works adopted clustering as machine learning algorithm in binary code analysis, such as K-means Clustering [21, 39], Hierarchical Clustering [36], Affinity Propagation Clustering [58], Spectral Clustering [15] proposed by Ng et al. [32].

**Rule-based**  Schultz et al. [43] employed a special rule based learning system [11] which is a set-valued extension of the original RIPPER [10] to detect Malware.

**Weighted Prefix Tree**  The prefix tree can reflect the assemble code clearly and it has strong capability on retrieval. With up to $l$ instructions, a prefix tree is generated in Byteweight [3]. To avoid the confusion of similarity instructions, immediate number normalization and call & jump instruction normalization are adopted. With many prefix trees from the training set, a weight prefix tree can be generated to elaborate the likelihood for each node in the prefix tree.
4.4.3 Deep Learning

**Recurrent Neural Network** Recurrent Neural Network (RNN) [20] is a type of Artificial Neural Network with effective long short-memory. Katz et al. [23] proposed the encoder-decoder RNN using seq2seq model for decompilation, while Clone-Slicer [59] took RNN to get embedding vector of code tokens at the lexical level for code clone detection.

**Deep Neural Network** Deep Neural Network (DNN) is a complex non-linear ANN with multiple hidden layers. In binary analysis code, Genius [57] adopted DNN to embed ACFGs into vectors for binary code similarity detection, and Saxe et al. [42] used DNN as a classifier to detect Malware.

**Perceptron** Multiple Layer Perceptron is a feed-forward ANN. In binary code analysis, Shin et al. [45] adopted multi-layer perceptron (MLP), and Liangboonprakong et al. [26] used MLP as one of their classification models. Besides, Gavriluț et al. [18] proposed one-sided perceptrons to detect Malware.

4.5 Application

4.5.1 Code Clone detection

Code Clone detection is to detect the similarity between codes and is always used for vulnerabilities or Malware detection. Table 4.5 lists a few previous research for code clone detection. For features, David et al. [13], discovRE. [14], and Gemini [57] introduced graph based features, while the other featured code based features. For machine learning algorithm, Saebjørnsen et al. [41], David et al. [13], and Clone Hunter [58] adopted Clustering to split code samples into clusters and identify further code clone, while the other used various neural network for code clone detection except Esh [12] (Logistic Function).
### Table 4.5: Binary Code Clone Detection

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Features</th>
<th>ML Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saebjornsen et al. [41]</td>
<td>2009</td>
<td>Tokens</td>
<td>Clustering</td>
</tr>
<tr>
<td>David et al. [13]</td>
<td>2014</td>
<td>CFG &amp; Execution trace</td>
<td>Clustering</td>
</tr>
<tr>
<td>Esh [12]</td>
<td>2016</td>
<td>Instructions</td>
<td>Logistic function</td>
</tr>
<tr>
<td>discovRE. [14]</td>
<td>2016</td>
<td>CFG</td>
<td>K-NN</td>
</tr>
<tr>
<td>Gemini [57]</td>
<td>2017</td>
<td>ACFG</td>
<td>Neural Network</td>
</tr>
<tr>
<td>Clone Hunter [58]</td>
<td>2018</td>
<td>Tokens</td>
<td>Clustering</td>
</tr>
<tr>
<td>Clone-Slicer [59]</td>
<td>2018</td>
<td>Tokens</td>
<td>RNN</td>
</tr>
</tbody>
</table>

### Table 4.6: Function Recognition

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Features</th>
<th>ML Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rosenblum et al. [38]</td>
<td>2007</td>
<td>FEP based and Idiom features</td>
<td>MRF model interface</td>
</tr>
<tr>
<td>Rosenblum et al. [40]</td>
<td>2008</td>
<td>FEP based and Idiom features</td>
<td>CRF model interface</td>
</tr>
<tr>
<td>Byteweight [3]</td>
<td>2014</td>
<td>CFG</td>
<td>Weight Prefix Tree</td>
</tr>
<tr>
<td>Shin et al. [45]</td>
<td>2015</td>
<td>Byte Sequences</td>
<td>RNN</td>
</tr>
<tr>
<td>FID [54]</td>
<td>2017</td>
<td>Lexical, Syntactic and Stack</td>
<td>Linear SVC, AdaBoost and GradientBoosting</td>
</tr>
</tbody>
</table>

### 4.5.2 Function Recognition

In binary code, the boundary of functions is hard to identify and this task is defined as function recognition. Table 4.6 lists some previous works for the purpose of function recognition. Both Rosenblum et al. [38, 40] adopted FEP based features and idiom features with MRF(CRF) model interface. Byteweight [3] used CFG to represent functions and generated learning model through Weight Prefix Tree, while Shin et al. [45] extracted byte sequence from binary files and construct identifier via RNN. Besides, FID [54] generated learning model via Linear SVC, AdaBoost and GradientBoosting.
Table 4.7: Malware detection.

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Features</th>
<th>ML Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schultz et al. [43]</td>
<td>2001</td>
<td>PE features and Byte sequence</td>
<td>RIPPER, Native Bayes, and Multi-Native Bayes</td>
</tr>
<tr>
<td>Gavriluț et al. [18]</td>
<td>2009</td>
<td>N/A</td>
<td>One-Sided Perceptron</td>
</tr>
<tr>
<td>Rieck et al. [36]</td>
<td>2011</td>
<td>MIST</td>
<td>Hierarchical Clustering and Nearest prototype classification</td>
</tr>
<tr>
<td>Liangboonprakong et al. [26]</td>
<td>2013</td>
<td>String Pattern</td>
<td>C4.5, Multilayer perceptron, and SVM</td>
</tr>
<tr>
<td>Saxe et al. [42]</td>
<td>2015</td>
<td>Byte Histogram Feature and PE Features</td>
<td>DNN</td>
</tr>
<tr>
<td>Shijo et al. [44]</td>
<td>2015</td>
<td>PSI and API-calls</td>
<td>SVM and Random Forest</td>
</tr>
<tr>
<td>Zak et al. [62]</td>
<td>2017</td>
<td>Sectional Byte N-grams and Assembly N-gram features</td>
<td>Elastic-Net Regularized Logistic Regression, and Stacking</td>
</tr>
<tr>
<td>RMVC [50]</td>
<td>2018</td>
<td>Opcodes</td>
<td>RNN and CNN</td>
</tr>
<tr>
<td>Liu et al. [29]</td>
<td>2019</td>
<td>Image Features</td>
<td>MLP, KNN, and Random Forest</td>
</tr>
</tbody>
</table>

4.5.3 Malware Detection

To reduce the threat of Malare, the Malware detection in binary level becomes a method to deal with it. Table 4.7 shows the contrast of Malware detection. Liu et al. [29] took images feature as features, while the other all adopted code based features (PE features, Tokens, MIST, Printable Strings, N-grams sequences, opcode). For learning models, Liu et al. [29], RMVC [50], Saxe et al. [42], and Gavriluț et al. [18] used deep learning techniques, while the other employed supervised learning algorithms.

4.5.4 Vulnerability Discovery

Similar to Malware detection, the vulnerability discovery aims to find the bugs itself. Table 4.8 lists seven works to detect vulnerabilities. For features, Padmanabhuini et al. [33] and Genius [15] took graph based features, while the other adopted code based features. For machine learning model, Yamaguchi Fabian et al. [61] and Genius [15] employed
Table 4.8: Vulnerabilities discovery

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Features</th>
<th>ML Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padmanabhuni et al. [33]</td>
<td>2015</td>
<td>CFG-based Features</td>
<td>Native Bayes, MLP, Simple Logistic, and Sequential Minimum Optimization</td>
</tr>
<tr>
<td>Yamaguchi Fabian et al. [61]</td>
<td>2015</td>
<td>CFG-based Features</td>
<td>Clustering</td>
</tr>
<tr>
<td>VDISCOVER [19]</td>
<td>2016</td>
<td>Dynamic and static features</td>
<td>logistic regression, MLP of single hidden layer and random forest</td>
</tr>
<tr>
<td>NeuFuzz [55]</td>
<td>2019</td>
<td>Binary Programs</td>
<td>DNN</td>
</tr>
<tr>
<td>Change et al. [9]</td>
<td>2019</td>
<td>Execution Paths</td>
<td>RNN</td>
</tr>
<tr>
<td>PATCHDETECTOR [16]</td>
<td>2019</td>
<td>Tokens</td>
<td>Deep feed-forward Neural Network</td>
</tr>
</tbody>
</table>

Table 4.9: Authorship Recognition.

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Features</th>
<th>ML Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rosenblum et al. [39]</td>
<td>2011</td>
<td>Idiom and graphlets features</td>
<td>SVM and LMNN</td>
</tr>
<tr>
<td>Caliskan et al. [5]</td>
<td>2015</td>
<td>Instruction feature</td>
<td>Random Forest</td>
</tr>
<tr>
<td>Meng et al. [30]</td>
<td>2017</td>
<td>Instruction, data flow, and context features</td>
<td>CRF and SVM</td>
</tr>
</tbody>
</table>

clustering to split vulnerabilities classes, while the other all used deep learning techniques. Besides, Padmanabhuni et al. [33] and VDISCOVER [19] also adopted supervised learning algorithms except using deep learning.

4.5.5 Authorship Recognition

To identify the author of a certain program by coding style, the authorship recognition becomes a task. Table 4.9 lists a few research about authorship recognition. Rosenblum et al. [39] learned idiom and graphlets features via SVM and LMNN. Caliskan et al. [5] constructed learning model via Random Forest. Besides, Meng et al. [30] used instruction, data flow, and context features to represent samples and learn features through CRF and SVM.
Table 4.10: Other Types of Applications.

<table>
<thead>
<tr>
<th>Title</th>
<th>Features</th>
<th>ML Model</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosfelt et al. [21]</td>
<td>Instruction, Category feature</td>
<td>SVM, Native Bayes model, Decision Tree and K-means Clustering</td>
<td>Cryptographic algorithm classification</td>
</tr>
<tr>
<td>Katz et al. [23]</td>
<td>String</td>
<td>RNN</td>
<td>Decompilation</td>
</tr>
<tr>
<td>Rosenblum et al. [37]</td>
<td>FEP based features</td>
<td>SVM</td>
<td>Recovering Toolchain Provenance</td>
</tr>
</tbody>
</table>

4.5.6 Other

This paper also introduces three other types of application. Hosfelt et al. [21] generated a classifier to identify various cryptographic algorithm, and Katz et al. [23] constructed a decompilation tool via learning binary and source code string. Besides, Rosenblum et al. [37] generated a framework to recover toolchain provenance.

4.6 Discussion

As discussed in section 4.4, we can find that deep learning algorithm become a common machine learning algorithm in Malware detection and vulnerabilities detection. Besides, there are also many researches on other binary code analysis area (Code Clone [57 59], function recognition [45], and decompilation [23]). Future work will rely more on deep learning systems for sophisticated analysis system. Moreover, statistical and formal learning method Clone-Slicer [59] and Clone-hunter [58] proposed by which could be a future binary code analysis direction.
Chapter 5: Network Protocol Packet Generation and Analysis Identification via
domain Specific Slicing

To identify source code functions, we propose a framework to automatically identify functions of packet generation and analysis through a domain-specific slicing.

5.1 Motivation

The network based protocol is the cornerstone of communication, and each information/packet follows the structure of a certain protocol. With the increasing demand for communication security, a growing number of vulnerabilities has been found in security protocol implementations. Besides, IoT devices grew dramatically in recent years and the security of IoT communication also became a serious problem. To deal with security problems of protocol implementations, we proposed a tool to automatically identify packet related function in source code through domain-specific slicing.

5.2 Framework

This function identification tool includes four modules which are pointer specific slicing, path generation, vector embedding and classification as shown in figure 5.1. Given the source code of a network based protocol implementation, our framework slice the source code and generate a CFG with marked sliced pointer specific statements in the slicing stage. Then, merging paths will be extracted from CFGs via random walk and path merging. After vector embedding, we will train the classification model through merging paths.

5.2.1 Program Slicing

Functions of packet generation usually use pointers to translate the content of fields to packet memories, while functions of packet analysis use pointer to pull field contents of out from
packet memory. Thus, it is easy to find that pointer is an import feature of packet generation and analysis. Here, we use Joern [60] to achieve pointer specify program slicing. Joern [60] is a graph based C/C++ code analysis platform. It will first analyze a program and phased into a CFG. In our case, Joern will take functions & pointer string as input and output a CFG with marked nodes of a sliced program.

5.2.2 Path Generation

With the slicing result from the previous phase, our framework will analysis the sliced CFG and generate paths.

**Random Walk**   The number of paths in a function is huge, so the size of path sets have to be limited. Thus, we employ Random Walk to extract limited-number paths. To make these limited-number paths representing their corresponding functions, two thresholds are applied: the number of statements and the minimum pointer related statements. The longest path have the most statements, while pointer related statements can represent packet related function effectively.

**Path Merging**   In packet related functions, there could be many statements calling other functions which could also involve in packet generation or analysis, so these called function should also be included in the path. Our framework used path merging to deal with this problem. For instance, given a packet generation function $A$, there could be function $B$ called by $A$ through a pointer, such as value assign, value transmit, and address transmit. If callee and called functions are directly merged together, same variable & pointer & structure may
5.2.3 Embedding

After we get merging paths via random walk and path merging, embedding should be used to transfer these paths into numeric vectors. In our framework, code signature will generate in lexical level and syntax level.

**Lexical Level** To capture code signature at the lexical level, our framework adopts RNNLM [31] with 100 hidden layers to embed merging paths. During RNN embedding, the input will be a set of one-hot vector to represent the current path. Given statements of a merging path, RNN embedding can output a probability distribution predicting the next term statements. Take figure 5.2 as an example. There is a merging path $P_1$ shown in
A. Statements

B. Recursive Auto-encoder

Figure 5.3: An illustration of RAE

Figure 5.3A and one of pointer related statement is ... \( * (p++) = 1 \). With hidden layer \( S_{t-2} - S_{t+4} \) and network parameters \( V, W, U \), RNN will embed this path into a probability distribution predicting the term in next path \( P_2 \) as \( o_{t-2} - o_{t+4} \).

**Syntax Level**  For syntax level, our framework adopts Recursive Autoencoder (RAE) [25] to combine statements for embedding. Recursive Autoencoder first encode terms in a statement together, and then combine statements. For instance, Path \( P_1 \) have two statements: \( s2n(i,p) \) and \( p+ = i \). During RAE, terms in first statement \( (s2n,(,i,,p,)) \) will be encoded and other statements repeat the same process. At last, all statements in paths will be encoded and output a code signature to represent syntax feature between these two statements.
5.2.4 Learning Algorithm

Our framework adopted the polynomial kernel Support Vector Machine as the machine learning algorithm. SVM maps sample points to high dimensional space and split samples into different classes through hyperplanes.

5.3 Evaluation

Experiment Setup: The experiments are conducted on an Intel(R) Core(TM) i7-3770 CPU 3.40 GHz with 15GB system memory, and the operating system is Ubuntu 16.04 LTS.

Experiment Software: We take six implementations to evaluate this framework, which are LibreSSL, OpenSSL, WolfSSL, Mosquitto, MQTT-C, and WolfMQTT. Three of them are TLS / SSL implementations, while the others are implementations of MQTT. And the learning process is on Weka [56], a lightweight machine learning tool.

During the evaluation, LibreSSL, OpenSSL, Mosquitto, and MQTT-C are set as the training set, while WolfSSL, WolfMQTT are set as the testing set. As shown in figure 5.1, there are 1320 paths of packet generation, 1360 paths of packet analysis and 1050 paths of other functions for training. On the other hand, there are 665 paths of packet generation, 800 paths of packet analysis, and 1000 paths of other functions.

<table>
<thead>
<tr>
<th>Table 5.1: Training and Testing Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Generation</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Training Set</td>
</tr>
<tr>
<td>Testing Set</td>
</tr>
</tbody>
</table>

To measure the accuracy, we give a few definition. Given true positive (TP), true negative (TN), false positive (FP), and false negative (FN), we have:

\[
Precision = \frac{TP}{TP + FP}
\]  
(5.1)
Recall = \frac{TP}{(TP + FN)} \quad (5.2)

F - measure = 2 \cdot \frac{Precision \cdot Recall}{(Precision + Recall)} \quad (5.3)

MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (5.4)

The evaluation result is shown in figure 5.2, the true positive, precision, recall, F-measure, and MCC are high, while false positive rate is small.

Table 5.2: Preliminary Result

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
<th>MCC</th>
<th>Catogrey</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.995</td>
<td>0.000</td>
<td>1.000</td>
<td>0.995</td>
<td>0.997</td>
<td>0.996</td>
<td>Analysis</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>1.000</td>
<td>0.998</td>
<td>0.997</td>
<td>Generation</td>
</tr>
<tr>
<td>1.000</td>
<td>0.003</td>
<td>0.996</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>Other</td>
</tr>
<tr>
<td>0.998</td>
<td>0.001</td>
<td>0.998</td>
<td>0.998</td>
<td>0.998</td>
<td>0.997</td>
<td>Total</td>
</tr>
</tbody>
</table>

5.4 Discussion

As shown in figure 5.2, the general True Positive is 99.8% which means our framework can correctly classification. Because limited time, the training and testing sets are not enough and more other paths should be learned. In our work, the difference between Packet Related and Other is so huge that lead to high accuracy. We consider a bigger training set as future work to add in our framework.
Chapter 6: Conclusion

We designed TOSS, an online server binary customization framework. It can analyze the binary program and rewrite the program into a new one with just-enough features according to users’ demand. TOSS identifies the code to customize by tainting the network packets and propagate the taint through the instructions in the program binary. Then, TSE enlarges the scope of inputs the customized program can take and perform static binary rewriting to generate a customized program. Our experiment results demonstrate that TOSS is an efficient tool to generate a slim program with only necessary code segments and reduce the attack surface.

Besides, The study of machine learning binary code analysis discusses many frameworks in different aspects, such as feature extraction, feature embedding, machine learning techniques, and research target. This paper introduces the basic design principles of binary code analysis and it will help readers to have an overview of machine learning based binary code analysis.

Last, a packet generation and analysis functions identification tool is proposed and it can automatically identify packet related function via pointer specific slicing. As shown in table 5.2, the good testing performance demonstrates that our framework can be a good packet related function identifier. Known the source code packet functions, we can directly change the source code to modify the structure of protocol and make it unreadable except communication users. In the future, our work can be further adapted in automatically customized protocol structure and source code change.
Bibliography


