AI for Cybersecurity

A Handbook of Use Cases

Peng Liu, Tao Liu, Nanqing Luo, Zitong Shang,
Haizhou Wang, Zhilong Wang, Lan Zhang, and Qingtian Zou

Penn State Cyber Security Lab

Contents

Chapter 1 Introduction

1.1 Why a Handbook?

Regarding why we are motivated to write a handbook of “AI for Cybersecurity" use cases, we have three main reasons. First, *DL (Deep Learning) and RL (Reinforcement Learning) have been increasingly applied in solving cybersecurity challenges.* This trend is well reflected in the following four observations.

Observation 1. Some critical cybersecurity problem-solving processes (*e.g.*, reverse engineering) still require a large amount of manual effort from professionals/engineers. In order to reverse engineer malware code, engineers will often use many tools. Some of the most important ones are as follows. The industry has substantial incentives to use AI to reduce the amount of involved manual effort.

- Disassemblers (e.g. IDA Pro): A disassembler takes binary code as input and produces assembly code.
- Decompilers also are available for converting binary code into native code, although they are not available for all architectures.
- Debuggers (e.g. GDB): Reverse engineers use debuggers to trace the execution of a program in order to gain insights into what the program is doing. Debuggers also let one control certain aspects (*e.g.*, values in memory) of the program while it is running.
- PE Viewers (e.g. PE Explorer): PE (for Windows Portable Executable file format) viewers extract important information (*e.g.*, dependencies) from executables.

Observation 2. Quite a few important security-driven classification tasks (*e.g.*, malware classification, (session-level) traffic classification) still could not be accurately conducted. The industry has substantial incentives to use AI to further improve the accuracy.

Observation 3. AI is applicable in performing cyber operations. Given a variety of **data sources** (*e.g.*, indicators of compromises, IDS alerts, network traffic, event logs, threat intelligence), **cyber operations** aim to answer four questions: What has happened? What is the impact? Why did it happen? What should I do? Cyber operations are performed to let the **enterprise system** achieve acceptable business continuity in the presence of known and unknown cyberattacks. Cyber operations are by nature data-driven.

Enterprise systems (*e.g.*, networks, platforms, applications, files and databases) rely on security teams – ranging from few security analysts to large-scale cyber operation centers – to defend against and respond to severe cyber threats. Despite that security teams have been trying very hard to cope with such threats, many enterprises are still suffering from significant data breaches and loss (*e.g.*, business continuity loss) caused by attacks and intrusions. The SolarWinds APT (Advanced Persistent Threat) campaign and the Colonial Pipeline Ransomware Attack are just two recent examples illustrating the very limited capabilities of enterprise cyber operations. Today’s cyber operations suffer from several limitations, including the following: (a) Slowness: human cognition has a bottleneck when processing

high volume of cyber data; (b) Do not see the big picture: stovepiped organizational cyber-operation processes are harmful; (c) Mindsets: unrealistic mindsets on "what are trusted" (*e.g.*, the mindset used in defending SolarWinds APT) are harmful; (d) Human resources: the performance of junior analysts is in general much worse than experts.

With the rise of DL, AI technologies provide exciting new opportunities (*e.g.*, accurately detect and predict cyber threats, improve agility, reduce the costs and improve human analysts' job performance, and improve the level of automation) in addressing the challenges faced by security teams. For example, (a) the Endpoint Detection & Response solution framework of BlackBerry is powered by Cylance AI, one of the first Machine Learning (ML) models for cybersecurity. (b) Splunk security analysts are training various DNNs to serve various purposes on a daily basis.

Observation 4. Even some already highly successful tools may benefit substantially from DL and RL. For example, although fuzzing tools are nowadays a standard security testing tool in the industry, their efficiency is still a concern, and the industry has substantial incentives to use AI to further improve the fuzzing efficiency.

The second reason why we write a handbook is that *the existing survey papers have limited usefulness for engineers, security analysts, and students taking an "AI for Cybersecurity" course.* Partially due to the above observations, more and more survey papers (*e.g.*, [3, 14]) are published to summarize the new research progress on applying DL and RL to solve cybersecurity challenges. Although these survey papers provide systemization of the newly developed understandings (and knowledge) about *why* DL and RL could be applied to effectively solve particular cybersecurity challenges, they don't provide a **hands-on introduction** to *how* to apply DL and RL to solve a particular cybersecurity problem. As a result, these survey papers don't provide one with a hands-on learning opportunity, and there is still a wide gap between survey papers written by researchers and the hands-on learning/training needs of engineers, analysts and students.

The third reason is that *a more useful book for engineers, analysts and students should include the following information:*

- This book should provide beginners with a "jump start" learning experience. By "jump start", we mean that a beginner does not need to have any experience in applying DL and RL to solve a security problem.
- This book should provide experienced engineers and security analysts with a reference book of use cases. Rather than reading the book from beginning to end, this book is intended to be consulted for information on specific matters (*e.g.*, use cases, code snippets).
- Instead of providing a lengthy discussion on the general principles, each chapter just describes how a specific security problem gets solved, *i.e.*, each chapter just focuses on one use case.
- Each use case should be presented in a straightforward way so that the readers can easily navigate to what they are mostly interested in, such as code snippets and field-by-field presentation of the training data samples.
- The technical details should be self-contained: there is no need to go to elsewhere for additional technical details.
- The use cases should cover the entire machine learning pipeline, so that the learning experiences

of an engineer, an analyst, or a student can be most close to what is going on in the real world.

- Learning by doing: datasets and source code snippets should be provided to the readers so that the learning experiences can be as hands-on as the readers wish.

1.2 The Use Cases Intend to Solve Various Cybersecurity Challenges through A Unified DL Pipeline

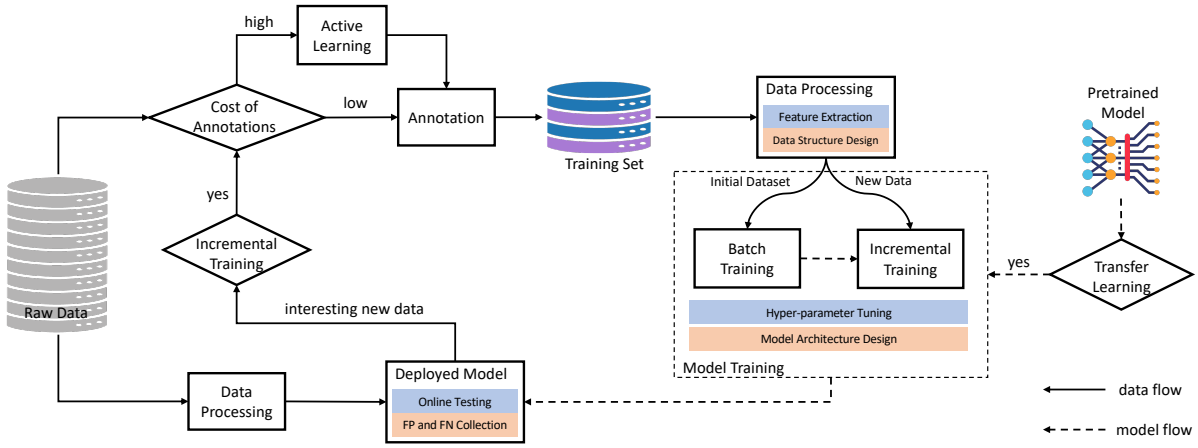


Figure 1.1: A unified DL pipeline could be used to solve various cybersecurity challenges.

Although the machine-learning-based problem solving procedures for different cybersecurity problems are often different in certain aspects, we advocate “applying DL in a systematic way" and believe that “ML pipeline standardization" is a good thing in the cybersecurity industry. We observe that the industry could be benefited a lot by ML pipeline standardization when applying DL and RL to solve cybersecurity challenges.

Since almost all the use cases presented in this book are applying DL to solve cybersecurity problems, here we synthesize a unified DL pipeline, which is shown in Figure 1.1. The main steps and characteristics of the DL pipeline are as follows.

- The DL pipeline consists of two **sets** of cyclic or non-cyclic workflows. Set A workflows *produce* DL models: they take the Raw Data as input, and produces a model which can be deployed in production systems. In contrast, Set B workflows *consume* DL models: they take (newly arrived) Raw Data as input, and output classification or prediction results. Model deployment (*e.g.*, how to deploy a newly produced model) is in the intersection of Set A and Set B.
- Depending on several technical (*e.g.*, whether the data annotation cost is high) or business factors (*e.g.*, whether incremental training is a business need), different workflow instances are often observed in different use cases.
- Given a repository of Raw Data, the first step of a Set A workflow is to annotate each unit of raw data with a label. If the annotation cost is high (*e.g.*, when a lot of manual effort is required), Active Learning [54] will be conducted to properly sample a subset of data units.

As described in [54], the main active learning issues include uncertainty sampling, diversity sampling, combining uncertainty sampling and diversity sampling, and active transfer learning.

- The data annotation step will result in the Training Set which is a set of *labeled* units of raw data. Note that the Training Set should be as *balanced* as possible.
- Since the Raw Data usually cannot be directly used to train a DL model, the Data Processing step of the Set A workflow will transform each member in the Training Set to a *training data sample*. Feature Extraction and Data Structure Design are two dominant challenges in this step.
- When a pretrained model is suitable for solving the corresponding cybersecurity problem, the Model Training step would become a Transfer Learning step in which the learning task designated to solving the cybersecurity problem would be treated as a downstream task.
- When no pretrained model is suitable for solving the cybersecurity problem, the Model Training step will firstly train a model from scratch using the initial set of training data samples. This sub-step is called Batch Training. When new data samples arrive, Incremental Training will usually be conducted to using the new data samples to enhance the existing model(s) (*e.g.*, make it more accurate), but no longer from scratch.
- During the Model Training step, hyper-parameter tuning and model architecture design are two major challenges to address.
- Since the computing environment of Model Training is often quite different from that of the Deployed Model, model deployment itself involves several challenges and issues, including how to conduct online testing, how to compress a DNN model, how to employ appropriate hardware acceleration, how to manage the dependencies between the libraries called during the execution of the model, how to audit a DNN site, and how to reliably upgrade a DNN site.
- A Set B workflow is quite straightforward. For example, when newly arrived unit of raw data needs to be classified, the unit will be firstly sent to the Data Processing component, which conducts the same kind of data processing described above. Then the corresponding data sample will be fed into the Deployed Model as an input.
- Cyclic workflows, though not always required, could be helpful in some circumstances. For example, when the audited classification/prediction results (of the Deployed Model) are further analyzed, false positives and false negatives could be identified, and it is not surprising that these data samples could make Incremental Learning substantially more effective. Hence, the industry has incentives to add the corresponding units of raw data to the Training Set. As a result, a cyclic workflow is formed.
- Since the DL pipeline is dedicated to solving cybersecurity problems, it has a few domain-specific characteristics. For example, the Data Structure Design challenge addressed in the Data Processing step of a Set A workflow is often associated with in-depth domain knowledge which only a cybersecurity expert would have. Insights on “which data structure is most appropriate to represent the raw data" are often hard to be automatically obtained or learned.

1.3 How to Properly Use This Handbook?

Engineers, security analysts and students can use this handbook to get a hands-on introduction to *how* to apply DL and RL to solve a particular cybersecurity problem. Using the code snippets and dataset links provided in the handbook, the readers can achieve “learning by doing" at use case level.

In particular, the handbook can be used by engineers, security analysts and students in several different ways to serve various needs:

- A reader can simply view the book as a set of use cases of how AI is used to solve a particular cybersecurity problem. When the reader intends to solve a new cybersecurity problem, he or she could start with a use case in the book which solves a similar problem.
- Since it is a handbook, a reader can directly navigate to a specific use case in the book without worrying about the content preceding it. The reader do not need to follow the order in which the book contents are presented. Each use case provides the reader with an independent hands-on learning experience.
- After reading a chapter of the book, it is often a good idea to download the corresponding datasets and source code, and quickly train an AI model. This book intends to make “learning by doing" easier for the readers.
- A reader could directly search for certain kind of code snippet. In this way, the reader can treat the book as an indexed repository of code snippets.
- A reader could directly search for a particular field inside a data sample belonging to certain kind of cybersecurity data. In this way, the reader can treat the book as a “dictionary" of cybersecurity data fields.
- It should be noticed that the book is not a textbook and hence probably should not be treated as a textbook. The book assumes that the readers already know the basic concepts and principles of *why* DL and RL could be applied to effectively solve particular cybersecurity challenges.
- A reader could use the book to gain a big-picture understanding of real-world AI applications in the cybersecurity domain. Each use case in the book illustrates the entire machine learning pipeline involved in solving a particular cybersecurity problem.

1.4 Organization of Rest of The Book

The rest of the book is organized as follows. In Chapter 2, two use cases are presented in which AI conducts two particular reverse engineering tasks. In Chapter 3, one use case is presented in which AI detects Android malware. In Chapter 4, one use case is presented in which AI detects abnormal events in two distinct sets of sequential data. In Chapter 5, one use case is presented in which AI detects DNS cache poisoning attack in enterprise networks. In Chapter 6, one use case is presented in which AI conducts PC malware detection. In Chapter 7, one use case is presented in which AI detects binary code similarity. In Chapter 8, one use case is presented in which AI conducts malware clustering.

Chapter 2 AI Conducts Two Reverse Engineering Tasks

2.1 The Security Problem

Reverse engineering (RE) is used to recover the semantic information of the original source code from binaries [16]. Since various commercial software and malicious software do not provide their source code, RE has a variety of applications in cybersecurity. For example, to deploy some security strategies (*e.g.*, CFI [2]) on binary-only programs, RE is adopted to reveal the hierarchy structures (*e.g.*, code section, functions, basic blocks, and etc.) of the binary file. For another example, to analyze ransomware and find the crypto keys, analysts need to first generate flow graphs of (sometime obfuscated) the ransomware.

Unfortunately, manual RE may require a lot of labor efforts from the analysts. A human reverse-engineer has to guess the boundary of procedures, follow the control and data flow in these procedures, infer the semantics of (frequently) accessed data, and finally aggregate all these information together to develop a holistic understanding of the program logic of the inspected executable. Therefore, more automatic RE methods are very attractive to the analysts, which can be achieved by adopting AI and deep learning technologies.

In this chapter, we focus on two basic yet essential RE tasks, which are foundations of some advanced RE tasks:

1. *Function Boundary Identification*. Given the raw bytes of `text` section in a binary, function boundary identification aims to detect the beginning and the end of functions. Figure 2.1a shows an example of a function boundary identification problem. Label `1` and `-1` indicate that current byte is the beginning and the end of a function, respectively. Other bytes inside a function are labeled with `0`. Shin, et al. designed a model [67] to identify function boundaries through the recurrent neural network.
2. *Function Signature Generation*. Given the binary code of functions, the function signature generation is to predict the type of the return values, the number of parameters, and the type of each parameter. Figure 2.1b shows an example of a function signature generation problem. [16] developed a deep learning based scheme to learn function type information.

Even though these two RE tasks may seem simple at the first glance. It is not easy to achieve high accuracy through traditional machine learning or data mining methods. The challenge is mainly due to that compilers may emit binary code in the backend in various hard-to-predict patterns.

2.2 Related Work

Researchers had developed some RE tools to ease the burden of human analysts. For example, IDA [24] is developed by Hex-Rays and widely used by security analysts to help analyze the malware.

401130:	add	$0x20,%rsp	48	0
			83	0
			c4	0
			20	0
401134:	pop	%rbp	5d	0
401135:	retq		c3	-1
401136:	push	%rbp	55	1
401137:	mov	%rsp,%rbp	48	0
			89	0
			e5	0
40113a:	sub	$0x20,%rsp	48	0
			83	0
			ec	0
			20	0

(a) Instruction Sequence (b) Raw Bytes (c) Labels

(a) A data sample of function boundary. Label 1 indicates that corresponding byte is the beginning of a function.

```
Function:
    f3 0f 1e fa             endbr64
    48 0f be 16             movsbq (%rsi),%rdx
    48 89 f8                mov    %rdi,%rax
    48 8d 3d c2 0d 00 00    lea    0xdc2(%rip),%rdi
    48 89 d1                mov    %rdx,%rcx
    48 39 fa                cmp    %rdi,%rdx
    74 1a                   je     1264 <strcpy+0x34>
    31 d2                   xor    %edx,%edx
    0f 1f 40 00             nopl   0x0(%rax)
    88 0c 10                mov    %cl,(%rax,%rdx,1)
    48 83 c2 01             add    $0x1,%rdx
    4c 0f be 04 16          movsbq (%rsi,%rdx,1),%r8
    4c 89 c1                mov    %r8,%rcx
    49 39 f8                cmp    %rdi,%r8
    75 ec                   jne    1250 <strcpy+0x20>
    c3                      retq
Signature:
    number of args: 2
    args types: [char*, char *]
    return type: char *
```

(b) A data sample of function signature.

Figure 2.1: Illustration of data samples.

These tools are developed based on domain knowledge, that is the code generation conventions, to carry out the disassembly and the analysis. However, many factors can affect the generated binary code, such as the programming language used, the compiler and its options, and different architectures and operating systems. Furthermore, some commercial software and malware are obfuscated to hide their logic against RE. Therefore, developing rule-based methods is not an easy task and will require huge amount of human efforts.

In recently years, machine learning methods have been gradually introduced to tackle various RE tasks (*e.g.*, function boundary identification, function prototype inference, etc.) and showed big potential. Among these methods, deep learning performed promising results. Researchers observed two characteristics that make deep learning effective in RE: firstly, comparing with traditional machine learning, deep learning has strong representation learning ability and can discover intricate structures in high-dimensional data [41, 61]. Secondly, even though deep learning generally require a larg dataset to train a high-quality model, it is not a challenge for most RE problems to generate large amount training samples.

2.3 DL Pipeline

In this section, we follow the unified DL pipeline (Figure 1.1) introduced in chapter 1 to illustrate the deep learning approaches for RE. Different RE tasks often share some common characteristics. Therefore, in each of the following sub-sections, we will firstly present the common characteristics shared by the two RE tasks. After the common characteristics are presented, we will present the distinct characteristics of each of the two RE tasks.

2.3.1 Raw Data

The RE always starts from stripped binaries (*i.e.*, binaries without any debug information), which contain both instructions and data. It is not challenging to gather raw data because both binaries and source code (which can be easily compiled to binary file) is widely available. In most cases, researchers collect their binary from some utility programs (*e.g.*, `binutils`, and `coreutils`) and open source projects (*e.g.*, compilers, and GNU projects).

For example, [67] adopts a dataset which consists of binaries from 3 popular Linux packages: `binutils`, `coreutils` and `findutils`. The dataset include 2000 different binaries, which are compiled from the programs in the packages using 4 optimization levels (`O0-O3`) with two compilers (`gcc` and `icc`) targeting two instruction sets (`x86` and `x64`). For `x86` binaries, there are `1,237,798` distinct instructions which make up `274,285` functions. Similarly for x64, there are `1,402,220` distinct instructions which make up `274,288` functions.

[16] extended the first dataset with 5 more packages, leading to a total of 8 packages: `binutils`, `coreutils`, `findutils`, `sg3utils`, `utillinux`, `inetutils`, `diffutils`, and `usbutils`. In terms of the number of binaries, this dataset is extended to 5168 different binaries. For `x86` binaries, there are `1,598,937` distinct instructions which constitute `370,317` functions while for `x64`, there are a total of `1,907,694` distinct instructions which make up `370,145` functions. The location of the these dataset will be listed in section 2.8.

Function Boundary Identification. For the function boundary identification, the raw data are bytes sequences of the equal length, which are generated by cutting the `text` section in binaries with a frame window. Short sequences are extended to the required length through padding (typically filled with zeros).

Function Signature Generation. Function signature generation assumes that the boundary of functions is known, so the raw data are all bytes of instructions from a function.

2.3.2 Data Annotation

Most RE tasks can obtain their ground truth from the source code. That is, when training the models, the source code is assumed to be available, and then the trained models are directly used to process binary without the assistance of any source code. In RE tasks, desired high level information can be easily acquired if source code is available, and it is not very challenging to pass this information from source code to binary (*e.g.*, by parsing the DWARF debug information [71]).

Labels for Function Boundary Identification Problem. For each sequence of bytes, a sequence of labels are there to indicate whether corresponding bytes are start of functions.

Labels for Function Function Signature Generation Problem. The function signature generation problems is a collection of small tasks which usually be solved through several model. Therefore, there are three kinds of labels (number of arguments, types of each arguments, and type of return value) for each function.

```
def instructions2int8(instructions):
    uint8 = []
    for ins in instructions:
        b = ins.binary
        for i in range(len(b)):
            uint8.append(b[i])
    return uint8
```

Code 2.1: Code for raw-byte encoding.

2.3.3 Data Processing

Data processing extracts instruction sequences from binary files. Several tools (`objdump`, `readelf`, `pyelftools`) can help to achieve the goal. Further processing of the extracted instruction sequences is dependent upon the adopted embedding methods.

2.3.3.1 Embedding Methods

When applying deep learning to these RE tasks, the first question is: what kind of encoding methods should be adopted to represent the data? Given the instruction sequence as the raw data, it is necessary to encode instructions in a deep-learning-friendly data structure so that a deep learning algorithm can take them as input and learn important features.

Traditionally, the binary instructions can be represented in two forms: machine code, and assembly code. The machine code is the a representation of instruction that can be executed by the CPU, and the assembly code is a string representation of instruction that can be easier for the human to understand. The following example shows the machine code and assembly code of an `sub` instruction.

$$83\ ec\ 0c \qquad sub\ \$0xc,\ \%esp$$

As summarized by [44], recently researchers had tried many different embedding methods, which can be grouped into three categories: 1) directly adopt raw bytes as input embedding, 2) select encoding features selected from the assembly code, or 3) automatically learn to generate a vector representation for each instruction through some representation learning models. We will briefly introduce them and show their pros and cons in this section.

Raw-byte Encoding. The most naive encoding method is to apply a simple encoding on the machine code of each instruction, and then feed (a sequence of) the encoded instructions into a deep neural network. For example, one-hot encoding adopted by DeepVSA [36] can convert each byte of machine code into a 256-dimensional vector. One of these dimensions in the vector is 1 and the others are all 0. Instead of the one-hot vector, Shin et al. [67] directly adopt the binary encoding of instruction's machine code, namely, encoding each byte as a 8-dimensional vector, in which each dimension presents the corresponding digit in the binary representation of that byte. Figure 2.2 shows how the raw bytes of the aforementioned instruction are encoded through the this method. Code 2.1 shows the piece of code for raw-byte encoding.

In general, the raw bytes encoding is the most straightforward and simple method, which does not require disassembling and other computationally expensive process. On the other hand, the

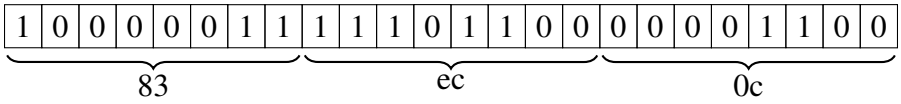


Figure 2.2: Binary encoding.

```
def read_asmcode_corpus(asmcode_corpus):
    asmcode_corpus = inst2vec.refine_asmcode(asmcode_corpus)
    list_instruction = []

    asmcode_corpus_list_split = asmcode_corpus.split('\n')

    for one_instruction in asmcode_corpus_list_split:

        one_instruction=one_instruction.replace(',',' ') # for split
        one_instruction=one_instruction.split()
        list_instruction.append(one_instruction)

    return list_instruction

word_list = read_asmcode_corpus(asmcode_corpus):
model = Word2Vec(word_list, size=vectorsize, window=128, min_count=1, workers=4, iter = 10)
```

Code 2.2: Code for Instruction2Vec [1].

disadvantage is that the semantic meaning of each instruction is hidden in the raw bytes encoding. For example, the X86 Instruction Set Architecture (ISA) adopts variable-length encoding, which means that common instructions typically have a shorter encoding so that they need less space in instruction cache; whereas less common instructions have a longer encoding. Therefore, the semantic meaning represented by bytes in different instructions is different, which pose big challenge for the deep neural networks to understand the instruction. Consequently, even the raw bytes contains as much information as disassembled code, it is questionable whether the neural networks can effectively learn the semantic meaning from such raw byte encoding.

Encoding of Disassembled Instructions. Since assembly code carries semantic meaning of instructions in a more comprehensible way, some researchers choose and encode features from assembly code. For example, [45] adopts opcodes extracted from assembly code, and encodes them through one-hot encoding, but this method completely ignores the information from operands. *Instruction2Vec* proposed by [42] adopts both opcode and operand from instructions, and encodes each instruction as a nine-dimensional feature vector.

Representation Learning for Instruction. Another methods is to learn an instruction embedding through representation learning. For example, *Asm2Vec* [21] makes use of the *PV-DM* neural network [40] to jointly learn instruction embeddings and function embedding. Specifically, this model learns latent lexical semantics between tokens and represents an assembly function as an internally weighted mixture of collective semantics. The learning process does not require any prior knowledge about assembly code, such as compiler optimization settings or the correct mapping between assembly functions [21]. Experiments show that *Asm2Vec* is more resilient to code obfuscation and compiler optimizations than aforementioned two embedding methods.

Pretrained Model for Instruction Embedding. Although recent progress in instruction represen-

[1]Copy from https://github.com/onstar99/VulnerabilitySystem

tation learning is encouraging, there are still some challenges to learn a good representation. As mentioned in [44], the complex internal formats of instructions are not easy to learn. For instance, in x86 assembly code, the number of operands can vary from zero to three; an operand could be a CPU register, an expression for a memory location, an immediate constant, or a string symbol; some instructions even have implicit operands, etc.

Secondly, some high level semantics (*e.g.*, data flow) hidden in instruction sequence can not be easily captured by deep learning model without some specific designs. Therefore, pretrained models (*e.g.*, [44] and [38]) with specific pre-training tasks and a large dataset to reveal complex internal structures and high level semantics are proposed. Such a well-trained model can be applied to many downstream tasks and achieve good results.

2.3.3.2 Principles to Guide Selection of Embedding Method

As stated by [44], "compared to the first two choices, automatically learning instruction-level representation is more attractive for two reasons: 1) it avoids manually designing efforts, which require expert knowledge and may be tedious and error-prone; and 2) it can learn higher-level features rather than pure syntactic features and thus provide better support for downstream tasks."

Despite the advantages of automatically learning representation for instruction, we observe two disadvantages: a) it requires a larger training set to automatically learn good representation for instructions. b) representation learning for binary code will consuming more computation resource. Accordingly, pre-trained models were put forward to alleviate these two limitations. Firstly, even through it is a common challenge to collect a large training set with labels for a specific task, it is not the case to generate a large dataset without any labels, since there are a large number of programs available in the Internet. By carefully designing a pre-training task (*e.g.*, mask language model), a good pre-trained model can be trained purely through a large unlabeled dataset. Secondly, a well trained pre-trained model could be reused by many downstream tasks. That means other researchers can directly adopt the pre-trained model to generate instruction representation. They only need to fine-tune some layers (*e.g.*, output layer) to get a model for a specific task, which will required fairly small amount of resource.

In summary, firstly a pre-trained model is the best option if available. Secondly, when a suitable pre-trained model is not available and the users have enough training data and computational resource, they can adopt representation learning to learn instruction embedding. Finally, if there is no large amount of training data or computational resource, encoding of disassembled instructions is the best option.

2.4 Model Architecture

Deep learning models for most RE works can share the same component that learns representation for instruction sequences. In this section, we will first introduce the model architecture design for representation learning, then show the model architectures for the specific reverse engineering tasks.

```
    tokens1 = [self.vocab.sos_index]
    segment1 = [1]
    i = 1
    for ins in bb_pair[0].split(";")[-5:]:
        if ins:
            for token in ins.split():
                tokens1.append(self.vocab.stoi.get(token, self.vocab.unk_index))
                segment1.append(i)
            tokens1.append(self.vocab.eos_index)
            segment1.append(i)
            i += 1
```

Code 2.3: Code for Tokenization [2].

2.4.1 Common Component: Learning the Embedding for Instructions

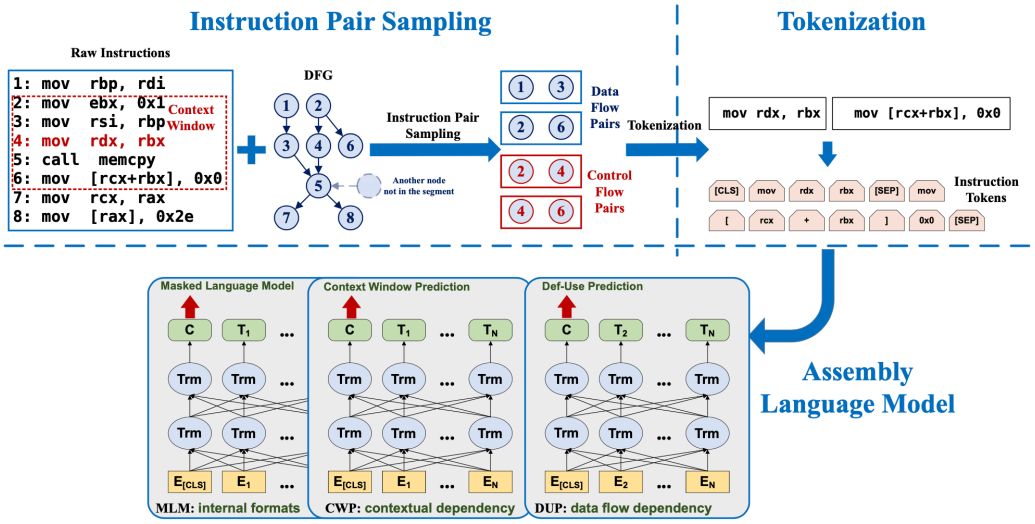


Figure 2.3: Overview of Pre-trained model (copied from [44]).

Deep learning is supposed to be good at building models to learn intricate structures and catch the underlying complicated features, even from high-dimensional data [41, 61]. However, it is better that the developer can represent the data sample in a data structure that can direct expose its internal formats relationships to the deep learning model.

Accordingly, to capture the complex internal formats of instructions and relationships among opcode and operands inside each instruction, it worths to adopt a fine-grained strategy to decompose instructions as mentioned by [44]. To achieve this goal, PalmTree [44] considers each instruction in a code sequence as a sentence and decompose it into basic tokens, and adopt the Transformer [75] to learn embedding for each sentence. The piece of code for tokenization is shown in Code 2.3, and an example of tokenized assemble code is shown in left of Figure 2.3. Taking the sequence of tokens as inputs, PalmTree adopts the Transformer to learn the embedding for instructions.

[2]Copy from https://github.com/palmtreemodel/PalmTree/

```
1     # NLL(negative log likelihood) loss of is_next classification result
2     dfg_next_loss = self.dfg_next_criterion(dfg_next_sent_output, data["dfg_is_next"])
3     cfg_next_loss = self.cfg_next_criterion(cfg_next_sent_output, data["cfg_is_next"])
4
5     # NLLLoss of predicting masked token word
6     mask_loss = self.masked_criterion(mask_lm_output.transpose(1, 2), data["dfg_bert_label"])
7
8     # Adding next_loss and mask_loss
9     loss = dfg_next_loss + cfg_next_loss + mask_loss + comp_loss
```

Code 2.4: Code for Pretrained Model [3].

In order to enable the designed deep neural network to understand the internal structures of instructions and dependency among instructions, PalmTree makes use of three pre-training tasks as shown in Figure 2.3:

1. A Masked Language Model (MLM) to understand the internal structures of instructions. MLM predicts the masked tokens within instructions.
2. A Context Window Prediction (CWP) to capture the relationships between instructions. CWP infers the word/instruction semantics by predicting two instructions' co-occurrence within a sliding window in control flow.
3. Def-Use Prediction (DUP) to learn the data dependency (or def-use relation) between instructions. DUP predicts if two instructions have a def-use relation.

Code 2.4 shows the code snippet for the pre-training tasks. Line 6, Line 3, and Line 2 calculates the loss for MLM, CWP, and DUP, respectively. Line 9 aggregates the total loss.

The transformer's encoder produces a sequence of hidden states as output and each hidden states is corresponding to a token from the input. The PalmTree adopts the mean pooling of the hidden states of the second last layer to represent the instruction representation. However, other pooling methods such as max pooling and sum pooling are also possible to achieve the similar purpose.

NOTING: The purpose of the pre-training tasks is to learn good representation for instructions through unsupervised learning. Therefore, pre-training tasks are not always necessary if the labeled training set of a downstream task is big enough to learn a good model.

2.4.2 Model Architecture For Function Boundary Identification

It is very straightforward to solve the function boundary identification problem after instruction embedding is learned. A simple output layer (*e.g.*, full connection layer) can be adopted to predict the label for each instructions, that is whether the instruction is start or end of a function.

2.4.3 Model Architecture For Function Signature Generation

PalmTree presents the model for function signature generation based on the pre-retrained model as shown in Figure 2.4. Firstly, the model leverages the pre-trained model shown in [44] to learn embedding for instructions in a target function. Secondly, a recurrent neural network (RNN) or its

[3]Copy from https://github.com/palmtreemodel/PalmTree

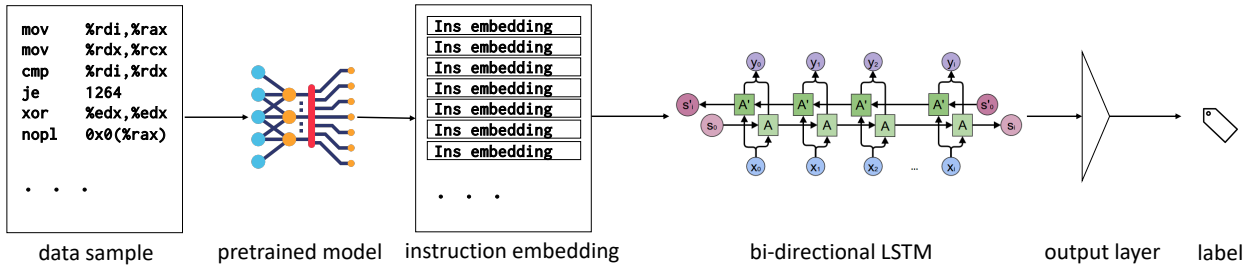


Figure 2.4: Overview of model for function signature generation.

```
1
2     def predict(self, inputs, source):
3         # generate embedding for each bytes from the instruction sequence.
4         inputs_embedded = self.embedding_layer(x_input)
5
6         # generate representation for bytes through bi-directional RNN/LSTM.
7         inputs_embedded = self.encoder_biRNN(inputs_embedded)
8
9         # predict label for each bytes based on the learned representation through an output layer.
10        class_pred = self.class_pred_net(inputs_embedded)
11        return class_pred
```

Code 2.5: Code for function boundary identification [4].

variants (LSTM or GRU) are adopt to aggregate the instruction embedding of the target function. Thirdly, a output layer are adopt to predict the type signature for the target function. Code 2.5 shows the implementation of the model.

2.5 Model Training Issues

There are several issues that could be faced in the model training phase. Firstly, training a pre-trained model often requires a huge amount of GPU resources, which may not be affordable to most users.

Secondly, the dataset could be extremely imbalanced. When solving the function boundary identification problem, a function with thousands of bytes only has one beginning byte and one end byte. When solving the function signature generation problem, numbers of input arguments are unevenly distributed among the candidate functions. There are two strategies that can be adopted to solve the unbalanced data issue: 1) balancing dataset by downsampling the majority classes; 2) strengthen penalty for lost minority class by adopting a weighted lose function.

Thirdly, trade-off between precision and recall is widely noticed in machine learning, which can be tuned through a weighted loss function. This can help to achieve a high recall by sacrificing some precision for some specific security problems (*e.g.*, binary rewriting).

[4]Copy from https://github.com/shensq04/EKLAVYA

Table 2.1: Model performance on function type signature generation.

Design	*Accuracy*	*Standard Deviation*
One-hot	0.309	0.0338
Instruction2Vec	0.311	0.0407
word2vec	0.856	0.0884
PalmTree	0.8747	0.0475

Table 2.2: Model performance on function boundary identification.

Design	**ELF x86**			**ELF x86-64**		
	Precision	Recall	F1	Precision	Recall	F1
One-hot	0.9775	0.9534	0.9653	0.9485	0.8991	0.9232

2.6 Model Performance

Since PalmTree [44] provides the model performance on the function boundary identification problem, Table 2.1 directly copies the result from the original paper. The model performance on function boundary identification with one-hot embedding method is evaluated in [67], and here we refer their model performance in Table 2.2.

2.7 Deployed Model

The models for RE can be deployed in two scenarios: Firstly, it can be integrated into the RE platforms such as IDA Pro, or Binary Ninja. However, since the parameter size of pre-trained models is usually large, a substantial amount of computational resource is usually required.

Therefore, there is another possible scenario for model deployment – it could be deployed as an online service. Specifically, a company can deploy the model together with the data processing component on a cloud server. And users can upload binaries they want to analyze onto the server, the server then will process the binaries, predict the label for the corresponding data samples, and send the results back to users.

2.8 Source Code and Dataset

The source code of PalmTree [44] is publicly available in `https://github.com/palmtreemodel/PalmTree`. The original implementation of the function type signature generation [16] is available in `https://github.com/shensq04/EKLAVYA`. There is no source code public available for the function boundary identification work [67].

We implement the model introduced in subsection 2.4.2 and release their source code on GitHub: `https://github.com/PSUCyberSecurityLab/AIforCybersecurity`.

Besides, as mentioned in subsection 2.3.1, both of the two papers adopt some publicly available dataset to prepare their training set. These dataset, consists of binary programs are available at

https://security.ece.cmu.edu/byteweight/. Besides, we prepare some ready-to-used training set in our GitHub repository.

2.9 Remaining Issues

We notice two limitations of the existing deep learning models for RE. Firstly, a model trained on one platform usually could not be generalized to binaries from other platforms, with different compiler tool-chains, or even with different compiler options. Although it is possible to train one model for each kind of binaries, it could take lots of human effort and resources due to that large quantity of platforms. For example, ARM [50] has 10 versions of processors and more than 10 versions of architectures. GCC [32] and LLVM [39] both contain 4 levels of optimization. Therefore, it deserves further research to enhance a model's generalization ability in conducting RE tasks.

Secondly, even through the existing research works show great potential in deep-learning-assisted RE, we notice that the current deep learning applications can only be used to conduct very simple RE tasks and are not very capable for conducting more complex RE tasks (*e.g.*, CFG, DFG generation, and pointer analysis). A main reason seems that the current schemes are not very capable of learning high level semantic information. Fortunately, we see more and more researchers begin to learn higher level semantic from binary code [78, 79]

Finally, binary files contain both code and data, and some of them are intermixed together (*e.g.*, jump tables [6]) due to the characteristics of von Neumann architectures. However, currently deep learning applications in RE are mainly focus on the RE of code. Hence, how to learn the memory layout (*e.g.*, data structures) of a program deserves in-depth research. In the literature, there is very limited research work in learning data structures and memory layout [12, 79].

Chapter 3 AI Detects Android Malware

3.1 The Security Problem

Android malware is simply any kind of malicious software or code designed to harm Android user's mobile device, such as viruses, trojans, ransomware, adware, spyware or phishing apps. Such malware can be downloaded or installed from a variety of places including malicious downloads in emails, visiting suspicious websites or clicking links from unknown senders. Once malware invades Android user's mobile, it can do several types of harmful things including stealing and selling your sensitive information, spying your mobile usage like text messages, phone calls, etc., or demanding for a ransom.

Though Android users' mobile devices expose their attack surface to side-loading an app, app markets such as Google Play and Amazon AppStore still play an essential role in today's mobile ecosystem, through which the majority of mobile apps are published, updated, and distributed to users [33]. Before an app is approved to be published to an app market, such as Google Play, T-Market, an Android application developer firstly uploaded either their new app or updated one to the app market. To obtain approval, the submitted app will be analyzed by the app market to decide whether it is malicious or not. Finally, the check report will be sent back to the application developer saying that either the app is detected as malicious or the app gets approval to be published.

3.2 Android Malware Example

In the past, Android ransomware made use of a special permission called "SYSTEM_ALERT-_WINDOW", which can draw a window that stays on top of all other windows regardless of which button is pressed. Those Android ransomware intentionally misuse this permission to display their ransom note which occupies the screen, making the device unusable until users pay the ransom.

```
1    PendingIntent ransomActivity = HelperOne.getIntent(context);
2    Notification.Builder notificationBuider = new Notification.Builder(context, Helper.notID);
3    notificationBuilder.setSmallIcon(HelperThree.getIconID("iaevwhnmfj"));
4    notificationBuilder.setContentTitle(context.getString(HelperThree.getID("zebkpsro")));
5    notificationBuilder.setContentText(context.getString(HelperThree.getID("qgukbdhnjc")));
6    notificationBuilder.setCategory("call");
7    notificationBuilder.setAutoCancel(true);
8    notificationBuilder.setFullScreenIntent(ransomeActivity, true);
9    notificationBuilder.setWhen(System.currentTimeMillis());
10   Helper.createNotificationChannel(context);
11   NotificationManager v4 = (NotificationManager)HelperThree.getSystemService(context, "notification");
12   if (v4 != null) {
13       v4.notify(Integer.parseInt(Helper.notID), notificationBuilder.build());
14   }
```

Code 3.1: The notification with full intent and set as "call" category [1].

[1]Copy from https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/

Details about a new Android ransomware variant were revealed on October 8, 2020 [68]. Basically, to release its ransom note, it utilizes the "call" notification and "onUserLeaveHint()" jointly to trigger the ransom screen. As shown in Code 3.1, the malware creates a notification builder. Method `setCategory` is to make this notification need immediate user action. Method `setFullScreenIntent` is to wire the notification to a GUI so that it pops up when the user clicks it.

```
public class RansomActivity extends Activity {
    public static volatile RansomActivity activityObj;
    @Override // android.app.Activity
    public void onBackPressed() {
    }
    @Override // android.app.Activity
    protected void onCreate(Bundle) {
        super.onCreate(bundle);
        RansomActivity.activityObj = this;
        HelperTwo.putActivityObj(this);
        HelperThree.checkPresenceOfVirtualMachine(this);
        this.getWindow().addFlags(RansomActivity.getFlags());
        try {
            Context context = this.getApplicationContext();
            this.setContentView(WebViewAggregator.getWebView(context, context.getCacheDir().getPath()));
        }
        catch(Exception exception) {
            exception.printStackTrace();
        }
    }
    @Override // android.app.Activity
    protected void onUserLeaveHint() {
        super.onUserLeaveHint();
        this.starActivity(new Intent(this, RansomActivity.class));
    }
    public static int getFlags() {
        return 0x680480;
    }
}
```

Code 3.2: The malware overriding onUserLeaveHint[2]

To continuously occupy the screen with ransom note, the malware needs to invoke the automatic pop-up of the ransomware screen without user interaction. As shown in Code 3.2, the malware overrides the `onUserLeaveHint()` callback function of the Activity class. Whenever the malware screen is pushed to the background, the callback function will be called, bringing the in-call activity to the foreground. Since the malware has already hooked the `RansomActivity` intent with the notification type set to be "call", a chain of function calls has been formed for the malware to occupy the screen.

3.3 Machine Learning Pipeline for the Use Case

Since DL is not applied in this chapter, the unified DL pipeline shown in Figure 1.1 could not be adopted in this use case. The machine learning pipeline adopted in this use case is proposed in [33] and shown in Figure 3.1. Starting from an App, its metadata will be extracted through the dynamic

[2]Copy from https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/

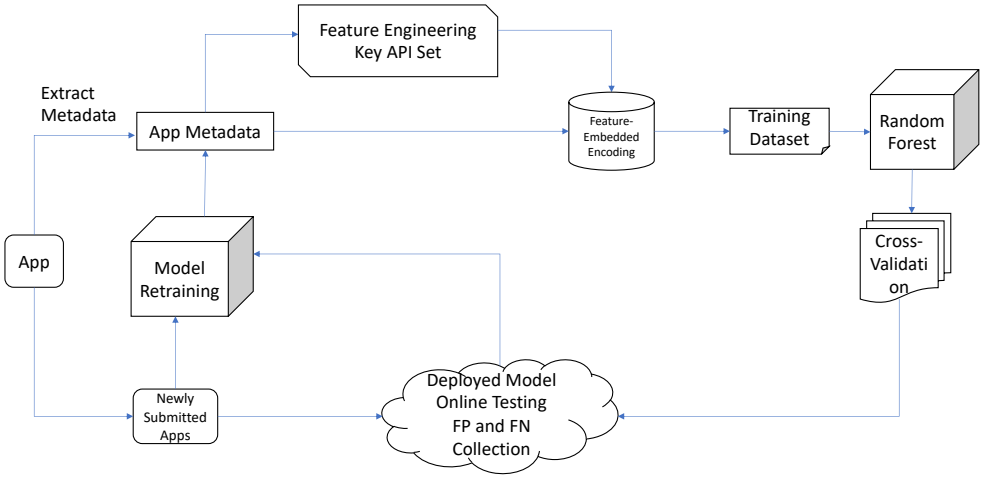


Figure 3.1: Machine learning pipeline for Android malware detection (proposed in [33]).

analysis engine built in [33] based on Google's official Android emulator [4] and the Xposed Hooking framework [85]. The intended metadata includes requested permissions, invoked APIs, and used intents. To achieve high UI coverage, the dynamic analysis engine adopts *Monkey* UI exerciser [74] to generate UI event streams at both application and system levels. With these extracted metadata, the Feature Engineering component is responsible for selecting the key API set for further feature-embedded encoding. Afterwards, these encoded feature vectors will be used by the Random Forest machine learning algorithm as the Training Dataset. After cross-validation, the trained model will be deployed in the production environment for online testing. In the meanwhile, false positives and false negatives will be collected to prepare for model retraining. The periodic retraining time is set to be one month in [33]. During model retraining, the newly submitted apps and the existing app dataset will go through the whole procedure described above to obtain the retrained model to accommodate the evolving Android malware.

3.4 Feature Engineering

In machine learning and pattern recognition, a feature is an individual measurable property or characteristic of a phenomenon [8]. "Choosing informative, discriminating and independent features is a crucial element of effective algorithms in pattern recognition, classification and regression and features are usually numeric" [27]. In this Android malware detection through dynamic API-level ML technique, APIs (application programming interface) are selected as features to train machine learning models, which later on will be used to classify one Android app as malware or not. For example, if one API was called when the app was running, then feature value one will be noted for this API regarding the tested app, or value zero will be given.

An API specification is a detailed description which guides the developer to use such an interface to fulfill a specific requirement or functionality. One of the APIs, which are selected as features, is

android.telephony.SmsManager sendTextMessage. This API is specifically responsible for sending a text-based SMS and its specification is shown in Code 3.3 [66].

```
1    public void sendTextMessage (String destinationAddress, // Sting: This value cannot be null.
2            String scAdress, // String: This value may be null.
3            String text, // String: This value cannot be null.
4            PendingIntent sentIntent, // PendingIntent: This value may be null.
5            PendingIntent deliveryIntent, // PendingIntent: This value may be null
6            long messageId // long: An id that uniquely identifies the message requested to be sent. Used for
                 logging and diagonostics purposes. The id may be 0.)
```

Code 3.3: sendTextMessage.

Feature engineering is to identify features to better represent the target problem being handled by machine learning and further improve its accuracy. It usually uses domain-specific knowledge or automated methods to extract, select, or construct the right features. For this Android malware detection use case, the key to achieving high accuracy is feature selection, which is essentially API selection in the approach proposed in [33]. Given that the current Android SDK provides over 50,000 APIs, a main issue is whether all of the APIs should be selected. As demonstrated in [33], only a small portion of APIs need to be selected. (In fact, 426 key APIs are selected as features.) Regarding why only a small portion of APIs, the main reasons are as follows.

- Since this technique records invocation of APIs during the dynamic emulation of the tested app, the dynamic analysis time will be substantially impacted by the number of selected APIs.
- A better detection accuracy can be achieved by selectively tracking a smaller portion of APIs than all 50,000 APIs.
- Some APIs complement each other with regard to functionality, which can be combined together to enhance the accuracy.

3.4.1 Understanding Trade-offs for API Selection

There are three aspects to understand the insights behind API selection: the statistical correlation between an API and the malice of apps, time consumption for dynamic analysis, accuracy of malware detection [33].

- Based on evaluation of the *Spearman's rank correlation coefficient* (SRC [51]), 260 APIs are selected, including 247 APIs with $SRC \geq 0.2$ and 13 APIs with $SRC \leq -0.2$ respectively.
- Based on a quantitative distribution, an average detection time of less than 5 minutes per app can be achieved by tracking no more than the top-490 APIs.
- Based on comparing malware detection accuracy in terms of tracking top-n correlated APIs using random forest classifier, better precision and recall could be achieved by tracking fewer APIs with effective strategies.

3.4.2 Key API Selection Strategy

There are four steps for API selection [33].

- Select APIs with the highest correlation with malware (Set-C), which results in top-260 highly correlated APIs.
- Select APIs that need restrictive permissions (Set-P), which has 112 APIs. To ensure the privacy/security requirement of user information, permissions to access specific information or execute particular functions need to be granted for an app [83]. There two tools Axplorer [20] and PScout [22] can be utilized to do selection.
- Select APIs that perform sensitive operations (Set-S), which include 70 identified APIs. Selection is based on domain knowledge, considering the sensitive operations previously utilized to realize attacks [13, 26, 90, 95].
- Combing the above, which leads to a total of 426 key APIs, i.e., Set-P∪Set-S∪Set-C, as shown in Figure 3.2.

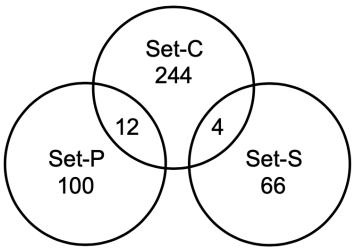


Figure 3.2: Number of APIs in $Set-C$, $Set-P$, $Set-S$ and their overlaps [33].

3.4.3 Further Enriching the Feature Space

Due to the limitation that certain key API invocations can be bypassed through other mechanisms such as Java reflection and intents, such “concealed" API invocations need extra mitigation. Considering the fact that it is necessary for an app to request permission before invoking those hidden APIs [30], two auxiliary feature are added [33], which are the requested permissions and the used intents, respectively. As a result, together with these added features, the best performance is achieved, i.e., the precision increases from 96.8% to 98.6%, recall rises from 93.7% to 96.7%.

3.5 Training Data

There are 501,971 new and updated apps in the app dataset used in [33], which are submitted to T-Market from March to December 2017. A malice label (Malicious or Benign) has been provided for each app by T-Market, which is taken as the ground truth. In total, there are 463,273 benign apps and 38,698 malicious apps in the dataset.

In machine learning, a feature vector is basically a collection of features. Each app has its own set of features. Different features are represented by different numeric or symbolic feature values, and multiple feature values can be combined to form a feature vector. Machine learning algorithms typically require numerical feature vectors in order for the algorithms to do processing and statistical analysis [28].

The Random Forest algorithm is adopted to classify an app as malware or non-malware. During the dynamic emulation of each app, the invocation status of the tracked APIs (API calls' names and parameters) is logged. As shown in Code 3.4, each emulation is assigned with an unique task id. For example, task id 20170718000003300 corresponds to an app named "meimeicaicaicai". One-Hot encoding [33] is employed to convert the log to a feature vector comprising a total of n bits, where n is the total number of tracked APIs. As shown in Code 3.5, each bit corresponds to a tracked API – if the API is invoked, the corresponding bit is set as 1; otherwise it is 0.

```
1  -1  0  ProcessId{:}2871{|}PackageURI{:}file:///data/share1/task/com.tle.erfpoz.apk{|}FileMd5{:}45
       e6ef0e6b6fa2dc122ee6d18e29fca1{|}FileName{:}/data/share1/task/com.tle.erfpoz.apk{|}ProcessName{:}
       app_process{|}OperatorProcessName{:}system_server 329 InstallPackage 5441 5441 0 20170718000003300
2  -1  0  Priority{:}1000{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.net.conn.CONNECTIVITY_CHANGE;
       android.intent.action.ACTION_POWER_CONNECTED;android.intent.action.DATA_CHANGED;android.intent.action
       .USER_PRESENT{|}ClassName{:}com.pz.test.TestReceive{|}OperatorProcessName{:}system_server 329
       DefineReceiver 5443 5443 1 20170718000003300
3  -1  0  Priority{:}1000{|}PackageName{:}com.tle.erfpoz{|}Actions{:}com.diamondsks.jaaakfd.com.mo.action.
       ACTION;android.intent.action.USER_PRESENT{|}ClassName{:}com.ast.sdk.receiver.ReceiverM{|}
       OperatorProcessName{:}system_server 329 DefineReceiver 5443 5443 2 20170718000003300
4  -1  0  Priority{:}2147483647{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.provider.Telephony.
       SMS_RECEIVED{|}ClassName{:}com.mj.jar.pay.InSmsReceiver{|}OperatorProcessName{:}system_server 329
       DefineReceiver 5443 5443 3 20170718000003300
5  -1  0  Priority{:}2147483647{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.provider.Telephony.
       SMS_RECEIVED;SEND_SMS_ACTION1;SEND_SMS_ACTION2;GET_SMS_ACTION;android.intent.action.USER_PRESENT{|}
       ClassName{:}com.android.mtools.MyReceiver{|}OperatorProcessName{:}system_server 329 DefineReceiver
       5443 5443 4 20170718000003300
6  -1  0  Priority{:}0{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.net.conn.CONNECTIVITY_CHANGE;
       android.intent.action.USER_PRESENT;android.intent.action.BOOT_COMPLETED{|}ClassName{:}com.b.ht.JDR{|}
       OperatorProcessName{:}system_server 329 DefineReceiver 5443 5443 5 20170718000003300
7  -1  0  Priority{:}2147483647{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.provider.Telephony.
       SMS_RECEIVED;android.net.conn.CONNECTIVITY_CHANGE;android.intent.action.BATTERY_CHANGED;android.
       intent.action.SIM_STATE_CHANGED;android.intent.action.NOTIFICATION_ADD;android.intent.action.
       SERVICE_STATE;android.intent.action.NOTIFICATION_REMOVE;android.intent.action.NOTIFICATION_UPDATE;
       android.bluetooth.adapter.action.STATE_CHANGED;android.intent.action.ANY_DATA_STATE;android.net.wifi.
       STATE_CHANGE;android.intent.action.BOOT_COMPLETED;android.intent.action.SCREEN_ON;android.intent.
       action.USER_PRESENT{|}ClassName{:}com.door.pay.sdk.sms.SmsReceiver{|}OperatorProcessName{:}
       system_server 329 DefineReceiver 5443 5443 6 20170718000003300
8  -1  0  Priority{:}2147483647{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.intent.action.
       BOOT_COMPLETED;android.intent.action.USER_PRESENT{|}ClassName{:}com.emag.yapz.receiver.BootReceiver
       {|}OperatorProcessName{:}system_server 329 DefineReceiver 5443 5443 7 20170718000003300
9  -1  0  Priority{:}2147483647{|}PackageName{:}com.tle.erfpoz{|}Actions{:}android.provider.Telephony.
       SMS_RECEIVED;android.provider.Telephony.SMS_DELIVER{|}ClassName{:}com.emag.yapz.receiver.
       ZPayReceiver2{|}OperatorProcessName{:}system_server 329 DefineReceiver 5443 5443 8 20170718000003300
```

Code 3.4: App running log dataset raw samples[3].

3.6 Machine Learning

A Random Forest is a machine learning algorithm that integrates multiple trees through the idea of ensemble learning. Its basic unit is a decision tree, as shown in Figure 3.3. A decision tree is a tree structure (can be binary or non-binary). Each non-leaf node represents a test on a feature attribute, and each branch represents the output of the feature attribute within a certain value range, and each leaf node stores a classification result. The process of using a decision tree to make a decision is to start from the root node, test the corresponding feature attributes in the items to be classified, and select the

[3]Copy from https://apichecker.github.io/

[4]Copy from https://apichecker.github.io/

```
[[0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0,
     1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0,
     1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1,
    0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0,
     1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
     0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0]]
```

Code 3.5: A feature vector sample[4].

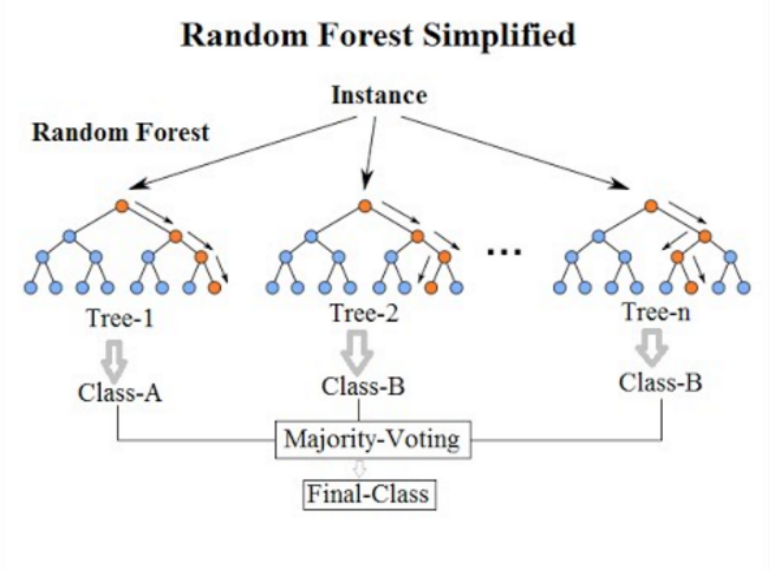


Figure 3.3: Random Forest simplified [63].

output branch according to its value until it reaches the leaf node, and the classification result stored in the leaf node is used as the decision result. The random forest algorithm is essentially an ensemble method, which can effectively reduce over-fitting. It basically is to sample several small sets from the original training dataset, and then train the model on each small set. It takes the average (regression) or vote (classification) of all model outputs.

Code 3.6 shows a code snippet used to implement the above-described Random Forest machine learning algorithm. In the experiments conducted in [33], the hyperparameters of the model are configured based on domain knowledge, which are included in the repository created by the authors of [33], which will be shortly mentioned in Section 3.9.1.

[5]Copy from `https://apichecker.github.io/`

```
    for i in range(1, 9):
        kf = KFold(n_splits=10)
        round = 1
        x = np.array(X)
        y = np.array(y)
        for train_index, test_index in kf.split(X, y):
            round += 1
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]

            if i == 1:
                clf = BernoulliNB()
            elif i == 2:
                clf = RandomForestClassifying(n_estimators=15)
            elif i == 3:
                clf = tree.DecisionTreeClassifier()
            elif i == 4:
                clf = LinearRegression()
            elif i == 5:
                clf = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto')
            elif i == 6:
                clf = svm.SVC(gamma='auto')
            elif i == 7:
                clf = MLPClassifier(hidden_layer_sizes=(100), solver='adam', alpha=0.0001)
            elif i == 8:
                clf = MLPClassifier(hidden_layer_sizes=(100,75,50,20), solver='adam', alpha=0.0001)
            else:
                clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state
                    =0)

            print("Training Classification Model %d", i)
```

Code 3.6: Training Classification Model [5]

3.7 Model Deployment

The above-described ML pipeline is proposed in [33] and is denoted as **APIChecker**. APIChecker has been deployed and running since March 2018. It is able to check around 10k apps every day using a single commodity server with 16 emulators running concurrently on 16 cores. To be specific, it takes an app's APK file as input, which will be installed on an idle Android emulator. Afterwards, APIChecker runs the app and logs all the required information for feature engineering. Lastly, the malice of the app will be determined by the random forest classifier. This whole process consumes 1.92 minutes on average and the app security analysis occupies 1.4 minutes. In a range of 12 months from March 2018 to February 2019, APIChecker detected around 2.4k suspicious apps per month. A detect result sample can be seen in Code 3.7.

The accuracy results indicate that through the 12 months, the per-month precision is over 98% (min: 98.5%, max: 99.0%) and the recall is over 96% (min: 96.5%, max: 97.0%) [33]. In the follow-up work [34], they update the deployment as shown in Figure 3.4 instead of the original monolithic back-to-back execution sequence.

[6]Copy from https://apichecker.github.io/

```
1   1500307357          0  Android 2D689C45E1A19A8AABAD5F159148C4BF 41 20170717000897473    0       com.ctftek
        .screensmanager 384:BsWpJtAIg92SUj5LwJVa6DvOI5RA5adc2tJrH79C5HmMdaSCi4LEr:BsWpoIzdwJ86DGKndc2xPutj4s
        ScreensManager
2   1500307359          0  Android 49EB2793B7D82CCC0B6B1945A5000C66 41 20170717000897610    0       cn.sjjsd.
        jieqianzhinan 192:f2Kk/+J2WPKEXyWZrDSq5YTnYTyNI2Nxu:f2Z/BWyE7TYLNw jieqiangonglue
3   1500307352          0  Android 3BABD0BFCB0083AADA3579F7A4D3DD10 41 20170717000897628    0       net.upper.
        master.support.texas.storm.threw 48:WaNdrVyASXnG1hrX3+3pSr0rkADNlo+ec3w3lCC:WatVhVghqcAcC File Manager
4   1500307357          0  Android CFD7BA9525B1DD19292CB994E6C74259 41 20170717000897828    0       com.android.
        providers.settings 24:W9ELJTEXLfKELsIY5OELa0MWELzKEL8rhELwEL5oELjeqELsfOELY7sVy:
        WaVMDdUZxMRXdwrGHFD3o4fZ87Iy NULL
5   1500307393          0  Android AF61439E6849742D041A73B40FBEFC19 41 20170717000897887    0       com.android.
        backupconfirm 12:WMTEXX6ELQyFavr4QIELi0XuYrKNCMIELMNdNhM6ELrVHELRkteNCqIELhqxa:
        WMTEXX6ELQTWELi8rELYd3hELtELGthe NULL
6   1500307391          0  Android 47880DEBAD50540BA26D1E35ECB50AE6 41 20170717000898022    0       aimoxiu.theme
        .mx596cdb6c9408f43ddc4e52db 96:WHdWKVgdd2XYP44/gecBKB+nyx/10LTzXKxZ3nSYaDr8:a5iSwkXiXdA8 NULL
7   1500307349          0  Android 33451C6FAAFC295D854621108E69318C 41 20170717000898023    0       com.policydm
         192:ip/qkHfoTu3ryu8ICvx/LmtGt589WYagc/:ip/Nfnbyu8vvxDmtk58IYags NULL
8   1500307443          0  Android 61861FCC685D2E3782CAB89B5549D971 41 20170717000898120    0       com.example.
        lab10_1 24:W9ELmTEuW39hnnL8J2a8q8sAg30FRE50K:Wauhs9zyHgC7 lab10_1
9   1500307446          0  Android 5B3F359DA058ADFCE6390BD913E30BB4 41 20170717000898271    0       com.yiguo.
        recordinganimation 6:+AegSkyCNLKttclFr3RwWggFhFshX8EUr4R+vC6iZy4xUqCQlDcqkjDy0sn:WgSky+
        Lgc336WggTEMDr4QvsovqFOqkc NULL
10  1500307443          0  Android AE7A680CB81175AB9107F48FADF054CE 41 20170717000898471    0       com.example.
        exam1 24:W9ELmKcTE3D7gevNWh9nmZsRhIcVTTTaNevpTfqANYoa:WaRU2YKM9B1H5NNFa exam1
11  1500307204          0  Android 584A0A9FA6F4B6CFEAF727D83F5E3747 41 20170717000898587    0       com.qqreader
         768:frwkS+GnlGGbHjjqZk1lfrosk6yZ0CyRYCsVpQaa:frwHlbHjssNrooCy8s QQyuedu
```

Code 3.7: Detect result sample[6].

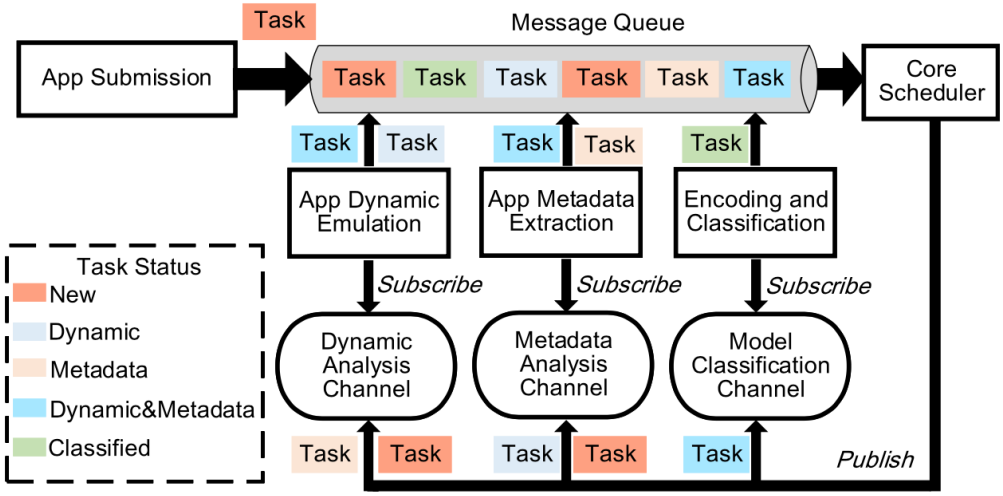


Figure 3.4: APIChecker's distributed deployment in T-Market [34].

3.8 System Evolution

The system needs evolution, which specifically is to update the selected key APIs in a periodic time as one month. There are two reasons, one is new apps are added into T-market's database constantly; the other one is the API base updates themselves since Android SDK has regular update in every several months [33].

3.9 Code, Data, and Other Issues

3.9.1 Dataset and Tool Release

Authors of [33] have released at `https://apichecker.github.io` the list of their selected key Android APIs, part of their analysis logs, and their implemented efficient emulator.

3.9.2 Other Issues

- How robust is APIChecker against sophisticated attackers?
  Based on scanning the source code of Android SDK (level 27), the calculated usage of the selected key APIs can cover up to 5,242 (10.5%) APIs [33], which makes it difficult and arduous for even experienced attackers to escape from detection of APIChecker if not impossible.
- Can we further reduce the number of key APIs?
  Based on the fact that similar detection performance [33] with using all 426 key APIs can be achieved by only tracking the top-150 high-ranking key APIs, it seems feasible to further reduce the number of key APIs. In the meanwhile, the analysis time per app is also reduced.
- Can APIChecker be used by other app markets?
  All the techniques used in APIChecker can be reasonably applied to other app markets.

3.9.3 Related Work

There are quite a few ML-based techniques in the existing works, including but not limited to permission-based footprint detection scheme [96], static analysis [1, 5, 29, 35, 83], dynamic analysis [11, 26, 82, 84]. Due to the lack of practical landing experiences and effectiveness evaluation of such solutions when they are deployed at market scale, the authors of [33] present their experience on constructing and deployment of an ML-based approach to this security problem with a major Android app market, i.e., Tencent App Market.

To make a ML-based malware detection system reliably work at market level without causing significant overhead for average app runtime, the authors of [33] adopt API-centric analysis approach with focus on both API feature selection and app runtime analysis reduction. They provided their innovative key API selection strategies and optimized dynamic analysis engine. Lastly, they commercially deployed their solution and update the ML model in a timely manner.

Chapter 4 AI Detects Abnormal Events in Sequential Data

4.1 The Security Problem

Anomaly detection has been broadly considered in a variety of industries to achieve a greater level of security. Anomaly detection looks for unanticipated changes (i.e., deviations from what is normal) in the observed behavior. When employed in an intrusion detection system (IDS), it suspects and anticipates attacks if outliers are identified.

Suppose that the network activities during a specific time interval are being monitored, and we use e_i to represent the ith activity. Let us use $E = \{e_1, e_2, e_3, ..., e_n\}$ be the collection of all the network activities during a long period of time. Anomaly detector views e_i as an event and $\{e_i, e_{i+1}, ..., e_{i+j}\}$ as a sequence of events.

Usually, anomaly detection has two basic assumptions: a) abnormal event sub-sequences rarely exist in normal behavior; b) their features differ from the normal events significantly. Therefore, security analysts conduct anomaly detection over the dataset E to identify the time intervals (sub-sequences of events) that are anomalous compared to all other time intervals.

4.2 Dataset

We introduce two datasets, UNSW-NB15 [56] and SOSP2009 [87].

4.2.1 The UNSW-NB15 Dataset

The Cyber Range Lab of the Australian Centre for Cyber Security (ACCS) released UNSW-NB15 in 2015. The dataset includes a hybrid of real normal activities and synthetic network attack behaviours which is generated by The IXIA PerfectStorm tool.

First, it should noticed that a flow is a sequence of packets from source IP to destination. The dataset contains 2,218,761 (87.35%) benign flows and 321,283 (12.65%) attack ones, that is, 2,540,044 flows in total. To be more specific, the dataset holds nine types of attacks, namely, Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms which were captured by Tcpdump tool. The packets within the dataset are characterized by 49 different features provided by the Argus packet processor, Bro and twelve additional algorithms.

The mostly used training set (UNSW_NB15_training-set.csv [1]), containing 175,341 flows, and test set (UNSW_NB15_testing-set.csv[2]), containing 82,332 flows, are part of the whole dataset and

[1] https://research.unsw.edu.au/projects/unsw-nb15-dataset

[2] https://research.unsw.edu.au/projects/unsw-nb15-dataset

Table 4.1: UNSW-NB15 data distribution.

Class type	Training samples	Training samples percentage	Testing samples	Testing samples percentage
Normal	56000	31.94	37000	44.94
Analysis	2000	1.14	677	0.82
Backdoors	1746	1.00	583	0.71
DoS	12264	6.99	4089	4.97
Exploits	33393	19.05	11132	13.52
Fuzzers	18184	10.37	6062	7.36
Generic	40000	22.81	18871	22.92
Reconnaissance	10491	5.98	3496	4.25
Shellcode	1133	0.65	378	0.46
Worms	130	0.07	44	0.05
Total	175341	100	82332	100

Table 4.2: HDFS dataset.

System	Training data	Test data	Logs	Number of Log keys	anomalies	normalies
HDFS	4,855 normal 1,638 abnormal	553,366 normal 15,200 abnormal	11,175,629	29	16,838(blocks)	558,223(blocks)

publicly available to help researchers develop data-driven attack detection techniques. Table 4.1 shows the samples for each class and their percentage.

4.2.2 SOSP 2009 Log Dataset

Table 4.2 provides a simple overview of SOSP 2009 Log Dataset which is actually a Hadoop File System (HDFS) console-log dataset. This log set is generated in a private cloud environment using benchmark workloads. The log entries are manually labeled through handcrafted rules to identify the anomalies. Specifically, the log dataset holds over 11 million log entries from Hadoop map-reduce operations that ran on 203 Amazon EC2 nodes over two days. Each log entry has a block identifier (denoted as block *ID*), and each HDFS block may be considered as a thread in its own right.

Each log entry corresponds to a distinct event. And all the events are classified into a set of classes: each class is represented by a **log key** k_i. Furthermore, each *session* is defined as a sequence of events happening to a particular block ID. (No session involves two or more block IDs.) Since no block ID appears in two or more sessions, we can directly label a block ID as normal or abnormal. As shown in Table 4.2, the labels (i.e., normal or abnormal) are provided at the session level, and there are 558,223 normal sessions and 16,838 abnormal sessions. The whole dataset consists of a normal training set (4,855 parsed sessions), a normal testing set (553,366 parsed sessions), an abnormal training set (1,638 parsed sessions) and an abnormal testing set (15,200 parsed sessions).

4.3 Data Processing

4.3.1 SOSP 2009 Log Dataset

Code 4.1 is an example of the raw HDFS log which records system behaviors, source IP address, destination IP address after “INFO" text. “blk_-1608999687919862906" is a block ID. Table 4.3 shows some normal block IDs and some abnormal block IDs.

```
1  081109 203518 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src:
       /10.250.19.102:54106 dest: /10.250.19.102:50010
2  081109 203518 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/
       job_200811092030_0001/job.jar. blk_-1608999687919862906
3  081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src:
       /10.250.10.6:40524 dest: /10.250.10.6:50010
4  081109 203519 145 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src:
       /10.250.14.224:42420 dest: /10.250.14.224:50010
5  081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_-1608999687919862906
       terminating
6  081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_-1608999687919862906
       terminating
7  081109 203519 145 INFO dfs.DataNode$PacketResponder: Received block blk_-1608999687919862906 of size 91178
       from /10.250.10.6
8  081109 203519 145 INFO dfs.DataNode$PacketResponder: Received block blk_-1608999687919862906 of size 91178
       from /10.250.19.102
9  081109 203519 147 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_-1608999687919862906
       terminating
10 081109 203520 145 INFO dfs.DataNode$DataXceiver: Receiving block blk_7503483334202473044 src:
       /10.250.19.102:34232 dest: /10.250.19.102:50010
```

Code 4.1: Example raw data in the HDFS log dataset

Table 4.3: Some normal and abnormal block IDs in the HDFS dataset.

BlockId	Label	BlockId	Label
blk_-3544583377289625738	Anomaly	blk_-50273257731426871	Normal
blk_-9073992586687739851	Normal	blk_4394112519745907149	Normal
blk_7854771516489510256	Normal	blk_3640100967125688321	Normal
blk_1717858812220360316	Normal	blk_-40115644493265216	Normal
blk_-2519617320378473615	Normal	blk_-8531310335568756456	Anomaly

In the data processing phase, we firstly re-crafted the log keys into three key sets $\mathcal{K}_0$ (new base), $\mathcal{K}_1$, and $\mathcal{K}_2$. In total, they hold 31, 101 and 304 log keys, respectively. Since labels are given to sessions and block IDs, it is safe to use log keys to encode each event (i.e., log entry) without modifying the original event sequences. The statistics of each key set under configuration seqlen = 10 is shown in Table 4.4. These key sets are from the same log, except that $\mathcal{K}_1$ and $\mathcal{K}_2$ discard less information by reattaching add-on strings; for example:

$$\begin{aligned} k_i \in \mathcal{K}_0 &: \text{ "Received block"} \\ 1^{\text{st}} \text{ add-on} &:\text{"of size } 20-30\text{MB "} \\ 2^{\text{st}} \text{ add-on} &:\text{"from 10.250.*"} \\ k_j \in \mathcal{K}_1 &:\text{"Received block of size } 20-30\text{MB from 10.250.*"} \end{aligned} \tag{4.1}$$

Table 4.4: Statistics of Event Key Sets under seqlen = 10.

	$\lvert\mathcal{K}_*\rvert$ Size	Normal Patterns	Abnormal Patterns	Undecidable Patterns
$\mathcal{K}_0$	31	13,056	11,099	4,806
$\mathcal{K}_1$	101	220,912	35,662	19,925
$\mathcal{K}_2$	304	1,868,327	103,863	49,856

There are three types of add-on strings:

- For two log keys that each involves a filepath, we attach one of the 32 filepath add-ons (e.g. /user/root/randtxt/temporary/task*/part*).
- For two log keys that each involves a filesize, we attach one of the seven 10-MB interval add-ons (e.g. 0-10 MB).
- For the events that involve an IP address, we attach add-ons from the following rules:
  - If a log entry involves both a source IP address and a destination IP address, we check and attach add-ons that represent whether it is "within the localhost"; if not, we then check whether it is "within the subnet" or "between subnets" by the IP prefixes (e.g. 10.251.7*).
  - If a log entry involves either a source IP address or a destination IP, we attach add-ons that represent directions and IP prefixes (e.g., from 10.251.7*)

The IP-prefix granularity is different for $\mathcal{K}_1$ and $\mathcal{K}_2$: for $\mathcal{K}_1$, it uses the first two decimal numbers (e.g., 10.251.*), and for $\mathcal{K}_2$ it uses just one more decimal number (e.g., 10.251.7*). We split $\mathcal{K}_0$ by attaching add-ons, but we also discard keys that have zero occurrence.

Code 4.2 shows the data processing code snippet.

```
def readSequence(filepath, args):
    seq = []
    for line in open(filepath, 'r'):
        obj = json.loads(line)
        evt = obj['event']
        task = evt['task']

        if args.logkeys != 0:
            if 'filepath_type' in evt: task += ', filepath_type=' + evt['filepath_type']
            if task in [
                'INFO dfs.DataNode$PacketResponder: Received block BLK of size SIZE from IP',
                'INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: DST is added to BLK
                    size '
                'SIZE']:
                if 'size' in evt: task += ', size=' + str(int(evt['size']) // 10000000) + '0 MB'
        if args.logkeys != 0:
            precision = args.logkeys
            # parse IPs
            if 'src_ip' in evt and 'dst_ip' in evt:
                if evt['src_ip'] == evt['dst_ip']:
                    task += ', traffic=localhost'
                elif '.'.join(evt['src_ip'].split('.')[: precision]) == '.'.join(evt['dst_ip'].split('.')[:
                    precision]):
                    task += ', traffic=subnet'
                else:
                    task += ', traffic=intranet'
            elif 'src_ip' in evt:
                src_ip = '.'.join(evt['src_ip'].split('.')[: precision])
                if precision > 2: src_ip = src_ip[: src_ip.rfind('.') + 2]
```

```
            task += ', src_ip=' + src_ip
        elif 'dst_ip' in evt:
            dst_ip = '.'.join(evt['dst_ip'].split('.')[: precision])
            if precision > 2: dst_ip = dst_ip[: dst_ip.rfind('.') + 2]
            task += ', dst_ip=' + dst_ip
    seq.append(task)
return seq
```

Code 4.2: SOSP2009 data processing

For split of training dataset and testing dataset: the training dataset for $\mathcal{K}_0$ and $\mathcal{K}_1$ consists of 200,000 normal sessions, and the training dataset for $\mathcal{K}_2$ consists of 100,000 normal sessions. The number of sessions in dataset for $\mathcal{K}_2$ is smaller due to the computational limitation within our experiment environment. The testing datasets for $\mathcal{K}_0$, $\mathcal{K}_1$, and $\mathcal{K}_2$ each includes all 16,868 abnormal sessions. Beside abnormal sessions, the testing dataset for $\mathcal{K}_0$ and $\mathcal{K}_1$ include 200,000 normal sessions, and the testing dataset for $\mathcal{K}_2$ include 100,000 normal sessions.

4.3.2 UNSW-NB15 Dataset

First, Table 4.5 shows several raw data samples in the UNSW-NB15 dataset.

SRC IP	SRC PORT	DST IP	DST PORT	PROTOCOL	STATUS
59.166.0.0	1390	149.171.126.6	53	udp	CON
10.40.170.2	0	10.40.170.2	0	arp	INT
175.45.176.2	23357	149.171.126.16	80	tcp	FIN
192.168.241.243	259	192.168.241.243	49320	icmp	URH
10.40.182.3	0	10.40.182.3	0	arp	INT

Table 4.5: Some raw data samples in the UNSW-NB15 dataset.

The data processing code snippet is shown in Code 4.3. It extracts source IP, source port, destination IP, and destination port, and then combine them as an event which is loaded into a sequence. More importantly, label and log keys are inevitable. For log keys, some occurrences with values of “-" and “spaces" are removed. Some text types that include numeric values are transformed to number types. The feature column’s median is used to replace null values. Finally, we attach some add-on keys and make it a log key sequence (more details are in “data/unswnb15/key.py").

```
# define how to read sequences from file
def readSequences(ip, filename):
    sequence = {}
    label = {}
    with open(os.path.join(args.input, ip, filename), 'rt') as fin:
        csvfin = csv.reader(fin, delimiter=',')
        for line in csvfin:
            datetime = data.unswnb15.key.getDateTimeFromLine(line)
            srcip = line[data.unswnb15.key.srcip]
            dstip = line[data.unswnb15.key.dstip]
            dstport = line[data.unswnb15.key.dsport]

            subject = '-'.join(['from', srcip, 'to', dstip, 'on', str(datetime.day), str(datetime.hour),
                                str(datetime.minute // args.window_size)])
            slabel = data.unswnb15.key.getLabelFromLine(line)
```

```
            skeystr = data.unswnb15.key.getKeyFromLine(line, args.logkeys, args.key_divisor)
            if subject not in sequence:
                sequence[subject] = list()
            sequence[subject].append(skeystr)
            if subject not in label:
                label[subject] = list()
            label[subject].append(slabel)
    ret = []
    for subject in sequence:
        notNoneLabels = [l for l in label[subject] if l is not None]
        ret.append((
            sequence[subject], subject,
            ','.join(sorted(set(notNoneLabels))), len(notNoneLabels)))
    return ret
```

Code 4.3: UNSW-NB15 data processing

4.4 Model Architecture

This use case applies anomaly detection on discrete events or logs. Inspired by natural language processing (NLP), researchers have recently adapted related NLP to anomaly detection for discrete logs by considering discrete events as words and logs as sentences [47, 48]. Accordingly, a commonly adopted detection strategy is as follows: first, let the model predict upcoming log events; second, if the predicted events are adequately different from the observed events (in the real system), raise intrusion *alerts* against the observed events.

However, it is observed in [93] that this strategy might not be able to fully exploit distinctive characteristics of a sequence, because an operation is affected not only by previous operations, but also by later events. Therefore, this use case will utilize a bi-directional LSTM model proposed in [93] to help security analysts achieve higher accuracy in anomaly detection.

In a nutshell, the bi-directional LSTM model consists of four major components (see Figure 4.1): an embedding layer, a deep LSTM autoencoder, an event classifier, and an anomaly critic. The embedding layer E embeds a sequence of events S into an embedded distribution X; the autoencoder analyzes (encodes) X and reconstructs (decodes) the categorical logit distribution Y; the event classifier transforms Y into a categorical probability distribution P; and the critic compares P to S and reports whether S is normal or abnormal.

4.4.1 Bi-directional LSTM Network Architecture

Different from LSTM, a bi-directional LSTM contains a forward and backward layer which connects to the same output layer. Subsequently, the concatenation of each output from one LSTM makes up the Bi-LSTM and the concatenated vector will be passed to the next LSTM layer. Finally, the last linear layer will have activation function such as softmax function. The architecture of Bi-LSTM is shown in Figure 4.2.

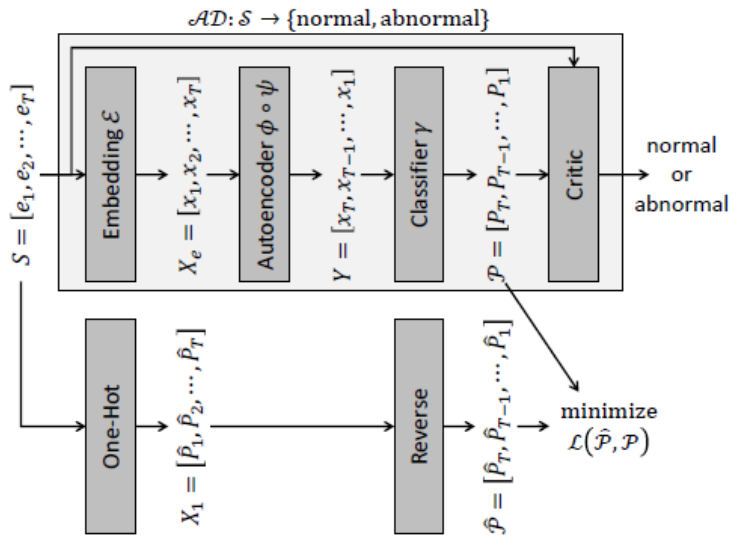


Figure 4.1: Architectural overview of DabLog [93].

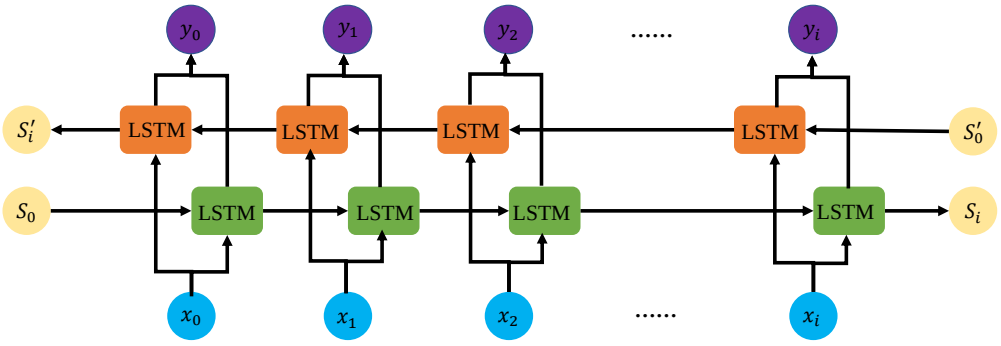


Figure 4.2: Architectural overview of BiLSTM.

4.4.2 Embedding Layer

Since we take a sequence of discrete event $S = [e_t \mid 1 \leq t \leq T]$ as input, we need to embed $e_t \in \mathcal{K}$ into an embedding vector that the model could recognize, where $\mathcal{K} = \{k_i \mid 1 \leq k \leq V\}$ is the set of discrete event keys of vocabulary size $V = |\mathcal{K}|$. In addition to discrete log key $\mathcal{K}$, we also add three special padding keys: begin-of-sequence, end-of-sequence, and unknown (shown in Code 4.4 which is implemented by Codebook class in file “models/util.py"). On one hand, begin-of-sequence and end-of-sequence provide additional characteristics to help autoencoders and predictors. On the other hand, the unknown key is to improve computational performance.

```
class Codebook(object):
    PAD = '<pad>'
    BEGIN = '<sequence>'
    END = '</sequence>'
    UNKNOWN = '<unknown>'

    def __init__(self, codebook=None, threshold=0.01):
        self.encode, self.decode, self.types = None, None, None
        if codebook is not None:
            if isinstance(codebook, dict):
                codebook = set(codebook.keys())
            if isinstance(codebook, list):
                codebook = set(codebook)
            if isinstance(codebook, set):
                for preserved in [self.PAD, self.BEGIN, self.END, self.UNKNOWN]:
                    if preserved in codebook:
                        codebook.remove(preserved)
                self.types = set(codebook)
                self.decode = {i: k for i, k in enumerate([self.PAD, self.BEGIN, self.END, self.UNKNOWN] +
                    sorted(self.types))}
                self.encode = {k: i for i, k in self.decode.items()}
                self.types = codebook | {self.PAD, self.BEGIN, self.END, self.UNKNOWN}
```

Code 4.4: Codebook

We train an additional embedding layer along with other layers instead of utilizing other existing embedding approaches to generate word embedding vectors since the embedding function can be customized in this way (shown in Code 4.5 which is implemented by Dablog class in file “models/dablog.py").

```
self.label_input = keras.layers.Input(shape=(None,), name='Label_Input')
self.label_embed = keras.layers.Embedding(self.n_labels, self.config.hidden_unit, mask_zero=True, name='
    Label_Embed')
self.label_dense = self.label_embed(self.label_input)
```

Code 4.5: Embedding Layer Code Implementation

4.4.3 Deep LSTM Autoencoder

Different from a common deep autoencoder which learns the identity function of the normal data and reconstructs the normal data distribution, the deep LSTM autoencoder proposed in [37] tries to reconstruct the logit inputs of categorical events from the embedding layer.

To tackle time-sensitive events, the objective function of this autoencoder also follows the typical deep autoencoder. We set X is the input distribution, $\psi \circ \phi(X)$ is the target distribution so the loss function can be written like:

$$\begin{aligned} \phi, \psi &= \arg\min_{\phi,\psi} \|X - Y\|^2 \\ &= \arg\min_{\phi,\psi} \|\text{rev}\,(X_e) - (\psi \circ \phi)\,(X_e)\|^2 \\ &= \arg\min_{\phi,\psi} \|(\text{rev} \circ \mathcal{E})(S) - (\psi \circ \phi \circ \mathcal{E})(S)\|^2 \end{aligned} \tag{4.2}$$

where ϕ is the encoder function, ψ is the decoder function, $\mathcal{E}$ is the embedding function which maps S to X, and rev is a function that reverses the distribution matrix.

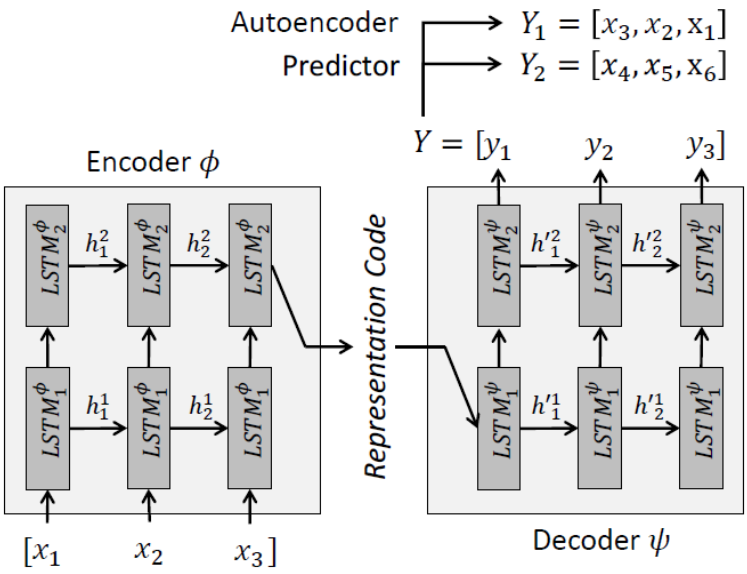


Figure 4.3: Deep LSTM Autoencoder Network [93].

The reason why we use rev function is that Y is in reverse order from X_e and due to LSTM's hidden state. As illustrated in Figure 4.3, we can simply represent LSTM network by:

$$h_t = \text{LSTM}\,(x_t, h_{t-1}) \tag{4.3}$$

where h_t is the hidden states at time step t, and x_t is the current data point. As we could see, the LSTM iteratively calculates this function from step 1 to the last time step. Similar to function stack, we can think this encoder procedure like pushing x_t into a stack h_T, decoder procedure is popping x_t out from a stack h_T. Therefore, X_e and Y are in reverse order.

This autoencoder is not conditional which means that it does not provide a condition data point $\hat{y}_\tau = e_{T-\tau+1}$ to the decoder ϕ when it is decoding $\hat{y}_{\tau+1}$ for any $\hat{y}_\tau$ in $Y = [y_\tau \mid 1 \leq \tau \leq T]$ though conditional predictors can provide better results. First, conditional autoencoder acts as a hint which tells predictors which suffix should be decoded, whereas provides no additional purpose for the autoencoder. Second, because adjacent events normally have significant short-term dependencies, it is not optimal to offer a condition that causes the model to quickly pick up short-term dependencies but ignore long-term connections. Code 4.6 shows a code snippet implementing the autoencoder.

```
1 elif mode == 'autoencoder':
2         # autoencoder layer
```

```
        nn = keras.models.Sequential(name='RNN')
        nn_sizes = [int(nn_size / i) for i in range(1, self.config.hidden_layer + 1)]
        nn_sizes = [] + nn_sizes + nn_sizes[:: -1]
        for i in range(1, self.config.hidden_layer):
            nn.add(keras.layers.LSTM(nn_sizes[i], activation='relu', return_sequences=True, name='LSTM_' + str
                (i)))
        i = self.config.hidden_layer
        if self.config.use_repeat_vector: # bi-direction
            nn.add(keras.layers.LSTM(nn_sizes[i], activation='relu', return_sequences=False, name='LSTM_' +
                str(i)))
            nn.add(keras.layers.RepeatVector(seqlen))
        else:
            nn.add(keras.layers.LSTM(nn_sizes[i], activation='relu', return_sequences=True, name='LSTM_' + str
                (i)))

        for i in range(self.config.hidden_layer + 1, 2 * self.config.hidden_layer + 1):
            nn.add(keras.layers.LSTM(nn_sizes[i], activation='relu', return_sequences=True, name='LSTM_' + str
                (i)))
        nn.add(keras.layers.TimeDistributed(keras.layers.Dense(n_labels, activation='softmax'), name='Softmax'
            ))
        # model
        if n_floats == 0:
            model = keras.models.Model(inputs=label_input, outputs=nn(label_dense))
        else:
            model = keras.models.Model(inputs=[label_input, float_input], outputs=nn(merge))
        model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])
```

Code 4.6: Autoencoder

4.4.4 Event Classifier and Anomaly Critic

With the embedding layer and the autoencoder, it is still not enough to achieve our goal: predict a sequence whether normal or abnormal. So right after the deep autoencoder, we add an additional single layer feed forward neural network γ for prediction. The layer γ acts a multi-class classifier that takes input Y which is the output of decoder and generates a probabilistic matrix $\mathcal{P} = [P_\tau \mid 1 \leq \tau \leq T]$, where $\mathcal{P}_\tau = [P_i \mid 1 \leq i \leq V]$ and Pi can be interpreted as likelihood of the discrete event $e_t = e_{T-\tau+1}$ being an instance of discrete event key k_i. We choose γ as an event classifier rather than a scalar reconstruction error used in some typical anomaly detection autoencoders since identifying time-sensitive anomaly by examining discrete events is more like a language processing problem. What I mean is that we can view sequence S as sentences, events e_t as words so we care more about wording than embedding to find fitting words and unfitting words, which are equivalent to normal events and abnormal events in S.

In order to train γ, first we generate true probabilistic matrix via one-hot representation of S denoted as $X_1 = onehot(S) = \left[\hat{P}_t \mid 1 \leq t \leq T\right]$, where $\hat{P}_t = [\hat{p}_i \mid 1 \leq i \leq V]$, and $\hat{p}_i \in [0, 1]$ and $\sum_i \hat{p}_i = 1$.Second, similar to embedding layer, we also include begin-of-sequence, end-of-sequence, and unknown in the one-hot function. Note that the probabilistic matrix $\hat{P}$ generated from one-hot representation is in reverse order with X. Therefore, the embedding-encoder-decoder-classifier network tries to minimize the function:

Table 4.6: Parameter details of DabLog.

Layer (type)	Output Shape	Parameter
Label_Input (InputLayer)	[(None, None)]	0
Label_Embed (Embedding)	(None, None, 64)	13696
RNN (Sequential)	(None, 10, 214)	92502

$$\mathcal{E}, \phi, \psi, \gamma = \underset{\mathcal{E},\phi,\psi,\gamma}{\arg\min} \| (\text{rev} \circ \text{onehot}\ )(S) - (\gamma \circ \psi \circ \phi \circ \mathcal{E})(S) \| \tag{4.4}$$

Narrowly, the objective function of this event classifier γ is based on categorical cross entropy loss defined by this formula:

$$\mathcal{L}(\hat{\mathcal{P}}, \mathcal{P}) = \sum_{\tau}^{T} L\left(x_{T-\tau+1}, P_{\tau}\right) = -\sum_{\tau}^{T} \sum_{i}^{V} \hat{p}_i \times \log\left(p_i\right) \tag{4.5}$$

For the wording options in anomaly detection, rank-based and threshold-based are two common ones adopted by previous work. In order to compare with the existing work, DabLog adopted rank-based criterion, though it has drawback which will be discussed in the last part. Say a discrete event e_t is an instance of $\hat{k}_i$, a rank-based criterion will consider e_t anomalous if p_i is not in top-N prediction (as shown in Figure 4.4).

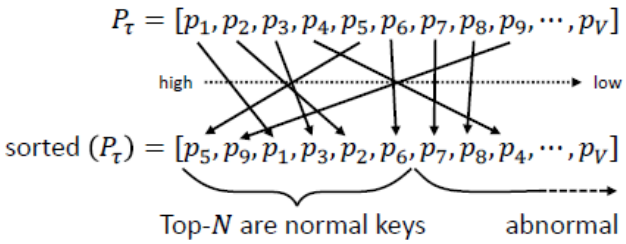


Figure 4.4: An example of the rank-based criterion[93].

4.5 Hyperparameter Tuning

There are two settings of hyperparameters including dataset tuning and neural network hyperparameters. For dataset processing, we have mentioned in Data processing section, seqlen (sequence length) is set as 10 default, and developers can choose different log keys according to the scenario, 0 means 32, 1 means 78, 2 denotes 104, and 3 denotes 1129. The parameter windows size specifies the number of minutes for each sequence and we default set it as 15. The key divisor denotes the number of key divisor which is default set as 100.

For model hyperparameters, we have 64 hidden units for every layer, two hidden layers, and batch size is 256. More details related to hyperparameters are shown below. For anomaly criterion, rank threshold is 0.05 and distance threshold is 0.05.

Table 4.7: Parameter details of RNN in DabLog.

Layer (type)	Output Shape	Parameter
LSTM_1 (LSTM)	(None, None, 64)	33024
LSTM_2 (LSTM)	(None, 32)	12416
Repeat vector (RepeatVector)	(None, 10, 32)	92502
LSTM_3 (LSTM)	(None, 10, 32)	8320
LSTM_4 (LSTM)	(None, 10, 64)	24832
Softmax (TimeDistributed)	(None, 10, 214)	13910

4.6 Model Deployment

Ideally, the deployed model can determine whether an event and a sequence is abnormal or normal in real-time. However, there are several factors that could make model deployment quite challenging.

- The first factor is that the model throughput may or may not meet the real-time requirements. To met all the real-time requirements, the production system needs to choose a platform (e.g., CPU, GPU, memory) that provides adequate computational resources. In addition, it should be noticed that containerization is a useful technique for deploying DL models. Because containers (e.g. docker) make scaling easy, it's a popular platform for model deployment. Containerized code also makes it simpler to update the deployed model. This reduces the likelihood of downtime and improves maintenance efficiency.
- The second factor is whether there is a need to deploy the model at an edge device. If so, a lightweight model execution environment, such as TensorFlow lite[3], is often required.

4.7 Evaluation

We leverage accuracy and F1 Score to compare the performance of different models.

4.7.1 Comparison between DabLog and Baseline

First, we want to examine how DabLog is better than the baseline predictor. As for baseline, it is similar to DeepLog[23] and nLSALog[89] which is implemented by two-layered LSTM network, a multi-class classifier, and a rank-based critic as shown in Figure 4.5. Similar to DabLog, we did not directly use one-hot representation instead we train a customized embedding layer. Just like DeepLog and nLSALog, the event classifier is activated by softmax function and they both utilize rank-based critic to evaluate whether a sample is normal or abnormal.

Figure 4.6 and Figure 4.7 illustrate the F1 score trends of baseline and DabLog model based on different keys. The x-axis represents the variable ranking threshold in a normalized form $\theta_N = \frac{N}{|\mathcal{K}_*|} \times 100\%$. It is equivalent that use Top-N threshold out of $|\mathcal{K}_*|$. The Y-axis represents F1 score of the models. Besides DabLog and Baseline, we also include a trivial frequency model, reports anomalous sequences by checking whether a sequence includes any event that is not an instance of top-N most frequent keys.

[3] https://www.tensorflow.org/lite

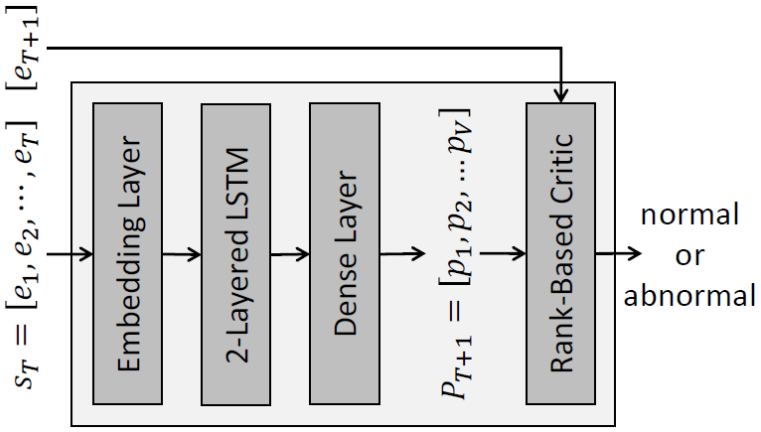
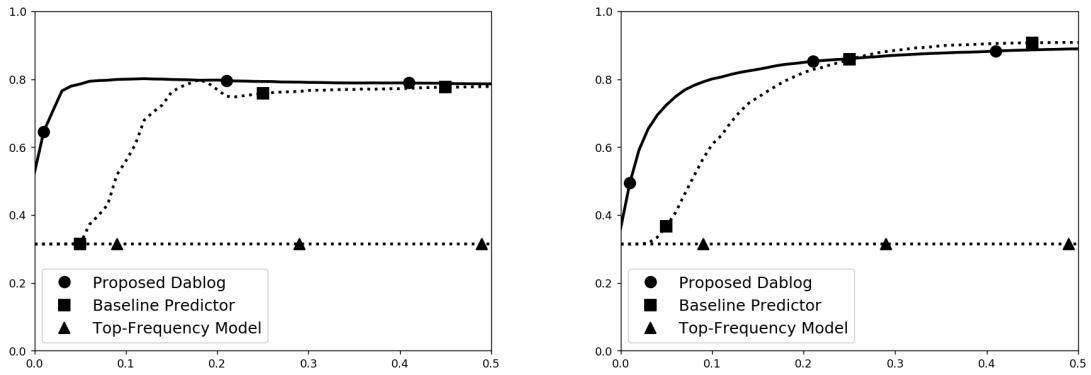


Figure 4.5: Baseline Predictor-Based Model[93].

For SOSP2009 dataset, from Figure 4.7a, Baseline has its peak F1 score 87.32% at θ_N = 10%, DabLog reaches its peak F1 score 97.18% at θ_N = 9%. By comparing the trends, it could easily get that DabLog has a higher peak and wider range hence DabLog is more advantageous for critics than Baseline and other two: DeepLog, nLSALog. Similar as Figure 4.7b, DabLog has its peak F1 score 94.15% at θ_N = 6%, whereas baseline only achieves F1 score 80.47% at θ_N = 6%.

For UNSW-NB15 dataset, from Figure 4.6a and Figure 4.6b, both Baseline and DabLog have a smooth curve and wide plateu which means UNSW-NB15 dataset has less infrequent data point. Besides, they both achieve similar F1 score but DabLog still has a slight higher F1 score. Therefore, above the comparisons, DabLog is more advantageous than Baseline.



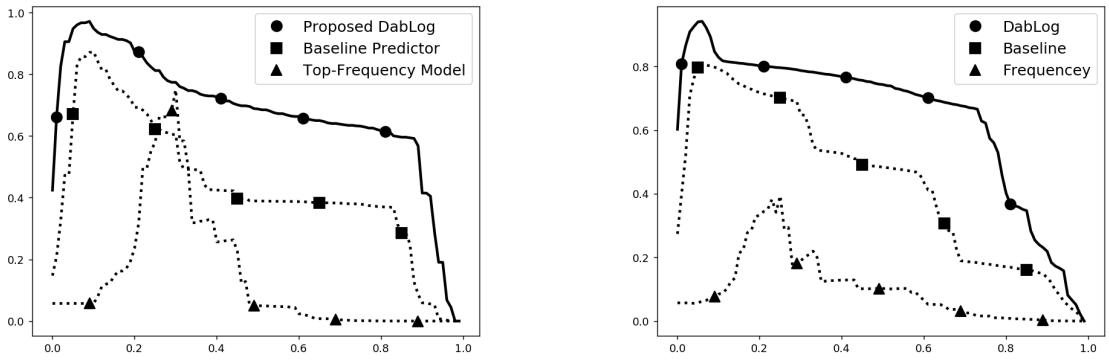
(a) 366 keys upon UNSW-NB15 dataset. **(b)** 706 keys upon UNSW-NB15 dataset.

Figure 4.6: Comparison between baseline and DabLog trained on UNSW-NB15.

4.7.2 Benefit of Bi-directional LSTM

Although the results show that DabLog is more advantageous, it is still a little abstract to understand the fundamental reasons. So in this part, we aim to introduce the fundamental difference between DabLog, the bidirectional one and baseline and other non-bidirectional ones.

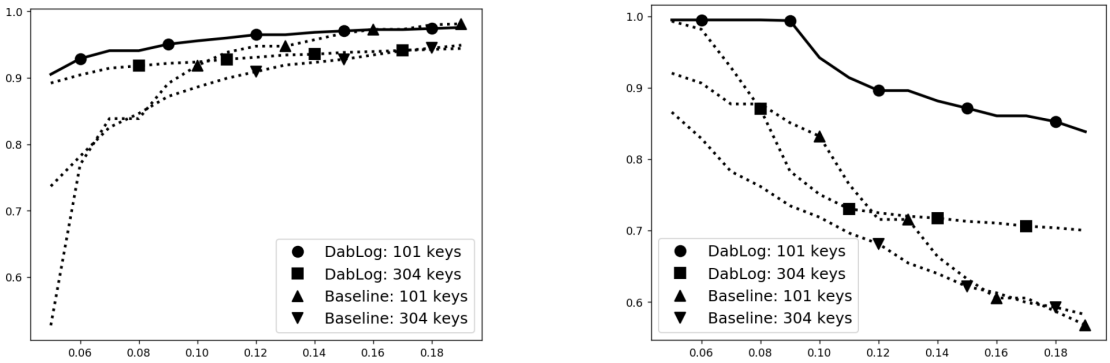
First, we illustrate that Dablog has a better performance in precision than baseline upon 101 keys and 304 keys (shown in Figure 4.8a). That shows DabLog is more advantageous than baseline since



(a) 101 keys upon SOSP2009 dataset. **(b)** 304 keys upon SOSP2009 dataset.

Figure 4.7: Comparison between baseline and DabLog trained on SOSP2009.

higher precision means lower false positive rates. And Figure 4.8b shows that while recall rate of Dablog and baseline both decrease with θ_N, Dablog decreases slower since baseline cannot identify structural abnormal sequences as it would have more FNs. And We will explain this in the later part.



(a) Precision upon different keys. **(b)** Recall rate upon different keys.

Figure 4.8: Comparison between baseline and DabLog trained on SOSP2009 dataset.

As we said some frameworks like baseline, DeepLog, and nLSALog cannot idetify structural abnormal sequences, we would like to borrow the example from DabLog to explain it.

We select such a case that DabLog reports an abnormal case as abnormal, whereas baseline report it as normal. In this session, it has total 34 events which are listed in Table 4.8. DabLog reported the subsequences s_{23}, s_{28}, s_{29}, and s_{30} abnormal, where $s_{23} = [e_{14}, \ldots, e_{23}]$ and $s_{30} = [e_{21}, \ldots, e_{30}]$.

These subsequences are considered abnormal because DabLog could not correctly reconstruct particularly the 21st event e_{21}. That is, the key k_3 is not within the top- 9% reconstructions for e_{21}. Top- 9% reconstructions include k_4, variants of k_5, variant of k_6="add StoredBlock: blockMap updated ...", and k_7="EXCEPTION: block is not valid ...". These event keys, except k_7, are frequent keys each dominates over 0.1% of the dataset. Interestingly, here DabLog expects not only frequent keys, but also an extremely rare event key k_7 (which dominates 0.0017%) before k_5. Since these expected keys in top- 9% reconstructions for e_{21} are related to exception, verification, or blockMap updates, we believe that the reconstruction distribution is derived for causality relationship with e_{23} (which is related block transmission) rather than for e_{19} (which is related to block deletion), even though DabLog knows a deletion is asked at e_{19} as it has correctly reconstructed e_{19}. Our interpretation is

that, DabLog expects a cause at e_{21} that leads to the exception at e_{23}, and it is the absence of causality before e_{23} making the sequence structurally abnormal.

In contrast, Baseline does not predict e_{21} to be any of these keys after $s_{20} = [e_{11}, \ldots, e_{20}]$. In other words, Baseline does not expect a cause at e_{21}, because it cannot foresee $e_{23} = k_5$. With the fundamental limitation of unable to exploit bi-directional causality, Baseline is incapable of detecting such a structurally abnormal session. Therefore, we believe it is necessary for an anomaly detection methodology to see sequences as atomic instances and examine the bi-directional causality as well as the structure within a sequence. Single-direction anomaly detection like Baseline cannot identify structurally abnormal sequences.

Table 4.8: Example of abnormal sequence events.

e_{14}		Starting thread to transfer block
e_{15}		Receiving block within the localhost
e_{16}		blockMap updated: $10.251.*$ added of size $60 - 70$MB
e_{17}		Received block within the localhost
e_{18}		Transmitted block within the subnet
e_{19}	k_1	ask 10.251.* to delete block(s)
e_{20}	k_2	blockMap updated: 10.251.* added of size 60-70 MB
e_{21}	k_3	Deleting block /mnt/hadoop/dfs/data/current/...
e_{22}	k_4	Verification succeeded for block
e_{23}	k_5	Got exception while serving block within the subnet
e_{24}		Got exception while serving block within the subnet
e_{25}		Verification succeeded for block
e_{26}		delete block on 10.251.*
e_{27}		delete block on 10.251.*
e_{28}		delete block on 10.251.*
e_{29}		ask 10.251.* to delete block(s)
e_{30}		Deleting block /mnt/hadoop/dfs/data/current/...

4.8 Code, Data, and Other Issues

We have made our code public on GitHub[4].

4.8.1 Additional Approaches and Datasets

Autoencoder is not the only approach to tackle anomaly detection, some other sequence reconstruction solutions such as combining predictions from predictors of different directions could also work well. We also provide a curated list of anomaly detection projects on our GitHub page[5]. Furthermore, additional datasets should be helpful. DabLog requires datasets that include the relationships among low-level events and detailed events, such as system call events. Therefore, we provide a dataset database: loghub[6] which contains a collection of log events.

[4] https://github.com/PSUCyberSecurityLab/AIforCybersecurity/tree/main/Chapter4-BiLSTM-For-Anomaly_detection

[5] https://github.com/PSUCyberSecurityLab/AIforCybersecurity/tree/main/Chapter4-BiLSTM-For-Anomaly_detection#misc

[6] https://github.com/logpai/loghub

Table 4.9: Experiments upon $\mathcal{K}_1$ and with θ_N = 7.5%.

	seqlen = 10	seqlen = 30	Intersection	Union
TP	16,367	14,910	14,630	16,647
FP	2,424	1,224	1,059	2,589
TN	197,576	198,776	198,941	197,411
FN	471	1,928	2,208	191
FP Rate	1.21%	0.61%	0.52%	1.29%
Recall	97.20%	88.54%	86.88%	98.86%
Precision	87.10%	92.41%	93.25%	86.54%
F_1 Score	91.87%	90.44%	89.95%	92.29%

4.8.2 Combining Different Models

There are many different ideas on how to build an anomaly detection model, and one may want to combine the advantages of different models. As a result, a hybrid model could be constructed. From Table 4.9, we can see that the intersection of the outputs of two models could reduce the false positive rate. Moreover, composite models such as ensemble learning could assist us to further improve the overall DL performance.

4.8.3 Embedding Layer

Although our embedding layer learns the embedding function γ by training an additional embedding layer, it still has the drawback of handling unknown event keys. For instance, if an event key is not in the training data, then it will be seen as unknown keys which might be wrongly judged as anomaly event. There are two common solutions to this issue. One is that we add rare events so that unknown is trained even if it is a rare event. However, in security applications, unknown events are always frequent and wrongly treat unknown events as rare events might cause high FPs. The other option is to leverage a trained embedding layer such as Mimic embedding[60] to build an event embedding model to map from event to embedding.

4.8.4 Prediction Criterion

Rank-based criterion and threshold-based criterion are two common wording options adopted by previous anomaly detection methods. However, both rank-based and threshold-based criteria brutally divide the probabilistic distribution into normal and abnormal, while omitting the relationship between the event keys. The problem is that the critic may wrongly see a normal key as abnormal and vice versa. So we are wondering if we could design such a criterion: to incorporate the embedding distance with rank-based criterion, which means we have both top-N threshold and also checks the labels of neighbors within a region.

Chapter 5 AI Detects DNS Cache Poisoning Attack

5.1 The Security Problem

This use case intends to demonstrate the basic and classic application of deep learning for detecting network attacks. In this use case, the network attack in question is DNS cache poisoning attack. A major functionality of Domain Name Service (DNS) is to provide the mapping between the domain names and IP addresses. When a client program on a user machine refers to a domain name, the domain name needs to be translated to an IP address for network communication. The DNS servers are responsible to perform such translation.

The distributed Internet-wide DNS system has a hierarchical structure that contains root name servers, top-level domain name servers, and authoritative name servers. Some examples are the public DNS servers `8.8.8.8` and `8.8.4.4` provided by Google, and recently released `1.1.1.1` by Cloudflare. These name servers are referred as the global DNS servers. Due to the geological distance between user machines and the global DNS servers, it is very costly to contact the global DNS servers every time when certain mapping information is needed. To reduce the cost, organizations often deploy their own DNS servers, referred as **local DNS servers**, within the LAN to cache the most commonly used mappings between domain names and IP addresses. Generally, when a user machine needs to make connection with a destination machine, it will contact the local DNS server first to resolve the domain name. If the local DNS server does not cache the DNS record for this domain name, it will send out a DNS query to a global DNS server to get the answer for the user machine. The user machine gets to know the IP address after receiving the response.

DNS cache poisoning attack can target local DNS servers. When the local DNS server receives a query which it does not have the corresponding records (first stage), it will inquire the global DNS server (second stage). On receiving the response (third stage), the local DNS server will save this record in its **cache**, and forward the response to the user machine (fourth stage). However, the DNS server cannot verify the response at the third stage, and this is where the attacker can fool the local DNS server. Pretending as the global DNS server, the attacker can send a spoofed DNS response to the local DNS server with the falsified DNS record. As long as the fake response arrives earlier than the real one, the local DNS server will forward it to the user machine and save the falsified record to its cache, as illustrated in Figure 5.1. When new queries about the same domain name come in, the local DNS server will not send a query to the global DNS server again because the corresponding record has been cached. Consequently, it will answer the user machine with the falsified record, until the record expires or the cache is flushed.

DNS cache poisoning attack is difficult to detect with traditional approaches like signature-based and rule-based intrusion detection, because attackers intentionally craft

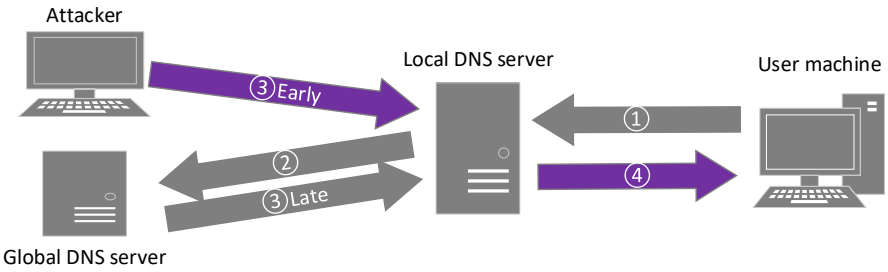


Figure 5.1: DNS cache poisoning illustration.

their packets to follow the protocol and make them look very similar to genuine DNS packets. Accordingly, there are substantial incentives to apply machine learning to detect this attack. However, DNS cache poisoning attack is a session-based attack. Attack packets have to be crafted based on user/server packets, and the attack may need one or more attack packets. As a result, each data sample needs to contain information from multiple packets, which makes feature engineering a daunting barrier.

In this use case, demonstrated in a prior work [97], deep learning is applied to help cross the barrier.

5.2 Raw Data Generation and Collection

In this use case, the raw data, both benign and malicious, is generated by the original authors themselves, but not from an existing public dataset, because no public dataset offers network packet data for DNS cache poisoning at the time. For full details, please refer to the original paper [97].

The authors leverage protocol fuzzing to mutate the contents of network packets, specifically, the values of some fields in the packets. Using protocol fuzzing for data generation has the following benefits: 1) With protocol fuzzing, a large variety of malicious network packets for a chosen network attack can be generated at a fast speed. 2) Since the network packets are all generated from the chosen network attacks, they can be labeled as malicious packets automatically without many human efforts. 3) Protocol fuzzing can generate data with more variations than real world data, or even data that are not yet observed in real world. 4) Protocol fuzzing can generate and cover malicious data samples which may otherwise be overlooked when applying deep learning. In deep learning, the changed values for the fuzzing fields may make the malicious data samples misclassified as benign. With protocol fuzzing, if the malicious data are generated in attacks, they will be labeled as malicious automatically, so they will not be omitted in the malicious data sets. In addition, the above-mentioned merits remain when protocol fuzzing is leveraged to generate the needed benign network packets.

It should be noted that the fuzzing is done in a way that the packet's validity and the session's validity are not harmed. To preserve validity at the packet level, certain fields that affect the packets' validity are never fuzzed, such as fields of checksum values and packet lengths. The values of those fields should not be arbitrarily changed. When fuzzing other fields, their values should always be within their valid ranges. For example, the most basic valid range for a field of one byte is $[0, 1, 2, \ldots, 255]$. To preserve validity at the session level, when choosing the fields to be fuzzed, only fields which can keep the attack session complete and successful are chosen.

The testbed contains three machines: a local DNS server, a user machine, and an attacker machine. The user machine is configured to contact the local DNS server to resolve domain names.

In the malicious scenario, the user machine is to ask for the IP address of one specific domain name from the local DNS server using command dig. The domain name is one that does not have a corresponding record on the local DNS server, thus enabling the DNS cache poisoning attack towards it. The attacker machine sniffs for DNS queries with that specific domain name sent out from the local DNS server, and responds them with fuzzed DNS responses with falsified IP addresses. Then, the DNS cache gets poisoned, and the user machine gets the falsified DNS record. The user machine is to send out DNS queries periodically, so that the above process happens lots times and a large amount of data can be generated. However, as discussed earlier, if the local DNS server has the record for the domain name in its cache, it will not send out DNS queries for it, which is why the DNS cache of the local DNS server is flushed periodically, so that it remains vulnerable in different iterations. If the attack is successful, the falsified IP addresses can be seen on the results of dig.

In the benign scenario, a list containing 4098 domain names is prepared. During each iteration, the user machine randomly chooses one domain name from the list, and asks the local DNS server for its IP address. To resemble the malicious scenario, the cache of local DNS server is also flushed periodically so that the local DNS server always needs to communicate with the global DNS server.

This use case uses network traffic data, which is network logs. The packet capturing happens at the victim's side, that is, the local DNS server.

Table 5.1 shows several examples of the captured DNS cache poisoning packets. All packets shown are from one DNS cache poisoning session. `192.168.100.128` represents the user machine, and `192.168.100.50` represents the local DNS server. Packet 1 shows the user machines asks the local DNS server for DNS record of `www.example.net`; packets 2 to 5 are local DNS server's query packets to the global DNS server; packet 6 is the attack packet sent to the local DNS server, whose IP address is spoofed to be that of the global DNS server, and the DNS record is also falsified; packet 7 is local DNS server's response to the user machine after the (spoofed) response is received from the global DNS server.

Table 5.1: Sample captured DNS cache poisoning session packets.

No.	Time (s)	Source	Destination	Info
1	0.000000000	192.168.100.128	192.168.100.50	Standard query A www.example.net OPT
2	0.000577082	192.168.100.50	192.33.4.12	Standard query NS <Root>OPT
3	0.000616206	192.168.100.50	192.33.4.12	Standard query A www.example.net OPT
4	0.000672609	192.168.100.50	192.33.4.12	Standard query AAAA E.ROOT-SERVERS.NET OPT
5	0.000712695	192.168.100.50	192.33.4.12	Standard query AAAA G.ROOT-SERVERS.NET OPT
6	0.019987042	192.33.4.12	192.168.100.50	Standard query response A www.example.net A 10.0.2.5 NS ns1.example.net NS ns2.example.net A 1.2.3.4 A 5.6.7.8
7	0.020299614	192.168.100.50	192.168.100.128	Standard query response A www.example.net A 10.0.2.5 NS ns1.example.net NS ns2.example.net A 1.2.3.4 A 5.6.7.8 OPT

5.3 Labeling DNS Sessions

This use case aims to train a DL model to classify DNS sessions into “malicious" or “benign". To achieve this, a labeled data set is needed. Since raw data has been collected in the previous section, this section will focus on labeling the collected data.

This attack is session-based. Each session, as shown in Table 5.1, is marked by the DNS query packet sent by the user machine. The malicious impact can only be generated when the attack session is complete. Therefore, the label is also assigned based on session: if the session is a successful DNS cache poisoning attack, then the session is labeled as malicious; otherwise, the session is labeled as benign.

Luckily, the malicious and benign raw data in this use case have already been separated. The malicious and benign data are generated and collected separately. The benign raw data does not have malicious sessions involved, and the malicious raw data does not have unsuccessful attacks or benign sessions involved. Therefore, during data processing, the benign and malicious raw data can be processed and labeled separately.

5.4 Feature Extraction and Data Sample Representation

Network packets from DNS cache poisoning attack form sessions which consist of queries and answers. Therefore, each data sample should include information from multiple network packets. The intuitive idea is to let a data sample hold every byte in every packet in a session. However, this is a bad idea. On one hand, without any feature extraction, the data sample can contain sensitive private information that may harm privacy, like the domain to query. On the other hand, the model may learn what are not intended to be learned, like the media access control (MAC) address of machines, which are better treated as signatures.

```
def process_malicious(malicious):
    malicious_bytes=list()
    i=0
    j=-1
    for item in tqdm(malicious,ascii=True,desc="Processing malicious data"):
        cap=pyshark.FileCapture(item,display_filter='dns and not tcp',use_json=True,include_raw=True)
        try:
            while(True):
                pkt=cap.next()
                if pkt.ip.src_host=='192.168.100.18' and pkt.ip.dst_host=='192.168.100.50': # chop by session
                    malicious_bytes.append([])
                    j+=1
                malicious_bytes[j].append(np.array(list(int(ele) for ele in pkt.get_raw_packet()[14:26]+pkt.
                    get_raw_packet()[34:54]))) # select bytes of interest
                i+=1
        except StopIteration:
            pass
        cap.close()

    np.save(os.path.join('data','malicious_bytes.npy'),np.array(malicious_bytes,dtype= object),allow_pickle=
        True)
```

Code 5.1: Processing malicious raw data.

Therefore, instead of using the whole packet, 32 bytes are chosen from every DNS packet, as listed in Table 5.2. All bytes in the lowest ETH layer are omitted. They are excluded purposefully to rule out the impact of MAC addresses: the MAC address may be treated as signatures to detect malicious packets. The chosen 32 bytes include bytes in the IP layer, the UDP layer, and part of the DNS layer. Only part of DNS layer data is used because the query and records are excluded. Those sections contain the domain names and IP addresses, which are excluded for similar reasons as above: it is not desired that the neural network learns the "malicious" domain. After the packet data processing, every packet is represented as a fixed-length sequence of 32 integer numbers ranging from 0 to 255. The whole captured network traffic is represented as a sequence of 32-integer vectors, and the sequence length is equal to the number of packets in the network log.

Now that the whole network log can be represented as a long sequence of many sequences of vectors containing 32 integers, the next step is to chop the whole log (long sequence) by session. Specifically, whenever a DNS query packet from the user machine is observed, the sequence is chopped right before it. As a result, a number of variate-length sequences can be generated, and the sequence length here is equal to the number of packets in one session.

The following step is to unify the data representation. Specifically, the variate-length sequence should be processed into fixed-length sequence(s). Generally, there are two ways to do this. One is to find the longest sequence, and pad all other shorter sequences to the same length with dummy data. The other way is to apply sliding windows, so that long sequences can be converted to several shorter sequences. Both have been proven to be effective in multiple scenarios in deep learning, and, in this use case, the sliding window approach is used. After applying sliding window, a number of fixed-length sequences

Table 5.2: Fields of interest.

Layer	Fields	Size in bytes	Explanation
IP	Version	4/8	For IPv4, this is always equal to 4.
	IHL	4/8	Internet header length.
	DSF	1	Differentiated service field, which includes differentiated services code point and explicit congestion notification.
	TLen	2	The entire packet size in bytes.
	ID	2	An identification field which is primarily used for uniquely identifying the group of fragments of a single IP datagram.
	Flags	3/8	3 bits for controlling or identifying fragments.
	FragOff	13/8	13 bits for specifying the offset of a particular fragment relative to the beginning of the original unfragmented IP datagram.
	TTL	1	Time to live field, which limits a datagram's lifetime.
	port	1	This field defines the protocol used in the data portion of the IP datagram.
	chksum	2	Header checksum for error-checking of the header.
UDP	src_port	2	Source port number.
	dst_port	2	Destination port number.
	hd_len	2	The length in bytes of the UDP header and UDP data.
	chksum	2	Checksum field for error-checking of the UDP header and UDP data.
DNS	TID	2	Transaction ID for synchronization between DNS servers/clients.
	flags	2	Control flags
	q	2	The number of entries in the question section.
	AnRR	2	The number of resource records in the answer section.
	AuRR	2	The number of resource records in the authoritative section.
	AdRR	2	The number of resource records in the additional section.

```
def prep(xList, window_size, window_step):
    X = []
    for i in range(len(xList)):
        line = xList[i]
        n = len(line)
        for j in range(0, n-window_size+1, window_step):
            if j+window_size <= len(line):
                X.append(line[j:j+window_size])
            else:
                X.append(line[n-window_size:n])
    return np.array(X)
```

Code 5.2: Codes for appying sliding window.

are generated, and the sequence length here is equal to the sliding window length. Each element inside is a vector containing 32 integers.

The last step is to change the integers. Those integers are converted from bytes, but integers may not be the most suitable representation for the byte. For example, the `flags` field in the `DNS` layer has two bytes, but these two bytes are made up of control bits. As a result, close numbers do not necessarily mean that the functionalities represented are close. For example, in the case of response DNS packet, the lower byte (eight bits) in the `flags` field of `DNS` layer contains five control flags: bit 0 to 3 indicate the reply code; bit 4 indicates whether to accept non-authenticated data or not; bit 5 indicates whether the answer is authenticated or not; bit 6 is reserved in response DNS packet; and bit 7 indicates whether recursion is available or not. Incidentally, `0x00`, where all bits are zero, and `0x10`, where all but bit 4 are zero, only differ in one bit, but the numeral difference is 16. In a word, bit representation might be more natural in this use case. Correspondingly, the integers in the vectors

```python
def int2bin(num,padding=False,pad_to=8):
    lst=[]
    while num>1:
        lst.insert(0,num%2)
        num=num//2
    lst.insert(0,num)
    if padding:
        while len(lst)<pad_to:
            lst.insert(0,0)
    return lst
```

Code 5.3: Codes for transforming an integer to bits.

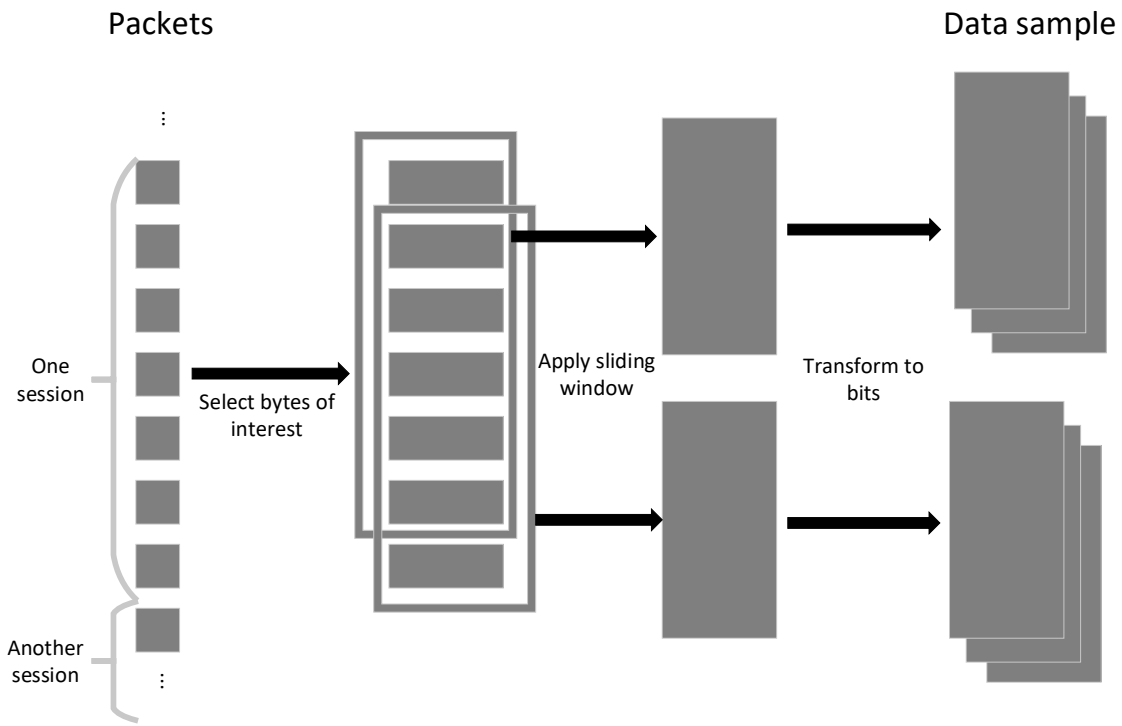


Figure 5.2: Data processing demonstration.

are transformed to bits.

In the end, all data samples are ready, and the data representation is similar to images with three dimensions. The first dimension is packet index. The second dimension is the byte in the packet. The third dimension is the bit in the byte. The corresponding label to data samples are also ready: if it comes from an attack session, it is labeled as malicious; or else it is labeled as benign. The whole process is demonstrated in Figure 5.2:

1. A whole network log is first chopped by sessions, and one of the sessions contains seven packets.
2. From each of the seven packets, bytes of interest (32 bytes in this use case) are chosen.
3. After that, a sliding window of length six is applied, resulting to two sequences containing 6 vectors consisting of 32 integers. Such sequences can also be viewed as matrices of shape $6 * 32$.
4. Finally, every integer is transformed to bits. Because each integer origins from one byte, 8 bits are enough to represent all integers. Therefore, the resulting data samples are of shape $6 * 32 * 8$, like images which have 6 rows, 32 columns, and 8 channels.

5.5 Data Set Construction

Now that the data samples and corresponding labels are ready, the next step is to construct the training set, the test set, and the validation set. However, before the splitting the whole data set into them, two kinds of data samples have to be ruled out from the whole data set, because they cannot help the classification.

The first kind is called **double-dipping** data samples, which refer to same data samples that appear in multiple classes. As shown in Figure 5.2, one session may result in one or more data samples. As a result, benign sessions and malicious sessions may generate same data samples. Such data samples are what the benign and malicious sessions share in common, so they cannot help the classification. That is why double-dipping data samples are ruled out from the whole data set.

The second kind is called **duplicated** data samples, which refer to data samples that appear multiple times in the same class. In this use case, duplicated data samples refer to what benign sessions or malicious sessions share in common. Such data samples should only appear once in the whole data set, or else the data variety of the whole set will be harmed. That is why duplicated data samples are ruled out from the whole data set, so that they only appear once in each class.

After double-dipping and duplicated data samples are ruled out, there is another step for data balancing. Specifically, it is most desirable if the data set is both balanced and large enough. Lack of training data can result in poor results, while biased data sets may result in biased models. The data sets after processing are usually unbalanced. To mitigate such unbalancing, a common practice is to down-sample the one with more data samples, but this will reduce the size of data set. To deal with this dilemma, the criterion of "a balanced data set" is loosened: as long as the benign to malicious ratio is within the range of $[\frac{2}{3}, \frac{3}{2}]$, then there is no need for down-sampling. (This means that both the benign and malicious take up more than 40% but less than 60% of whole data, so the data set as a whole is adequately balanced.)

The final step after data balancing is to split the data set into training, test, and validation set. First, out of the whole data set, 20% are kept aside as test set, never to be used in training. Second, the rest 80% of the whole data set are further split into four folds of the same size, upon which 4-fold cross validation is applied.

5.6 Model Architecture

Because the data samples are image-like, the convolutional neural network (CNN), which has been proven to work well towards such data samples in the computer vision field, is selected. The neural network architecture is shown in Figure 5.3.

5.7 Parameter Tuning

There are two sets of parameters in this use case. One set corresponds to the sliding window, and the other corresponds to the hyperparameters inside the neural network.

```
full_set=set()

# remove duplicates
tmp = []
for ele in X_mal_tmp:
    tmp.append(tuple(ele.reshape((window_size*32,)).tolist()))
full_set = full_set | set(tmp)
X_mal_tmp = np.array(list(set(tmp)),dtype=np.uint8).reshape((len(tmp),window_size,32))

tmp = []
for ele in X_ben_tmp:
    tmp.append(tuple(ele.reshape((window_size*32,)).tolist()))
full_set = full_set | set(tmp)
X_ben_tmp = np.array(list(set(tmp)),dtype=np.uint8).reshape((len(tmp),window_size,32))

# print dataset statistics before removing double-dipping
print("Original size of dataset for k = %d" % k)
print("Malicious: %d" % len(X_mal_tmp))
print("Benign: %d" % len(X_ben_tmp))
print("Merged size: %d" % len(full_set))

if len(full_set)!=(X_ben_tmp.shape[0]+X_mal_tmp.shape[0]): # check the necessity for removing double-
    dipping
    # remove double-dipping
    X_mal = []
    X_ben = []
    print("Double dipping exist (%d!=%d)! Remove double dipping..."%(len(full_set),(X_ben_tmp.shape[0]+
        X_mal_tmp.shape[0])))
    for item in full_set:
        item=np.array(item,dtype=np.uint8).reshape((window_size,32))
        if (item in X_ben_tmp) and (item in X_mal_tmp):
            continue
        elif item in X_ben_tmp:
            X_ben.append(item)
        elif item in X_mal_tmp:
            X_mal.append(item)
        if (len(X_ben) % 1000 == 0) or (len(X_mal) % 1000 == 0):
            print("############################")
            print(str(k) + " :")
            print("Malicious: %d" % (len(X_mal)))
            print("Benign: %d" % (len(X_ben)))
            print("Progress: %d/%d" %
                (len(X_mal)+len(X_ben), len(full_set)))
    X_ben=np.array(X_ben)
    X_mal=np.array(X_mal)
else:
    X_ben=X_ben_tmp
    X_mal=X_mal_tmp

# remove unused variable to save memory
del(X_ben_tmp)
del(X_mal_tmp)
gc.collect()

# print data set statistics after removing double-dipping
print("############################")
print("Double dipping removing finished.")
print("Malicious: %d" % len(X_mal))
print("Benign: %d" % len(X_ben))
```

Code 5.4: Codes removing duplicated and double-dipping data samples.

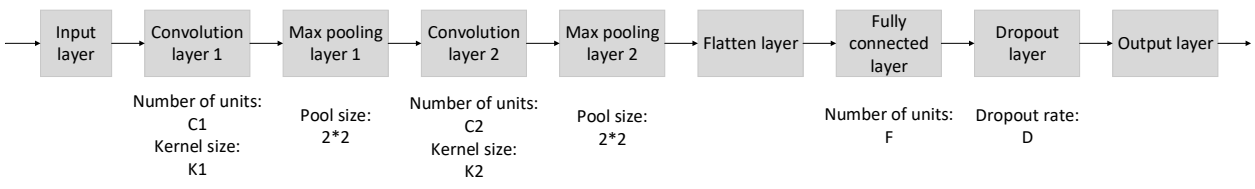


Figure 5.3: Neural network structure for DNS cache poisoning detection.

```
# imports are omitted
def build_model(window_size, hidden_tensor, kernel_size, dropout_rate):
    inputs = tf.keras.layers.Input(shape=(window_size, 32,8), name='Input')
    y = tf.keras.layers.Conv2D(filters=hidden_tensor[0], kernel_size=kernel_size[0], padding='same',
        activation='relu', name='Hidden0')(inputs)
    y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='Pooling0')(y)
    y = tf.keras.layers.Conv2D(filters=hidden_tensor[1], kernel_size=kernel_size[1], padding='same',
        activation='relu', name='Hidden1')(y)
    y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same', name='Pooling1')(y)
    y = tf.keras.layers.Flatten(name='Flatten')(y)
    y = tf.keras.layers.Dense(hidden_tensor[2], activation='relu', name='Dense')(y)
    y = tf.keras.layers.Dropout(dropout_rate, name='Dropout')(y)
    probs = tf.keras.layers.Dense(2, activation='softmax', name='Output')(y)
    model = tf.keras.models.Model(inputs, probs, name='classifier')
    return model
```

Code 5.5: Codes for model building.

The first set of parameters, which are about the sliding window, includes the window length w_size and window step w_step. Window length defines the length of the sliding window, and the window step defines the size of the sliding window's movement. For example, in Figure 5.2, the window size is six, and the window step is one. For w_size, $\{4, 6, 8, 10, 12\}$ are tried; for st, $\{1, 2, 4, 6, 8\}$ are tried. Note that, depending on the first set of parameters, the size of data sets can differ.

The second set of parameters, which are model hyperparameters, includes the number of units in some hidden layers (the convolutional layer 1, 2 and the fully connected layer, $N\_hidden \in \{(C_1 = 16, C_2 = 16, F = 8), (C_1 = 32, C_2 = 16, F = 8), (C_1 = 32, C_2 = 32, F = 16), (C_1 = 32, C_2 = 32, F = 8), (C_1 = 64, C_2 = 32, F = 16), (C_1 = 64, C_2 = 64, F = 16)\}$), the kernel size ($K\_size \in \{(K_1 = 2 * 2, K_2 = 2 * 2), (K_1 = 3 * 3, K_2 = 3 * 3), (K_1 = 4 * 4, K_2 = 4 * 4)\}$) in each CNN layer, and the dropout rate ($D \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$) in the dropout layer.

Code 5.5 shows how the classification neural network is built. Doubtlessly, the second set of parameters are necessary for model building. The window size from the first set of parameters is also needed for model building, because it is directly related to the shape of input data samples to the neural network.

Because it is unknown that which values will result in the best results beforehand, all parameter combinations are tried. Also, because 4-fold cross validation is applied, a total of $5*5*6*3*5*4 = 9000$ models are trained. The best model is chosen based upon the average performance on the validation set across all four folds.

5.8 Evaluation results

Table 5.3 shows the data sets statistics for the chosen model, and Table 5.4 shows the evaluation results. Four metrics are selected for testing, which are accuracy (Acc), F1 score (F1), detection rate (DR), and precision (Pre).

Table 5.3: Data set statistics.

Data set	Size	Benign to malicious ratio
Training	30928	1.003:1
Test	7732	0.988:1

Table 5.4: CNN model performance.

Data set	*Accuracy*	*Precision*	*Detection rate*	*F1*
Training	0.9987	0.9997	0.9976	0.9987
Test	0.9973	0.9992	0.9953	0.9973

5.9 Model Deployment

After the model is trained and evaluated, the next step is to deploy the trained model. However, it should be noticed that the training environment differs from the deployment environment in several ways. For example, the production system often has different calculation speed, memory/storage size, network bandwidth, to name a few. Therefore, production system pays more attention to efficiency and costs, and models usually need additional optimizations before being deployed.

The neural networks in this use case are trained with TensorFlow [1], which contains TensorFlow Lite [2], a set of tools that "enables on-device machine learning by helping developers run their models on mobile, embedded, and IoT devices". It features optimizations addressing latency (no round-trip to a server), privacy (no personal data leaves the device), connectivity (no requirement for internet connectivity), size (reduced model and binary size) and power consumption (efficient inference and a lack of network connections).

In this use case, post-training quantization is applied (Code 5.6). It is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. The simplest form of post-training quantization statically quantizes only the weights from floating point to integer, which has 8-bit precision. During inference, weights are converted from 8-bit precision to floating point and computed using floating-point kernels. This conversion is done once and cached to reduce latency.

To further reduce latency, it dynamically quantizes activations based on their range to 8 bits and perform computations with 8-bit weights and activations. This optimization provides latency close to fully fixed-point inference. However, the outputs are still stored using floating point so that the speedup with dynamic-range ops is less than a full fixed-point computation.

Specifically, in this use case, the post-training model can be reduced from 452 KB to 41 KB, while the evaluation metrics (Acc, Pre, DR, and F1) do not downgrade on both the training set and the test set.

[1] https://www.tensorflow.org/

[2] https://www.tensorflow.org/lite/guide

```python
# some codes are from https://www.tensorflow.org/lite/performance/post_training_quantization
import os
import argparse

import tensorflow as tf
from tensorflow_addons.metrics import F1Score

FLAGS=None

if __name__ == "__main__":
    parser=argparse.ArgumentParser()
    parser.add_argument(
        '--n',
        type=int,
        required=True,
        help='ID of training.'
    )
    parser.add_argument(
        '--k',
        type=int,
        required=True,
        help='ID of model. The models to load should be placed in the model directory, named \"classifier-N-K-[0,1,2,3].h5\"'
    )
    FLAGS, unparsed =parser.parse_known_args()

    gpus = tf.config.list_physical_devices('GPU')
    if gpus:
        try:
            # Currently, memory growth needs to be the same across GPUs
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
            logical_gpus = tf.config.experimental.list_logical_devices('GPU')
            print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
        except RuntimeError as e:
            # Memory growth must be set before GPUs have been initialized
            print(e)
    else:
        tf.config.threading.set_inter_op_parallelism_threads(8)
        tf.config.threading.set_intra_op_parallelism_threads(8)

    for i in range(4):
        model=tf.keras.models.load_model(os.path.join("model","classifier-"+str(FLAGS.n)+"-"+str(FLAGS.k)+"-"+str(i)+".h5"),custom_objects={"metric":F1Score(num_classes=2)})

        converter=tf.lite.TFLiteConverter.from_keras_model(model)
        converter.optimizations=[tf.lite.Optimize.DEFAULT]
        tflite_quantilized_model=converter.convert()
        fi=open(os.path.join("model","tflite_quantilized_model-"+str(FLAGS.n)+"-"+str(FLAGS.k)+"-"+str(i)+".tflite"),'wb')
        fi.write(tflite_quantilized_model)
        fi.close()
```

Code 5.6: Codes for post training model optimization.

5.10 Remaining Issues

There are two major remaining issues regarding this use case.

Run-time efficiency optimization. Though initial run-time optimization is applied by post-training quantization, another part of run-time efficiency optimization is not taken into consideration, which is library dependence. The model is trained with TensorFlow, which depends on lots of other libraries by itself. As a result, even loading the model at inference time will consume quite some resources. To make matters worse, the data processing in this use case is also resource-hungry. As shown in Figure 5.2, the raw network packets need to go through four steps before data samples can be generated. Optimizations at data processing is also worth of considering.

Model updating. This use case only involves one iteration of the whole pipeline, from data gathering to model deploying. Updating the model is not taken into consideration. This makes sense to a certain extent, because the trained model is dedicated to detect just one network attack. If the trained model can already capture the attack characteristics well, there is little reason to update the model. However, given that more and more data are observed after the deployment, the size of labeled data are on the increase. Therefore, in the simplest case, one can retrain the whole model with new data, evaluate the new model, and compare it with the old one. If the new model prevails, the old model can then be replaced with the new one.

5.11 Code and Data Resources

Resources, including part of the raw data, scripts for data processing, model training, evaluations, and run-time optimizations are provided on the GitHub repository of this handbook under `https://github.com/PSUCyberSecurityLab/AIforCybersecurity/tree/main/Chapter5-DL-for-Detecting-DNS-Cache-Poisoning`. For detailed instructions on how to use the resources, please refer to the `README.md` there.

Chapter 6 AI Detects PC Malware

6.1 The Security Problem

Both the industry and the academic community have provided approaches to detect malware with program analysis. Traditional methods such as behavior-based signatures, dynamic taint tracking, and static data flow analysis require experts to manually investigate unknown files and locate the malicious behaviors. However, these approaches are not very scalable when there are a large number of malware variants to detect. Attackers can specifically modify their malware using various obfuscation techniques such as code encryption, reordering the program instructions, and dead code insertion technique to avoid detection based on the analysis of malware detection tools.

To detect malware more accurately and to better counter these detection evasion techniques, state-of-the-art research has leveraged deep-learning algorithms for malware detection with static and dynamic analysis. The dynamic analysis identifies programs' behaviors in a runtime environment. It requires sufficient inputs to cover all behaviors. Static analysis, on the other hand, is program-centric involving code disassembly, opcode sequences extraction, the control flow graphs analysis, and so on.

In this chapter, we introduce a malware detection method that (a) converts malware to images, sequential data, and graphs, and (b) leverages deep learning to learn the similarities among malware samples as well as the dissimilarities between malware and benign ware. The method has been demonstrated by Intel Labs and the Microsoft Threat Intelligence Team [69].

6.2 Raw Data

The malicious binaries used in this chapter are collected from VirusShare [76]. VirusTotal [77] is used to label the malware samples and identify the malware families. The dataset holds five families of malware samples:

- Virus: A virus modifies other system programs, so when a victim's file is executed, the virus code is also executed.
- Worm: A worm program will copy itself to other hosts by some media, like email.
- Trojan: Trojans pretend to be a benign program and collect user information without users' permission.
- Ransomware: Ransomware invades users' machines and encrypts their data for cryptocurrency.
- Adware: Adware will push malicious advertising to users.

The benign programs are directly retrieved from the software packages of Ubuntu 18.04LTS [73]. The categories of the installed packages vary including popular applications (such as Chrome, Firefox, and Zoom), security applications (such as Spybot2, SUPERAntiSpyware, and Malwarebytes), developer tools (such as Python, Github, and PuTTY), and so on.

6.3 Data Processing

The goal of malware classification is to identify malicious behaviors in software with static and dynamic features like control flow graph and system API calls. In this chapter, static analysis is used to examine malware so that we do not need to execute the binaries and monitor their behaviours. To apply deep learning, the first step is to represent malware data in a meaningful way. This chapter introduces three kinds of malware data representations: images, sequences and graphs.

6.3.1 Image Data Conversion

The raw binaries are read as a sequence of bytes, which then are converted into a value between 0 and 255. The sequential data is then reshaped into two dimensions so that we can directly apply computer vision based deep learning models. The width and height of the generated images are depend on the size of the original binaries. As shown in Code 6.1, the height of the images is calculated as the size of binaries divided by the width. And the final result is rounded up to an integer. Figure 6.1 is an example of the generated gray scale image.

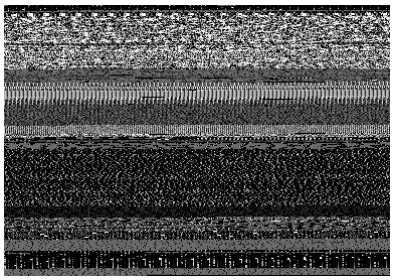


Figure 6.1: Malware gray image.

```
1  def get_size(data_length, width=64):
2      # source Malware images: visualization and automatic classification by L. Nataraj
3      # url : http://dl.acm.org/citation.cfm?id=2016908
4      if width is None: # with don't specified any with value
5          size = data_length
6          if (size < 10240):
7              width = 32
8          elif (10240 <= size <= 10240 * 3):
9              width = 64
10         elif (10240 * 3 <= size <= 10240 * 6):
11             width = 128
12         elif (10240 * 6 <= size <= 10240 * 10):
13             width = 256
14         elif (10240 * 10 <= size <= 10240 * 20):
15             width = 384
16         elif (10240 * 20 <= size <= 10240 * 50):
17             width = 512
18         elif (10240 * 50 <= size <= 10240 * 100):
19             width = 768
20         else:
21             width = 1024
22         height = int(size / width) + 1
23     else:
24         height = width
```

```
        width = int(data_length / width) + 1
    return (width, height)
}
```

Code 6.1: Code snippet of image conversion.

6.3.2 Sequential Data Conversion

The representation of sequential data is more straightforward. The binary is directly converted as sequences of bytes, which can be interpreted as an unsigned integer in the range 0–255. Because the size of binaries are not the same, the sequence is partitioned into several sub-sequences with the same length ($sequence\_length$ in Code 6.2). The last sub-sequence is padded with the $sequence\_length$ to make sure the inputs have the same size.

```
def get_sequence(f, sequence_length = 100):
    tmp = np.fromfile(f, np.uint8)
    tmp = np.array(tmp)
    left = len(tmp) % sequence_length
    length = len(tmp) if left == 0 else sequence_length - left + len(tmp)
    res = np.ones(length) * 256
    res[:tmp.shape[0]] = tmp
    res = np.reshape(res, (-1, sequence_length))
    return res
```

Code 6.2: Code snippet of sequence conversion.

6.3.3 Graph Data Conversion

A control flow graph (CFG) is a directed graph representation that illustrates all reachable paths of the program during execution shown as in Figure 6.2. The nodes of the CFG are the basic blocks of the program. Each basic block is a consecutive, single-entry code without any branching except at the end of the sequence. Edges in a CFG represent possible control flows in the program. The control flow enters only at the beginning of the basic block and leaves only at the end of the basic block. Each basic block can have multiple incoming/outgoing edges. Each edge corresponds to a potential program execution. The structure information contained by CFGs indicates a step-by-step execution process of programs which can be used to find unreachable code, find syntactic structure (like loops), and predict programs' defect [49, 59].

As shown in Code 6.3, the Python framework for analyzing binaries, called Angr, is used to extract CFG in those datasets. It is transformed to a direct graph. After extracting CFG from the executable files, we will construct an abstract directed graph $G =< V, E >$, consisting of the set V of nodes and the set E of edges. Each node v_i represents a basic block in a CFG, while a directed edge e_{ij} points from the first basic block v_i to the second basic block v_j. The feature of each basic block is the bytes sequence extracted with the same method as subsection 6.3.2.

```
def get_CFG(f, feature_dim):
    proj = angr.Project(f, main_opts={'backend':'blob','arch':'i386'}, load_options={'auto_load_libs':False})
    cfg = proj.analyses.CFGEmulated()
```

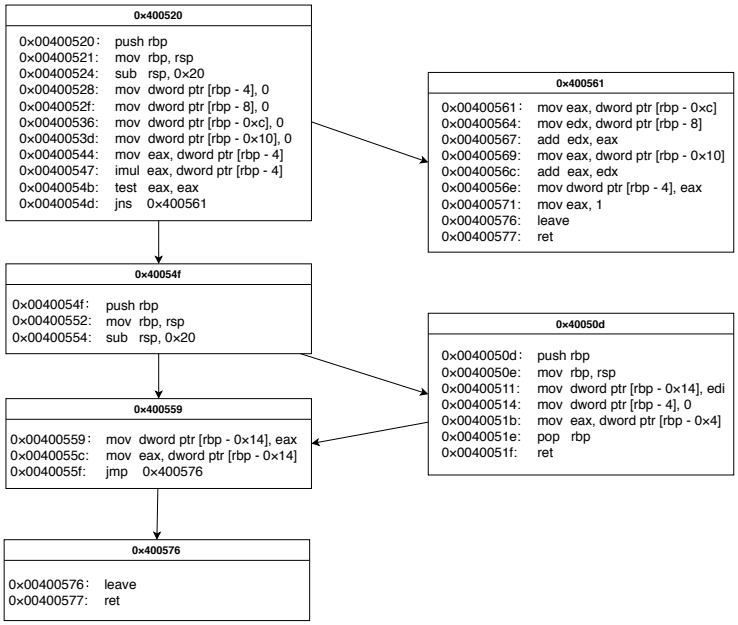


Figure 6.2: An example of CFG.

```
    block = proj.factory.block(proj.entry)
    blocks = {}
    node_features = []
    G = nx.DiGraph()
    idx = 0
    for n in cfg.graph.nodes():
        blocks[hex(n.addr)] = idx
        G.add_node(idx)
        idx += 1
        uint8 = []
        if n.block != None:
            block_instructions = n.block.capstone.__str__()
            vector = n.block.bytes.hex()
            b = bytearray.fromhex(vector)
            for i in range(len(b)):
                uint8.append(b[i])
        uint8.extend([256]*(feature_dim - len(uint8)))
        node_features.append(uint8)
    for k, v in cfg.graph.edges():
        G.add_edge(blocks[hex(k.addr)], blocks[hex(v.addr)])
    return G
```

Code 6.3: Code snippet of CFG conversion.

6.4 Model Training

In the previous section, we represent malware data in three alternative ways. In this section, we introduce three kinds of DL models to learn malware features from image-based data representation, sequence-based data representation, and graph-based data representation, respectively.

6.4.1 Model Architecture

6.4.1.1 CNN Models

The idea of transfer learning, which is widely employed in computer vision, is leveraged to train a malware detection model. The basic idea of transfer learning is to apply the knowledge gained from one field to another field so that we do not have to restart from scratch for every new model.

The pretrained model used in this section is resnet-50 [64]. Residual Networks (ResNets) [64] are deep convolutional networks that learn shortcut connections between convolutional layers, as shown as Figure 6.3. The intuition is to avoid the problem of vanishing gradients by skipping one or more layers.

As shown in Code 6.4, it is easy to load the resnet-50 network and its weights with Tensorflow. The weights of resnet-50 model are no longer updated during training. Several layers are created on top of the output of the base model. Once the model has been trained on the new data, it is suggested to unfreeze all or part of the base model and fine-tune the whole model again with a very low learning rate.

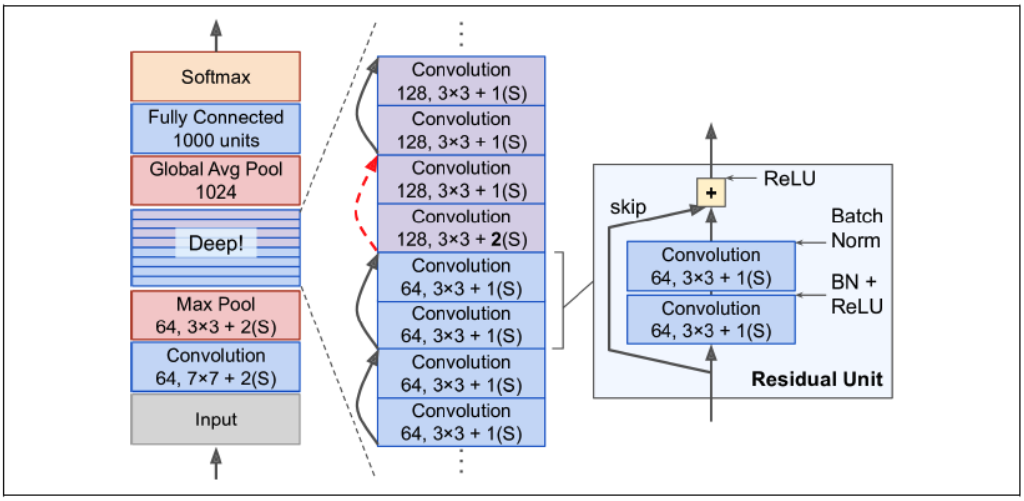


Figure 6.3: ResNet-50 [64].

```
class CNN_MAL(tf.keras.Model):
    def __init__(self, num_classes):
        """Initializes the CNN model
        :param num_classes: The number of classes in the dataset.
        """
        super(CNN_MAL, self).__init__(name="CNN_MAL")
        self.num_classes = num_classes
        self.image_shape = [-1, config.img_width, config.img_height, config.channel]
        def __graph__():
            self.preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
            self.base_model = tf.keras.applications.resnet50(input_shape=self.image_shape,
                                        include_top=False,
                                        weights='imagenet')
            self.base_model.trainable = False
            self.global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
            # Dropout, to avoid overfitting
            self.dropout_layer = tf.keras.layers.Dropout(0.2)
```

```
            # Readout layer
            self.prediction_layer = layers.Dense(num_classes)

        __graph__()

    def call(self, x_input, training=False):
        x_input = self.preprocess_input(x_input)
        x = self.base_model(x_input, training=False)
        x = self.global_average_layer(x)
        x = self.dropout_layer(x)
        logits = self.prediction_layer(x)
        return logits
```

Code 6.4: Code snippet of the CNN model.

6.4.1.2 RNN Models

As shown in Code 6.5, the RNN models have 5 layers:

1. Input Layer: The input of the neural networks are data generated by converting the binaries to a sequence of integers.
2. Embedding Layer: In this layer, the original input is transformed into a low dimension vector.
3. LSTM Layer: BiRNN is used to learn higher level features based on the embedding layer.
4. Summary Layer: We summarize the features of the entire sequence.
5. Output Layer: In this layer we use the sequence-level features to determine the classification results.

The embedding layer transforms the subsequences x_i^j to a low dimension vector h_i. Bidirectional basic RNN, BiLSTM, and BiGRU can be used to get a high level features based on embedding layer. The BiRNNs contains two RNNs for the forward and backward pass of the sequence. The output h_i^j of j-th subsequence of sequence i is the sum of the left output $\overrightarrow{h_i^j}$ and the right output $\overleftarrow{h_i^j}$.

$$h_i^j = \overrightarrow{h_i^j} + \overleftarrow{h_i^j} \tag{6.1}$$

We additionally apply layer normalization after the LSTM Layer. Since the sequence of one binary is divided into several subsequences, the sequence-level feature x_i are summarized by adding up the features of all subsequences $x_i = \sum_j^{x_i^j}$. Finally, dropout and classification layer are employed.

```
def __init__(self, num_classes, rnn_type):
    """Initializes the RNN model
    :param num_classes: The number of classes in the dataset.
    :param rnn_type: basic rnn, rnn, gru
    """
    super(RNN_MAL, self).__init__(name="RNN_MAL")
    self.num_classes = num_classes

    def __graph__():
        self.embedding_layer = layers.Embedding(config.max_features, config.hidden_units)
        if rnn_type == 'lstm':
            self.rnn = LSTM(config.hidden_units, return_state=False, return_sequences=True)
        elif rnn_type == 'rnn':
            self.rnn = RNN(config.hidden_units, return_state=False, return_sequences=True)
```

```
        elif rnn_type == 'gru':
            self.rnn = GRU(config.hidden_units, return_state=False, return_sequences=True)
        self.biRNN = Bidirectional(self.rnn)
        self.dropout_layer1 = layers.Dropout(config.dropout)
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.output_layer = layers.Dense(num_classes)
    __graph__()

def call(self, inputs, training):
    x_input = inputs[0]
    self.shape = inputs[1]
    inputs_embedded = self.embedding_layer(x_input)
    outputs = self.biRNN(inputs_embedded)
    outputs = self.layernorm1(outputs)
    outputs = tf.reshape(outputs, [-1, config.sequence_length * config.hidden_units * 2])
    outputs = self.across_sum_layer(outputs, training = training)
    outputs = self.dropout_layer1(outputs, training = training)
    logits = self.output_layer(outputs)
    return logits
def across_sum_layer(self, attention_r, training):
    def sub_windows_sum(index_span):
        indices = tf.range(index_span[0], index_span[1])
        bag_hidden_mat = tf.gather(attention_r, indices)
        return tf.reduce_sum(bag_hidden_mat, 0)
    sort_pooling = tf.map_fn(sub_windows_sum, self.shape, dtype=tf.float32)
    return sort_pooling
```

Code 6.5: Code snippet of the RNN model.

6.4.1.3 GNN Models

In this section, we use the DGCNN model [94] to embed structural information inherent in graph like data. The graph convolution layer takes the following propagation rule:

$$H_{l+1} = \sigma(\widetilde{D}^{-1}\widetilde{A}H_l W) \tag{6.2}$$

Here, $\widetilde{A} = A + I$ is the adjacency matrix A of the directed graph G with added self-connections I. $\widetilde{D}_{ii} = \sum_j \widetilde{A}_i j$ is its diagonal degree matrix. W is a layer-specific trainable weight matrix. $\sigma(\cdot)$ is a nonlinear activation function. H_l is the matrix of activations in the l-th layer; $H_0 = X$, where X denotes the node information matrix of graph G.

The graph convolutional layer propagates node features to neighboring nodes as well as the node itself to extract local substructure information. As shown in Figure 6.4 and the code snippet Code 6.6, multiple graph convolution layers are stacked to get high-level substructure features followed by a SortPooling layer and a classification layer.

```
def gcnn_layer(self, input_Z, W):
    AZ = tf.sparse.sparse_dense_matmul(self.ajacent, input_Z) # AZ
    AZ = tf.add(AZ, input_Z)                                  # AZ+Z = (A+I)Z
    AZW = tf.matmul(AZ, W)                                    # (A+I)ZW
    DAZW = tf.sparse.sparse_dense_matmul(self.dgree_inv, AZW) # D^-1AZW
    return tf.nn.tanh(DAZW)

def gcnn_layers(self, Z):
    Z1_h = []
    for i in range(len(self.gcnn_dims)):
```

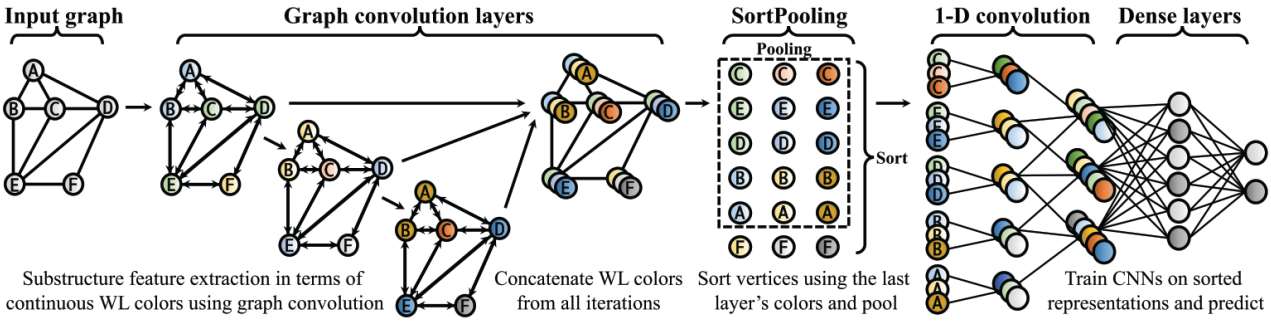


Figure 6.4: DGCNN [94].

```
        Z = self.gcnn_layer(Z, self.weight_matrix[i])
        Z1_h.append(Z)
    Z1_h = tf.concat(Z1_h, 1)
    return Z1_h
```

Code 6.6: Code snippet of graph convolutional layer.

The SortPooling layer extracts and sorts the vertex features based on the structural roles within the graph [94]. Here, the extracted vertex features are the continuous Weisfeiler Lehman (WL) colors. As with the DGCNN model, after we sort all the vertices using the last output layer's output, the $topk$ sorted vertices are send to s convolutional layer and a classification layer.

```
def sort_pooling_layer(self, gcnn_out):
    def sort_a_graph(index_span):
        indices = tf.range(index_span[0], index_span[1])
        graph_feature = tf.gather(gcnn_out, indices)

        graph_size = index_span[1] - index_span[0]
        k = tf.cond(self.k > graph_size, lambda: graph_size, lambda: self.k)
        top_k = tf.gather(graph_feature, tf.nn.top_k(graph_feature[:, -1], k=k).indices)

        zeros = tf.zeros([self.k - k, sum(self.gcnn_dims)], dtype=tf.float32)
        top_k = tf.concat([top_k, zeros], 0)
        return top_k

    sort_pooling = tf.map_fn(sort_a_graph, self.graph_indexes, dtype=tf.float32)
    return sort_pooling

def cnn1d_layers(self, inputs):
    total_dim = sum(self.gcnn_dims)
    graph_embeddings = tf.reshape(inputs, [-1, self.k * total_dim, 1]) # (batch, width, channel)
    first_conv = self.first_conv_layer(graph_embeddings)
    first_conv_pool = self.first_pooling_layer(first_conv)

    second_conv = self.second_conv_layer(first_conv_pool)
    return second_conv

def classification_layer(self, inputs):
    cnn1d_embed = self.flatten_layer(inputs)
    outputs = self.dense_layer(cnn1d_embed)
    return outputs
```

Code 6.7: Code snippet of SortPooling layer and classification layer.

Table 6.1: Dataset statistics.

Data set	Size	Benign	malicious
Training	16545	8294	8251
Validation	1241	593	648
Test	2896	1475	1421

Table 6.2: Model performance.

	Training dataset				Validation dataset				Test dataset			
Data set	***Acc***	***Rec***	***Pre***	***F1***	***Acc***	***Rec***	***Pre***	***F1***	***Acc***	***Rec***	***Pre***	***F1***
CNN	99.53	99.63	99.43	99.53	99.51	99.53.68	99.53	99.53	97.48	98.57	99.36	98.47
LSTM	98.30	97.88	98.69	98.29	99.19	98.91	99.53	99.22	98.92	98.09	99.71	98.90
GNN	96.78	94.44	99.07	96.70	96.68	94.21	99.09	96.59	96.93	94.42	99.9	96.70

6.4.2 Model Performance

Benign programs and malware are merged together, and 20% samples are randomly selected as the test dataset. We trained the model on 70% samples and validated the model on 10% samples. Also, to reduce variability on a limited data sample, we trained the evaluation model using 5-fold cross-validation.

All experiments are conducted on Ubuntu 16.04, using Python 3.7 and TensorFlow 2.3 with NVIDIA GTX980 Ti Graphics Processing Unit (GPU). Table 6.1 shows the final datasets statistics for the chosen model, and Table 6.2 shows the evaluation results. Five metrics are selected for testing, which are accuracy (Acc), Recall (Rec), precision (Pre), and F1 score (F1).

6.5 Model Deployment

After the model is trained and evaluated, the next step is to deploy the trained model. Regarding the production system design, the readers can refer to section 3.7. Note that a main difference from the production system architecture shown in section 3.7 is that instead of using an Android emulator, this use case conducts static analysis.

6.6 Remaining Issues

Traditional malware detection will create a unique signature so they can utilize the scanning algorithm to search for the signature in other potential malware. Malware analysis is used to create the signature leveraged in the signature-based detection system. Dynamic signature-based detection monitors the state transition of programs and behavioral signature e.g. server-to-client signature [25]. Static signature-based detection analyzes binary code of malware to identify sequences of code as the signatures [70]. Hybrid signature-based detection uses static and dynamic analysis to determine the maliciousness [55]. To detect unknown malware, the detection system needs to keep its signature database up to date. This method has been used to the predominant technique in the industry including anti-virus software, firewalls, email, and network gateways. The signature-based detection system is

very effective at detecting known malware but generally unproductive at detecting previously unknown malware. Attackers can reorder the malware code or insert useless code to avoid this kind of detection. Figure 6.5 shows a slice of code from a well-known malware family distributed by APT threat actor OceanLotus on the left, and a YARA signature to detect it on the right [80].

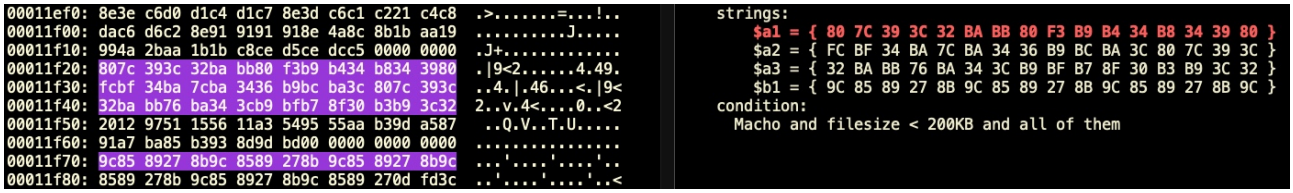


Figure 6.5: Malware example [80].

Dynamic analysis executes the programs in a virtual environment to monitor their behaviors and observe their functionality. Several tools can be used to safely execute suspicious programs: sandbox (e.g. Cuckoo, DefenseWall, Bufferzone); virtual machine (e.g. HoneyMonkey, VGround); emulator (e.g. TTAnalyze, K-Tracer). In general, it provides an emulated environment to run the suspicious applications. Features obtained by dynamic analysis are API calls, system calls, registry changes, memory writes, network patterns, etc [18, 57, 72]. Although dynamic analysis is potentially comprehensive, it is more computationally expensive and less widely used. This analysis technique is more time consuming and has high false positive. The malware starts an early check and immediately exits if it runs on virtual machines. Even worse, some of the malware intentionally exhibit some benign behaviors to trigger human analysts to draw incorrect conclusions about the intent of the malware.

In static-analysis-based malware detection, before a suspicious program file is executed, certain static features are extracted from the executable file to determine whether the file is malicious or not. Some work make use of the binary file itself as indicators to detect the malware [17, 62]. The characteristics of the binary files, such as PE import features, metadata, and strings, are also ubiquitously applied in malware detection [19]. Others leverage reverse engineering to understand the programs' architecture. Reverse engineering is employed to disassemble the program to extract high-level representation, including instruction flow graph, control flow graphs, call graph, and opcode sequences [19, 52, 86]. One advantage of static analysis is that it is usually substantially faster than dynamic analysis. Another advantage is that it can achieve better coverage.

Even though the performance of the models trained in this use case looks very good, adversarial attacks should be considered when we deploy these models in real-world. For example, code obfuscation is effective to evade signature-based detection because it could significantly change the syntactic of original malware. Similarly, obfuscated binaries might also be able to evade DL models. Code obfuscation tools serve two main purposes: (a) to protect intellectual properties; (b) to evade malware detection systems. There are a variety of code obfuscation techniques. A basic requirement of code obfuscation is that the program semantics must be preserved. Traditionally, attackers use obfuscation tricks, including dead-code insertion, register reassignment, subroutine reordering, instruction substitution and so on, to morph their malware to evade malware detection [15, 92]. Here we list the definitions of some widely-used obfuscation tricks:

- Semantic `Nops` Insertion: Inserting certain ineffective instructions, e.g., NOP, to the original

binary without changing its behavior;

- Register Reassignment: Switching registers while keeping the program code and its behavior same, such as registers EAX in the binary are reassigned to EBX;
- Instruction Substitution: Replacing some instructions with other equivalent ones, for example, xor can be replaced with sub;
- Code Transposition: Reordering the sequence of the instructions of a binary;

6.7 Code and Data Resources

Resources, including part of the raw data, scripts for data processing, model training and evaluations are provided on the GitHub repository of this handbook under https://github.com/PSUCyberSecurityLab/AIforCybersecurity/. For detailed instructions on how to use the resources, please refer to the `README.md` there.

Chapter 7 AI Detects Code Similarity

7.1 The Security Problem

In practice, binary code similarity is usually used to search known vulnerabilities, detect code plagiarism, and analyze malware families. However, measuring similarity for binary code is challenging. At binary level, there are several factors that could affect the effectiveness of measuring the code similarity, including the choice of the compiler, the optimization level and the processor architecture. These factors become the major challenges to generalize the code similarity detection, that is, cross-version code similarity detection. For example, a research work [58] creates signatures based on the control flow graph (CFG), which could be inconsistent between different processor architectures.

On one hand, the cross-version code similarity detection can be enabled if important features can be summarized into a high-level abstract representation, so that statistical-based methods becomes very effective. On the other hand, deep learning can also be leveraged to overcome the cross-version challenge, which is known to be capable of extracting abstract features.

This chapter introduces cross-version code similarity detection method based on deep learning. The problem is defined as follow: *given two pieces of binary code, measure them to decide whether they are compiled from the same source code.* Please note that the similarity of the program semantics (*e.g.*, codes written differently for identical purposes) is out of the scope of this chapter. Semantics clones are usually detected in source code level.

When applying deep learning in code similarity detection, usually two different input data representations can be used: a dedicated graph representation (e.g., Gemini [88]) and the raw byte sequence representation (*e.g.*, α-Diff [46]). Though both representations will be introduced, the focus of this chapter is the graph representation, because the sequence models are similar to the ones in other fields (*e.g.*, computer vision), so that less domain knowledge is required during the implementation.

7.2 Raw Data

Though binary similarity can be measured at different levels (e.g., function, loop), it is most common to make the comparison between functions. Thus, the raw data are binary codes of functions. Typically, the ground truth labels are needed for the training purpose, but labeling code clones is a time-consuming task. Therefore, in binary code similarity detection, to enable the supervised training, a very common way to generate dataset is compiling a piece of source code with different optimization levels and different compilers. For example, Code 7.1 shows a very simple loop written in C, and

```
1 for (int i = 0; i < 6; i++) {
2     sum[i] = v1[i] + v2[i];
3 }
```

Code (7.1) A code snippet of a simple loop.

```
0x715: movl   $0x0,-0x6c(%rbp)
0x71c: jmp    0x73f
0x71e: mov    -0x6c(%rbp),%eax
0x721: cltq
0x723: mov    -0x60(%rbp,%rax,4),%edx
0x727: mov    -0x6c(%rbp),%eax
0x72a: cltq
0x72c: mov    -0x40(%rbp,%rax,4),%eax
0x730: add    %eax,%edx
0x732: mov    -0x6c(%rbp),%eax
0x735: cltq
0x737: mov    %edx,-0x20(%rbp,%rax,4)
0x73b: addl   $0x1,-0x6c(%rbp)
0x73f: cmpl   $0x5,-0x6c(%rbp)
0x743: jle    0x71e
```

Code (7.2) Code generated using O0 compiler option.

```
0x745: movl   $0x0,0x34(%rsp)
0x74d: mov    0x20(%rsp,%rax,1),%edx
0x751: add    (%rsp,%rax,1),%edx
0x754: mov    %edx,0x40(%rsp,%rax,1)
0x758: add    $0x4,%rax
0x75c: cmp    $0x18,%rax
0x760: jne    0x74d
```

Code (7.3) Code generated using O1 compiler option.

Code 7.2 and Code 7.3 shows the code emitted using GCC 7.5.0 at two different optimization levels. As shown in the figures, the emitted codes are very different, and it is not a trivial to determine whether the two binary code snippets are compiled from the same source code. By generating code using different optimization levels and compilers, training data with the ground truth can be obtained with minimal amount of human efforts.

To summarize, the raw data are obtained by compiling source code with different compiler setting. The compiled program can be further divided into function, basic blocks, and/or loops. During the model training phase, the function can be identified using the symbol table, and during the production phase, the function are identified using existing tools, *e.g.*, IDAPro.

7.3 Data Processing

This section introduces the processes to convert the raw data into both sequence representation and graph representation. Both representations could be useful, as mentioned in section 7.1.

7.3.1 Sequence Representation

The sequence representation is essentially a vector formed using the raw bytes of the instruction sequence, which can be used as the input of the deep learning models. In other words, there will be very minimal information abstraction based on the domain knowledge, such as conducting flow analysis and dependency analysis. The most straightforward way is to form a vector with the raw bytes of the instruction sequence (similar to the Raw Byte Encoding introduced in subsubsection 2.3.3.1). The merit is, apparently, very simple and intuitive; but in some extreme cases, the padding due to different lengths of samples maybe too long. Another way is to make the sequence into 2D, creating a bitmap, so that the whole sequence could be distributed into 2 dimensions. The advantage of this structure is for some neural network such as convolutional neural nets (CNN), the relations between not only the adjacent bytes, but also further bytes could be learned. Figure 7.3 illustrate the two kinds of structures explained above. Note that there is no theoretical evidence showing any one of

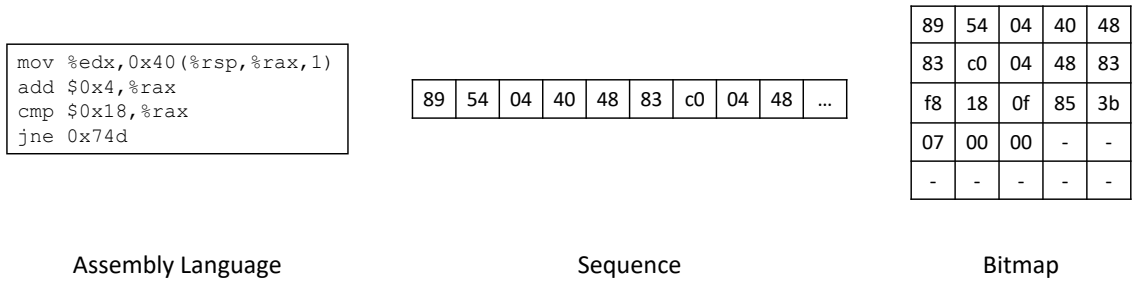


Figure 7.3: Different structures for sequence representation.

the structures is better than the other, and one should select an appropriate representation based on empirical study.

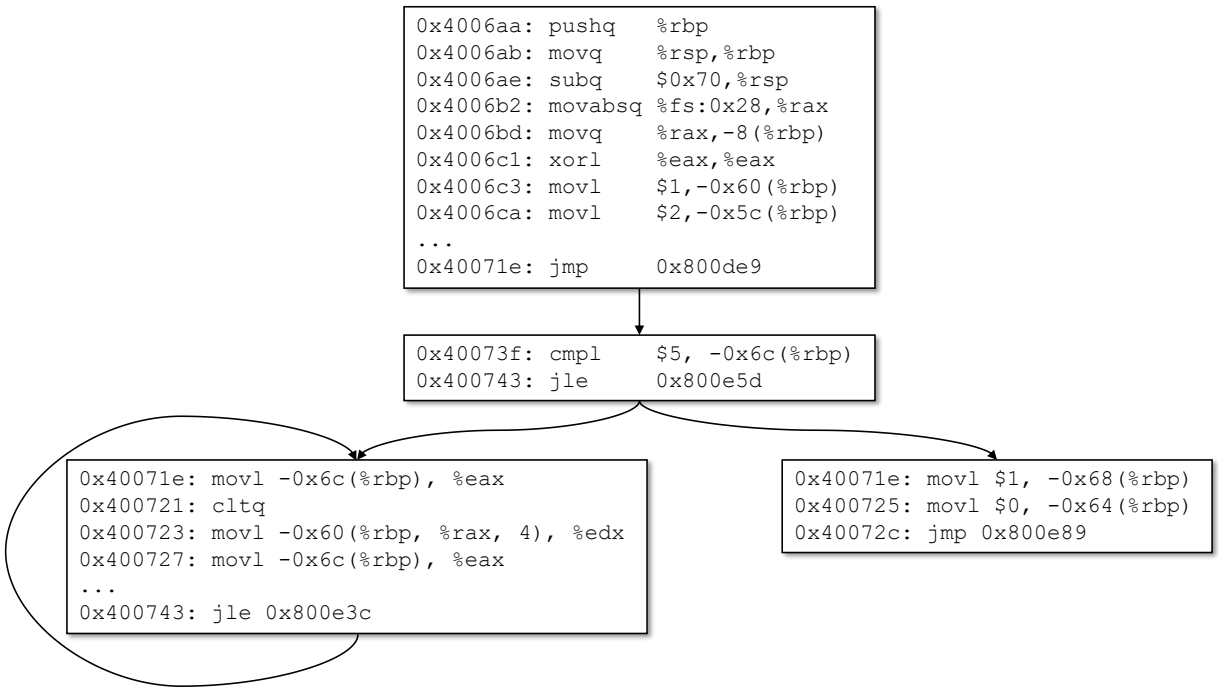
The raw bytes can naturally be mapped to decimal integer values, that is, 0 to 255, to enable the computation of the deep learning model. In case the padding is needed, the padding values can be mapped to an integer that is outside of the range. The downside of this mapping is the numerical characteristics will be carried to the deep learning model, which is undesired, because one cannot claim byte `0x10` is numerically smaller than `0xff`. One solution to this is to map the bytes into one-hot vectors, but this will greatly increase the size of the model.

7.3.2 Graph Representation

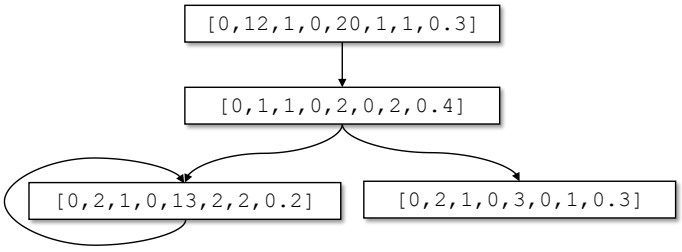
Most traditional code similarity detection methods will first abstract the program information into CFG, and some statistics, usually selected based on the domain knowledge, can be summarized for each basic block. As the graph neural networks (GNN) get mature, researchers start creating carefully designed graphs based on CFGs as the input of a deep learning model. The advantage is clear: compared to sequence representations, the information implied by the graph structure is preserved, providing the structural information as well as the other information.

In the graph representation, the structural information is carried natively, and all other information are carried in the node. A node in a standard CFG is a basic block, which is a sequence of instruction that can be executed sequentially without any jumps. These instructions may not be suitable for a deep learning model to learn from if not well-handled. Typically, nodes in the graphs for the input are fixed-length vectors with selected attributes, which is called node features. Node features can either be selected attributes based on domain knowledge, or embedding vectors from a model.

Let us use attributed control flow graph (ACFG) proposed by [31] as an example, which is used in Gemini [88], a graph neural network based binary code similarity detection method. ACFG is built upon the regular CFG, where each node contains 8 node features, including 6 basic block features and 2 graph structural features. Basic block features include: number of string constants, number of numeric constants, number of transfer instructions, number of calls, number of instructions, and number of arithmetic instructions; whereas graph structural features includes number of outgoing edges and the betweeness.



(a) Partial CFG based on source code shown in Figure 7.4c.



(b) ACFG generated from the CFG shown in Figure 7.4a.

```
1  int main()
2  {
3      int vec1[6] = {1, 2, 3, 4, 5, 6};
4      int vec2[6] = {5, 4, 3, 2, 1, 0};
5      int sum[6];
6      for (int i = 0; i < 6; i++) {
7          sum[i] = vec1[i] + vec2[i];
8      }
9  ...
10 }
```

(c) The corresponding source code.

Figure 7.4: The illustration of generating ACFG.

Figure 7.4 shows an example of the process to generate ACFG from a standard CFG. As shown in Figure 7.4a, the node in the graph is the basic block, which always end with a branch transferring instruction. Then, the 6 basic block features mentioned above can be extracted easily through simple program analysis, and 2 graph structural features can be calculated through graph analysis. Together the two kinds of features will form a node feature vector that is 8 elements in length for each node. The generated ACFG is ready to feed into the deep learning model.

ACFG is architectural independent. In other word, the challenge of the cross-architecture code similarity detection is alleviated by abstracting the information into ACFG. However, the downside is that since the node features selected based on heuristics and domain knowledge, some implicit patterns could be neglected.

7.3.3 Other Information

Previous subsections described the data processing of the input data to the deep learning model, but in order to perform final binary code similarity comparison, more information may need to be extracted from the raw data. For example, it is reasonable to consider inter-procedure relationships if the final goal is to measure the similarity between programs. For example, a common place to find inter-procedure relationships is the call graph, whose statistical attributes, such as number of in-edges and out-edges, can be very useful when comparing call graphs. Although there may be many kinds of information can be leveraged for code similarity detection, this chapter will only focus on measuring the similarity between functions, which is usually the most important part.

7.4 Model

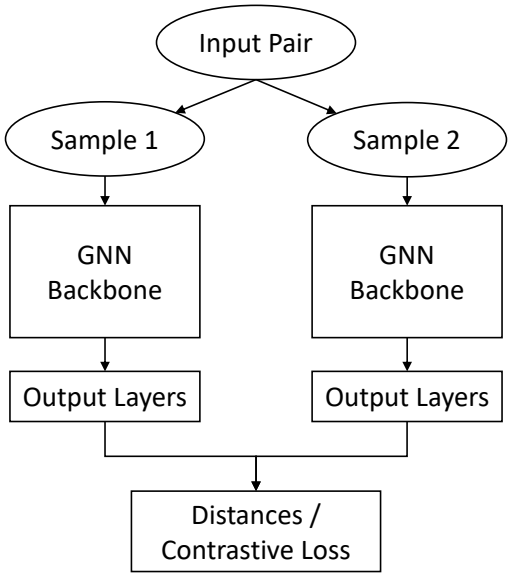


Figure 7.5: The Siamese model architecture using GNN as the backbone.

To obtain quantified similarity score, this use case applies deep learning to learn embedding

vectors for given inputs. Just as other deep learning based similarity detection applications such as face recognition, where the same person's faces result in similar embedding vectors, the binary codes compiled from the same function should also result in similar vectors. To achieve this goal, one of the most popular model architecture to use is the siamese network [10], which is initially proposed to verify hand-written signature. Therefore, in order to use the siamese network to train a model, labeled data is needed.

Regarding the backbone of the deep learning model, it is mostly depending upon the data representation. For sequence data, one can use convolutional neural network (CNN); whereas for graph data, one can use GNN. The focus of this use case is the graph model, as the CNN backbone is rather straightforward. The diagram for the model architecture is shown in Figure 7.5, and in the rest of this section, the GNN backbone and the siamese network will be introduced.

7.4.1 The Graph Neural Network Backbone

This subsection introduces the graph embedding network adopted by Gemini [88] in 2017. Although this model was proposed few years ago, the idea is the same as modern spatial GNNs: aggregate the node features from neighbor nodes to create the node embedding. Certainly, one may choose newer GNN backbone in pursuing better performance.

Before explaining the detail, let us present the formal definition. Given an ACFG $G = \{N, E\}$, where N is the set of nodes, and E is the set of edges, the GNN model f outputs an embedding vector y. The deep learning model firstly computes the node embedding u_n^t for each node $n \in N$ by aggregating the node features x_n of the node and its neighbors for t number of hops. Then the embedding for the whole graph y is obtained by aggregating all the node features.

That is the boring formal definition. In plain words, essentially the workflow is to first acquire the embedding for each node, and then aggregate the node embedding to get the graph embedding. The aggregation function here can be customized, but the most popular one is the simple summation. In other words, after the node embedding for all nodes is obtained, the graph embedding is just simple element-wise addition of all the node embedding.

Obviously, what is important is how to acquire the node embedding. Here let us use the node embedding computation in Gemini:

$$u_n^t = \tanh(Wx_n + \sigma(\sum_{n \in A_n} u_n)) \tag{7.1}$$

In Equation 7.1, W is the trainable weight, σ is a nonlinear transformation function, and A_n is the set of adjacent nodes of node n. The nonlinear transformation function can be trainable, so that it can be a fully-connected neural network. A very critical variable here is the number of iteration t, and each iteration means one hop over the graph. For example, when $t = 1$, the node embedding aggregates the information from the immediate neighbors; when $t = 2$, the node embedding aggregates the information from the neighbors of the immediate neighbors. Although not mentioned in Gemini paper, researchers nowadays are all aware of the oversmoothing issue, so that the total number of hops should not be too large.

Rather than showing the formal algorithm to produce the graph embedding, let us look at the

```
1  def graph_embed(X, msg_mask, N_x, N_embed, N_o, iter_level, Wnode, Wembed, W_output, b_output):
2      node_val = tf.reshape(tf.matmul( tf.reshape(X, [-1, N_x]) , Wnode),
3              [tf.shape(X)[0], -1, N_embed])
4
5      cur_msg = tf.nn.relu(node_val)
6      for t in range(iter_level):
7          # Message convey
8          Li_t = tf.matmul(msg_mask, cur_msg)
9          # Complex Function
10         cur_info = tf.reshape(Li_t, [-1, N_embed])
11         for Wi in Wembed:
12             if (Wi == Wembed[-1]):
13                 cur_info = tf.matmul(cur_info, Wi)
14             else:
15                 cur_info = tf.nn.relu(tf.matmul(cur_info, Wi))
16         neigh_val_t = tf.reshape(cur_info, tf.shape(Li_t))
17         # Adding
18         tot_val_t = node_val + neigh_val_t
19         # Nonlinearity
20         tot_msg_t = tf.nn.tanh(tot_val_t)
21         cur_msg = tot_msg_t
22
23     g_embed = tf.reduce_sum(cur_msg, 1)
24     output = tf.matmul(g_embed, W_output) + b_output
25
26     return output
```

Figure 7.6: Function to produce graph embedding in Gemini.

Python code[1] released by the authors of Gemini. As shown in Figure 7.6, in this implementation, variable `X` is the tensor containing all the node features for all nodes, while all edge information is stored in `msg_mask`, which is a binary matrix (*e.g.*, elements are either 0 or 1).

Codes in line 2 to line 5 correspond to the Wx_n part in Equation 7.1 in the first iteration to produce the embedding of the current node in the beginning. Then start from line 6, the loop of each iteration (*i.e.*, hop) begins. Notice that the message aggregation is done by multiplying a message mask tensor (`msg_mask`) to the node feature/embedding tensor (`cur_msg`). The message aggregation operation is shown in line 8. The rest of the loop body, from line 10 to 21, are straightforward, which is implementing the other parts of the Equation 7.1. At the end, in line 23, the node embedding are summed together to produce the embedding for the entire graph, and in line 24, one more linear layer is used to propagate the information.

As described earlier, the last layer of the GNN backbone (*e.g.*, a pooling layer) will output the embedding of the graph by aggregating node representation. Next section will introduce the whole network structure to learn from a pair of similar/dissimilar binary code.

7.4.2 Siamese Network

The siamese network is a popular representation learning architecture to learn embedding vectors, so that the distances between output vectors from similar inputs are minimized. To train a siamese architecture, pairs of similar/dissimilar data samples are needed, which is different from regular classification and regression tasks, where only the ground truth values are needed during the training time. This section first explains the overall architecture of the siamese network and then introduces

[1] https://github.com/xiaojunxu/dnn-binary-code-similarity/blob/master/graphnnSiamese.py

```
1 def contrastiveLoss(x1, x2, y, m):
2     # We assume the inputs are batched
3     distance = tf.sqrt( tf.reduce_sum( tf.square(x1 - y1) , 1) )
4
5     loss = y * distance + (1 - y) * (m - distance)
6     loss = tf.reduce_sum(loss)
7     return loss
```

Code 7.4: An implementation of the contrastive loss.

the training process.

Shown in Figure 7.5, the most important idea of the siamese network is simple: two sub-networks sharing the weights, each of which will take one sample in a input pair. The loss function is the distance between the outputs from the sub-networks, which should be minimized if the input is a pair of similar samples and maximized if the input is a pair of dissimilar samples. Modern siamese network use so-called contrastive losses that can address both similar pairs and dissimilar pairs as inputs. One example of the contrastive losses is shown in Equation 7.2, where D is the distance of the outputs from the two sub-networks, y is a binary scalar indicating whether the D is the distance of a similar pair or a dissimilar pair output, and m is the threshold maximum distance between the dissimilar pairs.

$$L = yD + (1 - y)\max(0, m - D) \tag{7.2}$$

Note that in Equation 7.2, D is from a similar pair if $y = 1$. Distance D is usually Euclidean distance, but it can be replaced by any distance or similarity measurements if needed. In fact, in the original Gemini [88] paper, the authors did not use any contrastive losses, but only the cosine similarity as the loss function. By minimizing L, the neural network can learn not only how two samples can be similar, but also how they can be different. For dissimilar samples, despite the differences in terms of the labels and classes, they are still in the same domain, and therefore, it is necessary to have a threshold m to cap the distances.

There are two important elements to implement for the siamese network: the loss function and the sub-networks sharing the weights. A simple implementation of the loss function shown in Equation 7.2 is shown in Code 7.4. Line 3 calculates the Euclidean distances between output vectors in the tensor `x1` and `x2`. Line 5 is the implementation of Equation 7.2. Line 6 calculate the scalar loss value by summing up the losses in a batch.

The model parameters of the sub-networks are shared, so essentially there is only one set of model parameters in the memory. The gradient will be calculated through backpropagation w.r.t. the loss function, which needs the two different outputs from two different inputs. Code 7.5 shows a code snippet modified from the Gemini repo. The function `graph_embed` is showing earlier in Figure 7.6. As one may noticed, since the sub-networks share the weight, there is only one copy for each weight, and all of them are defined in this snippet at line 5, line 7 to line 10, line 12, and line 14. For those who are not familiar with Tensorflow 1.X versions, `tf.placeholder` is a node on the computing graph that can hold data (e.g. input data) during the runtime, and these variables such as `X1`, `msg1_mask` will hold inputs from the dataset. Here, two different samples in a pair will be processed identically, which is shown at line 20 and line 25. After obtaining the two embedding vectors from different inputs (i.e. `embed1` and `embed2`), the loss function can obtained as shown in Code 7.4. Then the gradients of the

```
# N_x: dimension of the node feature
# N_embed: dimension of the hidden layer outputs
# N_o: dimension of the output

Wnode = tf.Variable(tf.truncated_normal(
    shape = [N_x, N_embed], stddev = 0.1, dtype = tf.float32))
Wembed = []
for i in range(depth_embed):
    Wembed.append(tf.Variable(tf.truncated_normal(
        shape = [N_embed, N_embed], stddev = 0.1, dtype = tf.float32)))

W_output = tf.Variable(tf.truncated_normal(
    shape = [N_embed, N_o], stddev = 0.1, dtype = tf.float32))
b_output = tf.Variable(tf.constant(0, shape = [N_o], dtype = tf.float32))

X1 = tf.placeholder(tf.float32, [None, None, N_x])
msg1_mask = tf.placeholder( tf.float32, [None, None, None])

embed1 = graph_embed(X1, msg1_mask, N_x, N_embed, N_o, ITER_LEVEL,
        Wnode, Wembed, W_output, b_output)

X2 = tf.placeholder(tf.float32, [None, None, N_x])
msg2_mask = tf.placeholder(tf.float32, [None, None, None])
embed2 = graph_embed(X2, msg2_mask, N_x, N_embed, N_o, ITER_LEVEL,
        Wnode, Wembed, W_output, b_output)
```

Code 7.5: A code snippet from Gemini to illustrate forward inference of the Siamese network.

loss function w.r.t. the parameters can be computed.

7.5 Code, Data and Other Issues

7.5.1 Code and Data Resources

The source code for training and testing Gemini in non-production environment can be found at https://github.com/xiaojunxu/dnn-binary-code-similarity.

7.5.2 Model Performance

The model performance varies depending the implementation, model, tuning, and the dataset. Similar to recommender models, the performance of a code similarity detection model usually uses Precision@k and Recall@k as the major metric.

We introduce the evaluation of α-Diff and Gemini here. α-Diff [46] evaluates the model using sequence data and CNN comprehensively. The average Recall@1 is 0.953 and Recall@5 is 0.996, which is significantly better than most traditional methods. When doing cross version binary code similarity on `coreutils`, the best Recall@1 is 0.738 and Recall@5 is 0.821 when comparing a newer version with an older version 14 years ago. Gemini evaluates the GNN model using graph data structure. Unfortunately, Gemini did not use Precision@k or Recall@k, but the author did provide ROC-AUC score to be 0.971, which can be considered as ROC@1.

Although based on the reported number, the overall performance of the graph model seems better than the sequence model, it is not fair to make the comparison just based on the numbers reported

by the two paper. The reason is that α-Diff is doing code similarity detection on program-level, and Gemini is on function-level. Therefore, readers should select model based on their own experiments and needs. In short, graph model relies more on program analysis domain knowledge, but it is easier to be extended and further customized compared to the sequence model. Sequence model is usually easier to implement and faster to train, but since it is taking encoded raw bytes as input, less customization is available.

7.5.3 Active Learning

As mentioned in section 7.4, labeled data is needed to train a siamese network. In most binary code similarity detection scenarios, the data is labeled by compiling a same pieces of source code into different binary files using different compiler settings. However, as shown by the authors of Gemini, the performance could be improved by including some samples that are similar in high-level semantics but different in the source code, so that manually labeling data may be occasionally needed. In such case, due to the high cost of manually labeling the data, active learning becomes useful.

In classical classification task, the model outputs the logits or the Softmax scores, which is the probability of the given input being each class. In other words, the output in a classification task is essentially how confident the model is when it is predicting. Based on this fact, it is possible to select those samples model predicts not so confidently to label, and discard other samples. In such a way, the human effort to label the data can be more efficient. Such techniques to select samples to label are one kind of active learning.

However, for a siamese network, it is not very practical to use the active learning techniques based on the Softmax score, because siamese network outputs the embedding of the inputs, which gives no information about whether the model is confident in inferring a sample. The difficulty of general active learning of neural networks is because neural networks are black boxes, resulting challenges to interpret which samples are important to them. Therefore, one way to resolve this issue is to have another sub-network to learn whether a network feels confident on some samples, which is proposed in [91]. Essentially, one can create a neural network to predict the loss value of a model, given the input data. If the predicted loss of a input sample is high, then the sample should be labeled.

Chapter 8 AI Conducts Malware Clustering

8.1 The Security Problem

Malware, short for malicious software, has been one of the major threats that Internet users are facing today. Many Internet security problems are caused by the existence of malware. Malware comes in a wide range of variations, including viruses, worms, Trojans, spyware, ransomware, scareware, bots, rootkits and hybrid malware. Each type of malware has its own purposes and proliferation ways. Anti-malware companies protect Internet users from the threats of malware by providing defense against changeable malware. Many automatic malware analysis tools are developed to detect malware by running malware in controlled environments and then generate reports regarding malware behaviors.

However, due to economic benefits that malware can usually bring, the variation, sophistication and quantity of malware increase significantly. Thousands of new malware samples are usually received by anti-malware vendors every day. Despite the fact that automatic malware analysis tools exist to examine the behavior of one piece of malware, it is only part of the defense and is not sufficient. When facing a large number of malware samples in a short period of time, which is always the situation Internet users are dealing with, there is a need for anti-malware vendors to have the ability to prioritize reports of malware samples that require most attention. One approach is to cluster malware samples that exhibit similar behaviors. This enables malware analysts to quickly and automatically detect and focus on malware that either is novel and interesting or requires immediate attention.

Although malware clustering (through unsupervised machine learning) can be very helpful, many existing malware clustering methods suffer from two main limitations: one is their limited capability to cluster related malware samples together, the other is their limited effectiveness to do large scale malware clustering. In this chapter, we present a use case based on a mature malware clustering technique that can already be used by anti-malware companies to combat increasing sophisticated malware. This use case is specifically described based on a previous research paper titled “Scalable, behavior-based malware clustering" [7]. The result shows this type of clustering approach is not only able to identify and group malware samples that exhibit similar behavior, but also very scalable.

8.2 Machine Learning Pipeline

Since the use case in this chapter does not utilize any deep learning techniques, we have a pipeline different from what we have introduced in Chapter 1. Figure 8.1 shows the overall process of malware clustering. Typically, there are 3 steps included. In the *first step*, analyses are carried on malware samples to gather information about the malware. Analyses carried on malware are either static or dynamic, with only one difference that whether the piece of malware is executed during the analysis process. In the *second step*, malware features are extracted based on the analysis result in the previous step. The features are always carefully chosen so that the extracted features can represent the different

behaviors of different malware samples clearly. Well-chosen features lay a solid foundation for precise malware clustering. In the *last step*, a clustering algorithm is used to sort malware samples into clusters based on the extracted features.

Figure 8.1: Basic pipeline of malware clustering.

The malware clustering technique described in this chapter utilizes dynamic analysis to gather information about malware samples. subsection 8.4.1 gives more details on how the analysis is done. Previous research has shown that static analysis has its drawback when dealing with runtime packing code and complex obfuscations, which are often used in sophisticated malware. Also, the possibility of writing semantically-equal programs that are greatly different in their code makes it a must to deploy dynamic analysis. Taint tracking is integrated during the dynamic analysis process to obtain important features that better capture the behavior of malware samples. The output of the dynamic analysis phase is an execution trace with taint information. The trace is then summarized into a **behavior profile**, an abstraction of the execution trace which contains information around system-call related resources, such as files or registry keys. Next, a behavior profile is transformed into a feature set. The output of this step is a set of features in a form that is suitable for the clustering algorithm. subsection 8.4.2 gives more details on how the transformation is done. In the last step, scalable clustering of malware samples is done by using an algorithm based on locality-sensitive hashing (LSH).

8.3 Example Data

Before we dive into details of how each step of this use case is carried out, we show an example of behavior profile and an example of extracted features to give readers a direct impression of what they look like. Figure 8.2a shows the corresponding behavior profile of a piece of program. Note that although the program is shown in C programming language, the algorithm of generating behavior profiles works directly on execution traces generated by dynamic analysis.

Figure 8.2b shows three examples of the individual features that are extracted based on control flow dependency information held in behavior profiles. Bear in mind that, for each malware sample, there will be one corresponding behavior profile and one set of features transformed from that.

8.4 Feature Extraction

Before describing the feature extraction process, we have to admit that building a customized dynamic malware analysis tool is beyond the scope of this chapter. Instead, we provide readers with some pointers on how to use existing dynamic malware analysis tools to get proper analysis traces.

(a) Example of behavior profile **(b)** Example of extracted features

Figure 8.2: Illustration of examples (copied from [7]).

8.4.1 Dynamic analysis with taint tracking

Unlike many previous malware clustering systems that directly use low-level data such as system calls, this use case enriches and generalizes the information collected from dynamic analysis with taint tracking. The upgraded information is summarized into behavioral profiles, which display malware behaviors in terms of Operating System (OS) Object and OS Operations.

OS Objects. "An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. Formally, an OS object is a tuple of the following form:

OS Object ::= (type, [object-names])
type ::= dll|file|hook|network|object|process|registry|service|sync|system-info|time" [7]

OS Operations. "An OS operation is a generalization of a system call. Formally, an operation is defined as:

OS Operation ::= (operation-name, [operation-attributes], return_value)" [7]

Behavioral Profile. As stated in [7], "A behavioral profile P is defined as an 8-tuple:

$$P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$$

where O is the set of all OS objects, OP is the set of all OS operations, $\Gamma \subseteq (O \times OP)$ is a relation assigning one or several operations to each object, and $\Delta \subseteq ((O \times OP) \times (O \times OP))$ represents the set of dependences. CV is the set of all compare operations of type label-value, while CL is the set of all compare operation of type label-label. $\Theta_{CmpValue} \subseteq (CV \times O)$ is a relation assigning label-value compare operations to an OS object. $\Theta_{CmpLabel} \subseteq (CL \times O \times O)$ is a relation assigning label-label compare operations to the two appropriate OS objects."

In general, system calls are treated as taint sources in the taint tracking system. To be more precise, out-arguments and return values of all system calls are tainted. Three types of information are included inside a behavior profile:

- *System Call Dependences.* Every in-argument of the system calls is checked. If tainted, a dependence between the taint origin system call and the current system call is created.
- *Control Flow Dependences.* Compare instructions that involve tainted data (result of system calls) are recorded. Both label-value comparison (comparison between untainted and tainted value) and label-label comparison (comparison between 2 tainted values) are summarized inside behavior profiles.
- *Network Analysis.* Analyze the relevant network behaviors.

Readers interested in carrying out the dynamic analysis on malware samples can refer to PANDA [1], a Platform for Architecture-Neutral Dynamic Analysis, for extraction of system call traces and taint information. Here, we list some available plugins that are relevant to our use case [2].

8.4.2 Feature Generation

Behavior profiles of malware samples need to be further processed in order to be suitable to use in later on clustering steps. As stated in [7], "A behavior profile $P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$ is transformed into a feature set based on the following algorithm:

For each object $o_i \in O$, and for each assigned $op_j \in OP | (o_i, a) \in \Gamma$, create a feature:
$f_{ij} = \text{“}op|\text{”} + name(o_i) + \text{“}|\text{”} + name(op_j)$
For each dependence $\delta_i \in \Delta = ((o_{i1}, op_{i1}), (o_{i2}, op_{i2}))$, we create a feature:
$f_i = \text{“}dep|\text{”} + name(o_{i1}) + \text{“}|\text{”} + name(op_{i1}) + \text{“} \rightarrow \text{”} + name(o_{i2}) + \text{“}|\text{”} + name(o_{i2})$
For each label-value comparison $\theta_i \in \Theta_{CmpValue} = (cmp, o)$, we create a feature:
$f_i = \text{“}cmp\_value|\text{”} + name(o) + \text{“}|\text{”} + name(cmp)$
For each label-label comparison $\theta_i \in \Theta_{CmpLabel} = (cmp, o_1, o_2)$, we create a feature:
$f_i = \text{“}cmp\_label|\text{”} + name(o_1) + + \text{“} \rightarrow \text{”} + name(o_2) + \text{“}|\text{”} + name(cmp)$

where $name()$ is a function that returns the name of an OS object, operation, or comparison as string; quotes denote a literal string; and '+' concatenates two strings."

8.5 Scalable Clustering

After the feature extraction process described in section 8.4, each malware sample is represented as a set of features. Beginning from this section, we are going to introduce the process of clustering malware samples based on the extracted features sets. To make the clustering process scalable, which enables anti-malware vendors to classify huge amount of received malware samples in a short period of time, a technique called Locality-Sensitive Hashing (LSH) is used. LSH is an algorithmic technique that hashes similar input items into the same "buckets" with high probability. Since similar items end up in the same buckets, this technique can be used for data clustering and nearest neighbor search

[1] https://github.com/panda-re/panda

[2] System call tracing: https://github.com/panda-re/panda/tree/dev/panda/plugins/syscalls2,
General tainting: https://github.com/panda-re/panda/tree/dev/panda/plugins/taint2,
Control flow tainting: https://github.com/panda-re/panda/tree/dev/panda/plugins/tainted_branch

	m_1	m_2
f_1	1	0
f_2	0	1
f_3	0	0
f_4	1	1
f_5	0	1

Table 8.1: Boolean matrix representing two malware samples.

[81]. The result of LSH presents a set consisting of pairs of similar malware samples. The set of pairs is then sorted by similarity, which allows us to produce a single-linkage hierarchical clustering. Figure 8.3 shows the process of finding the set consisting of similar pairs of malware samples.

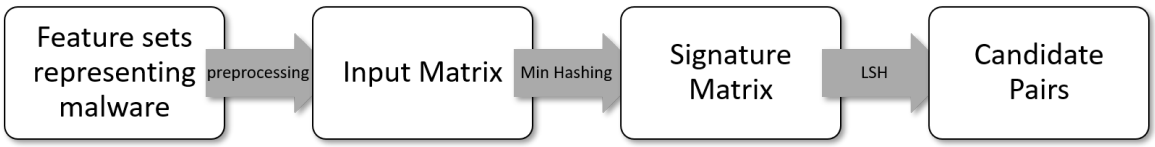


Figure 8.3: Overall LSH process.

8.5.1 Input Matrix

Malware samples need to be processed further into an input matrix before they can be handled by the LSH algorithm. Every malware sample can be represented as a set of features. Therefore, the set of all malware samples can be represented as a Boolean matrix, with rows representing all the features and columns representing all the samples. 1 in row x and column Y if and only if x is a member of Y. Table 8.1 depicts an example Boolean input matrix that represents 2 malware samples with five features in total. Note that features that are unique to one malware sample should be discarded while constructing the matrix, since they are seldom useful in finding similar malware pairs.

8.5.2 Jaccard Similarity

To determine the similarity of two malware samples, Jaccard similarity is used. Jaccard similarity of two sets, set U and set V, is defined as the size of the intersection divided by the size of the union of the sample sets:

$$Jaccard(U, V) = \frac{|U \cap V|}{|U \cup V|} \tag{8.1}$$

In our case, Jaccard similarity between two example malware samples in Table 8.1 is:

$$Jaccard(m_1, m_2) = \frac{|m_1 \cap m_2|}{|m_1 \cup m_2|} = \frac{1}{4} \tag{8.2}$$

8.5.3 Min Hashing

For a large number of malware samples and feature sets, the input matrix can be very sparse. Further processing of a sparse matrix can be very time and space consuming. Therefore, hashing is used to compress an input matrix into a signature matrix that is much more condense. Hash function H should hold the following properties:

1. If $similarity(m_1, m_2)$ is high then $Prob(H(m_1) == H(m_2))$ is high.
2. If $similarity(m_1, m_2)$ is low then $Prob(H(m_1) == H(m_2))$ is low.

The choice of hash functions is highly related to similarity metrics. For Jaccard similarity, min hashing is appropriate. The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets [43].To minhash an input matrix representing all malware samples, a permutation of the row indices is generated. The minhash of any column is the smallest index of the first row, in the permuted order, in which the column has a 1 [9]. The number of row-index permutations determines the number of minhash results. That is, for example, if there are 100 row-index permutations, then there is a signature column of length 100 for each column of the original input matrix. An example of Min Hashing with 3 row-index permutations, 7x4 input matrix and the resulting 3x4 signature matrix is shown in Figure 8.4.

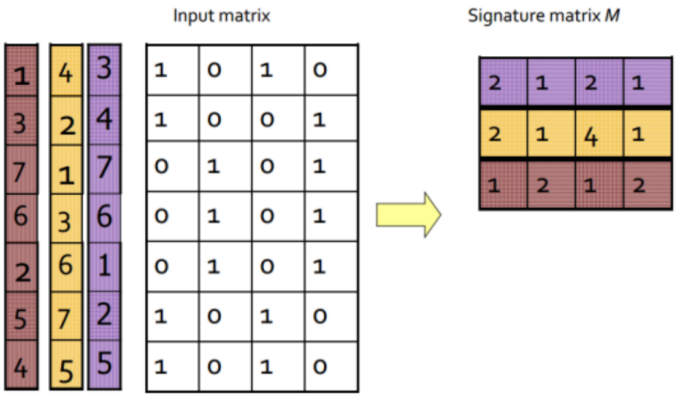


Figure 8.4: Example of Min Hashing (Copy from [9]).

Practical implementation of min hashing. In practice, it is inefficient to generate truly random permutations when the feature set is huge. One way to solve this problem is to use random linear functions in the form $h_i(x) = a_{i_1}x + a_{i_2} \mod P$, with P equals to the total number of features. Pseudo code of computing the signature matrix $M(i, c)$ is provided in **Algorithm 1**.

8.5.4 Locality-Sensitive Hashing

The goal of locality-sensitive hashing is to generate a set of candidate pairs whose similarity is greater than a threshold t. Note that we take the similarity of columns in the signature matrix as an indication of similarity of original columns in the input matrix. Since comparing all the rows in a signature matrix column individually can still be both time and space consuming, the intuition of LSH is that, we hash the columns of the signature matrix using several hash functions. If any of the two

Algorithm 1 Practical Min Hashing calculating $M(i, c)$ (copied from [53])

```
for each row r do
    for each hash function h_i do
        Compute h_i(r);
    end for
    for each column c do
        if c has 1 in row r then
            for each hash function h_i do
                if h_i(r) is smaller than M(i, c) then
                    M(i, c) := h_i(r);
                end if
            end for
        end if
    end for
end for
```

columns hash into the same bucket for at least one of the hash functions, we treat the two columns, which represent two malware samples, as a candidate pair.

Banding techniques. If we have a minhash signatures matrix, an effective way to choose the hashings is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets. Same hash function can be used for all the bands, but a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket [43]. Figure 8.5 shows an example of a hash function for one band.

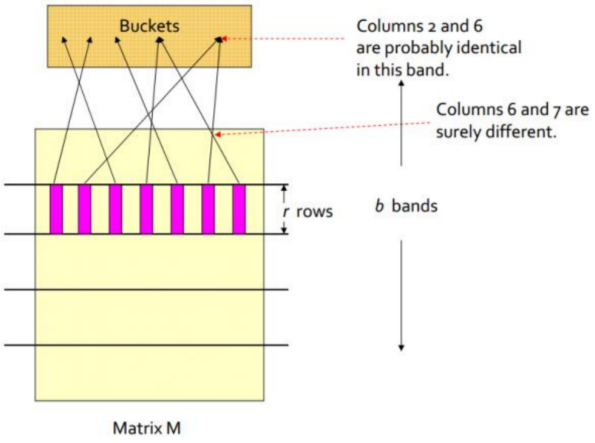


Figure 8.5: Hashing for one band (Copy from [9]).

Parameter choosing. As stated in [43], "Suppose we use b bands of r rows each, and suppose that a particular pair of documents have Jaccard similarity s. The probability the minhash signatures for these documents agree in any one particular row of the signature matrix is s. We can calculate the probability that these documents (or rather their signatures) become a candidate pair as follows:

1. The probability that the signatures agree in all rows of one particular band is s^r.
2. The probability that the signatures disagree in at least one row of a particular band is $1 - s^r$.

3. The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$.
4. The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

It may not be obvious, but regardless of the chosen constants b and r, this function has the form of an S-curve, as suggested in Figure 8.6. The threshold is roughly where the rise is the steepest, and for large b and r we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want. An approximation to the threshold is $(1/b)^{1/r}$." Therefore, since $t = (1/b)^{1/r}$, by only defining one parameter, a suitable t, with $br = n$, where n stands for the total number of features, we will be able to find appropriate r and b.

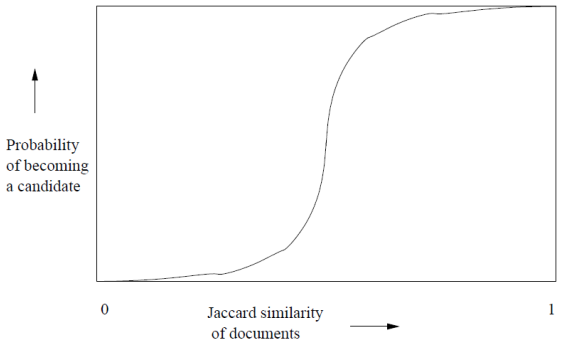


Figure 8.6: The S curve (Copy from [43]).

Code implementation. Some code that can be used to implement LSH is shown below in Code 8.1 and Code 8.2.

8.6 Clusters Deployment

As can be concluded from section 8.5, the result set S from locality-sensitive hashing can only be an approximation of the true set T of all near pairs. Therefore, there will possibilities that pairs that have a similarity lower than the threshold t that are included in set S. We remove them by calculating the similarity of each pair and discarding pairs that have a similarity lower than threshold. We get a set of candidate pairs whose similarity is greater than a threshold t. With this set, we can iterate over the set of pairs, sort them by similarity and produce a single-linkage agglomerative clustering.

We use python package `scikit-learn 1.0.2` to implement single-linkage agglomerative clustering. Note that parameter $affinity$ is a metric used to compute the linkage. "A distance matrix instead of a similarity matrix is needed as input for the fit method [65]." We can easily calculate the *Jaccard Distance* between a pair by *1 – Jaccard Similarity*. As for the optimal number of clusters to find, $n\_clusters$, *Dendrogram* method can be used to find the appropriate parameter.

[3]Copy from https://github.com/go2starr/lshhdc

[4]Copy from https://github.com/go2starr/lshhdc

```
class LSH_Util(object):
def __init__(self, length, threshold):
    self.length = length
    self.threshold = threshold
    self.bandwidth = self.get_bandwidth(length,
         threshold)
def hash(self, sig):
    for band in zip( * (iter(sig),) * self.
         bandwidth):
        yield hash(unicode(band))
def get_bandwidth(self, n, t):
    best = n, 1
    minerr = float("inf")
    for r in range(1, n + 1):
        try:
            b = 1. / (t ** r)
        except:
            return best
        err = abs(n - b * r)
        if err < minerr:
            best = r
            minerr = err
    return best
def get_threshold(self):
    r = self.bandwidth
    b = self.length / r
    return (1. / b) ** (1. / r)
def get_n_bands(self):
    return int(self.length / self.bandwidth)
```

Code 8.1: Code for LSH utility [3].

```
class LSH(object):
def __init__(self, width=10, threshold=0.5):
    self.width = width
    self.signer = MinHashSignature(width)
    self.hasher = LSH_Util(width, threshold)
    self.hashmaps = [defaultdict(list)
                     for _ in range(self.hasher.
                          get_n_bands())]

def add_sample(self, s, label=None):
    # A label for this sample
    if not label:
        label = s

    # Get minhash signature
    sig = self.signer.sign(s)

    # Contruct hashmaps storing the LSH hashes
    for band_idx, hshval in enumerate(self.
         hasher.hash(sig)):
        self.hashmaps[band_idx][hshval].append(
             label)

# Iterate through hashmaps to get candidate
     pairs
    ...
```

Code 8.2: Code for implementing LSH [4].

```
from sklearn.cluster import AgglomerativeClustering
distance_matrix = compute_distance_matrix()
model = AgglomerativeClustering(affinity='precomputed', n_clusters=n, linkage='single').fit(distance_matrix)
```

Code 8.3: Code for implementing single-linkage hierarchical clustering.

Since the result of LSH only provides information regarding pairs that have similarity higher than the threshold, information about subsequent clustering steps to merge clusters that have a similarity below threshold is not readily available. To obtain an exhaustive hierarchical clustering, exact hierarchical clustering still needs to be performed on representatives of readily available clusters and all the rest of the clusters. Although exact hierarchical clustering has a time complexity of $O(n^2)$, the result is acceptable since the number of the representatives and the rest of the clusters are usually not large.

8.7 Concluding Remarks

8.7.1 Time Complexity

For large datasets, the time complexity of the algorithm used in this use case is dominated by the cost of calculating the similarity of each pair. $|T| < nc$, where T is the true set of all near pairs, n is the number of malware samples, c is the size of the maximum cluster. The time complexity of calculating similarity of each pair is $O(ncd)$ with each pair costing a constant time $O(d)$. [7]

8.7.2 Limitations and Possible Extensions

For extremely large datasets, the time complexity may be high. Some aggressive clustering schemes can be used with some sacrifice in accuracy. Since features representing malware behavior are extracted by dynamic analysis, malware intentionally modified to behave differently from others in the same cluster will weaken the clustering performance.

8.7.3 Additional Resources

Some dynamic malware analysis tools that support taint tracking are as follows: https://panda.re/; https://www.usenix.org/conference/raid2019/presentation/davanian. More information about LSH can be found in http://www.mmds.org/.

Bibliography

[1] Yousra Aafer, Wenliang Du, and Heng Yin. "Droidapiminer: Mining Api-Level Features for Robust Malware Detection in Android". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2013, pp. 86–103.

[2] Martin Abadi et al. "Control-flow Integrity Principles, Implementations, and Applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[3] Y. Xin et al. "Machine Learning and Deep Learning Methods for Cybersecurity". In: *IEEE Access* (2018).

[4] *Android Emulator*. Android.com, 2020. URL: https://developer.android.com/studio/run/emulator.

[5] Steven Arzt et al. "Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[6] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics." In: *NDSS*. 2018.

[7] Ulrich Bayer et al. "Scalable, behavior-based malware clustering." In: *NDSS*. 2009.

[8] Christopher M Bishop. "Pattern Recognition". In: *Machine Learning* 128.9 (2006).

[9] Ivan Blekanov and Vasilii Korelin. "Hierarchical clustering of large text datasets using Locality-Sensitive Hashing". In: *IWAIT Workshop*. 2015.

[10] Jane Bromley et al. "Signature Verification Using a "Siamese" Time Delay Neural Network". In: *International Journal of Pattern Recognition and Artificial Intelligence* 7.04 (1993), pp. 669–688.

[11] Haipeng Cai et al. "Droidcat: Effective Android Malware Detection and Categorization via App-level Profiling". In: *IEEE Transactions on Information Forensics and Security* 14.6 (2018), pp. 1455–1470.

[12] Ligeng Chen, Zhongling He, and Bing Mao. "CATI: Context-Assisted Type Inference from Stripped Binaries". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2020, pp. 88–98.

[13] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 1037–1052.

[14] Yoon-Ho Choi et al. "Using Deep Learning to Solve Computer Security Challenges: A Survey". In: *Cybersecurity* (2020).

[15] Mihai Christodorescu and Somesh Jha. "Testing Malware Detectors". In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pp. 34–44.

[16] Zheng Leong Chua et al. “Neural Nets Can Learn Function Type Signatures from Binaries”. In: *26th USENIX Security Symposium USENIX Security 17)*. 2017, pp. 99–116.

[17] Zhihua Cui et al. “Detection of Malicious Code Variants Based on Deep Learning”. In: *IEEE Transactions on Industrial Informatics* 14.7 (2018), pp. 3187–3196.

[18] Dahl et al. “Large-scale Malware Classification Using Random Projections and Neural Networks”. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (2013), pp. 3422–3426.

[19] Leonardo De La Rosa et al. “Efficient Characterization and Classification of Malware Using Deep Learning”. In: *Proceedings - Resilience Week 2018, RWS 2018* (2018), pp. 77–83.

[20] Erik Derr. *axplorer*. 2017. URL: https://github.com/reddr/axplorer.

[21] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. “Asm2vec: Boosting Static Representation Robustness for Binary Clone Search Against Code Obfuscation and Compiler Optimization”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 472–489.

[22] dlgroupuoft. *PScout*. 2018. URL: https://github.com/dlgroupuoft/PScout.

[23] Min Du et al. “Deeplog: Anomaly detection and diagnosis from system logs through deep learning”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298.

[24] Chris Eagle. *The IDA pro book*. No Starch Press, 2011.

[25] Daniel R Ellis et al. “A Behavioral Approach to Worm Detection”. In: *Proceedings of the 2004 ACM workshop on Rapid malcode*. 2004, pp. 43–53.

[26] William Enck et al. “Taintdroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), pp. 1–29.

[27] *Feature (Machine Learning)*. accessed: 2022-1-07. Wikipedia, Wikipedia Foundation. URL: https://en.wikipedia.org/wiki/Feature_(machine_learning).

[28] *Feature Vector*. Accessed: 2022-01-10. URL: https://brilliant.org/wiki/feature-vector/.

[29] Adrienne Porter Felt et al. “Android Permissions Demystified”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 627–638.

[30] Adrienne Porter Felt et al. “Permission Re-Delegation: Attacks and Defenses.” In: *USENIX Security Symposium*. Vol. 30. 2011, p. 88.

[31] Qian Feng et al. “Scalable Graph-based Bug Search for Firmware Images”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 480–491.

[32] *GCC, the GNU Compiler Collection*. Accessed: 2021-10-01. URL: https://gcc.gnu.org.

[33] Liangyi Gong et al. “Experiences of Landing Machine Learning onto Market-scale Mobile Malware Detection”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–14.

[34] Liangyi Gong et al. “Systematically Landing Machine Learning onto Market-Scale Mobile Malware Detection”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.7 (2020), pp. 1615–1628.

[35] Michael Grace et al. “Riskranker: Scalable and Accurate Zero-day Android Malware Detection”. In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 2012, pp. 281–294.

[36] Wenbo Guo et al. “DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Post-mortem Program Analysis”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1787–1804.

[37] Michiel Hermans and Benjamin Schrauwen. “Training and analysing deep recurrent neural networks”. In: *Advances in neural information processing systems* 26 (2013), pp. 190–198.

[38] Hyungjoon Koo et al. “Semantic-aware Binary Code Representation with BERT”. In: *arXiv preprint arXiv:2106.05478* (2021).

[39] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[40] Quoc Le and Tomas Mikolov. “Distributed Representations of Sentences and Documents”. In: *International Conference on Machine Learning*. PMLR. 2014, pp. 1188–1196.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[42] Young Jun Lee et al. “Learning Binary Code with Deep Learning to Detect Software Weakness”. In: *KSII the 9th International Conference on Internet (ICONI) 2017 Symposium*. 2017.

[43] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. 2nd. USA: Cambridge University Press, 2014. ISBN: 1107077230.

[44] Xuezixiang Li, Qu Yu, and Heng Yin. “PalmTree: Learning an Assembly Language Model for Instruction Embedding”. In: *arXiv preprint arXiv:2103.03809* (2021).

[45] Yujia Li et al. “Graph Matching Networks for Learning the Similarity of Graph Structured Objects”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3835–3845.

[46] Bingchang Liu et al. “αdiff: Cross-version Binary Code Similarity Detection with Dnn”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 667–678.

[47] Liu Liu et al. “Insider Threat Identification using The Simultaneous Neural Learning of Multi-source Logs”. In: *IEEE Access* 7 (2019), pp. 183162–183176.

[48] Liu Liu et al. “Unsupervised Insider Detection through Neural Feature Learning and Model Optimisation”. In: *International Conference on Network and System Security*. Springer. 2019, pp. 18–36.

[49] Aravind Machiry et al. “Using Loops for Malware Classification Resilient to Feature-unaware Perturbations”. In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, Dec. 2018, pp. 112–123.

[50] Mark McDermott. “Presentation: The ARM Instruction Set Architecture”. In: (2008). URL: http://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf.

[51] John H. McDonald. *Spearman Rank Correlation*. 2019. URL: http://www.biostathandbook.com/spearman.html.

[52] Niall McLaughlin et al. “Deep Android Malware Detection”. In: *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy* (2017), pp. 301–308.

[53] *Mining of massive datasets*. URL: http://www.mmds.org/.

[54] Robert Monarch. *Human-in-the-Loop Machine Learning*. Manning Publications Corp., 2021.

[55] Akira Mori et al. “A Tool for Analyzing and Detecting Malicious Mobile Code”. In: *ICSE*. Vol. 2006. May 2006, pp. 831–834.

[56] Nour Moustafa and Jill Slay. “UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set)”. In: *2015 Military Communications and Information Systems Conference (MilCIS)*. IEEE. 2015, pp. 1–6.

[57] Robin Nix and Jian Zhang. “Classification of Android Apps and Malware Using Deep Neural Networks”. In: *Proceedings of the International Joint Conference on Neural Networks* 2017-May (2017), pp. 1871–1878.

[58] Jannik Pewny et al. “Leveraging Semantic Signatures for Bug Search in Binary Programs”. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 406–415.

[59] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. “Convolutional Neural Networks Over Control Flow Graphs for Software Defect Prediction”. In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2017, pp. 45–52.

[60] Yuval Pinter, Robert Guthrie, and Jacob Eisenstein. “Mimicking Word Embeddings using Subword RNNs”. In: *arXiv preprint arXiv:1707.06961* (2017).

[61] Samira Pouyanfar et al. “A Survey on Deep Learning: Algorithms, Techniques, and Applications”. In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36.

[62] Edward Raff et al. “Malware Detection by Eating a Whole exe”. In: *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[63] *Random Forest*. Accessed: 2022-1-11. URL: https://en.wikipedia.org/wiki/Random_forest.

[64] Suhita Ray. "Disease Classification within Dermascopic Images using Features Extracted by Resnet50 and Classification through Deep Forest". In: *arXiv preprint arXiv:1807.05711* (2018).

[65] *Scikit-learn*. INRIA, 2010. URL: https://scikit-learn.org/stable/index.html.

[66] *sendTextMessage*. Accessed: 2022-1-26. URL: https://developer.android.com/reference/android/telephony/SmsManager#sendTextMessage(java.lang.String,%5C%20java.lang.String,%5C%20java.lang.String,%5C%20android.app.PendingIntent,%5C%20android.app.PendingIntent,%5C%20long).

[67] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. "Recognizing Functions in Binaries with Neural Networks". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 611–626.

[68] *Sophisticated new Android Malware Marks the Latest Evolution of Mobile Ransomware*. microsoft.com, 2020. URL: https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/.

[69] *Stamina Scalable Deep Learning Whitepaper*. Accessed: 2021-09-30. URL: https://www.intel.com/content/dam/www/public/us/en/ai/documents/stamina-scalable-deep-learning-whitepaper.pdf.

[70] Andrew Sung et al. "Static Analyzer of Vicious Executables (SAVE)". In: *ACSAC*. 2005.

[71] *The DWARF Debugging Standard*. DWARF Standards Committee, 2012. URL: https://dwarfstd.org.

[72] Tobiyama et al. "Malware Detection with Deep Neural Network Using Process Behavior". In: *International Computer Software and Applications Conference* 2 (2016), pp. 577–582. ISSN: 07303157.

[73] *Ubuntu Software Packages*. https://packages.ubuntu.com/bionic/. Accessed: 2021-09-30.

[74] *UI/Application Exerciser Monkey in Android Studio*. Android.com, 2008. URL: https://developer.android.com/studio/test/monkey.html.

[75] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. 2017, pp. 5998–6008.

[76] *Virusshare*. Accessed: 2021-09-30. URL: https://virusshare.com/.

[77] *VirusTotal*. Accessed: 2019-09-30. URL: https://www.virustotal.com/.

[78] Zhilong Wang et al. "Identifying Non-Control Security-Critical Data in Program Binaries with a Deep Neural Model". In: *arXiv preprint arXiv:2108.12071* (2021).

[79] Zhilong Wang et al. "Spotting Silent Buffer Overflows in Execution Trace through Graph Neural Network Assisted Data Flow Analysis". In: *arXiv preprint arXiv:2102.10452* (2021).

[80] *What Is A Malware File Signature (And How Does It Work)?* Accessed: 2021-09-30. URL: https://www.sentinelone.com/blog/what-is-a-malware-file-signature-and-how-does-it-work/.

[81] Wikipedia. *Locality-sensitive hashing — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Locality-sensitive%20hashing&oldid=1062941845.

[82] Michelle Y Wong and David Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." In: *NDSS*. 2016.

[83] Dong-Jie Wu et al. "Droidmat: Android Malware Detection through Manifest and Api Calls Tracing". In: *2012 Seventh Asia Joint Conference on Information Security*. IEEE. 2012, pp. 62–69.

[84] Wen-Chieh Wu and Shih-Hao Hung. "DroidDolphin: A Dynamic Android Malware Detection Framework using Big Data and Machine Learning". In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*. 2014, pp. 247–252.

[85] *XposedBridge*. Rovo89, 2016. URL: https://github.com/rovo89/Xposed%20Bridge/wiki/Development-tutorial.

[86] Lifan Xu et al. "Hadm: Hybrid Analysis for Detection of Malware". In: *Proceedings of SAI Intelligent Systems Conference*. Springer. 2016, pp. 702–724.

[87] Wei Xu et al. "Largescale System Problem Detection by Mining Console Logs". In: *Proceedings of SOSP'09* (2009).

[88] Xiaojun Xu et al. "Neural Network-based Graph Embedding for Cross-platform Binary Code Similarity Detection". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 363–376.

[89] Ruipeng Yang et al. "NLSALog: An anomaly detection framework for log sequence in security management". In: *IEEE Access* 7 (2019), pp. 181152–181164.

[90] Tianda Yang et al. "Automated Detection and Analysis for Android Ransomware". In: *IEEE 7th International Symposium on Cyberspace Safety and Security*. 2015.

[91] Donggeun Yoo and In So Kweon. "Learning Loss for Active Learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 93–102.

[92] Ilsun You and Kangbin Yim. "Malware Obfuscation Techniques: A Brief Survey". In: *2010 International conference on broadband, wireless computing, communication and applications*. IEEE. 2010, pp. 297–300.

[93] Lun-Pin Yuan, Peng Liu, and Sencun Zhu. "Recomposition vs. Prediction: A Novel Anomaly Detection for Discrete Events Based On Autoencoder". In: *arXiv preprint arXiv:2012.13972* (2020).

[94] Muhan Zhang et al. "An End-to-end Deep Learning Architecture for Graph Classification". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[95] Yajin Zhou and Xuxian Jiang. “Dissecting Android Malware: Characterization and Evolution”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 95–109.

[96] Yajin Zhou et al. “Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.” In: *NDSS*. 2012.

[97] Qingtian Zou et al. “Deep Learning for Detecting Network Attacks: An End-to-end Approach”. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2021, pp. 221–234.