Using Deep Learning to Solve Computer Security Challenges: A Survey*

Yoon-Ho Choi[2,1], Peng Liu[1], Zitong Shang[1], Haizhou Wang[1], Zhilong Wang[1], Lan Zhang[1], Junwei Zhou[†3,1], and Qingtian Zou[1]

[1]The Pennsylvania State University, United States
[2]Pusan National University, Republic of Korea
[3]Wuhan University of Technology, China

Abstract

Although using machine learning techniques to solve computer security challenges is not a new idea, the rapidly emerging Deep Learning technology has recently triggered a substantial amount of interests in the computer security community. This paper seeks to provide a dedicated review of the very recent research works on using Deep Learning techniques to solve computer security challenges. In particular, the review covers eight computer security problems being solved by applications of Deep Learning: security-oriented program analysis, defending return-oriented programming (ROP) attacks, achieving control-flow integrity (CFI), defending network attacks, malware classification, system-event-based anomaly detection, memory forensics, and fuzzing for software security.

1 Introduction

Using machine learning techniques to solve computer security challenges is not a new idea. For example, in the year of 1998, Ghosh and others in [1] proposed to train a (traditional) neural network to do anomaly detection (i.e., detecting anomalous and unknown intrusions against programs); in the year of 2003, Hu and others in [2] and Heller and others in [3] applied Support Vector Machines to do anomaly detection (e.g., detecting anomalous Windows registry accesses).

The machine-learning-based computer security research investigations during 1990-2010, however, have not been very impactful. For example, to the best of our knowledge, none of the machine learning applications proposed in [1–3] has been incorporated into a widely deployed intrusion-detection commercial product.

Regarding why not very impactful, although researchers in the computer security community seem to have different opinions, the following remarks by Sommer and Paxson [4] (in the context of intrusion detection) have resonated with many researchers:

- Remark A: "It is crucial to have a clear picture of what problem a system targets: what specifically are the attacks to be detected? The more narrowly one can define the target activity, the better one can tailor a detector to its specifics and reduce the potential for misclassifications." [4]

*The authors of this paper are listed in alphabetic order.

†Corresponding author: junweizhou@msn.com

- Remark B: "If one cannot make a solid argument for the relation of the features to the attacks of interest, the resulting study risks foundering on serious flaws." [4]

These insightful remarks, though well aligned with the machine learning techniques used by security researchers during 1990-2010, may become a less significant concern with Deep Learning (DL), a rapidly emerging machine learning technology. Of course, it should be noticed that these remarks were raised before researchers started using DL to solve computer security challenges.

As stated in [5], "DL is a statistical technique that exploits large quantities of data as training sets for a network with multiple hidden layers, called a deep neural network (DNN). A DNN is trained on a dataset, generating outputs, calculating errors, and adjusting its internal parameters. Then the process is repeated hundreds of thousands of times until the network achieves an acceptable level of performance. It has proved to be an effective technique for image classification, object detection, speech recognition, and natural language processing—problems that challenged researchers for decades. By learning from data, DNNs can solve some problems much more effectively, and also solve problems that were never solvable before."

Now let's take a high-level look at how DL could make it substantially easier to overcome the challenges identified by Sommer and Paxson [4]. First, one major advantage of DL is that it makes learning algorithms less dependent on feature engineering. This characteristic of DL makes it easier to overcome the challenge indicated by Remark B. Second, another major advantage of DL is that it could achieve high classification accuracy with minimum domain knowledge. This characteristic of DL makes it easier to overcome the challenge indicated by Remark A.

Key observation. *The above discussion indicates that DL could be a game changer in applying machine learning techniques to solving computer security challenges.*

Motivated by this observation, this paper seeks to provide a *dedicated* review of the very recent research works on using Deep Learning techniques to solve computer security challenges. It should be noticed that since this paper aims to provide a dedicated review, non-deep-learning techniques and their security applications are out of the scope of this paper.

The remaining of the paper is organized as follows. In Section 2, we present a four-phase workflow framework which we use to summarize the existing works in a unified manner. In Section 3-10, we provide a review of eight computer security problems being solved by applications of Deep Learning, respectively. In Section 11, we will discuss certain similarity and certain dissimilarity among the existing works. In Section 12, we mention two further areas of investigation. In Section 13, we conclude the paper.

2 A four-phase workflow framework can summarize the existing works in a unified manner

We found that a four-phase workflow framework can provide a unified way to summarize all the research works surveyed by us. In particular, we found that each work surveyed by us employs a particular workflow when using machine learning techniques to solve a computer security challenge, and we found that each workflow consists of two or more phases. By "a unified way", we mean that every workflow surveyed by us is essentially an instantiation of a common workflow pattern which is shown in Figure 1.

2.1 Definitions of the four phases

The four phases, shown in Figure 1, are defined as follows. To make the definitions of the four phases more tangible, we use a running example to illustrate each of the four phases.

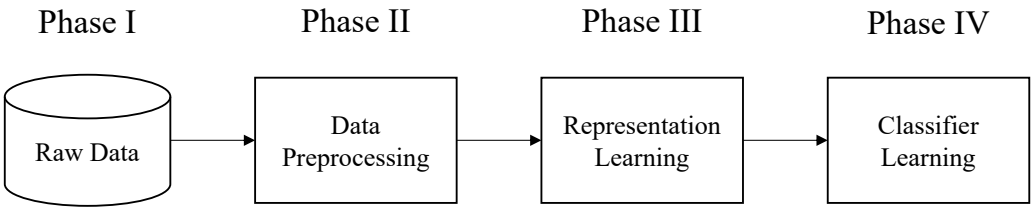


Figure 1: Overview of the four-phase workflow

Phase I.(Obtaining the Raw Data)

In this phase, certain raw data are collected.

Running Example: When Deep Learning is used to detect suspicious events in a Hadoop distributed file system (HDFS), the raw data are usually the events (e.g., a block is allocated, read, written, replicated, or deleted) that have happened to each block. Since these events are recorded in Hadoop logs, the log files hold the raw data. Since each event is uniquely identified by a particular (block ID, timestamp) tuple, we could simply view the raw data as n event sequences. Here n is the total number of blocks in the HDFS. For example, the raw data collected in [6] in total consists of 11,197,954 events. Since 575,139 blocks were in the HDFS, there were 575,139 event sequences in the raw data, and on average each event sequence had 19 events. One such event sequence is shown as follows:

```
081110 112428 31 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock:
  /user/root/rand/_temporary/_task_200811101024_0001_m_001649_0/
  part-01649.blk_-1033546237298158256
081110 112428 9602 INFO dfs.DataNode$DataXceiver:
  Receiving block blk_-1033546237298158256 src: /10.250.13.240:54015
  dest:/10.250.13.240:50010
081110 112428 9982 INFO dfs.DataNode$DataXceiver:
  Receiving block blk_-1033546237298158256 src: /10.250.13.240:52837
  dest:/10.250.13.240:50010
081110 112432 9982 INFO dfs.DataNode$DataXceiver:
  writeBlock blk_-1033546237298158256 received exception
  java.io.IOException:Could not read from stream
```

Phase II.(Data Preprocessing)

Both Phase II and Phase III aim to properly extract and represent the useful information held in the raw data collected in Phase I. Both Phase II and Phase III are closely related to feature engineering. A key difference between Phase II and Phase III is that Phase III is completely dedicated to representation learning, while Phase II is focused on all the information extraction and data processing operations that are **not** based on representation learning.

Running Example: Let's revisit the aforementioned HDFS. Each recorded event is described by unstructured text. In Phase II, the unstructured text is parsed to a data structure that shows the event type and a list of event variables in (name, value) pairs. Since there are 29 types of events in the HDFS, each event is represented by an integer from 1 to 29 according to its type. In this way, the aforementioned example event sequence can be transformed to:

```
22, 5, 5, 7
```

Phase III.(Representation Learning)

As stated in [7], "Learning representations of the data that make it easier to extract useful information when building classifiers or other predictors."

Running Example: Let's revisit the same HDFS. Although DeepLog [8] directly employed one-hot vectors to represent the event types without representation learning, if we view an event type as a word in a structured language, one may actually use the word embedding technique to represent each event type. It should be noticed that the word embedding technique is a representation learning technique.

Phase IV.(Classifier Learning)

This phase aims to build specific classifiers or other predictors through Deep Learning.

Running Example: Let's revisit the same HDFS. DeepLog [8] used Deep Learning to build a stacked LSTM neural network for anomaly detection. For example, let's consider event sequence {22,5,5,5,11,9,11,9,11,9,26,26,26} in which each integer represents the event type of the corresponding event in the event sequence. Given a window size $h = 4$, the input sample and the output label pairs to train DeepLog will be: {22,5,5,5 $\rightarrow$ 11 }, {5,5,5,11 $\rightarrow$ 9 }, {5,5,11,9 $\rightarrow$ 11 }, and so forth. In the detection stage, DeepLog examines each individual event. It determines if an event is treated as normal or abnormal according to whether the event's type is predicted by the LSTM neural network, given the history of event types. If the event's type is among the top g predicted types, the event is treated as normal; otherwise, it is treated as abnormal.

2.2 Using the four-phase workflow framework to summarize some representative research works

In this subsection, we use the four-phase workflow framework to summarize two representative works for each security problem. The summary is shown in Table 1. There are seven columns in the table. In the first column, we listed eight security problems, including security-oriented program analysis, defending return-oriented programming (ROP) attacks, control-flow integrity (CFI), defending network attacks (NA), malware classification (MC), system-event-based anomaly detection (SEAD), memory forensics (MF), and fuzzing for software security. In the second column, we list the very recent two representative works for each security problem. From the 3*rd* to 6*th* columns, we sequentially describe how the four phases are deployed at each work. In the 6*th* column, we list the Deep Learning models such as deep neural network (DNN), recurrent neural network (RNN),long short-term memory (LSTM), and convolutional neural network (CNN). In the 7*th* column, we list the evaluation results for each work in terms of accuracy (ACC), precision (PRC), recall (REC), F_1 score (F_1), false positive rate (FPR) and false negative rate (FNR), respectively.

Table 1: Solutions using Deep Learning for eight security problems. The metrics in the Evaluation column include accuracy (ACC), precision (PRC), recall (REC), F_1 score (F_1), false positive rate (FPR), and false negative rate (FNR).

Security Problem	Works	Phase I	Phase II	Phase III	Phase IV	Evaluation
Security Oriented Program Analysis [9–12]	RFBNN [9]	Dataset comes from previous paper [13], consisting of 2200 separate binaries. 2064 of the binaries were for Linux, obtained from the coreutils, binutils, and findutils packages. The remaining 136 for Windows consist of binaries from popular open-source projects. Half of the binaries were for x86, and the other half for x86-64.	They extract fixed-length subsequences (1000-byte chunks) from code section of binaries, Then, use "one-hot encoding", which converts a byte into a $\mathbb{Z}^{256}$ vector	N/A	Bi-directional RNN	ACC: 98.4% PRE: N/A REC: 0.97 F_1: 0.98 FPR: N/A FNR: N/A
	EKLAVYA [10]	They adopted source code from previous work [9] as their rawdata, then obtained two datasets by using two commonly used compilers: gcc and clang, with different optimization levels ranging from O0 to O3 for both x86 and x64. They obtained the ground truth for the function arguments by parsing the DWARF debug information. Next, they extract functions from the binaries and remove functions which are duplicates of other functions in the dataset. Finally, they match caller snipper and callee body.	Tokenizing the hexadecimal value of each instruction.	Word2vec technique to compute word embeddings.	RNN	ACC: 81.0% PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A
Defending Return Oriented Programming Attacks [14–16]	ROPNN [14]	The data is a set of gadget chains obtained from existing programs. A gadget searching tool, ROPGadget is used to find available gadgets. Gadgets are chained based on whether the produced gadget chain is executable on a CPU emulator. The raw data is represented in hexadecimal form of instruction sequences.	Form one-hot vector for bytes.	N/A	1-D CNN	ACC: 99.9% PRE: 0.99 REC: N/A F_1: 0.01 FPR: N/A FNR: N/A
	HeNet [15]	Data is acquired from Intel PT, which is a processor trace tool that can log control flow data. Taken Not-Taken (TNT) packet and Target IP (TIP) packet are the two packets of interested. Logged as binary numbers, information of executed branches can be obtained from TNT, and binary executed can be obtained from TIP. Then the binary sequences are transferred into sequences of values between 0-255, called pixels, byte by byte.	Given the pixel sequences, slice the whole sequence and reshape to form sequences of images for neural network training.	N/A	DNN	ACC: 98.1% PRE: 0.99 REC: 0.96 F_1: 0.97 FPR: 0.01 FNR: 0.04
Achieving Control Flow Integrity [16–18]	Barnum [17]	The raw data, which is the exact sequence of instructions executed, was generated by combining the program binary, get immediately before the program opens a document, and Intel® PT trace. While Intel® PT built-in filtering options are set to CR3 and current privilege level (CPL), which only traces the program activity in the user space.	The raw instruction sequences are summarized into Basic Blocks with IDs assigned and are then sliced into manageable subsequences with a fix window size 32, founded experimentally. Only sequences ending on indirect calls, jumps and returns are analyzed, since control-flow hijacking attacks always occur there. The label is the next BBID in the sequence.	N/A	LSTM	ACC: N/A PRE: 0.98 REC: 1.00 F_1: 0.98 FPR: 0.98 FNR: 0.02

Continued on Next Page...

Table 1 – Continued

Security Problem	Works	Phase I	Phase II	Phase III	Phase IV	Evaluation
	CFG-CNN [18]	The raw data is instruction level control-flow graph constructed from program assembly code by an algorithm proposed by the authors. While in the CFG, one vertex corresponds to one instruction and one directed edge corresponds to an execution path from one instruction to another. The program sets for experiments are obtained from popular programming contest CodeChief.	Since each vertex of the CFG represents an instruction with complex information that could be viewed from different aspects, including instruction name, type, operands etc., a vertex is represented as the sum of a set of real valued vectors, corresponding to the number of views (e.g. *addq 32,%rsp* is converted to linear combination of randomly assigned vectors of *addq value, reg*). The CFG is then sliced by a set of fixed size windows sliding through the entire graph to extract local features on different levels.	N/A	DGCNN with different numbers of views and with or without operands	ACC: 84.1% PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A
Defending Network Attacks [19–25]	50b(yte)-CNN [19]	Open dataset UNSW-NB15 is used. First, tcpdump tool is utilised to capture 100 GB of the raw traffic (i.e. PCAP files) containing benign activities and 9 types of attacks. The Argus, Bro-IDS (now called Zeek) analysis tools are then used and twelve algorithms are developed to generate totally 49 features with the class label. In the end, the total number of data samples is 2,540,044 which are stored in CSV files.	The first 50 bytes of each network traffic flow are picked out and each is directly used as one feature input to the neural network.	N/A	CNN with 2 hidden fully connected layers.	ACC: N/A PRE: N/A REC: N/A F_1: 0.93 FPR: N/A FNR: N/A
	PCCN [20]	Open dataset CICIDS2017, which contains benign and 14 types of attacks, is used. Background benign network traffics are generated by profiling the abstract behavior of human interactions. Raw data are provided as PCAP files, and the results of the network traffic analysis using CICFlowMeter are pvodided as CSV files. In the end the dataset contains 3,119,345 data samples and 83 features categorized into 15 classes (1 normal + 14 attacks).	Extract a total of 1,168,671 flow data, including 12 types of attack activities, from original dataset. Those flow data are then processed and visualized into grey-scale 2D graphs. The visualization method is not specified.	N/A	Parallel cross CNN.	ACC: N/A PRE: 0.99 REC: N/A F_1: 0.99 FPR: N/A FNR: N/A
Malware Classification [26–37]	Rosenberg [36]	The android dataset has the latest malware families and their variants, each with the same number of samples. The samples are labeled by VirusTotal. Then Cuckoo Sandbox is used to extract dynamic features (API calls) and static features (string). To avoid some anti-forensic sample, they applied YARA rule and removed sequences with less than 15 API calls. After preprocessing and balance the benign samples number, the dataset has 400,000 valid samples.	Long sequences cause out of memory during training LSTM model. So they use sliding window with fixed size and pad shorter sequences with zeros. One-hot encoding is applied to API calls. For static features strings, they defined a vector of 20,000 Boolean values indicating the most frequent Strings in the entire dataset. If the sample contain one string, the corresponding value in the vector will be assigned as 1, otherwise, 0.	N/A	They used RNN, BRNN, LSTM, Deep LSTM, BLSTM, Deep BLSTM, GRU, bi-directional GRU, Fully-connected DNN, 1D CNN in their experiments	ACC: 98.3% PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A

Continued on Next Page…

Table 1 – Continued

Security Problem	Works	Phase I	Phase II	Phase III	Phase IV	Evaluation
	DeLaRosa [26]	The windows dataset is from Reversing Labs including XP, 7, 8, and 10 for both 32-bit and 64-bit architectures and gathered over a span of twelve years (2006-2018). They selected nine malware families in their dataset and extracted static features in terms of bytes, basic, and assembly features.	For bytes-level features, they used a sliding window to get the histogram of the bytes and compute the associated entropy in a window; for basic features, they created a fixed-sized feature vector given either a list of ASCII strings, or extracted import and metadata information from the PE Header(Strings are hashed and calculate a histogram of these hashes by counting the occurrences of each value); for assembly features, the disassembled code generated by Radare2 can be parsed and transformed into graph-like data structures such as call graphs, control flow graph, and instruction flow graph.	N/A	N/A	ACC: 90.1% PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A
System Event Based Anomaly Detection [8, 38–42]	DeepLog [8]	More than 24 million raw log entries with the size of 2412 MB are recorded from the 203-node HDFS. Over 11 million log entries with 29 types are parsed, which are further grouped to 575,061 sessions according to block identifier. These sessions are manually labeled as normal and abnormal by HDFS experts. Finally, the constructed dataset HDFS 575,061 sessions of logs in the dataset, among which 16,838 sessions were labeled as anomalous	The raw log entries are parsed to different log type using Spell [43] which is based a longest common subsequence. There are total 29 log types in HDFS dataset	DeepLog directly utilized one-hot vector to represent 29 log key without represent learning	A stacked LSTM with two hidden LSTM layers.	ACC: N/A PRE: 0.95 REC: 0.96 F_1: 0.96 FPR: N/A FNR: N/A
	LogAnom [38]	LogAnom also used HDFS dataset, which is same as DeepLog.	The raw log entries are parsed to different log templates using FT-Tree [44] according the frequent combinations of log words. There are total 29 log templates in HDFS dataset	LogAnom employed Word2Vec to represent the extracted log templates with more semantic information	Two LSTM layers with 128 neurons	ACC: N/A PRE: 0.97 REC: 0.94 F_1: 0.96 FPR: N/A FNR: N/A
Memory Forensics [45–48]	DeepMem [45]	400 memory dumps are collected on Windows 7 x86 SP1 virtual machine with simulating various random user actions and forcing the OS to randomly allocate objects. The size of each dump is 1GB.	Construct memory graph from memory dumps, where each node represents a segment between two pointers and an edge is created if two nodes are neighbor	Each node is represented by a latent numeric vector from the embedding network.	Fully Connected Network (FCN) with ReLU layer.	ACC: N/A PRE: 0.99 REC: 0.99 F_1: 0.99 FPR: 0.01 FNR: 0.01

Continued on Next Page...

Table 1 – Continued

Security Problem	Works	Phase I	Phase II	Phase III	Phase IV	Evaluation
	MDMF [46]	Create a dataset of benign host memory snapshots running normal, non-compromised software, including software that executes in many of the malicious snapshots. The benign snapshot is extracted from memory after ample time has passed for the chosen programs to open. By generating samples in parallel to the separate malicious environment, the benign memory snapshot dataset created.	Various representation for the memory snapshots including byte sequence and image, without relying on domain-knowledge of the OS.	N/A	Recurrent Neural Network with LSTM cells and Convolutional Neural Network composed of multiple layers, including pooling and fully connected layers. for image data	ACC: 98.0% PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A
Fuzzing [49–53]	L-Fuzz [52]	The raw data are about 63,000 non-binary PDF objects, sliced in fix size, extracted from 534 PDF files that are provided by Windows fuzzing team and are previously used for prior extended fuzzing of Edge PDF parser.	N/A	N/A	Char-RNN	ACC: N/A PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A[1]
	NEUZZ [50]	For each program tested, the raw data is collected by running AFL-2.52b on a single core machine for one hour. The training data are byte level input files generated by AFL, and the labels are bitmaps corresponding to input files. For experiments, NEUZZ is implemented on 10 real-world programs, the LAVA-M bug dataset, and the CGC dataset.	N/A	N/A	NN	ACC: N/A PRE: N/A REC: N/A F_1: N/A FPR: N/A FNR: N/A[1]

[1] Deep Learning metrics are often not available in fuzzing papers. Typical fuzzing metrics used for evaluations are: code coverage, pass rate and bugs.

2.3 Methodology for reviewing the existing works

Data representation (or feature engineering) plays an important role in solving security problems with Deep Learning. This is because data representation is a way to take advantage of human ingenuity and prior knowledge to extract and organize the discriminative information from the data. Many efforts in deploying machine learning algorithms in security domain actually goes into the design of preprocessing pipelines and data transformations that result in a representation of the data to support effective machine learning.

In order to expand the scope and ease of applicability of machine learning in security domain, it would be highly desirable to find a proper way to represent the data in security domain, which can entangle and hide more or less the different explanatory factors of variation behind the data. To let this survey adequately reflect the important role played by data representation, our review will focus on how the following three questions are answered by the existing works:

- **Question 1:** Is Phase II pervasively done in the literature? When Phase II is skipped in a work, are there any particular reasons?
- **Question 2:** Is Phase III employed in the literature? When Phase III is skipped in a work, are there any particular reasons?
- **Question 3:** When solving different security problems, is there any commonality in terms of the (types of) classifiers learned in Phase IV? Among the works solving the same security problem, is there dissimilarity in terms of classifiers learned in Phase IV?

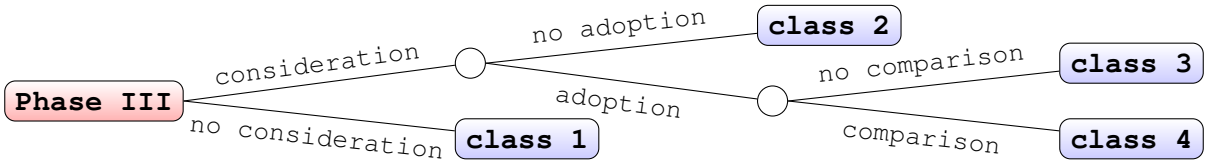


Figure 2: Classification tree for different Phase III methods. Here, `consideration`, `adoption`, and `comparison` indicate that a work considers Phase III, adopts Phase III and makes comparison with other methods, respectively.

To group the Phase III methods at different applications of Deep Learning in solving the same security problem, we introduce a classification tree as shown in Figure 2. The classification tree categorizes the Phase III methods in our selected survey works into four classes. First, class 1 includes the Phase III methods which do not consider representation learning. Second, class 2 includes the Phase III methods which consider representation learning but, do not adopt it. Third, class 3 includes the Phase III methods which consider and adopt representation learning but, do not compare the performance with other methods. Finally, class 4 includes the Phase III methods which consider and adopt representation learning and, compare the performance with other methods.

In the remaining of this paper, we take a closer look at how each of the eight security problems is being solved by applications of Deep Learning in the literature.

3 A closer look at applications of Deep Learning in solving security-oriented program analysis challenges

3.1 Introduction

Security-oriented program analysis is widely used in software security. For example, symbolic execution and taint analysis are used to discover, detect and analyze vulnerabilities in programs. Control flow analysis, data flow analysis and pointer/alias analysis are important components when enforcing many secure strategies, such as control flow integrity, data flow integrity and doling dangling pointer elimination. Reverse engineering was used by defenders and attackers to understand the logic of a program without source code.

In this section, we will review the very recent four representative works that use Deep Learning for security-oriented program analysis. We observed that they focused on different goals. Shin, et al. designed a model [9] to identify the function boundary. EKLAVYA [10] was developed to learn the function type. Gemini [12] was proposed to detect similarity among functions. DEEPVSA [11] was designed to learn memory region of an indirect addressing from the code sequence. Among these works, we select two representative works [9, 10] and then, summarize the analysis results in Table 1 in detail.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

3.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving security-oriented program analysis challenges, we observed the followings:

- **Observation 1:** *All of the works in our survey used binary files as their raw data.*

  Phase II in our survey had one similar and straightforward goal – extracting code sequences from the binary. Difference among them was that the code sequence was extracted directly from the binary file when solving problems in static program analysis, while it was extracted from the program execution when solving problems in dynamic program analysis.

- **Observation 2:** *Most data representation methods generally took into account the domain knowledge.*

  Most data representation methods generally took into the domain knowledge, i.e., what kind of information they wanted to reserve when processing their data. Note that the feature selection has a wide influence on Phase II and Phase III, for example, embedding granularities, representation learning methods. Gemini [12] selected function level feature and other works in our survey selected instruction level feature. To be specifically, all the works except Gemini [12] vectorized code sequence on instruction level.

- **Observation 3:** *To better support data representation for high performance, some works adopted representation learning.*

For instance, DEEPVSA [11] employed a representation learning method, i.e., bi-directional LSTM, to learn data dependency within instructions. EKLAVYA [10] adopted representation learning method, i.e., word2vec technique, to extract inter-instruciton information. It is worth noting that Gemini [12] adopts the Structure2vec embedding network in its siamese architecture in Phase IV (see details in Observation 7). The Structure2vec embedding network learned information from an attributed control flow graph.

- **Observation 4:** *According to our taxonomy, most works in our survey were classified into class 4.*

  To compare the Phase III, we introduced a classification tree with three layers as shown in Figure 2 to group different works into four categories. The decision tree grouped our surveyed works into four classes according to whether they considered representation learning or not, whether they adopted representation learning or not, and whether they compared their methods with others', respectively, when designing their framework. According to our taxonomy, EKLAVYA [10], DEEPVSA [11] were grouped into class 4 shown in Figure 2. Also, Gemini's work [12] and Shin, et al.'s work [9] belonged to class 1 and class 2 shown in Figure 2, respectively.

- **Observation 5:** *All the works in our survey explain why they adopted or not did not adopt one of representation learning algorithms.*

  Two works in our survey adopted representation learning for different reasons: to enhance model's ability of generalization [10]; and to learn the dependency within instructions [11]. It is worth noting that Shin, et al. did not adopt representation learning because they wanted to preserve the "attractive" features of neural networks over other machine learning methods – simplicity. As they stated, "first, neural networks can learn directly from the original representation with minimal preprocessing (or "feature engineering") needed." and "second, neural networks can learn end-to-end, where each of its constituent stages are trained simultaneously in order to best solve the end goal." Although Gemini [12] did not adopt representation learning when processing their raw data, the Deep Learning models in siamese structure consisted of two graph embedding networks and one cosine function.

- **Observation 6:** *The analysis results showed that a suitable representation learning method could improve accuracy of Deep Learning models.*

  DEEPVSA [11] designed a series of experiments to evaluate the effectiveness of its representative method. By combining with the domain knowledge, EKLAVYA [10] employed t-SNE plots and analogical reasoning to explain the effectiveness of their representation learning method in an intuitive way.

- **Observation 7:** *Various Phase IV methods were used.*

  In Phase IV, Gemini [12] adopted siamese architecture model which consisted of two Structure2vec embedding networks and one cosine function. The siamese architecture took two functions as its input, and produced the similarity score as the output. The other three works [9–11] adopted bi-directional RNN, RNN, bi-directional LSTM respectively. Shin, et al. adopted bi-directional RNN because they wanted to combine both the past

and the future information in making a prediction for the present instruction [9]. DEEPVSA [11] adopted bi-directional RNN to enable their model to infer memory regions in both forward and backward ways.

The above observations seem to indicate the following indications:

- **Indication 1:** *Phase III is not always necessary.*

  Not all authors regard representation learning as a good choice even though some case experiments show that representation learning can improve the final results. They value more the simplicity of Deep Learning methods and suppose that the adoption of representation learning weakens the simplicity of Deep Learning methods.

- **Indication 2:** *Even though the ultimate objective of Phase III in the four surveyed works is to train a model with better accuracy, they have different specific motivations as described in Observation 5.*

  When authors choose representation learning, they usually try to convince people the effectiveness of their choice by empirical or theoretical analysis.

- **Indication 3:** *Observation 7 indicates that authors usually refer to the domain knowledge when designing the architecture of Deep Learning model.*

  For instance, the works we reviewed commonly adopt bi-directional RNN when their prediction partly based on future information in data sequence.

4 A closer look at applications of Deep Learning in defending ROP attacks

4.1 Introduction

Return-oriented programming (ROP) attack is one of the most dangerous code reuse attacks, which allows the attackers to launch control-flow hijacking attack without injecting any malicious code. Rather, It leverages particular instruction sequences (called "gadgets") widely existing in the program space to achieve Turing-complete attacks [54]. Gadgets are instruction sequences that end with a *RET* instruction. Therefore, they can be chained together by specifying the return addresses on program stack. Many traditional techniques could be used to detect ROP attacks, such as control-flow integrity (CFI [55]), but many of them either have low detection rate or have high runtime overhead. Here, authors explore data-driven techniques to detect ROP attack, specifically methods using neural networks.

In this section, we will review the very recent three representative works that use Deep Learning for defending ROP attacks: ROPNN [14], HeNet [15] and DeepCheck [16]. We observed that none of the works considered Phase III, so all of them belong to class 1 according to our taxonomy as shown in Figure 2. The analysis results of ROPNN [14] and HeNet [15] are shown in Table 1. Also, we observed that three works had different goals. ROPNN [14] aims to detect ROP attacks. HeNet [15] aims to detect malware using CFI. DeepCheck [16] aims at detecting all kinds of code reuse attacks.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

4.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for defending return-oriented programming attacks, we observed the followings:

- **Observation 1:** *All the works [14–16] in this survey focused on data generation and acquisition.*

  In ROPNN [14], both malicious samples (gadget chains) were generated using an automated gadget generator (i.e. ROPGadget [56]) and a CPU emulator (i.e. Unicorn [57]). ROPGadget was used to extract instruction sequences that could be used as gadgets from a program, and Unicorn was used to validate the instruction sequences. Corresponding benign sample (gadget-chain-like instruction sequences) were generated by disassembling a set of programs. In DeepCheck [16] refers to the key idea of control-flow integrity [55]. It generates program's run-time control flow through new feature of Intel CPU (Intel Processor Tracing), then compares the run-time control flow with the program's control-flow graph (CFG) that generates through static analysis. Benign instruction sequences are that with in the program's CFG, and vice versa. In HeNet [15], program's execution trace was extracted using the similar way as DeepCheck. Then, each byte was transformed into a pixel with an intensity between 0-255. Known malware samples and benign software samples were used to generate malicious data benign data, respectively.

- **Observation 2:** *None of the ROP works in this survey deployed Phase III.*

  Both ROPNN [14] and DeepCheck [16] used binary instruction sequences for training. In ROPNN [14], one byte was used as the very basic element for data pre-processing. Bytes were formed into one-hot matrices and flattened for 1-dimensional convolutional layer. In DeepCheck [16], half-byte was used as the basic unit. Each half-byte (4 bits) was transformed to decimal form ranging from 0-15 as the basic element of the input vector, then was fed into a fully-connected input layer. On the other hand, HeNet [15] used different kinds of data. By the time this survey has been drafted, the source code of HeNet was not available to public and thus, the details of the data pre-processing was not be investigated. However, it is still clear that HeNet used binary branch information collected from Intel PT rather than binary instructions. In HeNet, each byte was converted to one decimal number ranging from 0 to 255. Byte sequences was sliced and formed into image sequences (each pixel represented one byte) for a fully-connected input layer.

- **Observation 3:** *Fully-connected neural network was widely used.*

  Only ROPNN [14] used 1-dimensional convolutional neural network (CNN) when extracting features. Both HeNet [15] and DeepCheck [16] used fully-connected neural network (FCN). None of the works used recurrent neural network (RNN) and the variants.

The above observations seem to indicate the following indications:

- **Indication 1:** *It seems like that one of the most important factors in ROP problem is feature selection and data generation.*

All three works use very different methods to collect/generate data, and all the authors provide very strong evidences and/or arguments to justify their approaches. ROPNN [14] was trained by the malicious and benign instruction sequences. However, there is no clear boundary between benign instruction sequences and malicious gadget chains. This weakness may impair the performance when applying ROPNN to real world ROP attacks. As oppose to ROPNN, DeepCheck [16] utilizes CFG to generate training basic-block sequences. However, since the malicious basic-block sequences are generated by randomly connecting nodes without edges, it is not guaranteed that all the malicious basic-blocks are executable. HeNet [15] generates their training data from malware. Technically, HeNet could be used to detect any binary exploits, but their experiment focuses on ROP attack and achieves 100% accuracy. This shows that the source of data in ROP problem does not need to be related to ROP attacks to produce very impressive results.

- **Indication 2:** *Representation learning seems not critical when solve ROP problems using Deep Learning.*

  Minimal process on data in binary form seems to be enough to transform the data into a representation that is suitable for neural networks. Certainly, it is also possible to represent the binary instructions at a higher level, such as opcodes, or use embedding learning. However, as stated in [14], it appears that the performance will not change much by doing so. The only benefit of representing input data to a higher level is to reduce irrelevant information, but it seems like neural network by itself is good enough at extracting features.

- **Indication 3:** *Different Neural network architecture does not have much influence on the effectiveness of defending ROP attacks.*

  Both HeNet [15] and DeepCheck [16] utilizes standard DNN and achieved comparable results on ROP problems. One can infer that the input data can be easily processed by neural networks, and the features can be easily detected after proper pre-process.

It is not surprising that researchers are not very interested in representation learning for ROP problems as stated in Observation 1. Since ROP attack is focus on the gadget chains, it is straightforward for the researcher to choose the gadgets as their training data directly. It is easy to map the data into numerical representation with minimal processing. An example is that one can map binary executable to hexadecimal ASCII representation, which could be a good representation for neural network.

Instead, researchers focus more in data acquisition and generation. In ROP problems, the amount of data is very limited. Unlike malware and logs, ROP payloads normally only contain addresses rather than codes, which do not contain any information without providing the instructions in corresponding addresses. It is thus meaningless to collect all the payloads. At the best of our knowledge, all the previous works use pick instruction sequences rather than payloads as their training data, even though they are hard to collect.

5 A closer look at applications of Deep Learning in achieving CFI

5.1 Introduction

The basic ideas of control-flow integrity (CFI) techniques, proposed by Abadi in 2005 [55], could be dated back to 2002, when Vladimir and his fellow researchers proposed an idea called program shepherding [58], a method of monitoring the execution flow of a program when it is running by enforcing some security policies. The goal of CFI is to detect and prevent control-flow hijacking attacks, by restricting every critical control flow transfers to a set that can only appear in correct program executions, according to a pre-built CFG. Traditional CFI techniques typically leverage some knowledge, gained from either dynamic or static analysis of the target program, combined with some code instrumentation methods, to ensure the program runs on a correct track.

However, the problems of traditional CFI are: (1) Existing CFI implementations are not compatible with some of important code features [59]; (2) CFGs generated by static, dynamic or combined analysis cannot always be precisely completed due to some open problems [60]; (3) There always exist certain level of compromises between accuracy and performance overhead and other important properties [61, 62]. Recent research has proposed to apply Deep Learning on detecting control flow violation. Their result shows that, compared with traditional CFI implementation, the security coverage and scalability were enhanced in such a fashion [17]. Therefore, we argue that Deep Learning could be another approach which requires more attention from CFI researchers who aim at achieving control-flow integrity more efficiently and accurately.

In this section, we will review the very recent three representative papers that use Deep Learning for achieving CFI. Among the three, two representative papers [17, 18] are already summarized phase-by-phase in Table 1. We refer to interested readers the Table 1 for a concise overview of those two papers.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

5.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for achieving control-flow integrity, we observed the followings:

- **Observation 1:** *None of the related works realize preventive*[1] *prevention of control flow violation.*

  After doing a thorough literature search, we observed that security researchers are quite behind the trend of applying Deep Learning techniques to solve security problems. Only one paper has been founded by us, using Deep Learning techniques to directly enhance the performance of CFI [17]. This paper leveraged Deep Learning to detect document malware through checking program's execution traces that generated by hardware. Specifically, the CFI violations were checked in an offline mode. So far, no works have realized Just-In-Time checking for program's control flow.

[1] We refer readers to [62] which systemizes the knowledge of protections by CFI schemes.

In order to provide more insightful results, in this section, we try not to narrow down our focus on CFI detecting attacks at run-time, but to extend our scope to papers that take good use of control flow related data, combined with Deep Learning techniques [18, 63]. In one work, researchers used self-constructed instruction-level CFG to detect program defection [18]. In another work, researchers used lazy-binding CFG to detect sophisticated malware [63].

- **Observation 2:** *Diverse raw data were used for evaluating CFI solutions.*

  In all surveyed papers, there are two kinds of control flow related data being used: program instruction sequences and CFGs. Barnum et al. [17] employed statically and dynamically generated instruction sequences acquired by program disassembling and Intel® Processor Trace. CNNoverCFG [18] used self-designed algorithm to construct instruction level control-flow graph. Minh Hai Nguyen et al. [63] used proposed lazy-binding CFG to reflect the behavior of malware DEC.

- **Observation 3:** *All the papers in our survey adopted Phase II.*

  All the related papers in our survey employed Phase II to process their raw data before sending them into Phase III. In Barnum [17], the instruction sequences from program run-time tracing were sliced into basic-blocks. Then, they assigned each basic-blocks with an unique basic-block ID (BBID). Finally, due to the nature of control-flow hijacking attack, they selected the sequences ending with indirect branch instruction (e.g., indirect call/jump, return and so on) as the training data. In CNNoverCFG [18], each of instructions in CFG were labeled with its attributes in multiple perspectives, such as opcode, operands, and the function it belongs to. The training data is generated are sequences generated by traversing the attributed control-flow graph. Nguyen and others [63] converted the lazy-binding CFG to corresponding adjacent matrix and treated the matrix as a image as their training data.

- **Observation 4:** *All the papers in our survey did not adopt Phase III.*

  We observed all the papers we surveyed did not adopted Phase III. Instead, they adopted the form of numerical representation directly as their training data. Specifically, Barnum [17] grouped the the instructions into basic-blocks, then represented basic-blocks with uniquely assigning IDs. In CNNoverCFG [18], each of instructions in the CFG was represented by a vector that associated with its attributes. Nguyen and others directly used the hashed value of bit string representation.

- **Observation 5:** *Various Phase IV models were used.*

  Barnum [17] utilized BBID sequence to monitor the execution flow of the target program, which is sequence-type data. Therefore, they chose LSTM architecture to better learn the relationship between instructions. While in the other two papers [18,63], they trained CNN and directed graph-based CNN to extract information from control-flow graph and image, respectively.

The above observations seem to indicate the following indications:

- **Indication 1:** *All the existing works did not achieve Just-In-Time CFI violation detection.*

It is still a challenge to tightly embed Deep Learning model in program execution. All existing work adopted lazy-checking – checking the program's execution trace following its execution.

- **Indication 2:** *There is no unified opinion on how to generate malicious sample.*

  Data are hard to collect in control-flow hijacking attacks. The researchers must carefully craft malicious sample. It is not clear whether the "handcrafted" sample can reflect the nature the control-flow hijacking attack.

- **Indication 3:** The choice of methods in *Phase II are based on researchers' security domain knowledge.*

6 A closer look at applications of Deep Learning in defending network attacks

6.1 Introduction

Network security is becoming more and more important as we depend more and more on networks for our daily lives, works and researches. Some common network attack types include probe, denial of service (DoS), Remote-to-local (R2L), etc. Traditionally, people try to detect those attacks using signatures, rules, and unsupervised anomaly detection algorithms. However, signature based methods can be easily fooled by slightly changing the attack payload; rule based methods need experts to regularly update rules; and unsupervised anomaly detection algorithms tend to raise lots of false positives. Recently, people are trying to apply Deep Learning methods for network attack detection.

In this section, we will review the very recent seven representative works that use Deep Learning for defending network attacks. [19,22,24] build neural networks for multi-class classification, whose class labels include one benign label and multiple malicious labels for different attack types. [20] ignores normal network activities and proposes parallel cross convolutional neural network (PCCN) to classify the type of malicious network activities. [21] applies Deep Learning to detecting a specific attack type, distributed denial of service (DDoS) attack. [23,25] explores both binary classification and multi-class classification for benign and malicious activities. Among these seven works, we select two representative works [19, 20] and summarize the main aspects of their approaches regarding whether the four phases exist in their works, and what exactly do they do in the Phase if it exists. We direct interested readers to Table 1 for a concise overview of these two works.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

6.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving network attack challenges, we observed the followings:

- **Observation 1:** *All the seven works in our survey used public datasets, such as UNSW-NB15 [64] and CICIDS2017 [65].*

The public datasets were all generated in test-bed environments, with unbalanced simulated benign and attack activities. For attack activities, the dataset providers launched multiple types of attacks, and the numbers of malicious data for those attack activities were also unbalanced.

- **Observation 2:** *The public datasets were given into one of two data formats, i.e., PCAP and CSV.*

  One was raw PCAP or parsed CSV format, containing network packet level features, and the other was also CSV format, containing network flow level features, which showed the statistic information of many network packets. Out of all the seven works, [21, 22] used packet information as raw inputs, [20, 23–25] used flow information as raw inputs, and [19] explored both cases.

- **Observation 3:** *In order to parse the raw inputs, preprocessing methods, including one-hot vectors for categorical texts, normalization on numeric data, and removal of unused features/data samples, were commonly used.*

  Commonly removed features include IP addresses and timestamps. [25] also removed port numbers from used features. By doing this, they claimed that they could "avoid over-fitting and let the neural network learn characteristics of packets themselves". One outlier was that, when using packet level features in one experiment, [19] blindly chose the first 50 bytes of each network packet without any feature extracting processes and fed them into neural network.

- **Observation 4:** *Using image representation improved the performance of security solutions using Deep Learning.*

  After preprocessing the raw data, while [20] transformed the data into image representation, [21–25] directly used the original vectors as an input data. Also, [19] explored both cases and reported better performance using image representation.

- **Observation 5:** *None of all the seven surveyed works considered representation learning.*

  All the seven surveyed works belonged to class 1 shown in Figure 2. They either directly used the processed vectors to feed into the neural networks, or changed the representation without explanation. One research work [19] provided a comparison on two different representations (vectors and images) for the same type of raw input. However, the other works applied different preprocessing methods in Phase II. That is, since the different preprocessing methods generated different feature spaces, it was difficult to compare the experimental results.

- **Observation 6:** *Binary classification model showed better results from most experiments.*

  Among all the seven surveyed works, [21] focused on one specific attack type and only did binary classification to classify whether the network traffic was benign or malicious. Also, [19, 20, 22, 24] included more attack types and did multi-class classification to classify the type of malicious activities, and [23, 25] explored both cases. As for multi-class classification, the accuracy for selective classes was good, while accuracy for other classes, usually classes with much fewer data samples, suffered by up to 20% degradation.

- **Observation 7:** *Data representation influenced on choosing a neural network model.*

The above observations seem to indicate the following indications:

- **Indication 1:** *All works in our survey adopt a kind of preprocessing methods in Phase II, because raw data provided in the public datasets are either not ready for neural networks, or that the quality of data is too low to be directly used as data samples.*

  Preprocessing methods can help increase the neural network performance by improving the data samples' qualities. Furthermore, by reducing the feature space, pre-processing can also improve the efficiency of neural network training and testing. Thus, Phase II should not be skipped. If Phase II is skipped, the performance of neural network is expected to go down considerably.

- **Indication 2:** *Although Phase III is not employed in any of the seven surveyed works, none of them explains a reason for it. Also, they all do not take representation learning into consideration.*

- **Indication 3:** *Because no work uses representation learning, the effectiveness are not well-studied.*

  Out of other factors, it seems that the choice of pre-processing methods has the largest impact, because it directly affects the data samples fed to the neural network.

- **Indication 4:** *There is no guarantee that CNN also works well on images converted from network features.*

  Some works that use image data representation use CNN in Phase IV. Although CNN has been proven to work well on image classification problem in the recent years, there is no guarantee that CNN also works well on images converted from network features.

From the observations and indications above, we hereby present two recommendations: (1) Researchers can try to generate their own datasets for the specific network attack they want to detect. As stated, the public datasets have highly unbalanced number of data for different classes. Doubtlessly, such unbalance is the nature of real world network environment, in which normal activities are the majority, but it is not good for Deep Learning. [22] tries to solve this problem by oversampling the malicious data, but it is better to start with a balanced data set. (2) Representation learning should be taken into consideration. Some possible ways to apply representation learning include: (a) apply word2vec method to packet binaries, and categorical numbers and texts; (b) use K-means as one-hot vector representation instead of randomly encoding texts. We suggest that any change of data representation may be better justified by explanations or comparison experiments.

7 A closer look at applications of Deep Learning in malware classification

7.1 Introduction

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. Malware and benign programs

can be collected from open datasets and online websites. Most research will extract semantic features from those programs using security domain knowledge. To classify those programs, features need to be parsed to a reasonable representation for Deep Learning models. The state of art classification approaches using Deep Learning can achieve high accuracy.

In this section, we will review the very recent twelve representative works that use Deep Learning for malware classification [26–37]. [26] selects three different kinds of static features to classify malware. [27–29] also use static features from the PE files to classify programs. [30] extracts behavioral feature images using RNN to represent the behaviors of original programs. [31] transforms malicious behaviors using representative learning without neural network. [32] explores RNN model with the API calls sequences as programs' features. [33, 34] skip Phase II by directly transforming the binary file to image to classify the file. [35, 36] applies dynamic features to analyze malicious features. [37] combines static features and dynamic features to represent programs' features. Among these works, we select two representative works [26, 36] and identify four phases in their works shown as Table 1.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

7.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving malware classification challenges, we observed the followings:

- **Observation 1:** *Features selected in malware classification were grouped into three categories: static features, dynamic features, and hybrid features.*

  Typical static features include metadata, PE import Features, Byte/Entorpy, String, and Assembly Opcode Features derived from the PE files [27–29]. De LaRosa, Kilgallon, et al. [26] took three kinds of static features: byte-level, basic-level (strings in the file, the metadata table, and the import table of the PE header), and assembly features-level. Some works directly considered binary code as static features [33, 34].

  Different from static features, dynamic features were extracted by executing the files to retrieve their behaviors during execution. The behaviors of programs, including the API function calls, their parameters, files created or deleted, websites and ports accessed, etc, were recorded by a sandbox as dynamic features [35]. The process behaviors including operation name and their result codes were extracted [30]. The process memory, tri-grams of system API calls and one corresponding input parameter were chosen as dynamic features [31]. An API calls sequence for an APK file was another representation of dynamic features [32, 36].

  Static features and dynamic features were combined as hybrid features [37]. For static features, Xu and others in [37] used permissions, networks, calls, and providers, etc. For dynamic features, they used system call sequences.

- **Observation 2:** *In most works, Phase II was inevitable because extracted features needed to be vertorized for Deep Learning models.*

  One-hot encoding approach was frequently used to vectorize features [28–30, 32, 36]. Bag-of-words (BoW) and n-gram were also considered to represent features [32]. Some works brought the concepts of word frequency in NLP to convert the sandbox file to

fixed-size inputs [35]. Hashing features into a fixed vector was used as an effective method to represent features [27]. Bytes histogram using the bytes analysis and bytes-entropy histogram with a sliding window method were considered [26]. In [26], De La Rosa and others embeded strings by hashing the ASCII strings to a fixed-size feature vector. For assembly features, they extracted four different levels of granularity: operation level (instruction-flow-graph), block level (control-flow-graph), function level (call-graph), and global level (graphs summarized). bigram, trigram and four-gram vectors and *n*-gram graph were used for the hybrid features [37].

- **Observation 3:** *Most Phase III methods were classified into class 1.*

  Following the classification tree shown in Figure 2, most works were classified into class 1 shown in Figure 2 except two works [30, 31], which belonged to class 3 shown in Figure 2. To reduce the input dimension, Dahl et al. [31] performed feature selection using mutual information and random projection. Tobiyama et al. generated behavioral feature images using RNN [30].

- **Observation 4:** *After extracting features, two kinds of neural network architectures, i.e., one single neural network and multiple neural networks with a combined loss function, were used.*

  Hierarchical structures, like convolutional layers, fully connected layers and classification layers, were used to classify programs [27, 29–34]. A deep stack of denoising autoencoders was also introduced to learn programs' behaviors [35]. De La Rosa and others [26] trained three different models with different features to compare which static features are relevant for the classification model. Some works investigated LSTM models for sequential features [32, 36].

  Two networks with different features as inputs were used for malware classification by combining their outputs with a dropout layer and an output layer [28]. In [28], one network transformed PE Metadata and import features using feedforward neurons, another one leveraged convolutional network layers with opcode sequences. Lifan Xu et al. [37] constructed a few networks and combined them using a two-level multiple kernel learning algorithm.

The above observations seem to indicate the following indications:

- **Indication 1:** *Except two works transform binary into images [33, 34], most works surveyed need to adapt methods to vectorize extracted features.*

  The vectorization methods should not only keep syntactic and semantic information in features, but also consider the definition of the Deep Learning model.

- **Indication 2:** *Limited researches convert their features using representation learning.*

  Because some works assume the dynamic and static sequences, like API calls and instruction, and have similar syntactic and semantic structure as natural language, some representation learning techniques like word2vec may be useful in malware detection. In addition, for the control-flow graph, call graph and other graph representations, graph embedding is a potential method to transform those features.

Though several researches have been done in malware detection using Deep Learning, it's hard to compare their methods and performances because of two uncertainties in their approaches. First, the Deep Learning model is a black-box, researchers cannot detail which kind of features the model learned and explain why their model works. Second, feature selection and representation affect the model's performance. Because they do not use the same datasets, researchers cannot prove their approaches – including selected features and Deep Learning model – are better than others. The reason why few researchers use open datasets is that existing open malware datasets are out of data and limited. Also, researchers need to crawl benign programs from app stores, so their raw programs will be diverse.

8 A closer look at applications of Deep Learning in system-event-based anomaly detection

8.1 Introduction

System logs recorded significant events at various critical points, which can be used to debug the system's performance issues and failures. Moreover, log data are available in almost all computer systems and are a valuable resource for understanding system status. Recently, a large number of scholars employed Deep Learning techniques [8, 38–42] to detect anomaly events in the system logs and diagnosis system failures. The raw log data are unstructured because their formats and semantics can vary significantly. To detect the anomaly event, the raw log usually was parsed to structure data, then transformed into a representation that fit into an effective Deep Learning model. Finally, the anomaly event was detected by Deep Learning based classifier or predictor.

In this section, we will review the very recent six representative works that use Deep Learning for system-event-based anomaly detection [8, 38–42]. DeepLog [8] utilizes LSTM that processes the system log as a natural language sequence, which automatically learns log patterns from the normal event, and detects anomalies when log patterns deviate from the trained model. LogAnom [38] employs Word2vec to extract the semantic and syntax information from log samples. Moreover, it uses sequential and quantitative features simultaneously. Desh [39] uses LSTM to predict node failures that occur in super computing systems from HPC logs. Andy Brown et al. [40] presented RNN language models augmented with attention for anomaly detection in system logs. LogRobust [41] uses FastText to represent semantic information of log events, which can identify and handle unstable log events and sequences. Christophe Bertero and others [42] map log word to a high dimensional metric space using Google's word2vec algorithm and take it as features to classify. Among these six works, we select two representative works [8, 38] and summarize the four phases of their approaches. We direct interested readers to Table 1 for a concise overview of these two works.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

8.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving security-event-based anomaly detection challenges, we observed the followings:

- **Observation 1:** *Most works in this survey adopted Phase II when parsing the raw log data.*

  We observe that all the works except for [42] adopted Phase II to parse their raw data. DeepLog [8] parsed the raw log to different log type based on longest common subsequences [43]. Desh [39] parsed the raw log to constant message and variable component. Loganom [38] parsed the raw log to different log templates according to the frequency of combinations of long words [44]. Andy Brown et al. [40] parsed the raw log into word and character tokens. LogRobust [41] extracted its log event by abstracting away the parameters in the message. Christophe Bertero et al. [42] considered logs as regular text without parsing.

- **Observation 2:** *Most works have considered and adopted Phase III.*

  Among these six works, only DeepLog represented the parsed data using the one-hot vector without learning. Moreover, Loganom [38] compared their results with DeepLog. According to taxonomy as shown in Figure 2, DeepLog belonged to class 1 and Loganom belonged to class 4, while the other four works belonged to class 3. Four works [38,39,41, 42] used word embedding techniques to represent the log data. Andy Brown et al. [40] employed attention vectors to represent the log messages. DeepLog [8] employed the one-hot vector to represent the log type without learning. We have engaged an experiment replacing the one-hot vector with trained word embeddings.

- **Observation 3:** *Evaluation results were not compared using the same dataset.*

  DeepLog [8] employed the one-hot vector to represent the log type without learning, that is to mean they employed Phase II but not Phase III. However, Christophe Bertero et al. [42] considered logs as regular text without parsing, and they adopted Phase III but not Phase II. The precision of the two methods was very high, i.e., greater than 95%. Unfortunately, the evaluation of the two methods used different datasets.

- **Observation 4:** *Most works empolyed LSTM in Phase IV.*

  Five works including [8, 38–41] employed LSTM in the Phase IV, while Christophe Bertero et al. [42] compare LSTM model with other classifiers, such as naive Bayes, neural networks and random forest.

The above observations seem to indicate the following indications:

- **Indication 1:** *Most of the recent works use trained representation to represent parsed data.*

  As shown in Table 2, we can find Phase III is very useful, which can improve detection accuracy.

Table 2: Comparison between word embedding and one-hot representation.

Method	FP [1]	FN [2]	Precision	Recall	F_1-measure
Word Embedding [3]	680	219	96.069%	98.699%	97.366%
One-hot Vector [4]	711	705	95.779%	95.813%	95.796%
DeepLog [5]	833	619	95%	96%	96%

[1]FP: number of false positives; [2]FN: number of false negatives; [3]Word Embedding: Log keys are embedded by Continuous Bag of words; [4] One-hot Vector: We reproduced the results according to DeepLog; [5] DeepLog: Orignial results presented in the work [8].

Indication 2: *Phase II and Phase III cannot be skipped simultaneously.*

Both Phase II and Phase III are not required. However, all methods have employed Phase II or Phase III.

- **Indication 3:** *Observation 2 indicates that the trained word embedding can improve the anomaly detection accuracy as shown in Table 2.*

- **Indication 4:** *Observation 4 indicates that most works adopt LSTM to detect anomaly event.*

  We can find that most works adopt LSTM to detect anomaly event, since log data can be considered as sequence and there can be lags of unknown duration between important events in a time series. LSTM has feedback connections, which cannot only process single data points, but also entire sequences of data.

In conclusion, neither Phase II nor Phase III is required in system event-based anomaly detection. However, Phase II can remove noise in raw data, and Phase III can learn a proper representation of the data. Both Phase II and Phase III have a positive effect on anomaly detection accuracy. Since the event log is text data that we cannot been fed into Deep Learning model directly, Phase II and Phase III can't be skipped simultaneously.

9 A closer look at applications of Deep Learning in solving memory forensics challenges

9.1 Introduction

In the field of computer security, memory forensics is security-oriented forensic analysis of a computer's memory dump. Memory forensics can be conducted against OS kernels, user-level applications, as well as mobile devices. Memory forensics outperforms traditional disk-based forensics because although secrecy attacks can erase their footprints on disk, they would have to appear in memory [45]. The memory dump can be considered as a sequence of bytes, thus memory forensics usually needs to extract security semantic information from raw memory dump to find attack traces. Several challenges exist in memory forensics such as lack of adequate domain knowledge, lack of robustness and low efficiency. Researchers have been using Deep Learning to address these challenges [45–48].

In this section, we will review the very recent four representative works that use Deep Learning for memory forensics [45–48]. DeepMem [45] recognized the kernel objects from raw memory dumps by generating abstract representations of kernel objects with a graph-based Deep Learning approach. MDMF [46] detected OS and architecture-independent malware from memory snapshots with several pre-processing techniques, domain unaware feature selection, and a suite of machine learning algorithms. MemTri [47] predicts the likelihood of criminal activity in a memory image using a Bayesian network, based on evidence data artefacts generated by several applications. Dai et al. [48] monitor the malware process memory and classify malware according to memory dumps, by transforming the memory dump into grayscale images and adopting a multi-layer perception as the classifier.

Among these four works [45–48], two representative works (i.e., [45, 46]) are already summarized phase-by-phase in Table 1. We direct interested readers to Table 1 for a concise overview of these two works.

Our review will be centered around the three questions raised in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

9.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving memory forensics challenges, we observed the followings:

- **Observation 1:** *Most methods used their own datasets for performance evaluation, while none of them used a public dataset.*

  DeepMem was evaluated on self-generated dataset by the authors, who collected a large number of diverse memory dumps, and labeled the kernel objects in them using existing memory forensics tools like Volatility. MDMF employed the MalRec dataset by Georgia Tech to generate malicious snapshots, while it created a dataset of benign memory snapshots running normal software. MemTri ran several Windows 7 virtual machine instances with self-designed suspect activity scenarios to gather memory images. Dai et al. built the Procdump program in Cuckoo sandbox to extract malware memory dumps. We found that each of the four works in our survey generated their own datasets, while none was evaluated on a public dataset.

- **Observation 2:** *Among the four works [45–48], two works [45, 47] employed Phase II while the other two works [46, 48] did not employ.*

  DeepMem [45] devised a graph representation for a sequence of bytes, taking into account both adjacency and points-to relations, to better model the contextual information in memory dumps. MemTri [47] firstly identified the running processes within the memory image that match the target applications, then employed regular expressions to locate evidence artefacts in a memory image. MDMF [46] and Dai et al. [48] transformed the memory dump into image directly.

- **Observation 3:** *Among four works [45–48], only DeepMem [45] employed Phase III for which it used an embedding method to represent a memory graph.*

  MDMF [46] directly fed the generated memory images into the training of a CNN model. Dai et al. [48] used HOG feature descriptor for detecting objects, while MemTri [47]

extracted evidence artefacts as the input of Bayesian Network. In summary, DeepMem belonged to class 3 shown in Figure 2, while the other three works belonged to class 1 shown in Figure 2.

- **Observation 4:** *All the four works [45–48] have employed different classifiers even when the types of input data are the same.*

  DeepMem chose fully connected network (FCN) model that has multi-layered hidden neurons with ReLU activation functions, following by a softmax layer as the last layer. MDMF [46] evaluated their performance both on traditional machine learning algorithms and Deep Learning approach including CNN and LSTM. Their results showed the accuracy of different classifiers did not have a significant difference. MemTri employed a Bayesian network model that is designed with three layers, i.e., a hypothesis layer, a sub-hypothesis layer, and an evidence layer. Dai et al. used a multi-layer perception model including an input layer, a hidden layer and an output layer as the classifier.

The above observations seem to indicate the following indications:

- **Indication 1:** *There lacks public datasets for evaluating the performance of different Deep Learning methods in memory forensics.*

  From Observation 1, we find that none of the four works surveyed was evaluated on public datasets.

- **Indication 2:** *From Observation 2, we find that it is disputable whether one should employ Phase II when solving memory forensics problems.*

  Since both [46] and [48] directly transformed a memory dump into an image, Phase II is not required in these two works. However, since there is a large amount of useless information in a memory dump, we argue that appropriate prepossessing could improve the accuracy of the trained models.

- **Indication 3:** *From Observation 3, we find that Phase III is paid not much attention in memory forensics.*

  Most works did not employ Phase III. Among the four works, only DeepMem [45] employed Phase III during which it used embeddings to represent a memory graph. The other three works [46–48] did not learn any representations before training a Deep Learning model.

- **Indication 4:** *For Phase IV in memory forensics, different classifiers can be employed.*

  Which kind of classifier to use seems to be determined by the features used and their data structures. From Observation 4, we find that the four works have actually employed different kinds of classifiers even the types of input data are the same. It is very interesting that MDMF obtained similar results with different classifiers including traditional machine learning and Deep Learning models. However, the other three works did not discuss why they chose a particular kind of classifier.

Since a memory dump can be considered as a sequence of bytes, the data structure of a training data example is straightforward. If the memory dump is transformed into a simple form in Phase II, it can be directly fed into the training process of a Deep Learning model,

and as a result Phase III can be ignored. However, if the memory dump is transformed into a complicated form in Phase II, Phase III could be quite useful in memory forensics.

Regarding the answer for Question 3 at Section 2.3, it is very interesting that during Phase IV different classifiers can be employed in memory forensics. Moreover, MDMF [46] has shown that they can obtain similar results with different kinds of classifiers. Nevertheless, they also admit that with a larger amount of training data, the performance could be improved by Deep Learning.

10 A closer look at applications of Deep Learning in security-oriented fuzzing

10.1 Introduction

Fuzzing of software security is one of the state of art techniques that people use to detect software vulnerabilities. The goal of fuzzing is to find all the vulnerabilities exist in the program by testing as much program code as possible. Due to the nature of fuzzing, this technique works best on finding vulnerabilities in programs that take in input files, like PDF viewers [52] or web browsers. A typical workflow of fuzzing can be concluded as: given several seed input files, the fuzzer will mutate or fuzz the seed inputs to get more input files, with the aim of expanding the overall code coverage of the target program as it executes the mutated files. Although there have already been various popular fuzzers [66], fuzzing still cannot bypass its problem of sometimes redundantly testing input files which cannot improve the code coverage rate [50,53]. Some input files mutated by the fuzzer even cannot pass the well-formed file structure test [52]. Recent research has come up with ideas of applying Deep Learning in the process of fuzzing to solve these problems.

In this section, we will review the very recent four representative works that use Deep Learning for fuzzing for software security. Among the three, two representative works [50, 52] are already summarized phase-by-phase in Table 1. We direct interested readers to Table 1 for a concise overview of those two works.

Our review will be centered around three questions described in Section 2.3. In the remaining of this section, we will first provide a set of observations, and then we provide the indications. Finally, we provide some general remarks.

10.2 Key findings from a closer look

From a close look at the very recent applications using Deep Learning for solving security-oriented program analysis challenges, we observed the followings:

- **Observation 1:** *Deep Learning has only been applied in mutation-based fuzzing.*

  Even though various of different fuzzing techniques, including symbolic execution based fuzzing [67], tainted analysis based fuzzing [68] and hybrid fuzzing [69] have been proposed so far, we observed that all the works we surveyed employed Deep Learning method to assist the primitive fuzzing – mutation-based fuzzing. Specifically, they adopted Deep Learning to assist fuzzing tool's input mutation. We found that they commonly did it in two ways: 1) training Deep Learning models to tell how to efficiently mutate the input to trigger more execution path [50, 53]; 2) training Deep Learning models to tell how to keep the mutated files compliant with the program's basic semantic

requirement [52]. Besides, all three works trained different Deep Learning models for different programs, which means that knowledge learned from one programs cannot be applied to other programs.

- **Observation 2:** *Similarity among all the works in our survey existed when choosing the training samples in Phase I.*

  The works in this survey had a common practice, i.e., using the input files directly as training samples of the Deep Learning model. Learn&Fuzz [52] used character-level PDF objects sequence as training samples. Neuzz [50] regarded input files directly as byte sequences and fed them into the neural network model. Mohit Rajpal et al. [53] also used byte level representations of input files as training samples.

- **Observation 3:** *Difference between all the works in our survey existed when assigning the training labels in Phase I.*

  Despite the similarity of training samples researchers decide to use, there was a huge difference in the training labels that each work chose to use. Learn&Fuzz [52] directly used the character sequences of PDF objects as labels, same as training samples, but shifted by one position, which is a common generative model technique already broadly used in speech and handwriting recognition. Unlike Learn&Fuzz, Neuzz [50] and Rajpal's work [53] used bitmap and heatmap respectively as training labels, with the bitmap demonstrating the code coverage status of a certain input, and the heatmap demonstrating the efficacy of flipping one or more bytes of the input file. Whereas, as a common terminology well-known among fuzzing researchers, bitmap was gathered directly from the results of AFL. Heatmap used by Rajpal et al. was generated by comparing the code coverage supported by the bitmap of one seed file and the code coverage supported by bitmaps of the mutated seed files. It was noted that if there is acceptable level of code coverage expansion when executing the mutated seed files, demonstrated by more "1"s, instead of "0"s in the corresponding bitmaps, the byte level differences among the original seed file and the mutated seed files will be highlighted. Since those bytes should be the focus of later on mutation, heatmap was used to denote the location of those bytes.

  Different labels usage in each work was actually due to the different kinds of knowledge each work wants to learn. For a better understanding, let us note that we can simply regard a Deep Learning model as a simulation of a "function". Learn&Fuzz [52] wanted to learn valid mutation of a PDF file that was compliant with the syntax and semantic requirements of PDF objects. Their model could be seen as a simulation of $f(x, \theta) = y$, where x denotes sequence of characters in PDF objects and y represents a sequence that are obtained by shifting the input sequences by one position. They generated new PDF object character sequences given a starting prefix once the model was trained. In Neuzz [50], an NN(Neural Network) model was used to do program smoothing, which simultated a smooth surrogate function that approximated the discrete branching behaviors of the target program. $f(x, \theta) = y$, where x denoted program's byte level input and y represented the corresponding edge coverage bitmap. In this way, the gradient of the surrogate function was easily computed, due to NN's support of efficient computation of gradients and higher order derivatives. Gradients could then be used to guide the direction of mutation, in order to get greater code coverage. In Rajpal and others' work [53], they designed a model to predict good (and bad) locations to mutate in input files based on the past mutations and corresponding code coverage information. Here, the x variable

also denoted program's byte level input, but the y variable represented the corresponding heatmap.

- **Observation 4:** *Various lengths of input files were handled in Phase II.*

  Deep Learning models typically accepted fixed length input, whereas the input files for fuzzers often held different lengths. Two different approaches were used among the three works we surveyed: splitting and padding. Learn&Fuzz [52] dealt with this mismatch by concatenating all the PDF objects character sequences together, and then splited the large character sequence into multiple training samples with a fixed size. Neuzz [50] solved this problem by setting a maximize input file threshold and then, padding the smaller-sized input files with null bytes. From additional experiments, they also found that a modest threshold gived them the best result, and enlarging the input file size did not grant them additional accuracy. Aside from preprocessing training samples, Neuzz also preprocessed training labels and reduced labels dimension by merging the edges that always appeared together into one edge, in order to prevent the multicollinearity problem, that could prevent the model from converging to a small loss value. Rajpal and others [53] used the similar splitting mechanism as Learn&Fuzz to split their input files into either 64-bit or 128-bit chunks. Their chunk size was determined empirically and was considered as a trainable parameter for their Deep Learning model, and their approach did not require sequence concatenating at the beginning.

- **Observation 5:** *All the works in our survey skipped Phase III.*

  According to our definition of Phase III, all the works in our survey did not consider representation learning. Therefore, all the three works [50, 52, 53] fell into class 1 shown in Figure 2.While as in Rajpal and others' work, they considered the numerical representation of byte sequences. They claimed that since one byte binary data did not always represent the magnitude but also state, representing one byte in values ranging from 0 to 255 could be suboptimal. They used lower level 8-bit representation.

The above observations seem to indicate the following indications:

- **Indication 1:** *No alteration to the input files seems to be a correct approach.*

  As far as we concerned, it is due to the nature of fuzzing. That is, since every bit of the input files matters, any slight alteration to the input files could either lose important information or add redundant information for the neural network model to learn.

- **Indication 2:** *Evaluation criteria should be chosen carefully when judging mutation.*

  Input files are always used as training samples regarding using Deep Learning technique in fuzzing problems. Through this similar action, researchers have a common desire to let the neural network mode learn how the mutated input files should look like. But the criterion of judging a input file actually has two levels: on the one hand, a good input file should be correct in syntax and semantics; on the other hand, a good input file should be the product of a useful mutation, which triggers the program to behave differently from previous execution path. This idea of a fuzzer that can generate semantically correct input file could still be a bad fuzzer at triggering new execution path was first brought up in Learn&Fuzz [52]. We could see later on works trying to solve this problem by using either different training labels [53] or use neural network to do program smoothing [50].

We encouraged fuzzing researchers, when using Deep Learning techniques, to keep this problem in mind, in order to get better fuzzing results.

- **Indication 3:** *Works in our survey only focus on local knowledge.*

  In brief, some of the existing works [50, 53] leveraged the Deep Learning model to learn the relation between program's input and its behavior and used the knowledge that learned from history to guide future mutation. For better demonstration, we defined the knowledge that only applied in one program as *local knowledge*. In other words, this indicates that the *local knowledge* cannot direct fuzzing on other programs.

Based on aforementioned observations, we would like to raise several interesting questions: 1) Can the knowledge learned from the fuzzing history of one program be applied to direct testing on other programs? 2) If the answer to question one is positive, we can suppose that *global knowledge* across different programs exists? Then, can we train a model to extract the *global knowledge*? 3) Whether it is possible to combine *global knowledge* and *local knowledge* when fuzzing programs?

11 Discussion

Using high-quality data in Deep Learning is important as much as using well-structured deep neural network architectures. That is, obtaining quality data must be an important step, which should not be skipped, even in resolving security problems using Deep Learning. So far, this study demonstrated how the recent security papers using Deep Learning have adopted data conversion (Phase II) and data representation (Phase III) on different security problems. Our observations and indications showed a clear understanding of how security experts generate quality data when using Deep Learning.

Since we did not review all the existing security papers using Deep Learning, the generality of observations and indications is somewhat limited. Note that our selected papers for review have been published recently at one of prestigious security and reliability conferences such as USENIX SECURITY, ACM CCS and so on [9]- [39], [40, 41], [45, 46], [49]- [53]. Thus, our observations and indications help to understand how most security experts have used Deep Learning to solve the well-known eight security problems from program analysis to fuzzing.

Our observations show that we should transfer raw data to synthetic formats of data ready for resolving security problems using Deep Learning through data cleaning and data augmentation and so on. Specifically, we observe that Phases II and III methods have mainly been used for the following purposes:

- To clean the raw data to make the neural network (NN) models easier to interpret
- To reduce the dimensionality of data (e.g., principle component analysis (PCA), t-distributed stochastic neighbor embedding (t-SNE))
- To scale input data (e.g., normalization)
- To make NN models understand more complex relationships depending on security problems (e.g. memory graphs)
- To simply change various raw data formats into a vector format for NN models (e.g. one-hot encoding and word2vec embedding)

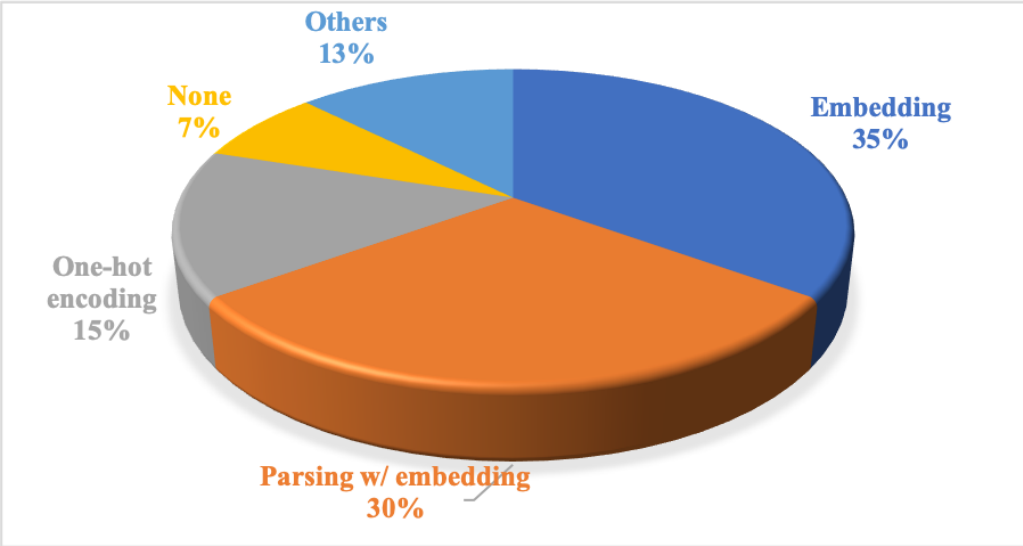


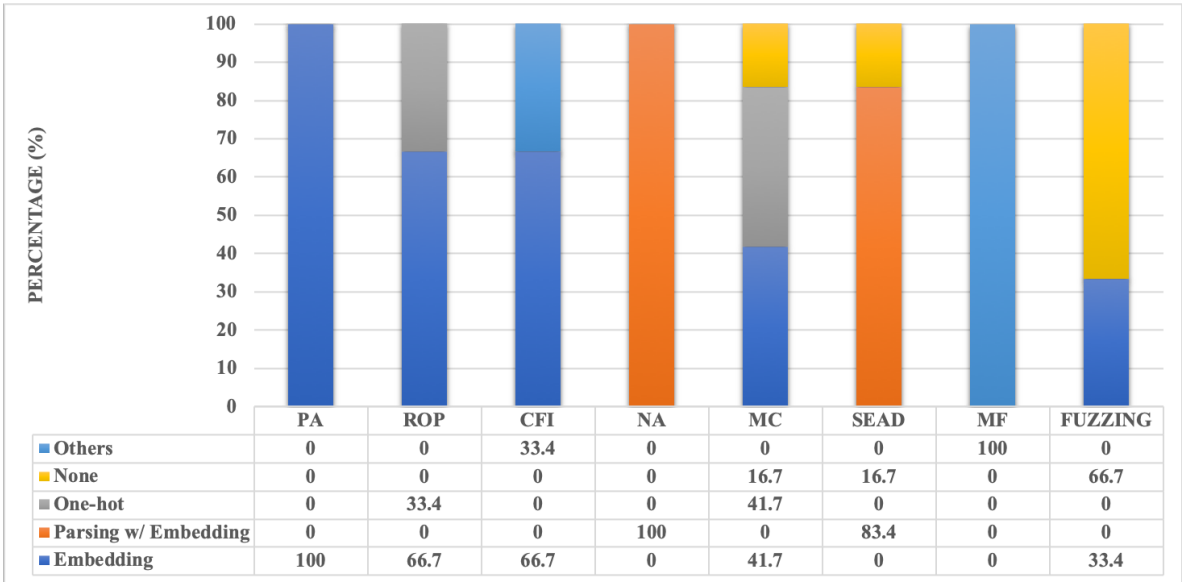
Figure 3: Statistics of Phase II methods on eight security problems

In this following, we do further consider the question, "What if Phase II is skipped?", not the question, "Is Phase III always necessary?". This is because most of the selected papers do not consider Phase III methods (76%), or adopt with no concrete reasoning (19%). Specifically, we demonstrate how Phase II has been adopted according to eight security problems, different types of data, various models of NN and various outputs of NN models, in depth. Our key findings are summarized as follows:

- How to fit security domain knowledge into raw data has not been well-studied yet.
- While raw text data are commonly parsed after embedding, raw binary data are converted using various Phase II methods.
- Raw data are commonly converted into a vector format to fit well to a specific NN model using various Phase II methods.
- Various Phase II methods are used according to the relationship between output of security problem and output of NN models.

11.1 What if Phase II is skipped?

From the analysis results of our selected papers for review, we roughly classify Phase II methods into embedding[2], parsing combined with embedding[3], one-hot encoding[4]and the $Others$[5].



	PA	ROP	CFI	NA	MC	SEAD	MF	FUZZING
Others	0	0	33.4	0	0	0	100	0
None	0	0	0	0	16.7	16.7	0	66.7
One-hot	0	33.4	0	0	41.7	0	0	0
Parsing w/ Embedding	0	0	0	100	0	83.4	0	0
Embedding	100	66.7	66.7	0	41.7	0	0	33.4

Figure 4: Statistics of Phase II methods for eight security problems

11.1.1 Findings on eight security problems

From Figure 3, we observe that most of papers over 93% use one of Phase II methods only except for papers less than 7%, most of which belong to fuzzing. Specifically, we observe that embedding (35%) and parsing combined with embedding (30%) are commonly used. We also observe that papers less than 13% rarely use various $Others$ methods. Especially, we observe that 15% of papers use one-hot encoding because most of security data include categorical input values, which are not directly analyzed by Deep Learning models.

From Figure 4, we also observe that according to security problems, different Phase II methods are used. First, PA, ROP and CIF should convert raw data into a vector format using embedding because they commonly collect instruction sequence from binary data. Second, NA and SEAD use parsing combined with embedding because raw data such as the network traffic and system logs consist of the complex attributes with the different formats such as categorical and numerical input values. Third, we observe that MF uses various data structures because memory dumps from memory layout are unstructured. Fourth, fuzzing generally uses no data conversion since Deep Learning models are used to generate the new input data with the same data format as the original raw data. Finally, we observe that MC commonly uses one-hot encoding and embedding because malware binary and well-structured security log files include categorical, numerical and unstructured data in general. These observations indicate that type of data strongly influences on use of Phase II methods. We also observe that only MF among eight security problems commonly transform raw data into well-structured data embedding a specialized security domain knowledge. This observation indicates that various conversion methods of raw data into well-structure data which embed various security domain knowledge

[5]The data conversion method that represents high-dimensional discrete variables into low-dimensional continuous vector [70].

[6]The data conversion method that constitutes an input data into syntactic components in order to test conformability after embedding.

[7]A simple embedding where each data belonging to a specific category is mapped to a vector of 0s and a single 1. Here, the low-dimension transformed vector is not managed.

[8]A set of data conversion methods which generate data structures including specific specific domain knowledge for different security problems, e.g., memory graph [45].

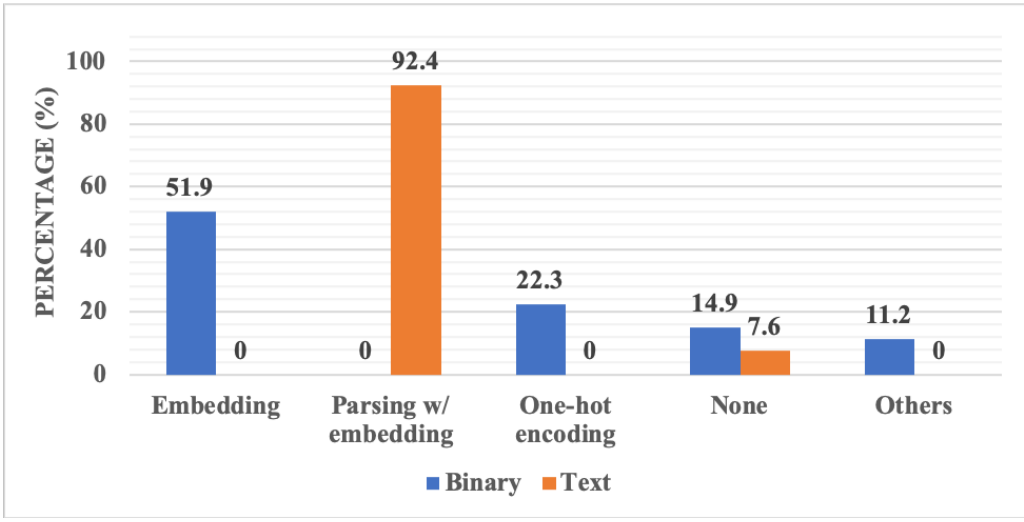


Figure 5: Statistics of Phase II methods on type of data

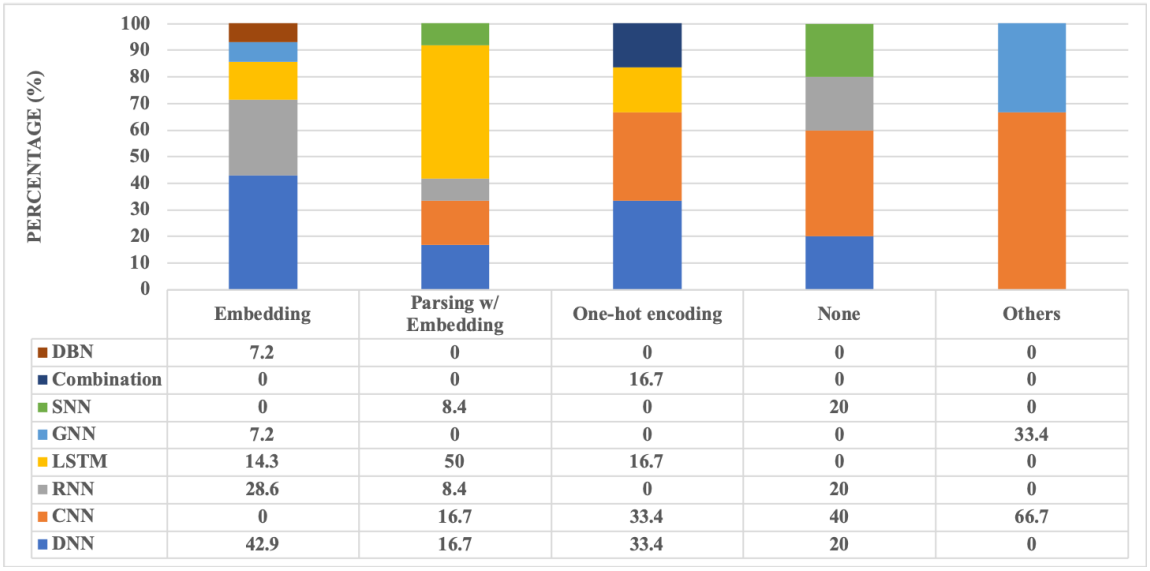
are not yet studied in depth.

11.1.2 Findings on different data types

Note that according to types of data, a NN model works better than the others. For example, CNN works well with images but does not work with text. From Figure 5 for raw binary data, we observe that 51.9%, 22.3% and 11.2% of security papers use embedding, one-hot encoding and $Others$, respectively. Only 14.9% of security papers, especially related to fuzzing, do not use one of Phase II methods. This observation indicates that binary input data which have various binary formats should be converted into an input data type which works well with a specific NN model. From Figure 5 for raw text data, we also observe that 92.4% of papers use parsing with embedding as the Phase II method. Note that compared with raw binary data whose formats are unstructured, raw text data generally have the well-structured format. Raw text data collected from network traffics may also have various types of attribute values. Thus, raw text data are commonly parsed after embedding to reduce redundancy and dimensionality of data.

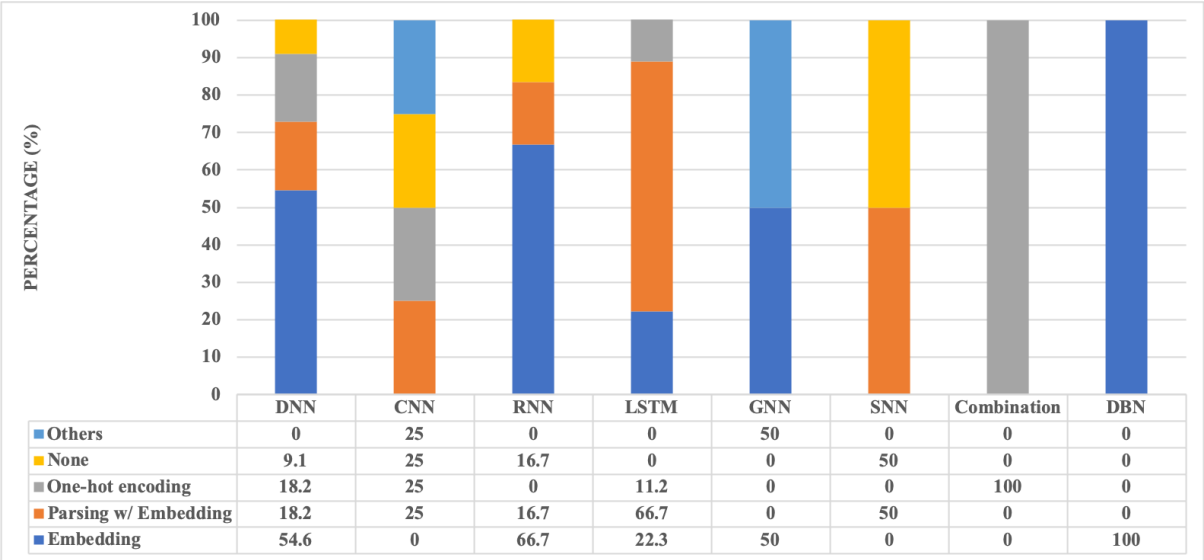
11.1.3 Findings on various models of NN

According to types of the converted data, a specific NN model works better than the others. For example, CNN works well with images but does not work with raw text. From Figure 6a, we observe that use of embedding for DNN (42.9%), RNN (28.6%) and LSTM (14.3%) models approximates to 85%. This observation indicates that embedding methods are commonly used to generate sequential input data for DNN, RNN and LSTM models. Also, we observe that one-hot encoded data are commonly used as input data for DNN (33.4%), CNN (33.4%) and LSTM (16.7%) models. This observation indicates that one-hot encoding is one of common Phase II methods to generate numerical values for image and sequential input data because many raw input data for security problems commonly have the categorical features. We observe that the CNN (66.7%) model uses the converted input data using the $Others$ methods to express the specific domain knowledge into the input data structure of NN networks. This is because general vector formats including graph, matrix and so on can also be used as an input value of the CNN model.



	Embedding	Parsing w/ Embedding	One-hot encoding	None	Others
DBN	7.2	0	0	0	0
Combination	0	0	16.7	0	0
SNN	0	8.4	0	20	0
GNN	7.2	0	0	0	33.4
LSTM	14.3	50	16.7	0	0
RNN	28.6	8.4	0	20	0
CNN	0	16.7	33.4	40	66.7
DNN	42.9	16.7	33.4	20	0

(a) Type of NN over Phase II methods



	DNN	CNN	RNN	LSTM	GNN	SNN	Combination	DBN
Others	0	25	0	0	50	0	0	0
None	9.1	25	16.7	0	0	50	0	0
One-hot encoding	18.2	25	0	11.2	0	0	100	0
Parsing w/ Embedding	18.2	25	16.7	66.7	0	50	0	0
Embedding	54.6	0	66.7	22.3	50	0	0	100

(b) Phase II methods over type of NN

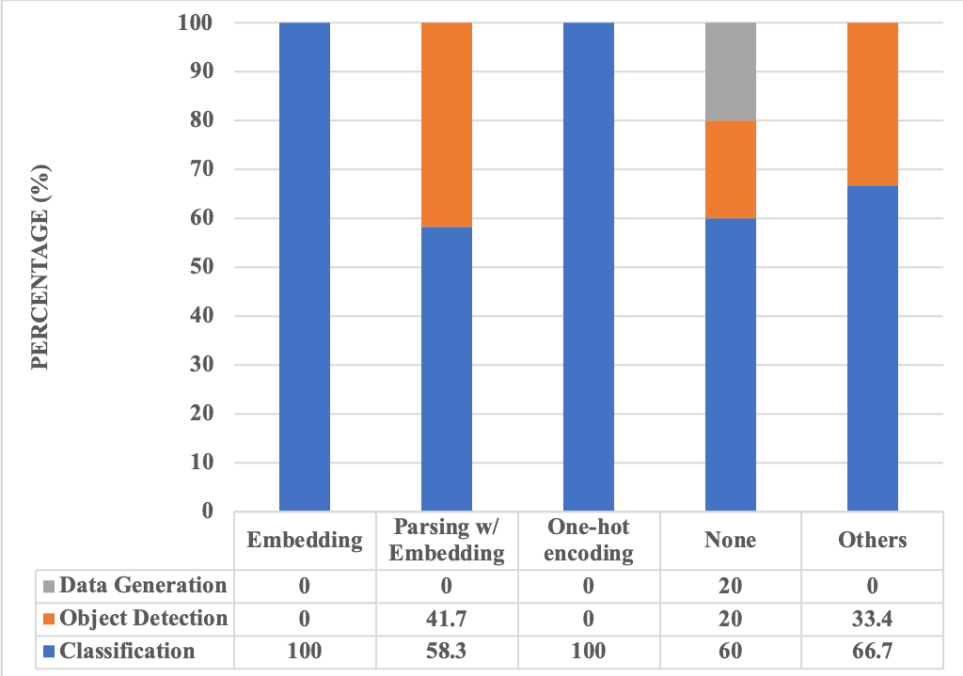
Figure 6: Statistics of Phase II methods for various types of NNs

From Figure 6b, we observe that DNN, RNN and LSTM models commonly use embedding, one-hot encoding and parsing combined with embedding. For example, we observe security papers of 54.6%, 18.2% and 18.2% models use embedding, one-hot encoding and parsing combined with embedding, respectively. We also observe that the CNN model is used with various Phase II methods because any vector formats such as image can generally be used as an input data of the CNN model.

11.1.4 Findings on output of NN models

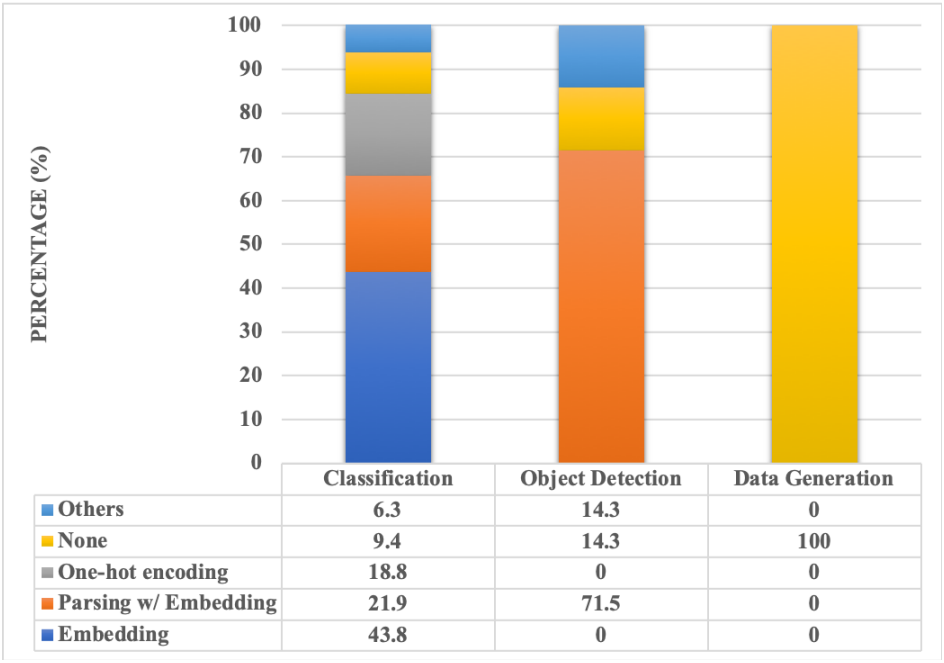
According to the relationship between output of security problem and output of NN, we may use a specific Phase II method. For example, if output of security problem is given into a class (e.g., normal or abnormal), output of NN should also be given into classification.

From Figure 7a, we observe that embedding is commonly used to support a security prob-



	Embedding	Parsing w/ Embedding	One-hot encoding	None	Others
Data Generation	0	0	0	20	0
Object Detection	0	41.7	0	20	33.4
Classification	100	58.3	100	60	66.7

(a) Output of NN over Phase II methods



	Classification	Object Detection	Data Generation
Others	6.3	14.3	0
None	9.4	14.3	100
One-hot encoding	18.8	0	0
Parsing w/ Embedding	21.9	71.5	0
Embedding	43.8	0	0

(b) Phase II methods over output of NN

Figure 7: Statistics of Phase II methods for various output of NN

lem for classification (100%). Parsing combined with embedding is used to support a security problem for object detection (41.7%) and classification (58.3%). One-hot encoding is used only for classification (100%). These observations indicate that classification of a given input data is the most common output which is obtained using Deep Learning under various Phase II methods.

From Figure 7b, we observe that security problems, whose outputs are classification, commonly use embedding (43.8%) and parsing combined with embedding (21.9%) as the Phase II

method. We also observe that security problems, whose outputs are object detection, commonly use parsing combined with embedding (71.5%). However, security problems, whose outputs are data generation, commonly do not use the Phase III methods. These observations indicate that a specific Phase II method has been used according to the relationship between output of security problem and use of NN models.

12 Further areas of investigation

Since any Deep Learning models are stochastic, each time the same Deep Learning model is fit even on the same data, it might give different outcomes. This is because deep neural networks use random values such as random initial weights. However, if we have all possible data for every security problem, we may not make random predictions. Since we have the limited sample data in practice, we need to get the best-effort prediction results using the given Deep Learning model, which fits to the given security problem.

How can we get the best-effort prediction results of Deep Learning models for different security problems? Let us begin to discuss about the stability of evaluation results for our selected papers for review. Next, let us discuss about the influence of security domain knowledge on prediction results of Deep Learning models.

12.1 How stable are evaluation results?

When evaluating neural network models, Deep Learning models commonly use three methods: train-test split; train-validation-test split; and k-fold cross validation. A train-test split method splits the data into two parts, i.e., training and test data. Even though a train-test split method makes the stable prediction with a large amount of data, predictions vary with a small amount of data. A train-validation-test split method splits the data into three parts, i.e., training, validation and test data. Validation data are used to estimate predictions over the unknown data. k-fold cross validation has k different set of predictions from k different evaluation data. Since k-fold cross validation takes the average expected performance of the NN model over k-fold validation data, the evaluation result is closer to the actual performance of the NN model.

From the analysis results of our selected papers for review, we observe that 40.0% and 32.5% of the selected papers are measured using a train-test split method and a train-validation-test split method, respectively. Only 17.5% of the selected papers are measured using k-fold cross validation. This observation implies that even though the selected papers show almost more than 99% of accuracy or 0.99 of F1 score, most solutions using Deep Learning might not show the same performance for the noisy data with randomness.

To get stable prediction results of Deep Learning models for different security problems, we might reduce the influence of the randomness of data on Deep Learning models. At least, it is recommended to consider the following methods:

- **Do experiments using the same data many time**: To get a stable prediction with a small amount of sample data, we might control the randomness of data using the same data many times.
- **Use cross validation methods, e.g. k-fold cross validation**: The expected average and variance from k-fold cross validation estimates how stable the proposed model is.

12.2 How does security domain knowledge influence the performance of security solutions using Deep Learning?

When selecting a NN model that analyzes an application dataset, e.g., MNIST dataset [71], we should understand that the problem is to classify a handwritten digit using a 28×28 black. Also, to solve the problem with the high classification accuracy, it is important to know which part of each handwritten digit mainly influences the outcome of the problem, i.e., a domain knowledge.

Even while solving a security problem, knowing and using security domain knowledge for each security problem is also important due to the following reasons. First, *different domain knowledge might generate different outcomes for each security problem using Deep Learning*. That is, different data need to be collected, converted and analyzed depending on domain knowledge of each security problem. For example, some ROP papers using Deep Learning [16] collect program instruction sequences and then, convert them into a control-flow graph (CFG) which helps for Deep Learning models to identify a binary branch information. On the other hand, some NA papers using Deep Learning [21] collect Pcap files including network packet level features and then, covert them into a vector which helps for Deep Learning models to identify a specific attack type such as distributed denial of service (DDoS) attack. That is, by using different input data which present different security domain knowledge, NN models show the different outcomes for each security problem.

Second, *using security domain knowledge might be useful in making the size of input data compact*. While the performance of NN models improves using the large amount of data with the high dimension, the training and evaluation (test) time also increase inefficiently. Since domain knowledge helps to clean the raw data to make NN models easier to interpret, security domain knowledge also helps to spend the small training and evaluation (test) time while keeping the same performance. However, due to the influence of the randomness of data on Deep Learning models, domain knowledge should be carefully adopted to avoid potential decrease of the accuracy.

Third, *security domain knowledge might be relevant to false outcomes*. The performance of Deep Learning models is commonly measured using four metrics: accuracy, precision, recall and F-1 measure. This indicates that the number of correct predictions over the number of predictions made is commonly important in Deep Learning. On the other hand, studies on security problems consider two additional metrics, i.e., false negative rate and false positive rate. This is because wrong outcomes may cause serious security breaches. Since security domain knowledge reduces the influence of randomness of data on NN models, security domain knowledge might be relevant to decreasing false negative rate and false positive rate.

Finally, *the performance of security solutions even using Deep Learning might vary according to datasets*. When evaluating different NN models, standard datasets such as MNIST for recognizing handwritten 10 digits and CIFAR10 [72] for recognizing 10 object classes are used for performance comparison of different NN models. However, there is no known state-of-the-art standard datasets for evaluating NN models on different security problems. Due to such a limitation, we observe that most security papers using Deep Learning do not compare the performance of different security solutions even when they consider the same security problem. Thus, it is recommended to generate and use a standard dataset for a specific security problem for comparison.

13 Conclusion

This paper seeks to provide a dedicated review of the very recent research works on using Deep Learning techniques to solve computer security challenges. In particular, the review covers eight computer security problems being solved by applications of Deep Learning: security-oriented program analysis, defending ROP attacks, achieving CFI, defending network attacks, malware classification, system-event-based anomaly detection, memory forensics, and fuzzing for software security. Our observations of the reviewed works indicate that the literature of using Deep Learning techniques to solve computer security challenges is still at an earlier stage of development.

References

[1] A. K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions against Programs. In Proceeding of Annual Computer Security Applications Conference (ACSAC), 1998.

[2] W. Hu, Y. Liao, and V. R. Vemuri. Robust Anomaly Detection using Support Vector Machines. In International Conference on Machine Learning (ICML), 2003.

[3] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo. One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses. In Proceedings of the Workshop on Data Mining for Computer Security, 2003.

[4] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning For Network Intrusion Detection. In 2010 IEEE Symposium on Security and Privacy (S&P), 2010.

[5] NSCAI Intern Report for Congress. Technical report, 2019. `https://drive.google.com/file/d/153OrxnuGEjsUvlxWsFYauslwNeCEkvUb/view`.

[6] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[7] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8):1798–1828, Aug 2013.

[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, pages 1285–1298, New York, NY, USA, 2017. ACM.

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

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural Nets Can Learn Function Type Signatures from Binaries. In 26th USENIX Security Symposium (USENIX Security 17), pages 99–116, 2017.

[11] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. {DEEPVSA}: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In 28th USENIX Security Symposium (USENIX Security 19), pages 1787–1804, 2019.

[12] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 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, pages 363–376. ACM, 2017.

[13] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In 23rd USENIX Security Symposium (USENIX Security 14), pages 845–860, San Diego, CA, August 2014. USENIX Association.

[14] Xusheng Li, Zhisheng Hu, Yiwei Fu, Ping Chen, Minghui Zhu, and Peng Liu. ROPNN: Detection of ROP Payloads Using Deep Neural Networks. arXiv preprint arXiv:1807.11110, 2018.

[15] Li Chen, Salmin Sultana, and Ravi Sahita. Henet: A Deep Learning Approach on Intel® Processor Trace for Effective Exploit Detection. In 2018 IEEE Security and Privacy Workshops (SPW), pages 109–115. IEEE, 2018.

[16] Jiliang Zhang, Wuqiao Chen, and Yuqi Niu. DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning. arXiv preprint arXiv:1905.01858, 2019.

[17] Carter Yagemann, Salmin Sultana, Li Chen, and Wenke Lee. Barnum: Detecting Document Malware via Control Flow Anomalies in Hardware Traces. In International Conference on Information Security, pages 341–359. Springer, 2019.

[18] 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), pages 45–52. IEEE, 2017.

[19] K Millar, A Cheng, HG Chew, and C-C Lim. Deep Learning for Classifying Malicious Network Traffic. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 156–161. Springer, 2018.

[20] Yong Zhang, Xu Chen, Da Guo, Mei Song, Yinglei Teng, and Xiaojuan Wang. PCCN: Parallel Cross Convolutional Neural Network for Abnormal Network Traffic Flows Detection in Multi-Class Imbalanced Network Traffic Flows. IEEE Access, 7:119904–119916, 2019.

[21] Xiaoyong Yuan, Chuanhuang Li, and Xiaolin Li. DeepDefense: Identifying DDoS Attack via Deep Learning. In 2017 IEEE International Conference on Smart Computing (SMARTCOMP), pages 1–8. IEEE, 2017.

[22] Remi Varenne, Jean Michel Delorme, Emanuele Plebani, Danilo Pau, and Valeria Tomaselli. Intelligent Recognition of TCP Intrusions for Embedded Micro-controllers. In International Conference on Image Analysis and Processing, pages 361–373. Springer, 2019.

[23] Chuanlong Yin, Yuefei Zhu, Jinlong Fei, and Xinzheng He. A Deep Learning Approach for Intrusion Detection using Recurrent Neural Networks. IEEE Access, 5:21954–21961, 2017.

[24] Serpil Ustebay, Zeynep Turgut, and M Ali Aydin. Cyber Attack Detection by Using Neural Network Approaches: Shallow Neural Network, Deep Neural Network and AutoEncoder. In International Conference on Computer Networks, pages 144–155. Springer, 2019.

[25] Osama Faker and Erdogan Dogdu. Intrusion Detection Using Big Data and Deep Learning Techniques. In Proceedings of the 2019 ACM Southeast Conference, pages 86–93. ACM, 2019.

[26] Leonardo De La Rosa, Sean Kilgallon, Tristan Vanderbruggen, and John Cavazos. Efficient Characterization and Classification of Malware Using Deep Learning. In Proceedings - Resilience Week 2018, RWS 2018, pages 77–83, 2018.

[27] Joshua Saxe and Konstantin Berlin. Deep Neural Network based Malware Detection using Two Dimensional Binary Program Features. In 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pages 11–20. IEEE, 2015.

[28] Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering Convolutional Networks for Malware Classification and Analysis. Proceedings of the International Joint Conference on Neural Networks, 2017-May:3838–3845, 2017.

[29] Niall McLaughlin, Jesus Martinez Del Rincon, Boo Joong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupe, and Gail Joon Ahn. Deep Android Malware Detection. Proceedings of the 7th ACM Conference on Data and Application Security and Privacy, pages 301–308, 2017.

[30] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware Detection with Deep Neural Network Using Process Behavior. In Proceedings - International Computer Software and Applications Conference, 2:577–582, 2016.

[31] George E. Dahl, Jack W. Stokes, Li Deng, and Dong Yu. Large-scale Malware Classification using Random Projections and Neural Networks. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 3422–3426, 2013.

[32] Robin Nix and Jian Zhang. Classification of Android Apps and Malware using Deep Neural Networks. Proceedings of the International Joint Conference on Neural Networks, 2017-May:1871–1878, 2017.

[33] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil D.B. Bruce, Yang Wang, and Farkhund Iqbal. Malware Classification with Deep Convolutional Neural Networks. 9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018 - Proceedings, 2018-Janua:1–5, 2018.

[34] Zhihua Cui, Fei Xue, Xingjuan Cai, Yang Cao, Gai Ge Wang, and Jinjun Chen. Detection of Malicious Code Variants Based on Deep Learning. IEEE Transactions on Industrial Informatics, 14(7):3187–3196, 2018.

[35] Omid E. David and Nathan S. Netanyahu. DeepSign: Deep Learning for Automatic Malware Signature Generation and Classification. Proceedings of the International Joint Conference on Neural Networks, 2015-Septe, 2015.

[36] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic Black-box End-to-End Attack against State of the Art API Call based Malware Classifiers. In International Symposium on Research in Attacks, Intrusions, and Defenses, 2018.

[37] Lifan Xu, Dongping Zhang, Nuwan Jayasena, and John Cavazos. HADM: Hybrid Analysis for Detection of Malware. 16:702–724, 2018.

[38] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Loganomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI'19, pages 4739–4745. AAAI Press, 2019.

[39] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18, pages 40–51, New York, NY, USA, 2018. ACM.

[40] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. Recurrent Neural Network Attention Mechanisms for Interpretable System Log Anomaly Detection. In Proceedings of the First Workshop on Machine Learning for Computing Systems, MLCS'18, pages 1:1–1:8, New York, NY, USA, 2018. ACM.

[41] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. Robust Log-based Anomaly Detection on Unstable Log Data. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pages 807–817, New York, NY, USA, 2019. ACM.

[42] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan. Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pages 351–360, Oct 2017.

[43] M. Du and F. Li. Spell: Streaming Parsing of System Event Logs. In 2016 IEEE 16th International Conference on Data Mining (ICDM), pages 859–864, Dec 2016.

[44] Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, D. Pei, J. Xu, Yu Chen, Hui Dong, Xianping Qu, and Lei Song. Syslog Processing for Switch Failure Diagnosis and Prediction in Datacenter Networks. In 2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS), pages 1–10, June 2017.

[45] Wei Song, Heng Yin, Chang Liu, and Dawn Song. DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, pages 606–618, New York, NY, USA, 2018. ACM.

[46] Rachel Petrik, Berat Arik, and Jared M. Smith. Towards Architecture and OS-Independent Malware Detection via Memory Forensics. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, pages 2267–2269, New York, NY, USA, 2018. ACM.

[47] Antonis Michalas and Rohan Murray. MemTri: A Memory Forensics Triage Tool Using Bayesian Network and Volatility. In Proceedings of the 2017 International Workshop on Managing Insider Security Threats, MIST '17, pages 57–66, New York, NY, USA, 2017. ACM.

[48] Yusheng Dai, Hui Li, Yekui Qian, and Xidong Lu. A Malware Classification Method Based on Memory Dump Grayscale Image. Digital Investigation, 27:30 – 37, 2018.

[49] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. NeuFuzz: Efficient Fuzzing with Deep Neural Network. IEEE Access, 7:36340–36352, 2019.

[50] Dongdong Shi and Kexin Pei. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. IEEE security & privacy, 2019.

[51] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep Reinforcement Fuzzing. In 2018 IEEE Security and Privacy Workshops (SPW), pages 116–122. IEEE, 2018.

[52] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 50–59. IEEE Press, 2017.

[53] Mohit Rajpal, William Blum, and Rishabh Singh. Not All Bytes are Equal: Neural Byte Sieve for Fuzzing. arXiv preprint arXiv:1711.04596, 2017.

[54] Hovav Shacham et al. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In ACM conference on Computer and communications security, pages 552–561. New York,, 2007.

[55] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. ACM Transactions on Information and System Security (TISSEC), 13(1):4, 2009.

[56] Jonathan Salwant. ROPGadget. `https://github.com/JonathanSalwan/ROPgadget`.

[57] Unicorn-The ultimate CPU emulator. `https://www.unicorn-engine.org/`.

[58] Vladimir Kiriansky, Derek Bruening, Saman P Amarasinghe, et al. Secure Execution via Program Shepherding. In USENIX Security Symposium, volume 92, page 84, 2002.

[59] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In 28th USENIX Security Symposium (USENIX Security 19), pages 1805–1821, Santa Clara, CA, August 2019. USENIX Association.

[60] Susan Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. ACM Trans. Program. Lang. Syst., 19(1):1–6, January 1997.

[61] Gang Tan and Trent Jaeger. CFG Construction Soundness in Control-Flow Integrity. In Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, pages 3–13. ACM, 2017.

[62] Zhilong Wang and Peng Liu. GPT Conjecture: Understanding the Trade-offs between Granularity, Performance and Timeliness in Control-Flow Integrity, 2019.

[63] Minh Hai Nguyen, Dung Le Nguyen, Xuan Mao Nguyen, and Tho Thanh Quan. Auto-Detection of Sophisticated Malware using Lazy-Binding Control Flow Graph and Deep Learning. Computers & Security, 76:128–155, 2018.

[64] 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), pages 1–6. IEEE, 2015.

[65] IDS 2017 Datasets, 2019. `https://www.unb.ca/cic/datasets/ids-2017.html`.

[66] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: A Survey. Cybersecurity, 1(1):6, 2018.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In In the Network and Distributed System Security Symposium (NDSS), volume 16, pages 1–16, 2016.

[68] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A Taint Based Approach for Smart Fuzzing. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 818–825, April 2012.

[69] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In 27th USENIX Security Symposium (USENIX Security 18), pages 745–761, Baltimore, MD, August 2018. USENIX Association.

[70] Google Developers. Embeddings. `https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture`.

[71] Yann LeCun and Corinna Cortes. MNIST Handwritten Digit Database. 2010. `http://yann.lecun.com/exdb/mnist/`.

[72] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 (Canadian Institute for Advanced Research). 2010. `http://www.cs.toronto.edu/~kriz/cifar.html`.