Journal of Computer Security 0 (0) 1
IOS Press

Deep Learning for Detecting Logic-flaw-exploiting Network Attacks: An End-to-end Approach

Qingtian Zou [a,*], Anoop Singhal [b], Xiaoyan Sun [c] and Peng Liu [a]
[a] *College of Information Sciences and Technology, The Pennsylvania State University, PA, USA*
E-mails: qzz32@psu.edu, pxl20@psu.edu
[b] *Security Test, Validation and Measurement Group, National Institute Standards and Technology, MD, USA*
E-mail: anoop.singhal@nist.gov
[c] *College of Engineering & Computer Science, California State University, Sacramento, CA, USA*
E-mail: xiaoyan.sun@csus.edu

Abstract. Network attacks have become a major security concern for organizations worldwide. A category of network attacks that exploit the logic (security) flaws of a few widely-deployed authentication protocols has been commonly observed in recent years. Such logic-flaw-exploiting network attacks often do not have distinguishing signatures, and can thus easily evade the typical signature-based network intrusion detection systems. Recently, researchers have applied neural networks to detect network attacks with network logs. However, public network data sets have major drawbacks such as limited data sample variations and unbalanced data with respect to malicious and benign samples. In this paper, we present a new end-to-end approach based on protocol fuzzing to automatically generate high-quality network data, on which deep learning models can be trained for network attack detection. Our findings show that protocol fuzzing can generate data samples that cover real-world data, and deep learning models trained with fuzzed data can successfully detect the logic-flaw-exploiting network attacks.

Keywords: Network Attack, Data Set, Protocol Fuzzing, Machine Learning

1. Introduction

Cyber attacks happen constantly with growing complexity and volume. As one of the most prevalent ways to compromise enterprise networks, network attack remains a prominent security concern. It can lead to serious consequences such as large-scale data breaches, system infection, and integrity degradation, particularly when network attacks are employed in attack strategies such as advanced persistent threats (APT) [1, 2]. Among the different types of network attacks, the *logic-flaw-exploiting network attacks* are very commonly seen.

Logic-flaw-exploiting network attacks refer to network attacks which exploit the logic flaws within the protocol specifications or implementations. Such attacks are very different from other attacks such as memory corruption for code reusing, command and control (C&C) over HTTP/HTTPS, and (distributed) denial of service with/without botnet, etc. Memory corruption is more about the server rather

*Corresponding author. E-mail: qzz32@psu.edu.

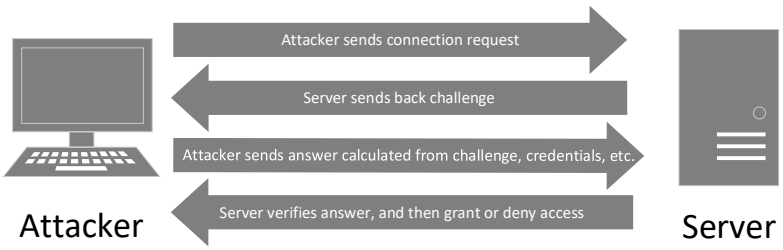


Fig. 1. PtH illustration.

than the protocol; C&C over HTTP/HTTPS assume the HTTP/HTTPS is running in a normal way; and (distributed) denial of service is usually accomplished by exhausting the server's resources instead of exploiting a logic flaw within a protocol.

To illustrate logic-flaw-exploiting network attacks, we present one motivating example in Fig. 1. As shown in Fig. 1, pass the hash (PtH), which has been observed to be used in more than ten APT campaigns in the recent years [3], exploits the weak authentication mechanism during remote login. The figure shows four stages during the PtH attack, which happens at the authentication stage during Windows remote login using the Server Message Block (SMB) protocol, or a newer version of it, referred to as SMB2. The authentication stage can be viewed as a sequence made up of client's authentication request, server's challenge, client's challenge response and server's authentication response. The client first sends a session setup request to the server; then the server responds to the client with a challenge; on receiving the challenge, the client uses the challenge and hashes to do calculations and sends back the result in challenge response packet; finally, the server verifies the result and sends back authentication response indicating whether authentication succeeds or not. This authentication logic assumes that only legitimate users have access to the hashes, which is not always the case in reality, and this makes PtH one of the logic-flaw-exploiting network attacks.

Detecting logic-flaw-exploiting network attacks is very important considering their common presence in APT campaigns. However, it is still a very challenging problem. Network attack detection methods can mainly be classified into two categories: *host-independent* methods and *host-dependent* methods. The host-independent methods solely rely on the network traffic, while the host-dependent methods [4–9] depend on additional data collected on the victim hosts. The host-dependent methods have some evident drawbacks: they have fairly high deployment costs and operation costs; they are error-prone due to necessary manual configuration by human administrators. Therefore, host-independent detection methods are highly desired as they can decrease deployment and operation costs while reducing the attack surface of detection system.

However, we found that the existing host-independent methods, including the classical intrusion detection approaches, often fall short in detecting some well-known and commonly used network attacks, such as the PtH, ARP poisoning, and Domain Name Service (DNS) cache poisoning attacks. Some traditional methods include signature-based, rule-based, and anomaly-detection-based methods, but they all have significant drawbacks, which will be explained later. Such drawbacks prevent them from being practically deployed in the real world.

Recently there is a trend for using machine learning (ML) and deep learning (DL) techniques to detect network attacks involved in APTs. Nevertheless, the deep learning approaches could also achieve mixed results [10, 11], if they do not address the following two challenges. The first challenge is the *generation*

of useful data sets. Neural networks require high-quality network traffic data and correct labels, which are hard to obtain in real world. The main reason is that real-world network traffic is often flooded with benign packets, which makes it difficult to label malicious network packets. Since lack of ground-truth is a challenge widely recognized in the network security community, how to assign correct labels to network packets in real-world network traffic is still an open problem. Although public data sets [12–16] for network attacks are available, they are barely useful in detecting logic-flaw-exploiting network attacks due to the following reasons: 1) Most data sets are generated (and synthesized) with various simulated or emulated types of benign and attack activities, and each type of attack is only launched for a few times. Therefore, these data sets are highly unbalanced and have limited variations, which downgrades the trained neural networks. 2) The malicious activities included in the public data sets, such as C&C, worm, and denial of service, etc., do not exploit logic flaws in the protocol specifications or implementations, and thus cannot be employed to detect the logic-flaw-exploiting network attacks.

The second challenge to apply the deep learning approaches is to *identify appropriate neural networks and train the models*. There are a variety of neural network architectures, including multi-layer perceptron (MLP) [17], convolutional neural network (CNN) [18], recurrent neural network (RNN) [19], etc., which have different characteristics and capabilities. Questions such as which architecture works best for network attack detection, and how to tune the hyper-parameters within models for optimization, are yet to be answered.

In this paper, we propose an *end-to-end approach* to detect the logic-flaw-exploiting network attacks. The end-to-end approach means it starts with acquiring data and ends with detecting attacks using the trained neural networks. *To address the data generation challenge* mentioned above, we propose a new protocol fuzzing-based approach to generate the network traffic data. Protocol fuzzing means the client generating mutated network packets in a specific protocol as input to feed into the server program. Using protocol fuzzing for data generation has the following benefits: 1) With protocol fuzzing, a large variety of *malicious* network packets for a chosen network attack can be generated at a fast speed. 2) Since the network packets are all generated from the chosen network attacks, they can be labeled as malicious packets automatically without much human efforts. 3) Protocol fuzzing can generate data with more variations than real world data, or even data that are not yet observed in real world. 4) Protocol fuzzing can generate and cover malicious data samples which may otherwise be overlooked when applying deep learning. In deep learning, the changed values for the fuzzing fields may make the malicious data samples misclassified as benign. With protocol fuzzing, if the malicious data are generated in attacks, they will be labeled as malicious automatically, so they will not be omitted in the malicious data sets. In addition, the above-mentioned merits remain when protocol fuzzing is leveraged to generate the needed *benign* network packets. It should be noted that our method is different from data synthesis. Data synthesis is to enhance existing data [20], while our method is to generate new data.

To address the neural network model training challenge, we propose the following procedures: 1) For network attacks that we can identify fields of interest, such as PtH attack, we directly examine the data, and then propose the suitable data representation and neural network architecture. 2) For other network attacks that the field of interests are not obvious, such as DNS cache poisoning and ARP poisoning attacks, we apply different neural network architectures to find out the ones with best performance. We propose to use accuracy, F1 score, detection rate, and false positive rate as the metrics to evaluate the neural networks from not only the performance perspective, but also the security perspective — the attack detection effectiveness. All models are trained on the data set with fuzzing involved. We then select the models that work the best, and evaluate them further on both the fuzzing data set and real attack data set with no fuzzing involved.

The main contributions of this work include: 1) We propose a deep learning based end-to-end approach to detect the logic-flaw-exploiting network attacks; 2) We propose to use protocol fuzzing to automatically generate high-quality network traffic data for applying deep learning techniques; 3) We propose a systematic procedure to evaluate and select the neural network models for logic-flaw-exploiting network attack detection; 4) We demonstrate the effectiveness of our approach with three classical logic-flaw-exploiting network attacks, including the PtH attack, DNS cache poisoning attack, and ARP poisoning attack. Some preliminary results have been published in Conference on Data and Applications Security and Privacy 2021 [21]. This paper includes new experiments and results not presented in the conference version, especially impacts of different data representations and probability thresholds, and interpretations on the model and generated data.

This remaining of the paper is structured as follows. Section 2 presents the related works. Section 3 discusses the network attack types that our approach can be applicable and the three specific network attacks used to demonstrate our approach. In section 4, we present the details for data generation using protocol fuzzing. In section 5, details about the neural network models trained with the generated data are provided. Section 6 evaluates the trained models. Section 7 presents the interpretations of our models. Section 8 discusses limitations of our experiment and approach. Section 9 concludes the paper.

2. Related Work

The research community has been tackling the network attack detection problem with both classical and novel approaches. In this section, we will discuss the research works related to network attack detection from different perspectives.

Traditional network attack detection approaches. Traditionally, people usually detect network attacks with approaches like signature-based, rule-based, and anomaly detection-based methods, some of which are still in use today. In the past, signature-based intrusion detection system (IDS) usually manually crafted signatures [22], which heavily depends on manual efforts. Nowadays, people focus more on automatically generating signatures [23]. However, signatures need to be constantly updated to align with newer attacks and signature-based detections can be easily evaded by slightly changing the attack payload. For some network attacks, signatures may not even exist. Similar problems also exist for rule-based methods [24, 25], which constantly need updates to the rules. As for anomaly detection-based methods, though they require much less manual efforts for updating, they tend to raise more false positives [26]. The reason is that anomaly detection-based methods fall short in distinguishing malicious activities and shifted normal user activity patterns, because such methods are often focusing on recognizing the previously-seen normal user activities. As a result, shifted normal user activity patterns often get misclassified as false positives which distract organizations' normal operations.

Traditional machine learning and deep learning for network attack detection. Network attacks are essential for APTs. The MITRE ATT&CK repository [27] includes some of the most famous and commonly used APT tactics and techniques. Tactics represent the purpose of techniques, such as initial access, privilege escalation, and lateral movements, etc. Techniques represent the approach by which an adversary achieves his/her goal, such as exploiting public-facing services, abusing valid accounts, phishing, etc. In some tactics, most techniques rely on network attacks. For example, for initial access [28], out of the nine listed techniques, six (drive-by compromise, exploit public-facing application, external remote services, phishing, supply chain compromise, and valid accounts) are related to network attacks. Another example is the lateral movement tactic [29], where all but one (replication through removable media) out of the nine listed techniques are network-related.

Table 1
UNSW-NB15 data set record distribution.

Type	Benign	Malicious								
		Fuzzers	Analysis	Backdoors	DoS	Exploits	Generic	Reconnaissance	Shellcode	Worm
Number of records	2,218,761	24,246	2,677	2,329	16,353	44,525	215,481	13,987	1,511	174

Generally speaking, common network attack types include probing, DoS, Remote-to-local (R2L), etc. To detect those attacks, some works focus on just one type of network attack and perform binary classifications. For example, MADE [30] employs machine learning to detect malware C&C network traffic, Ongun et al. [31] employs machine learning to detect botnet traffic, and DeepDefense [32] employs deep learning to detect distributed DoS (DDoS) attacks. Others [10, 11, 33–35] try multi-class classifications, which include one benign class and multiple malicious classes for different kinds of network attacks. The above-mentioned research works all use public data sets.

Network data sets for training and testing detection models. To apply deep learning for network attack detection, a data set is required on which the model can be trained. Raw data in the real world, unfortunately, is not good for applying deep learning, because real-world network traffics are flooded with benign data. Commonly used public data sets include KDD99 [15], NSL-KDD [13], UNSW-NB15 [14], CICIDS2017 [16], and CSE-CIC-IDS2018 [12]. The public data sets are all generated in test-bed environments, with simulated benign and malicious activities.

However, these data sets often have unbalanced number of benign and malicious data samples. Even for only malicious activities, multiple attack types may be included and the amount of malicious data samples for each type also varies a lot. For example, the NSL-KDD [13] contains 812,814 normal records and 262,178 malicious records in its training set. The benign to malicious rate is about 3.10:1. The UNSW-NB15 [14] contains 2,218,761 benign records and 321,283 malicious records (benign to malicious ratio is about 6.91:1), and the malicious records contain multiple types, which are highly unbalanced as well (i.e., some attack types hold a significantly greater number of records than other attack types). More importantly, we found that a main "missing piece" in these public data sets is that they do not focus on logic-flaw-exploiting network attacks such as PtH. Rather, they focus more on worms, Botnets, backdoors, DoS/DDoS, and so on.

Protocol fuzzing. Fuzzing is originally a black-box software testing technique, which looks for implementation bugs by feeding mutated data. A key function of fuzzers is to generate randomized data which still follows the original semantics. There are tools for building flexible and security-oriented network protocol fuzzers, such as SNOOZE [36]. Network protocol fuzzing frameworks such as AutoFuzz [37, 38] have also been presented. They either act as the client, constructing packets from the beginning, or act as a proxy, modifying packets on the fly.

In this paper, we use protocol fuzzing for a different purpose, which is to directly generate high-quality data sets that can be used to train neural networks. However, typical fuzzing methods aim to trigger program bugs, and it is not important to them whether sessions are complete or not. An interrupted session may actually indicate successful triggering of bugs. For example, boofuzz [39], a successor to the once preeminent open-source protocol fuzzer Sulley [40], sends out packets incrementally. In another word, if boofuzz is to fuzz three successive packets A, B, and C, it will send out packets in patterns like $[A]$, $[A, B]$, and $[A, B, C]$, without putting the session's completeness in consideration. However, we need

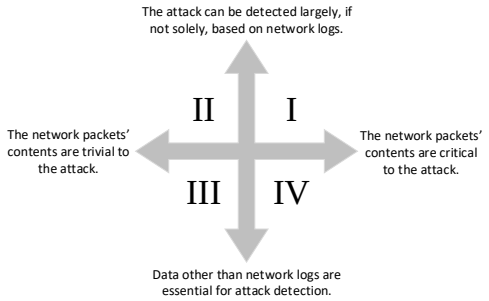


Fig. 2. Network attack categories.

to collect data for complete sessions, because complete sessions are necessary for either network attacks or benign activities. Therefore, instead of using the tools/frameworks mentioned earlier, we prepare our own fuzzing scripts for our specific requirements.

3. Problem Formulation

In principle, whether deep neural networks can play an essential role depends on the characteristics of the particular network attack in concern. To provide a more general guideline, we categorize network attacks into four categories like a Cartesian coordinate system as presented in Fig. 2. All network attacks are categorized into four quadrants, based on two criteria: (a) whether the network packets' contents are critical to the attack; (b) whether network logs are the key for network attack detection and whether they are sufficient, which means, whether some data other than network logs also plays a critical role for the detection.

We argue that the network attacks that fall into Region I are most likely to benefit from a carefully trained deep neural network. Accordingly, the protocol fuzzing for data generation is most useful for detecting the network attacks with the following characteristics: 1) The attack can be detected largely, if not solely, based on network logs. 2) The network packets' contents are critical to the attack. 3) There is not a distinctive signature for detecting the attack.

The rationale behind is as follows: (a) Protocol fuzzing is to generate fuzzed network traffic. If using solely network traffic is not sufficient and some other data is essential to detect the network attack, then protocol fuzzing has limited contribution to the attack detection. (b) Protocol fuzzing is to fuzz the network packets' contents, which means mutating the data in the packets. If the packets' contents are not important, then again fuzzing does help much with the attack detection. For example, if an attacker launches lateral movement attacks with stolen account username and password, the attack detection is more about monitoring the abuse of accounts. Network traffic is not helpful in this case because the attacker is using legitimate credentials and valid tools. Another example is resource exhaustion DoS attacks. The network traffic can be used to detect such attacks, but the packets' contents are of little use. The attack can be achieved by sending huge amounts of packets in a short time period, without changing packet contents significantly. (c) Lack of distinctive signature means traditional signature-based network attack detection mechanisms will fall short, while machine learning techniques become more appealing in this case.

Therefore, this paper will answer the question: is protocol fuzzing an effective approach in generating comprehensive data and trustworthy labels for applying deep learning to detect Region I attacks?

Without loss of generality, we have selected three representative network attacks in Region I, which are *PtH attacks, address resolution protocol (ARP) poisoning attacks*, and *domain name system (DNS) cache poisoning attacks*. PtH is a well-known attack for lateral movements, and has been reported to be used in more than ten APT campaigns in the recent years [3]. ARP poisoning attacks and DNS cache poisoning attacks are difficult to be identified by traditional approaches because both attacks involve spoofing. That is, attackers intentionally make the packets seem to be benign.

The three attacks can be detected by traditional approaches to some extent. For example, PtH may be detected by monitoring user login and login methods; ARP poisoning may be detected by monitoring the ARP table. However, such approaches are only effective after the attacks have succeeded, and they need to access certain log data on the victim machine. If the attacks can be detected at the network level, we can deploy the detection model to proactively raise alarms at an early stage. Other tools like Snort [25] may be able to detection those network attacks at the network level, but they are hardly effective against unknown attacks even of the same category, and their rules or signatures need to be constantly updated. To demonstrate the effectiveness of our approach, we have applied protocol fuzzing to generate network logs, trained and evaluated neural networks to detect the three attacks respectively based on the collected network traffic.

4. Generating Comprehensive Data

Because the available public data sets are barely useful for detecting the logic-flaw-exploiting network attacks, this paper will generate comprehensive data sets from scratch, including benign and malicious data sets. We have performed data generation for all three demonstration attacks including PtH, DNS cache poisoning, and ARP poisoning. ARP poisoning attack only requires one malicious packet for a successful attack, so we call it the single-packet attack. PtH and DNS cache poisoning attacks, however, need multiple malicious packets for one successful attack, so we call them multi-packet attacks. Below subsections discuss the general approach and implementation principles of protocol fuzzing followed by attack-specific details. All attacks are carried out thousands of times so that a fair amount of malicious data can be collected. Benign data generation also lasts long enough to gather the commensurate amount of data compared to malicious data. The network packet capturing is performed at the victim's side.

4.1. Protocol Fuzzing and The Implementation

In client-server enterprise computing, the server side protocol implementations are often complex and error-prone, and the clients are usually designed in accordance with the server. Hence, there is a need to achieve thorough testing of the server side implementation. That is why protocol fuzzing tools [36–38] are usually functioning at the client side, sending mutated network packets to the tested server programs, or acting as a proxy modifying packets on the fly and replaying them, so that unexpected errors on the tested server programs may be triggered. A main difference between protocol fuzzing and software fuzzing is that the protocol specification, especially its state transition diagram, will be used to guide the fuzzing process. In this way, fuzzing tests could be performed in a stateful manner.

In this paper, we leverage protocol fuzzing to change the contents of network packets, specifically, the values of some fields in the packets. If we are to fuzz a field, we will assign values chosen by ourselves, instead of the values chosen by the network client program. The fields to be fuzzed are chosen based on the following steps, as shown in Algorithm 1: 1) We list the fields in the packet of the attack-specific protocol. 2) We pick one field on the list and fuzz it by assigning values of our choices, rather than values

that are normally provided by the network programs. 3) We monitor how the attack goes after fuzzing the field. If the attack success rate is above 50%, we confirm that this field can be fuzzed. 4) After one field is fuzzed, we move on to the next field on the list, while keeping the already fuzzed field(s) still fuzzed. Though this algorithm seems naive, it requires minimum expert knowledge and is faster than a complete search. As for how the values are assigned (how the fields are fuzzed), details will be provided in the later paragraphs.

Result: `BList`, which stores fields to fuzz
input `AList` of all available fields;
initialize an empty `BList` to store fields to fuzz;
foreach *`field` in `AList`* **do**
    fuzz `field`;
    fuzz all fields in `BList`;
    launch the attack for hundreds of times;
    count successful attacks and calculate success rate;
    if *attack success rate is over 50%* **then**
        add `field` to `BList`;
    end
end

Algorithm 1: Select fields to be fuzzed.

To ensure the fuzzed packets are valid, we need to firstly make sure *AList*, the list of all candidate fuzzing fields, does not contain fields that will affect the packets' integrity, such as fields of checksum values and packet lengths. The values of those fields should not be arbitrarily changed. Furthermore, when we choose the fields to be fuzzed, we need to make sure the attack success rate after fuzzing this field is always above 50%. We use 50% as the threshold because the minimum field we tune is a bit, which only has two possible values, which are 0 and 1. Accordingly, if the attack success rate is below 50%, we assume that the attacker is likely to be concerned with the corresponding bit flipping. In another word, flipping this bit makes the attack (relatively) less successful than not flipping it, especially when the attacker has some other bits to flip with higher than 50% success rate.

We implemented a number of Python scripts to fuzz most fields and send the fuzzed network packets with open-source libraries such as scapy [41]. The fields are fuzzed by assigning values randomly in their valid ranges, so that the packets' integrity can be maintained. We will first take a look of the unfuzzed packet structure and corresponding documents, learn the valid range of that field, and then assign random values within that range. For example, one of the fields that are often fuzzed is the `time to live` value in the `IP` layer. By looking at the genuine packets and related documents, we learn that it is an 8-bit field, and the valid values are integers within $[0, 255]$. Hence, a random integer is generated from the range $[0, 255]$ and assigned to that field when fuzzing it. If the field to fuzz is a binary flag field, then the valid values only include 0 and 1, so we randomly generate an integer from range $[0, 1]$ and assign it to that field. Sometimes, binary flags are adjacent bits in the network packet, such as the `flags` field in the `IP` layer. For these, we treat all adjacent bits as one field and assign values to the entire field, instead of performing fuzzing to each individual bits respectively. Take the `flags` field as an example. It is a 3-bit field, and the first bit is reserved and must be 0, so the valid values are $0, 2, 4, 6$. When fuzzing those flags, we assign the whole 3-bit field with a randomly chosen value from the valid range, rather than assigning values to the last two bits separately.

However, some fields, such as the MAC addresses and IP addresses, may not be fuzzed as mentioned above using the Python scripts. 1) One reason is that we want to assign values in those fields in a way that the attacker machine itself can accept, such as MAC addresses. Though we can alter MAC addresses within Python scripts when crafting network packets, MAC addresses there will not be recognized by the machine itself. For those fields, we have created some additional scripts to complete the fuzzing. For MAC addresses, we will have another Bash script to randomly generate a MAC address, shut down the target network interface, re-initialize the network interface with the randomized MAC address, and then bring up the network interface. This Bash script is configured to auto-run periodically with a given time interval, so that the MAC address can be fuzzed in different network sessions. 2) Another reason not to use python scripts for some fields is that special procedures may be needed to assign valid values to these fields. For example, the IP addresses are not fuzzed using the Python scripts, even though they are accessible in the Python scripts. The IP addresses cannot be fuzzed by simply choosing a value randomly. Instead, a procedure is needed in order to assign valid values to the IP address field. Therefore, we deploy a Dynamic Host Configuration Protocol (DHCP) server in the local area network (LAN) to distribute IP addresses to machines connected to the LAN. If we are to fuzz the IP address of a host, we will connect it to the LAN, and make the DHCP server restart at a given time interval. Meantime, we'll flush the DHCP records with Bash scripts, so that the host will need to communicate with the DHCP server often for a new IP address.

4.2. ARP Poisoning Data Generation

ARP Poisoning Attack. When an Internet Protocol (IP) datagram is sent from one host to another in a LAN, the destination IP address must be resolved to a MAC address for transmission via the `ETH` layer. To resolve the IP address, a broadcast packet, known as an ARP request, is sent out on the LAN to ask which MAC address matches the destination IP. The destination host possessing this IP will respond with its MAC address in the ARP reply. ARP is a stateless protocol. Machines usually automatically cache all ARP replies they receive, regardless of whether they have requested them or not. Even ARP entries that have not yet expired will be overwritten when a new ARP reply packet is received. The host cannot verify where the ARP packets come from. This behavior is the vulnerability that allows ARP poisoning to occur. By spoofing ARP responses, the attacker can trick the victim into falsified mappings between IP addresses and MAC addresses, and thus intervening the network communication. For example, a machine within a LAN usually sends packets to the outside network through a router. By ARP poisoning, the attacker in the LAN may be able to redirect all traffic from and to the user machine to its own attacker machine. For example, at the user side, the attacker may map the router's IP address with the attacker's MAC address so that the traffic sent to the router is redirected to attacker. Similarly, at the router side, the user machine's IP address is mapped with the attacker's MAC address so that the traffic sent to the user machine is also redirected to the attacker. The attacker only needs to forward packets from the user machine and router to their intended targets, so that the attacker capture all network traffic without interrupting the network communication. One ARP poisoning attack illustration is presented in Fig. 3.

Data Generation for ARP Poisoning. For this attack, our test bed constructs a LAN that contains an Ubuntu machine configured as a router with DHCP server enabled, another Ubuntu as the user machine, and a Kali Linux as the attacker machine. Fuzzed fields include `MAC address` and `IP address`. The attacker machine is configured to send out fuzzed ARP responses with randomly generated IP addresses and MAC addresses periodically. The MAC address and IP address fuzzing methods was

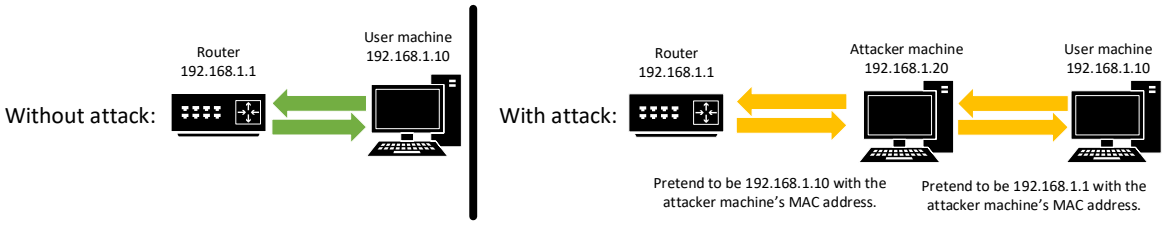


Fig. 3. ARP poisoning illustration.

discussed in Section 4.1. The victims ("router" and user machine) are configured to flush the IP-MAC mapping periodically. The IP addresses in spoofed packets are randomly generated from the same IP ranges distributed by the DHCP server, because the attacker is only interested in directing the traffic to an IP which actually exists in the LAN. On the user machine, we periodically run command $arp - a$ to print out the MAC-IP table. If MAC address in the table is not a valid machine in the LAN, then the attack is successful.

For benign data set generation, the user machine is configured to probe the presence of other machines by sending out ARP requests to all IP addresses within the LAN's IP range periodically so that enough data can be generated. In both the malicious and benign scenarios, the DHCP server is configured to flush the IP assignments to all machines within the LAN periodically, so that all machines' IP addresses within the LAN, except the DHCP server itself, will be changed in different iterations.

ARP poisoning can be achieved with one single malicious packet, and the detection neural network is also based on data samples constructed from individual packets. Therefore, one attack corresponds to one malicious network packet, and also one malicious data sample. In another word, the data sample's label directly indicate whether there is an attack or not.

4.3. PtH Data Generation

PtH Attack. PtH is a well-known technique for lateral movement. In remote login, plain text passwords are usually converted to hashes for authentication. Some authentication mechanisms only check whether hashes or the calculation results of them matches or not. PtH relies on these vulnerable mechanisms to impersonate normal users with dumped hashes. We assume that: (a) normal users use benign client programs that are usually authenticated through mechanisms other than just using hashes, and that (b) attackers cannot get the plain text passwords and have to rely on hashes to impersonate a normal user. We can capture the network packets at the server side and find out which kind of authentication mechanism is used. Login sessions that use those vulnerable authentication mechanisms can then be identified as PtH attack.

No.	Source	Destination	Protocol	Length	Info
26	172.29.36.125	172.29.151.231	SMB2	239	Session Setup Request, NTLMSSP_NEGOTIATE
27	172.29.151.231	172.29.36.125	SMB2	435	Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLENGE
28	172.29.36.125	172.29.151.231	SMB2	576	Session Setup Request, NTLMSSP_AUTH, User: CORP\Administrator
30	172.29.151.231	172.29.36.125	SMB2	151	Session Setup Response

Fig. 4. A subset of network packets during PtH attack.

Windows remote login processes, if not properly configured, can use such vulnerable authentication mechanisms. Windows remote login can be divided into three stages, protocol and mechanism negotiation (initial communication), authentication, and task-specific communication (afterwards communica-

tion). Each stage contains multiple network packets, and hashes are used in the authentication stage for impersonation, the details of which have been explained in section 1.

Data Generation for PtH. We set up a Windows 2012 Server R2 as the victim server machine, a Windows 7 as the user client machine, and another Kali Linux as the attacker machine. The data sets are automatically generated by protocol fuzzing, and the protocol of interest here is SMB/SMB2. SMB/SMB2 provides functions including file sharing, network browsing, printing, and inter-process communication over a network. In our data generation, more than 15 fields are fuzzed in each SMB/SMB2 packets, including `SMB flags`, `SMB capabilities`, and fields in SMB header, etc. Fig. 5 shows the fuzzing process for generating malicious data. We leverage the PtH script in Metasploit Framework to launch the attack. Boxes connected with solid lines are what happens at foreground, and boxes in the dash line area happen behind the scene. The process is to start the Metasploit Framework, set exploit parameters, start the exploitation, and then wait 25 seconds while monitoring the attack status. Waiting is necessary because the exploitation needs some time to complete. If the waiting time is too short, the attack may be stopped before completion. While the console is waiting at the foreground, the exploitation is on going at the background with fuzzed network packets. Network packets in all the three stages, initial communication, authentication, and afterwards communication, are fuzzed. After the exploitation, based on whether the attack succeeds or not, we may continue to establish C&C, like what a real attacker will do. (The C&C network traffic are mainly TCP packets, which are not used for attack detection. Details are discussed in subsection 5.2.) Finally, we quit all possibly established sessions and the Metasploit Framework, and then either start another fuzzing iteration to generate more data or stop. The sign of a successful PtH attack is an established reverse shell, which can be observed at the attacker's side.

The same fuzzing method has also been applied in the generation of benign data from normal network traffic. For the benign scenario, we first prepare a list of normal commands, including files reading, writing, network interactions, etc. For each benign fuzzing iteration, we first randomly choose a command from the list, and then use valid username, plain-text password, and tool to log in to the server and execute the command.

All the network packets from malicious and benign network traffic are captured using Wireshark at the victim server's side. Due to fuzzing, not all PtH attempts or benign access attempts can be guaranteed to be successful. For failed PtH attempts, we remove them from malicious data because they do not generate real malicious impact. These packets cannot be categorized as benign either because they are generated with attacker tools for malicious purpose, rather than for legitimate access. For failed benign accesses, we keep them in benign data, because normal user can also have failed logins due to typos, wrong passwords, etc.

In one PtH attack, there are packets for initial communications, authentication and afterwards communications. One data sample consists of multiple packets, and those packets may come from one, two, or all of the three stages above. Besides, one complete PtH attack or benign activity most certainly contains more packets than one data sample can represent. When labeling, if the session is malicious, then all data samples generated from this session is labeled malicious, and the same is also true for the benign cases.

4.4. DNS Cache Poisoning Data Generation

DNS Cache Poisoning Attack. A major functionality of Domain Name Service (DNS) is to provide the mapping between the domain names and IP addresses. When a client program on a user machine refers to a domain name, the domain name needs to be translated to an IP address for network communication. The DNS servers are responsible to perform such translation.

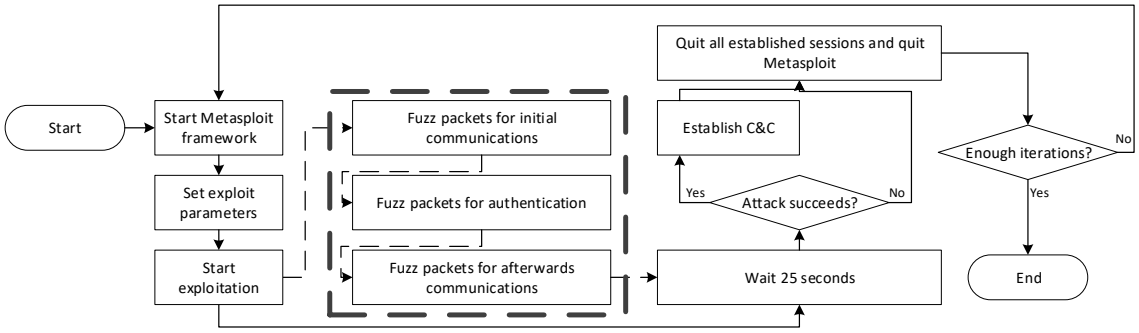


Fig. 5. Fuzzing process for generating malicious PtH network data.

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

DNS cache poisoning attack can target local DNS servers. When the local DNS server receives a query which it does not have the corresponding records (first stage), it will inquire the global DNS server (second stage). On receiving the response (third stage), the local DNS server will save this record in its cache, and forward the response to the user machine (fourth stage). However, the DNS server cannot verify the response at the second stage, and this is where the attacker can fool the local DNS server. Pretending as the global DNS server, the attacker can send a spoofed DNS response to the local DNS server with the falsified DNS record. As long as the fake response arrives earlier than the real one, the local DNS server will forward it to the user machine and save the falsified record to its cache, as illustrated in Fig. 6. When new queries about the same domain name come in, the local DNS server will not send a query to the global DNS server again because the corresponding record has been cached. Consequently, it will answer the user machine with the falsified record, until the record expires or the cache is flushed.

Data Generation for DNS Cache Poisoning. For this attack, ten fields, such as `time to live` values in different layers and resource records, are fuzzed in `IP` and `DNS` layers. The testbed contains three machines: a local DNS server whose DNS cache is flushed periodically, a user machine which sends out DNS queries to the local DNS server periodically, and an attacker machine which sniffs for DNS requests sent from the local DNS server and answer them with spoofed responses in the attack scenario, and does nothing in the benign scenario.

In the malicious scenario, we let the user machine to ask for the IP address of one specific domain name from the local DNS server using command *dig*. The domain name is one that does not have a

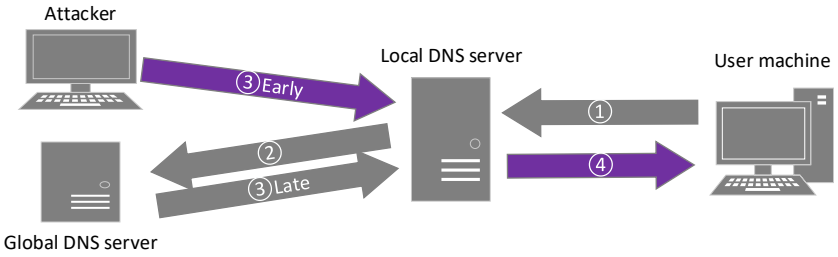


Fig. 6. DNS cache poisoning illustration.

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

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

The specific domain name used in the malicious scenario and the domain names used in the benign scenario do not overlap. Both the domain names and the IP addresses (falsified or genuine) are excluded during neural network training, so that the neural network will not learn the specific domain name or falsified IP address, which can be treated as signatures by the neural network. Details are provided in subsection 5.3.

5. Detection Models

In this section, we will discuss the network attacks' detection models trained with our generated data. Specifically, we will discuss how the data is processed before fed to the model, the model used, and some training details.

5.1. ARP Poisoning Detection

ARP is a stateless protocol on top of User Datagram Protocol (UDP), and the attack can be accomplished with an individual packet. Therefore, for detecting ARP poisoning attacks, every data sample represents one network packet. In addition, ARP packets have very simple packet structure, and it is not clear which fields are critical to detect this attack. For both the malicious and benign data, all the collected ARP packets' length is either 42 bytes or 60 bytes. For the 60-byte packets, the last 18 bytes are all zeros, which means that the meaningful data is only the first 42 bytes. Therefore, we select the first

Table 2
Fields of interest.

Layer	Fields	Size in bytes	Explanation
ETH	Dst_MAC	6	Destination MAC address
	Src_MAC	6	Source MAC address
	ETH_type	2	Indicate which protocol is encapsulated in the payload of the frame and is used at the receiving end by the data link layer to determine how the payload is processe
ARP	HTYPE	2	Network link protocol type. For Ethernet, this field is 1.
	PTYPE	2	Internetwork protocol for which the ARP request is intended. For IPv4, this has the value 0x0800.
	HLEN	1	Length of a hardware address. For Ethernet addresses, the length is 6.
	PLEN	1	Length of internetwork addresses. The internetwork protocol is specified in PTYPE. For IPv4 addresses, the length is 4.
	OpCode	2	Specifies the operation that the sender is performing: 1 for request, 2 for reply.
	SHA	6	Source hardware address. The MAC address of packet sender.
	SPA	4	Source internetwork address.
	THA	6	Target hardware address. The MAC address of packet target.
	TPA	4	Target internetwork address.
IP	Version	4/8	For IPv4, this is always equal to 4.
	IHL	4/8	Internet header length.
	DSF	1	Differentiated service field, which includes differentiated services code point and explicit congestion notification.
	TLen	2	The entire packet size in bytes.
	ID	2	An identification field which is primarily used for uniquely identifying the group of fragments of a single IP datagram.
	Flags	3/8	3 bits for controling or identifying fragments.
	FragOff	13/8	13 bits for specifying the offset of a particular fragment relative to the beginning of the original unfragmented IP datagram.
	TTL	1	Time to live field, which limits a datagram's lifetime.
	prot	1	This field defines the protocol used in the data portion of the IP datagram.
	chksum	2	Header checksum for error-checking of the header.
	src_add	4	Source IPv4 address.
	dst_add	4	Destination IPv4 address.
TCP	src_port	2	Source port number.
	dst_port	2	Destination port number.
	seq	4	Sequence number.
	ack	4	Acknowledgement number.
	hd_len	4/8	Size of the TCP header.
	flags	12/8	The first 3 bits are reserved, and the other 9 bits are control bits.
	window	2	Size of the receive window.
	chksum	2	Checksum field for error-checking of the TCP header.
	urgptr	2	Urgent pointer.
UDP	src_port	2	Source port number.
	dst_port	2	Destination port number.
	hd_len	2	The length in bytes of the UDP header and UDP data.
	chksum	2	Checksum field for error-checking of the UDP header and UDP data.
DNS	TID	2	Transaction ID for synchronization between DNS servers/clients.
	flags	2	Control flags
	q	2	The number of entries in the question section.
	AnRR	2	The number of resource records in the answer section.
	AuRR	2	The number of resource records in the authoritative section.
	AdRR	2	The number of resource records in the additional section.
SMB/SMB2	cmd	2	The command code of this packet.
	flags	4	Indicate how to process the operation
	NT_status	4	Status or error code.

42 bytes as input data (ETH and ARP layers' fields in Table 2). Every byte is treated as a number when fed to the neural network. As a result, each data sample is an integer list converted from one packet's bytes. The labeling is done towards each data sample, which is the 42 bytes generated from each ARP packet.

The neural network used for detecting ARP poisoning attack is simple with an input layer, a hidden layer, and an output layer with softmax activation function. We have tried four types of hidden layers (fully-connected layer, convolutional layer, recurrent layer, and LSTM layer). Thus, four types of neural networks are trained, namely multi-layer perceptron (MLP) model, convolutional neural network (CNN) model, recurrent neural network (RNN) model, and LSTM model. For each type of neural network, we have tried different numbers of units ($N\_hidden \in \{15, 20, 25, 30, 35\}$), and for CNN model, we

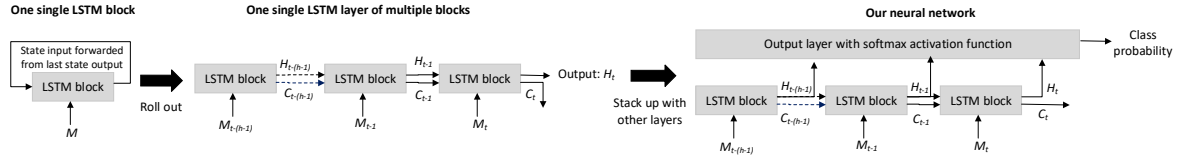


Fig. 7. LSTM neural network for PtH detection.

have also tried different kernel sizes ($kernel\_size \in \{2, 3, 4\}$) and different strides ($stride \in \{1, 2\}$). Therefore, a total of 45 models are trained.

5.2. PtH Detection

To detect PtH attack with neural networks, we have two key insights that help determine the representation of data samples: 1) Network communication for authentication is actually a sequence of different types of network packets in a certain order. An earlier packet can affect the packet afterwards. For example, the first several packets between a server and a client may be used to communicate and determine which protocol to use (e.g. SMB or SMB2), and packets afterwards will use the decided protocol. The attack is to get authenticated by the server, which requires a sequence of packets to accomplish. Therefore, each data sample should be a sequence of packets, rather than an individual packet. 2) PtH relies on authentication mechanisms that legitimate users usually don't use. The network packets for the benign and malicious authentication are different. Since both authentication methods use SMB/SMB2 packets, the differences between them thus exist in the fields of the SMB/SMB2 layer. Therefore, data in `SMB/SMB2` layer is used for PtH detection. In addition, the differences of field values between benign and malicious authentication will be helpful to distinguish them.

For this attack, we choose Long-short term memory (LSTM) as the architecture for the neural network. For each packet, we assign a type number to it to represent this packet as a whole. The packet types are defined by fields of interest (SMB/SMB2 layer's fields in Table 2) in the packets. If two packets have the same values in all those fields, then the two packets are given the same packet type number. Otherwise, different numbers are assigned. Please note that we only care about the differences of field values between two packets, but do not care about the values themselves. That's why we are using packet type numbers to represent the packets, rather than examining the content of each packet. The LSTM neural network takes a sequence of network packets' type numbers and outputs the binary label representing whether this sequence is PtH network traffic. Our neural network is presented in Fig. 7. The neural network consists of one LSTM layer, which contains multiple LSTM blocks to take inputs, and several fully connected layers, which take inputs from the LSTM layer and produce the final binary output. *M* stands for input; *H* stands for each LSTM block's output; and *C* stands for each LSTM block's state output. Subscripts stand for the time points, in which *h* stands for the window size. The output layer uses softmax activation function, so the raw outputs are the probabilities for the data sample to be benign or malicious, which add up to 1.

Now that each network packet is represented as a packet type number, the network log can be represented as a sequence of packet type numbers. The next step is to create data samples needed for neural network training. We complete this by the following steps: 1) We chop the sequence of packet type numbers to smaller sequences based on the unit of network communications. By identifying the start packet for each benign/malicious network communication, we chop the whole sequence into many *variate-length* sequences, and the beginning of every sequence is a start packet. 2) We further chop the smaller

variate-length sequences into one or more *fixed-length* sequences according to the window size and step size. The window size decides how many network packets are represented in one data sample, or, in another word, how many packet type numbers are included. The step size decides the shifting step when chopping large sequence. For example, if a sequence $[1, 2, 3, 4, 5, 6, 7, 8]$ is chomped with window size 4, and step size 2, then the resulting small sequences are $[1, 2, 3, 4]$, $[3, 4, 5, 6]$, and $[5, 6, 7, 8]$. 3) Depending on whether the sequence is from a benign traffic or malicious traffic, we assign a *binary label* to it to indicate whether this sequence is benign or malicious. This sequence becomes one data sample for our LSTM neural network. 4) If a fixed-length sequence appear in both the benign and malicious data, this sequence is removed from both of them because it cannot help with the classification. 5) We remove duplicate sequences in both the benign data and malicious data. After these two removal processes, all data samples are finally ready to be used.

We tried different data samples (windows size $w\_size \in \{4, 8, 12, 16\}$ and step size $st \in \{1, 2, 4, 6\}$) and different number of LSTM units ($N\_LSTM \in \{10, 20, 30\}$) in the LSTM layer. The values of window size and shift step size can affect the number of data samples, so different neural networks may be trained and evaluated with data sets of different sizes. Because it is unknown that which combination will achieve the best results, all combinations, a total of $4 * 4 * 3 = 48$ models, are tried.

5.3. DNS Cache Poisoning Detection

Network packets from DNS cache poisoning attack form sessions which consist of queries and answers. Therefore, each data sample should include data from multiple network packets. In addition, it is not clear which fields may be of importance, so we need to investigate the packet content, rather than simply generalizing the packets with packet types as we did in PtH detection.

Instead of using the whole packet, we use 40 bytes, from byte 15 to 54, of every packet and convert them to integers from 0 to 255 respectively. The first 14 bytes belong to the lowest `ETH` layer. They are excluded purposefully to rule out the impact of MAC addresses: the MAC address may be treated as signatures to detect malicious packets. The chosen 40 bytes include bytes in `IP` layer, `UDP` layer, and part of `DNS` layer (IP, UDP, and DNS layers' fields in Table 2). Only part of `DNS` layer data is used because the query and records are excluded. Those sections contain the domain names and IP addresses, which are excluded for similar reasons as above: we don't want the neural network to learn the "malicious" domain. After the packet data processing, every packet is represented as a fixed-length sequence of 40 integer numbers ranging from 0 to 255. The whole captured network traffic is represented as a sequence of 40-integer sequences.

The packet processing is different from the process in PtH detection. 1) In PtH detection, each component in a sequence is a packet type number, while in DNS cache poisoning attack detection, each component is one 40-integer sequence. 2) The resulting data sample is also different. In DNS cache poisoning attack, the data sample is a matrix of size $k * 40$, which represents k packets and 40 bytes in each packet. Each integer in the matrix ranges from 0 to 255. Therefore, the matrix can also be viewed as a grayscale image of size $k * 40$, and each integer at row i column j is the grayscale value of that pixel, corresponding to the jth selected byte in the ith packet.

Now that the data samples can be represented as images, we use a convolutional neural network (CNN) to do the classifications, which has been proven to work well in image classification problems. The labeling is done towards each data sample, which is the entire matrix, rather than an individual packet. During data processing, the malicious and benign data are processed separately. Matrices generated from malicious data are labeled as malicious, and matrices generated from benign data are labeled as benign.

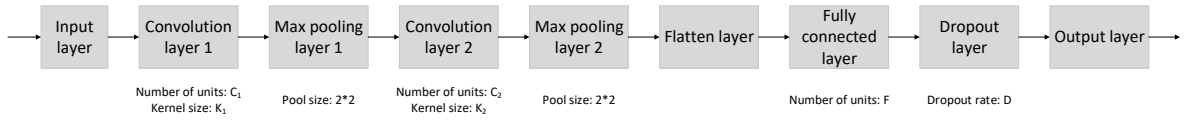


Fig. 8. The neural network structure for DNS cache poisoning detection.

Similar to PtH detection, we have trained a series of neural networks with different neural network hyper-parameters and data samples of different window sizes and window steps. That means we can adjust the number of packets k included in each data sample and thus change the size of matrix. The neural network structure is shown in Fig. 8. The output uses softmax activation function as well. Window size ($w\_size \in \{4, 6, 8, 10, 12\}$) and step size ($st \in \{1, 2, 4, 6, 8\}$) will affect the data samples. Besides, there are three hyper-parameters when constructing the detection neural networks, which are the number of units in some hidden layers (convolutional layer 1, 2 and fully connected layer, $N\_hidden \in \{(C_1 = 16, C_2 = 16, F = 8), (C_1 = 32, C_2 = 16, F = 8), (C_1 = 32, C_2 = 32, F = 16), (C_1 = 32, C_2 = 32, F = 8), (C_1 = 64, C_2 = 32, F = 16), (C_1 = 64, C_2 = 64, F = 16)\}$), the kernel size ($K\_size \in \{(K_1 = 2 * 2, K_2 = 2 * 2), (K_1 = 3 * 3, K_2 = 3 * 3), (K_1 = 4 * 4, K_2 = 4 * 4)\}$) in each CNN layer, and the dropout rate ($D \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$) in the dropout layer. Again, since it is unknown that which values will result in the best results beforehand, so we tried all combinations, and a total of $5 * 5 * 6 * 3 * 5 = 2250$ models with different combination of values are trained.

5.4. Model Selection

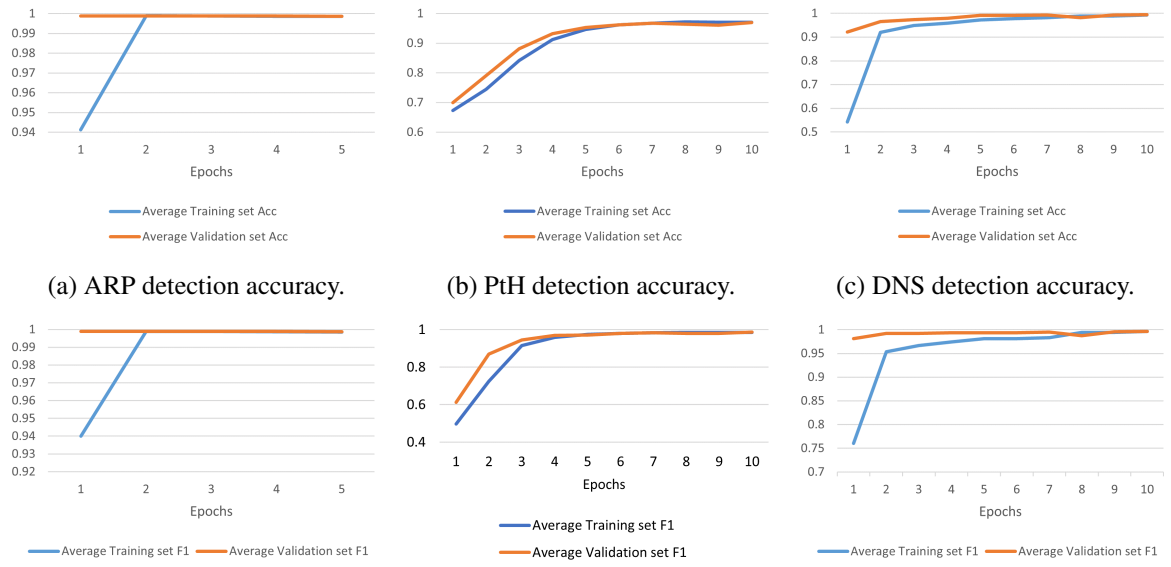
Before model selection, we perform *parameter optimization* for single-packet attacks, and we perform *data sample and parameter optimization* for multi-packet attacks. Data sample optimization means different data samples with different window size and/or step size are tried to select the data samples generating the best results, while parameter optimization means different model parameters are tried.

For model selection, we use metrics including accuracy (*Acc*), F1 score (*F1*), detection rate (*DR*) and false positive rate (*FPR*). Assuming the numbers of true positives, true negatives, false positives and false negatives are presented as TP, TN, FP, FN, respectively, then $Acc = (TP + TN)/(TP + TN + FP + FN)$, $F1 = TP/(TP+0.5*(FP+FN))$, $DR = TP/(TP+FN)$, and $FPR = FP/(TN+FP)$. *Acc* and *F1* are two of the commonly used metrics used for classification problems, and *DR* and *FPR* are important for intrusion detection. *DR* is defined by the ratio of detected attacks number to the total number of attacks, so *DR* directly shows the detector's ability of detecting attacks. *FPR* is defined by the ratio of number of attacks mis-classified as benign samples to the total number of benign samples, so it directly shows how likely the detector will raise false alarms, interrupting the normal works of organizations and enterprises. We call the best-performing model as the one that gets the highest average of *Acc* and *F1*, denoted as

$$P = \frac{Acc + F1}{2},$$

and the best-detecting model as the one that gets the highest average of *DR* and $1 - FPR$, denoted as

$$D = \frac{DR + 1 - FPR}{2}.$$



(a) ARP detection accuracy. (b) PtH detection accuracy. (c) DNS detection accuracy.

(d) ARP detection F1 score. (e) PtH detection F1 score. (f) DNS detection F1 score.

Fig. 9. Average accuracies and F1 scores of the best-performing models on the training and validation set with respect to the number of epochs. We limit the number of epochs so that validation results do not downgrade too much.

If FPR cannot be calculated (no benign data sample), we let $D = DR$. We simply take the average because all the chosen metrics are important for evaluations and we have no preference on any of them.

The generated fuzzing data set is randomly split into two parts: 80% as the training set, and 20% as the test set. The training set is then further randomly split into four parts of about the same size, upon which 4-fold cross-validation is employed to avoid over-fitting. All the reported results are the average results among four folds. The best-performing and best-detecting models are selected based on the average P and D results on the validation set across all four folds.

6. Evaluations

In this section, we will provide evaluation results on the selected best-performing and best-detecting models. Besides the training set and test set, which are split from the fuzzing set, we also prepare another malicious set without fuzzing, referred to as the real attack set. All the deep learning experiments are done on a Windows 10 machine with Intel Core i7-7820HK, 32 GB of RAM, and a GTX 1080 Mobile graphics card for acceleration. As for the software, we use Python 3.8.5 and TensorFlow 2.4.1.

For comparison, we have trained traditional machine learning models, including k-nearest neighbor (kNN) models, support vector machine (SVM) models with various kernels, decision tree (DT) models, and random forest (RF) models. For ARP poisoning, we have also added one recent work [42] using deep learning for detection. They are trained, selected, and evaluated on the same data sets. We use R 3.6.1 and several packages for different types of models, including *caret* for kNN models, *e1071* for SVM models, *rpart* for DT models, and *randomForest* for RF models.

The fields and the bytes are defined by the protocol, as listed in Table 2, but the term “feature” has slightly different meanings in traditional machine learning and deep learning. Because we want to pro-

vide a unified view of both deep learning and traditional machine learning, we hereby define a feature as the smallest component in the model input. For PtH and single-packet attacks (ARP poisoning), the traditional machine learning models' data samples and features are the same as those for deep learning models. However, for DNS cache poisoning, the same data sample and feature cannot be used because the input space is too large for traditional machine learning models to handle. Therefore, we employ principal component analysis (PCA) for dimension reduction, and only select the top-rated one-fifth PCA features. On average, they can explain about 97.09% of the original data.

The parameters in deep learning models have been discussed earlier. For kNN models, we optimize `k`, the number of neighbors; for SVM model with linear kernel (SVM-Linear), we optimize `C`, the cost constant; for SVM model with polynomial kernel (SVM-Poly), we optimize `degree`, the polynomial degree, `scale`, the scaling factor constant, and `C`, the cost constant; for SVM model with radial basis kernel (SVM-Radial), we optimize `sigma`, the precision parameter for the radial basis function, and `C`, the cost constant; for DT models, we optimize `cp`, the complexity parameter; for RF models, we optimize `mtry`, the number of randomly selected features for each tree in the forest.

6.1. Data Set Description

Table 3 shows the data set statistics. The data set contains fuzzed set (split into training set and test set) and non-fuzzed set (real attack set). A data set that is balanced and large enough in size is essential for training the models effectively. Lack of training data can result in poor results, while biased data sets may result in biased models. Although we have collected similar amounts of raw benign and malicious data in section 4, the number of benign and malicious data samples still varies after processing in section 5. As a result, the data sets after processing are usually unbalanced. To mitigate such unbalancing, a common practice is to down-sample the one with more data samples, but this will reduce the size of data set. To deal with this dilemma, we loosen the criterion of "a balanced data set": as long as the benign to malicious ratio is within the range of $[\frac{2}{3}, \frac{3}{2}]$, then there is no need for data set balancing. (This means that both the benign and malicious take up more than 40% but less than 60% of whole data, so the data set as a whole is adequately balanced.) Otherwise, we perform data set balancing with down-sampling. Specifically, if the benign data sets have significantly more data samples than the malicious data sets, we down-sample the benign data sets to match the size of malicious data sets, and vice versa.

6.2. Best-performing Models

Table 4 presents the evaluation results on the best-performing models for each network attack. Rows highlighted with yellow color indicate the best-performing deep learning models, and rows highlighted with green color indicate the best-performing machine learning models. All models get acceptable to good results on training set and test set. *For multi-packet attacks, deep learning models are substantially better than traditional machine learning models, especially on real attack set.* In PtH detection, the LSTM model achieves near 99% accuracy on the real attack set, while machine learning models cannot reach $1/4$ accuracy. In DNS cache poisoning detection, the CNN model's accuracy on the real attack set is 100%, while machine learning model can reach about 47% accuracy at most. Selected deep learning models' F1 scores are also far better than those of traditional machine learning models. For ARP poisoning detection, deep learning models do not have many advantages over traditional machine learning models, and all models' performances downgrade on real attack set comparing to those of training set and test set. The reason is that the real attack set for ARP poisoning is generated on a different LAN, with different valid MAC and IP addresses.

Table 3
Data set statistics.

Attacks	Set	Size	Benign to malicious ratio
ARP poisoning	Training	9584	1.005:1
	Test	2400	0.982:1
	Real attack	17471	0:1
PtH* (best-performing)	Training	3932	1.364:1
	Test	983	1.329:1
	Real attack	214	0:1
PtH* (best-detecting)	Training	2556	0.974:1
	Test	640	0.839:1
	Real attack	192	0:1
DNS cache poisoning*	Training	30928	1.003:1
	Test	7732	0.988:1
	Real attack	263	0:1

* For multi-packet attacks, we only list the data set statistics corresponding to the best-performing or best-detecting models.

(a) ARP detection rates. (b) PtH detection rates. (c) DNS detection rates.

(d) ARP false positive rates. (e) PtH false positive rates. (f) DNS false positive rates.

Fig. 10. Evaluation results on the best-detecting models.

6.3. *Best-detecting Models*

Fig. 10 presents the evaluation results of Best-detecting models. Results are shown as bar charts so that it is easy to compare between different model types and across different data sets. FPRs on real attack sets are not presented because there is no negative data sample, so FPR cannot be calculated. Except for LSTM model in PtH detection, in all other cases, the best-detecting model is the same as

Table 4
Evaluation results on best-performing models.

Attacks[1]	DL or ML[2]	Model type[3]	Data sample[4]	Model setting[5]	Training set		Test set		Real attack set	
					Acc	F1	Acc	F1	Acc	F1
ARP	DL	MLP	NA	hidden_units=15	99.91%	0.9991	99.75%	0.9975	72.84%	0.8429
		CNN		hidden_units=35	99.94%	0.9994	99.79%	0.9979	73.02%	0.8441
		RNN		hidden_units=15	99.91%	0.9991	99.75%	0.9975	72.83%	0.8428
		LSTM		hidden_units=15	99.91%	0.9991	99.75%	0.9975	72.83%	0.8428
	ML	kNN		k=3	99.90%	0.9990	99.93%	0.9993	81.99%	0.9010
		SVM-Linear		C=1	99.87%	0.9987	99.90%	0.9990	72.83%	0.8428
		SVM-Poly		C=10, degree=3, scale=0.01	99.96%	0.9996	99.93%	0.9993	72.83%	0.8428
		SVM-Radial		C=1, sigma=0.04	99.97%	0.9997	99.93%	0.9993	72.83%	0.8428
		DT		cp=0.45	99.84%	0.9984	99.90%	0.9990	82.35%	0.9032
		RF		mtry=4	99.97%	0.9997	99.93%	0.9993	72.83%	0.8428
PtH	DL	LSTM-P	WS=16, SS=1	hidden_units=30	98.45%	0.9865	98.07%	0.9831	98.96%	0.9948
	ML	kNN	WS=12, SS=2	k=3	96.77%	0.9682	96.53%	0.9658	23.44%	0.3797
		SVM-Linear	WS=12, SS=2	C=10	96.89%	0.9694	96.72%	0.9674	13.02%	0.2304
		SVM-Poly	WS=8, SS=4	C=10, degree=2, scale=0.01	97.75%	0.9779	94.69%	0.9479	23.44%	0.3797
		SVM-Radial	WS=8, SS=4	C=10, sigma=0.0115	98.07%	0.9810	93.72%	0.9378	18.23%	0.3084
		DT	WS=12, SS=2	cp=0.05	94.70%	0.9467	95.44%	0.9533	18.23%	0.3084
		RF	WS=12, SS=2	mtry=40	100.00%	1.0000	97.99%	0.9798	14.06%	0.2466
DNS	DL	CNN	WS=12, SS=1	$C_1 = 32, C_2 = 32, F = 16, K_1 = (2,2), K_2 = (2,2), D = 0.1$	99.87%	0.9987	99.73%	0.9973	100.00%	1.0000
	ML	kNN	WS=4, SS=1	k=10	98.67%	0.9867	98.35%	0.9834	0.00%	0.0000
		SVM-Linear	WS=12, SS=6	C=0.1	96.01%	0.9608	95.17%	0.9527	0.00%	0.0000
		SVM-Poly	WS=12, SS=7	C=0.1, degree=3, scale=0.02	99.63%	0.9963	98.70%	0.9870	0.00%	0.0000
		SVM-Radial	WS=12, SS=8	C=10, sigma=0.1647	100.00%	1.0000	98.66%	0.9867	0.00%	0.0000
		DT	WS=12, SS=9	cp=0.05	87.01%	0.8771	86.88%	0.8754	47.01%	0.6395
		RF	WS=12, SS=10	mtry=21	100.00%	1.0000	97.50%	0.9752	34.19%	0.5096

[1] The three attacks are ARP poisoning, pass the hash, and DNS cache poisoning, respectively.
[2] DL stands for deep learning, and ML stands for traditional machine learning.
[3] For multi-packet attacks, only the deep learning models proposed in section 5 are presented.
[4] NA stands for not available, because there is no different data samples for single-packet attack. WS stands for window size, and SS stands for step size.
[5] Explanation for the DL model settings are provided in section 5. ML model settings are explained at the beginning of section 6

best-performing model. The best detecting model for PtH detection has the window size of 16, the step size of 2, and 30 hidden units in the LSTM layer.

It can be inferred from the figure that, similar to the best-performing case, all models get acceptable to good results on training set and test set. For single-packet attack detections, deep learning models do not have many advantages over traditional machine learning models. *For multi-packet attacks, deep learning models have clear advantages over traditional machine learning models, especially on real attack set.* Because there is no negative data sample in the real attack set, $DR = Acc$, which has been discussed in the previous subsection. As for FPRs, though they cannot be calculated in the real attack set, it can be seen that deep learning models achieve generally lower FPRs comparing to traditional machine learning models on the training and test sets.

6.4. Impact of Data Representation

We also studied the impact of data representation on model training. For PtH attack, the change of data representation can indeed help with detection. To prove this, we created two data sets: data set (a) based on network packet type numbers, and data set (b) based on the raw bytes converted from the

corresponding network packets' selected field values. We compared the models trained with (a) and (b) respectively. With (b), we trained additional neural networks of the following four types: MLP, CNN, RNN, and LSTM. All these models have about the same complexity (hyper-parameters like number of layers and number of units in each layer) as PtH detection LSTM models (discussed in the previous subsection) trained on (a). Our results show that the best models of MLP and RNN trained on (b) only reach around 80% accuracies. The best models of CNN and LSTM trained on (b) reach around 97% accuracies. Referring back to Table 4, it is clear that none of them is as good as the LSTM model trained on (a). Therefore, the packet type number representation method is better in PtH detection compared to the raw byte representation method.

For ARP poisoning and DNS cache poisoning, they do not have obvious fields that may benefit from change of data representation. Therefore, we retain as much raw information as possible from the network packets, and keep the data representations close to the raw packets.

6.5. Impact of Probability Threshold

The raw outputs of the output layer of each detection neural networks are the probabilities for the input data sample to be benign or malicious, which add up to 1. The raw outputs can be converted to classification results. If the probability for malicious class is beyond a given threshold, the data sample will be classified as malicious. Otherwise, it will be classified as benign. When the threshold gets larger, the model is more likely to classify a data sample as benign: as a result, the false positive rate will decrease, but the false negative will increase. Hence, different thresholds correspond to different FPR and DR (or true positive rate) pairs

In order to understand the trade-offs between false positive rates and false negative rates, we examine the impact of probability thresholds in this experiment. We select the best-performing deep learning models for detecting each network attack, and re-run the evaluation experiments with 9 different probability thresholds (i.e., 0.1, 0.2, . . . , 0.9). The false positive rate and detection rate pairs for each threshold are shown in Table 5. It should be noticed that each pair of (FPR, DR) is actually a point in the corresponding ROC curve. (We do not show the plotted ROC curves since majority of the thresholds result in the same pair of false positive rate and false negative rate values.)

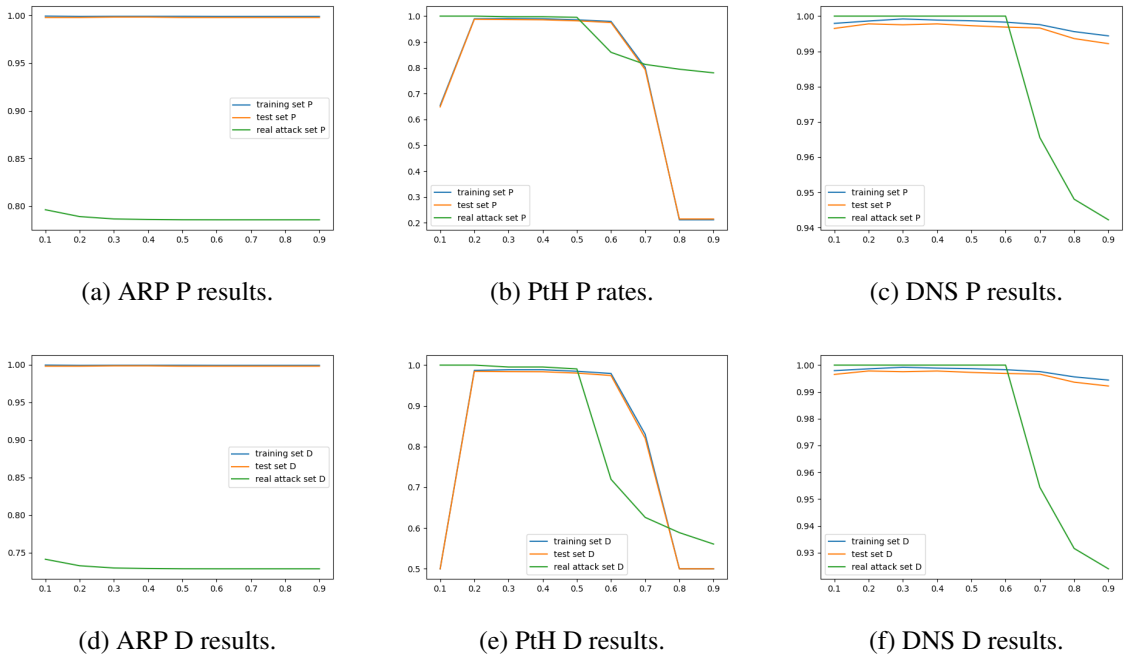
In addition, the P and D results are shown in Fig. 11. It can be inferred that the best probability threshold is not always 0.5. For ARP poisoning detection, the best threshold is 0.1. For PtH and DNS cache poisoning detections, there are multiple threshold values besides 0.5 that have almost the same performances. Depending on whether the defender prefers higher detection rates or lower false positive rates, changing the probability threshold is a worthy option.

How to choose the probability threshold: For instance, let's assume that $p_{b1}, p_{b2}, p_{b3}, \ldots, p_{bn}$ are the probabilities of being malicious for the n benign data samples in the data set, respectively, and $p_{m1}, p_{m2}, p_{m3}, \ldots, p_{mk}$ are those for the k malicious data samples, respectively. If $\max_{i\in\{1,2,\ldots,n\}}(p_{bi}) < \min_{i\in\{1,2,\ldots,k\}}(p_{mi})$, then the probability threshold can be easily set to $\max_{i\in\{1,2,\ldots,n\}}(p_{bi})$, so that the model will not misclassify anything. However, if $\max_{i\in\{1,2,\ldots,n\}}(p_{bi}) > \min_{i\in\{1,2,\ldots,k\}}(p_{mi})$, then there is a trade-off: there is no such a threshold to perfectly separate benign from malicious. If the defender wants the model to be sensitive to attacks and can tolerate some false alarms, he/she can set the threshold to be $\min_{i\in\{1,2,\ldots,k\}}(p_{mi})$; or, if the defender wants a smoother business process with fewer false alarms and only uses the detection model as one of the intrusion indicators, he/she can set the threshold to be $\max_{i\in\{1,2,\ldots,n\}}(p_{bi})$. Also, the defender can come up with a unified metric, like the P and D values mentioned earlier in this paper, and draw curves as in Fig. 11 to find out the best probability threshold.

Table 5
Receiver operating characteristic curve (ROC) points for the selected best-performing DL models.

	ARP poisoning				PtH				DNS cache poisoning			
	Train		Test		Train		Test		Train		Test	
Th. *	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR
1	0	0	0	0	0	0	0	0	0	0	0	0
0.9	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0001	0.9889	0.0000	0.9844
0.8	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0001	0.9913	0.0003	0.9875
0.7	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0001	0.9952	0.0003	0.9935
0.6	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0001	0.9967	0.0005	0.9943
0.5	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0003	0.9976	0.0008	0.9953
0.4	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0006	0.9983	0.0010	0.9966
0.3	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0010	0.9994	0.0021	0.9971
0.2	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0027	0.9999	0.0036	0.9992
0.1	0.0004	0.9985	0.0017	0.9959	0.0222	0.9929	0.0261	0.9947	0.0041	1.0000	0.0067	0.9997
0	1	1	1	1	1	1	1	1	1	1	1	1

* Thresholds. When the malicious probability is above the threshold, the data sample is classified as malicious.



(a) ARP P results. (b) PtH P rates. (c) DNS P results.

(d) ARP D results. (e) PtH D results. (f) DNS D results.

Fig. 11. Probability threshold's impact on models' P and D results.

7. Interpreting Why the Detection Models Achieved High Detection Accuracy

The results presented in the previous section show that the trained (network attack detection) models can achieve high detection accuracy. In this section, we would like to provide a few domain-specific interpretations on this observation. In particular, we seek to answer two research questions: (1) Do

fuzzed packet fields indeed influence model accuracy? (2) Does the protocol fuzzing approach has the ability to generate comprehensive data?

7.1. Influence of Fuzzed Packet Fields on Detection Accuracy

One of the problems we are curious about is how influential the fuzzed fields are to the accuracy of the trained neural networks. To answer this question, we have leveraged *permutation feature importance* [43], a widely-adopted procedure to make machine learning more interpretable, to measure the increase or decrease in the accuracy of the model after we permute a particular feature's values. We obtained the permutation feature importance measurements for ARP poisoning. For DNS cache poisoning, we visualized the learned features in the first convolutional layer, which provide insights about which pixels of the "image" are emphasized.

Permutation feature importance is a model inspection technique that can be used for fitted models. It is defined as the decrease in a model score, which is the F1 score in our experiments, when a single feature value is randomly shuffled [43]. This technique benefits from being model agnostic and can be calculated many times with different permutations of the feature, and is often used to inspect how influential some features are to the model in various deep learning applications [44–46]. The drawback is that, when several features are correlated, the measurement becomes inaccurate. Our results show that, for ARP poisoning, the top three most influential features all belong to fuzzed fields, which are `destination MAC address`, `target MAC address`, and `target IP address`.

In summary, our experiments show that the 3 most influential features for each of the trained models usually involve one or more fuzzed packet fields. However, for the CNN for DNS cache poisoning identification, a fuzzed field, `time to live` value in the `IP` layer, is one of the fields that are least influential. Although our experiments did not find any fuzzed field as the most influential field on the first layer (of the model) in detecting DNS cache poisoning, it's still possible that a fuzzed field is most influential on a deeper layer. Deep learning models are still mostly "black boxes". Furthermore, not all fuzzed fields may be presented in the data samples. For example, for PtH, the fields used in the mapping (and embedding) do not concern any fuzzed fields. Another possibility is that, though some fuzzed fields' influences are not directly reflected from our measurements above, the fuzzed fields may affect other fields, which are in turn influential to the models.

7.2. Can Protocol Fuzzing Generate Comprehensive Training Data?

A comprehensive training data set is a key factor in many successful applications (e.g., image classification) of deep learning. However, how to formally assess the comprehensiveness of the training data in deep-learning-based detection of network attacks is an issue largely neglected in the literature. In this subsection, we seek to take the first step towards systematically addressing this issue. In particular, we introduce a new notion, called **"covered-by"**, to capture an inherent connection between protocol fuzzing and training-data comprehensiveness.

The intuition behind this notion is as follows. First, a network packet is essentially a set of fields. Although real-world packets are usually not byte-to-byte identical to the packets generated through protocol fuzzing, the content in a particular field of a real-world packet may be identical to the corresponding field of a fuzzed packet. Hence, it seems appropriate to do field-content-based data comprehensiveness assessment. Second, one major advantage of deep learning is that it requires less feature engineering than conventional machine learning techniques (e.g., Support Vector Machines). Accordingly, although

a data sample in the training set involves the information from multiple fields, the deep learning algorithm typically treats each element (e.g., a byte or a pixel) in the data sample in a unified way, and does not pre-extract semantic-carrying features. Based on this observation, we do not assess data comprehensiveness based on a space of semantic-carrying features.

Based on the above intuition, we define the notion of "covered-by" as follows: 1) When a real-world network packet is being "examined" by a deep learning model, it will need to be firstly transformed to a data sample *a* (or a portion of *a*) expected by the model. An element in the data sample is *covered-by* a data sample *b* generated through the proposed protocol fuzzing method if A) the element's content is identical to the corresponding element in *b*, B) the packet field used to derive the element in *a* has the same content as the field used to derive the corresponding element in *b*, and C) the above-mentioned packet field is a being-fuzzed field in the proposed protocol fuzzing. Note that since the definition intends to capture an inherent connection between protocol fuzzing and data comprehensiveness, it ignores the elements that are not derived from a being-fuzzed field. 2) Given a set of data samples S_1 (generated through real-world network packets) and a training set of data samples S_2 generated through the proposed protocol fuzzing method, the *coverage rate* is defined as the ratio of x to $(x+y)$. Here x is the total number of the elements in S_1 that are covered-by one or more data samples in S_2, and y is the total number of the elements in S_1 that are derived from a being-fuzzed field but not covered-by any data sample in S_2.

It should be noticed that coverage rate is not applicable to the deep learning algorithms that employ a mapping from an entire network packet to a type assigned with a particular type number. Accordingly, because such a mapping is done when detecting PtH, we should not use coverage rate to measure the comprehensiveness of the training data generated for detecting PtH.

Measuring the coverage rates. To measure the coverage rates for the three network attacks, we have collected additional non-fuzzed data and measured the coverage rates with respect to the fuzzed data collected in Section 4. The non-fuzzed data are collected by launching each kind of network attack with different parameters (e.g., IP addresses, MAC addresses, process names, etc.) for hundreds of times so that data collected in different attack sessions are not identical to each other.

We have measured the coverage rates for ARP poisoning and DNS cache poisoning. Our results show that, for all the three network attacks, the coverage rates are all 100%. In addition, we also assess the comprehensiveness of the training data generated for detecting PtH in an ad hoc way, and found that for each being-fuzzed field, every value observed in the non-fuzzed network packets appeared at least once in the fuzzed data. In summary, our results show that a correlation between high detection accuracy and high coverage rates exists.

8. Discussions and Limitations

In previous sections, we have shown that our new approach, protocol fuzzing, is helpful in generating data on which neural networks can be trained. However, our approach still has some limitations.

Efficiency: Training a neural network requires a large amount of data samples. However, the number of data samples can be affected in many ways. For example, protocol fuzzing in nature cannot guarantee that all malicious/benign activities are successful. Although the fuzzed values are in a valid range, the network packets with fuzzed values may still get rejected by the server or trigger some unexpected circumstances, leading to an interrupted session. Those data are probably useless as discussed in section 4.3. Also, the removal of duplicate (same data in one class) and double-dipping (same data among

different classes) data samples will also affect the number of data samples. In a word, not all collected data can be used as data sample for neural network training.

Another factor that affects the efficiency is the time consumed by each benign/malicious activity. Except for some simple activities like MAC-IP address resolving with only several ARP packets, other complicated activities need time to get carried out, especially those containing hundreds or more network packets. Also, depending on the mechanism of packet processing, the client/server may also need more time before it can respond. For example, in PtH data generation, one successful attack contains 300 to 400 packets (and not all of them are usable to generate data sample), and some time intervals between adjacent packets can be as large as 0.5 second. In addition, in our experiments, we manually inserted idle time intervals. The purpose is to stabilize the activity. For example, in PtH experiment, after starting the exploitation, the foreground is put to sleep for 25 seconds. This time interval is reserved so that the exploitation can continue to run to reach a successful end. If this time interval is removed or too short, then the attack process is very likely to end at the middle of exploitation. In a word, each data generation iteration takes time to complete.

Because of the two points above, our data generating efficiency can still be improved. Take PtH as an example. We spent about 4 days running 5,000 attack iterations, of which 611 failed. The total amount of network packets captured is 497,956, of which 103,718 are SMB/SMB2 packets that may be used to generate data sample. However, the final number of data samples is only in the thousands, as shown in Table 3.

Neural networks for various network attacks: Though we have verified our idea on three chosen network attacks, we trained separate neural networks for different attacks and did not train a neural network to detect various network attacks. It is difficult to train such a neural network because different network attacks have different characteristics, which may need different data representations and neural network architectures, as shown in section 5.

Interpretability: In subsection 7.1, we have tried to interpret the trained neural networks in two widely-used ways. However, there are still other interpretation techniques we have not tried, which may show some further insights about the neural networks and the attacks themselves. Our future work may provide additional interpretations about the trained neural networks.

Use of large language models (LLMs): Recently, some pre-trained LLMs (e.g., BERT [47], RoBERTa [48], SpanBERT [49], etc.) have been released and proven to be effective not only in natural language processing [50] (the proposed usage), but also in other tasks like object detection [51]. On one hand, such models could be effective in network attacks detections, and network traffic data could be transformed and fed into LLMs for the purpose of network attack detections. On the other hand, we observe that no existing work has done such data transformation in the literature to detect network attacks. Nevertheless, because LLMs have good potential, we will investigate the potential of LLMs in detecting network attacks in future work.

9. Conclusion

This paper presents an end-to-end approach to detect the logic-flaw-exploiting network attacks using deep learning. The end-to-end approach begins with data generation and collection, and ends with attack detection with trained neural network. We address two major challenges in applying deep learning for logic-flaw-exploiting network attack detection: the generation of useful data sets and the training of appropriate neural network models. In this paper, we propose a protocol fuzzing-based approach for data

generation, and study how to evaluate and compare different deep learning architectures from perspectives of both the neural network performance and the detection effectiveness in security. We show the effectiveness of our approach with three specific demonstration attacks, including PtH, DNS cache poisoning, and ARP poisoning. We have generated high quality network traffic data using protocol fuzzing, trained neural networks with generated data, and evaluated the trained models from the perspective of both neural network performance and attack detection. Some of those results have been published in an earlier work of ours [21]. In addition, we have also tried interpreting our trained models and discussed the limitations of our experiments and approach. Our future works include improving efficiency, finding more fuzzable fields, trying more complicated attacks and interpreting methods, and so on.

Disclaimer

This paper is not subject to copyright in the United States. Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

Acknowledgment

This work was supported by NIST 60NANB20D180.

References

[1] S.M.e.a. Milajerdi, HOLMES: Real-time APT Detection through Correlationof Suspicious Information Flows, in: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019.

[2] Q.e.a. Zou, An Approach for Detection of Advanced Persistent Threat Attacks, *IEEE Computer Magazine* **53**(12) (2020), 92–96.

[3] Use Alternate Authentication Material: Pass the Hash, Sub-technique T1550.002 - Enterprise | MITRE ATT&CK®, 2021, [Online; accessed 3. Mar. 2021]. https://attack.mitre.org/versions/v8/techniques/T1550/002.

[4] P. Arote and K.V. Arya, Detection and prevention against ARP poisoning attack using modified ICMP and voting, in: *2015 International Conference on Computational Intelligence and Networks*, IEEE, 2015, pp. 136–141.

[5] S. Goswami, N. Hoque, D.K. Bhattacharyya and J. Kalita, An Unsupervised Method for Detection of XSS Attack., *IJ Network Security* **19**(5) (2017), 761–775.

[6] E. Hogan, J.R. Johnson and M. Halappanavar, Graph coarsening for path finding in cybersecurity graphs, in: *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, 2013, pp. 1–4.

[7] S. Kumar and S. Tapaswi, A centralized detection and prevention technique against ARP poisoning, in: *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, IEEE, 2012, pp. 259–264.

[8] H.-M.e.a. Sun, DepenDNS: Dependable mechanism against DNS cache poisoning, in: *International Conference on Cryptology and Network Security*, Springer, 2009, pp. 174–188.

[9] L.e.a. Yuan, DoX: A peer-to-peer antidote for DNS cache poisoning attacks, in: *2006 IEEE International Conference on Communications*, Vol. 5, IEEE, 2006, pp. 2345–2350.

[10] O. Faker and E. Dogdu, Intrusion detection using big data and deep learning techniques, in: *ACMSE 2019 - Proceedings of the 2019 ACM Southeast Conference*, 2019, pp. 86–93. https://dl.acm.org/citation.cfm?id=3314439.

[11] K. Millar, A. Cheng, H.G. Chew and C.C. Lim, Deep learning for classifying malicious network traffic, in: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 11154 LNAI, 2018, pp. 156–161. http://link.springer.com/10.1007/978-3-030-04503-6_15.

[12] IDS 2018 | Datasets | Research | Canadian Institute for Cybersecurity | UNB, 2020, [Accessed Jul 4 2020]. https://www.unb.ca/cic/datasets/ids-2018.html.

[13] L. Dhanabal and S. Shantharajah, A study on NSL-KDD dataset for intrusion detection system based on classification algorithms, *International Journal of Advanced Research in Computer and Communication Engineering* **4**(6) (2015), 446–452.
[14] N. Moustafa and J. 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 2015 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2015.
[15] B. Pfahringer, Winning the KDD99 classification cup: bagged boosting, *ACM SIGKDD Explorations Newsletter* **1**(2) (2000), 65–66.
[16] I. Sharafaldin, A.H. Lashkari and A.A. Ghorbani, Toward generating a new intrusion detection dataset and intrusion traffic characterization, in: *ICISSP 2018 - Proceedings of the 4th International Conference on Information Systems Security and Privacy*, Vol. 2018-Janua, 2018, pp. 108–116. https://www.scitepress.org/Papers/2018/66398/66398.pdf.
[17] C.M. Bishop et al., *Neural networks for pattern recognition*, Oxford university press, 1995.
[18] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, MIT press, 2016.
[19] D.E. Rumelhart, G.E. Hinton and R.J. Williams, Learning representations by back-propagating errors, *nature* **323**(6088) (1986), 533–536.
[20] S.T. Jan, Q. Hao, T. Hu, J. Pu, S. Oswal, G. Wang and B. Viswanath, Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation, in: *The 41st IEEE Symposium on Security and Privacy (IEEE SP)*, 2020.
[21] Q. Zou, A. Singhal, X. Sun and P. Liu, Deep Learning for Detecting Network Attacks: An End-to-End Approach, in: *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, 2021, pp. 221–234.
[22] C. Taylor, W. Harrison, A. Krings, N. Hanebutte and M. McQueen, Low-Level network attack recognition: a signature-based approach, *IEEE Proc. PDCS'2001* (2001), 570–574.
[23] S. Kaur and M. Singh, Automatic attack signature generation systems: A review, *IEEE Security & Privacy* **11**(6) (2013), 54–61.
[24] J. Choi, C. Choi, B. Ko and P. Kim, A method of DDoS attack detection using HTTP packet pattern and rule engine in cloud computing environment, *Soft Computing* **18**(9) (2014), 1697–1703.
[25] Snort - Network Intrusion Detection & Prevention System, 2019, [Online; accessed 10. Apr. 2019]. https://www.snort.org/.
[26] M. Amini, R. Jalili and H.R. Shahriari, RT-UNNID: A practical solution to real-time network-based intrusion detection using unsupervised neural networks, *computers & security* **25**(6) (2006), 459–468.
[27] MITRE ATT&CK®, 2021, [Online; accessed 20. Aug. 2021]. https://attack.mitre.org.
[28] Initial Access, Tactic TA0001 - Enterprise | MITRE ATT&CK®, 2021, [Online; accessed 20. Aug. 2021]. https://attack.mitre.org/versions/v9/tactics/TA0001.
[29] Lateral Movement, Tactic TA0008 - Enterprise | MITRE ATT&CK®, 2021, [Online; accessed 20. Aug. 2021]. https://attack.mitre.org/versions/v9/tactics/TA0008.
[30] A. Oprea, Z. Li, R. Norris and K. Bowers, MADE: Security Analytics for Enterprise Threat Detection, in: *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, Association for Computing Machinery, 2018, pp. 124–136–. ISBN 978-1-4503-6569-7. doi:10.1145/3274694.3274710.
[31] T. Ongun, T. Sakharaov, S. Boboila, A. Oprea and T. Eliassi-Rad, On Designing Machine Learning Models for Malicious Network Traffic Classification, *arXiv:1907.04846 [cs, stat]* (2019), arXiv: 1907.04846. http://arxiv.org/abs/1907.04846.
[32] X. Yuan, C. Li and X. Li, DeepDefense: Identifying DDoS Attack via Deep Learning, in: *2017 IEEE International Conference on Smart Computing, SMARTCOMP 2017*, 2017. https://ieeexplore.ieee.org/abstract/document/7946998/.
[33] C. Yin, Y. Zhu, J. Fei and X. He, A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks, *IEEE Access* **5** (2017), 21954–21961. https://ieeexplore.ieee.org/abstract/document/8066291/.
[34] Y. Zhang, X. Chen, D. Guo, M. Song, Y. Teng and X. Wang, PCCN: Parallel Cross Convolutional Neural Network for Abnormal Network Traffic Flows Detection in Multi-class imbalanced Network Traffic Flows, *IEEE Access* (2019), 1–1. https://ieeexplore.ieee.org/abstract/document/8787567/.
[35] P. Mishra, V. Varadharajan, U. Tupakula and E.S. Pilli, A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection, *IEEE Communications Surveys Tutorials* **21**(1) (2019), 686–728–. doi:10.1109/COMST.2018.2847722.
[36] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer and G. Vigna, SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr, in: *Springer*, 2006, pp. 343–358. https://link.springer.com/chapter/10.1007/11836810_25http://link.springer.com/10.1007/11836810_25.
[37] D. Aitel, The advantages of block-based protocol analysis for security testing, *Immunity Inc., February* **105** (2002), 106. http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.pdf.
[38] S. Gorbunov and A. Rosenbloom, AutoFuzz: Automated Network Protocol Fuzzing Framework, *International Journal of Computer Science and Network Security* **10**(8) (2010), 239–245. http://people.csail.mit.edu/sergeyg/publications/autofuzz.pdfhttp://paper.ijcsns.org/07_book/html/201008/201008036.html.
[39] jtpereyda, boofuzz, 2021, [Online; accessed 20. Aug. 2021]. https://github.com/jtpereyda/boofuzz.

[40] Openrce, sulley, 2021, [Online; accessed 20. Aug. 2021]. https://github.com/OpenRCE/sulley.
[41] P.B. Community. and the, Scapy, 2021, [Online; accessed 24. Aug. 2021]. https://scapy.net.
[42] A.P. Psathas, L. Iliadis, A. Papaleonidas and D. Bountas, A Hybrid Deep Learning Ensemble for Cyber Intrusion Detection, in: *Proceedings of the 22nd Engineering Applications of Neural Networks Conference*, L. Iliadis, J. Macintyre, C. Jayne and E. Pimenidis, eds, Proceedings of the International Neural Networks Society, Springer International Publishing, 2021, pp. 27–41–. ISBN 978-3-030-80568-5. doi:10.1007/978-3-030-80568-5_3.
[43] L. Breiman, Random forests, *Machine learning* **45**(1) (2001), 5–32.
[44] F. Galkin, A. Aliper, E. Putin, I. Kuznetsov, V.N. Gladyshev and A. Zhavoronkov, Human microbiome aging clocks based on deep learning and tandem of permutation feature importance and accumulated local effects, *BioRxiv* (2018), 507780.
[45] Y. Liu, A. Jain, C. Eng, D.H. Way, K. Lee, P. Bui, K. Kanada, G. de Oliveira Marinho, J. Gallegos, S. Gabriele et al., A deep learning system for differential diagnosis of skin diseases, *Nature Medicine* (2020), 1–9.
[46] M. Zamani Joharestani, C. Cao, X. Ni, B. Bashir and S. Talebiesfandarani, PM2. 5 Prediction based on random forest, XGBoost, and deep learning using multisource remote sensing data, *Atmosphere* **10**(7) (2019), 373.
[47] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *arXiv:1810.04805 [cs]* (2019), arXiv: 1810.04805. http://arxiv.org/abs/1810.04805.
[48] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, Roberta: A robustly optimized bert pretraining approach, *arXiv preprint arXiv:1907.11692* (2019).
[49] M. Joshi, D. Chen, Y. Liu, D.S. Weld, L. Zettlemoyer and O. Levy, Spanbert: Improving pre-training by representing and predicting spans, *Transactions of the Association for Computational Linguistics* **8** (2020), 64–77.
[50] I. Beltagy, M.E. Peters and A. Cohan, Longformer: The long-document transformer, *arXiv preprint arXiv:2004.05150* (2020).
[51] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov and S. Zagoruyko, End-to-end object detection with transformers, in: *European Conference on Computer Vision*, Springer, 2020, pp. 213–229.