JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1189-1211 (2022)
DOI: 10.6688/JISE.202211_38(6).0006

SandboxNet: A Learning-Based Malicious Application Detection Framework in SDN Networks*

PO-WEN CHI[1], YU ZHENG[1], WEI-YANG CHANG[2]
AND MING-HUNG WANG[3,+]
[1]*Department of Computer Science and Information Engineering*
National Taiwan Normal University
Taipei, 106 Taiwan
[2]*Department of Information Engineering and Computer Science*
Feng Chia University
Taichung, 407 Taiwan
[3]*Department of Computer Science and Information Engineering*
National Chung Cheng University
Chiayi, 621 Taiwan
E-mail: neokent@gapps.ntnu.edu.tw; 60747041s@gapps.ntnu.edu.tw;
m0907194@o365.fcu.edu.tw; tonymhwang@cs.ccu.edu.tw

Software Defined Networking (SDN) is a concept that decouples the control plane and the user plane. So, the network administrator can easily control the network behavior through its own programs. However, the administrator may unconsciously set up some malicious programs on SDN controllers so that the whole network may be under the attacker's control. In this paper, we discuss the malicious software issue on SDN networks. We use the idea of the **sandbox** to propose a sandbox network called SanboxNet. We emulate a virtual isolated network environment to verify the SDN application functions. With continuous monitoring, we can locate the suspicious SDN applications if the system detects some pre-defined malicious behaviors. We also apply machine learning (ML) techniques to identify unknown malicious applications. Considering the sandbox-evading issue, in our work, we make the emulated networks, and the real-world networks will be indistinguishable to the SDN controller.

Keywords: software defined networking, intrusion detection, SDN application, machine learning, software testing

1. INTRODUCTION

Software Defined Networking (SDN) is a paradigm shift on network technology. The idea was first proposed by McKeown *et al*. in [1][1]. Unlike legacy networks where network engineers implement protocols on network devices following well-specified standards and

Received October 23, 2021; revised December 7, 2021; accepted February 14, 2022.
Communicated by Meng Hsun Tsai.
+ Corresponding author.
* This work was supported by Ministry of Science and Technology, Taiwan, under the Grant MOST 110-2221-E-003-002-MY3, MOST 110-2927-I-194-001, and MOST 107-2218-E-035-009-MY3.
[1]Actually, there were some earlier works about how to process network packets dynamically according to the administrator's programs. However, these works are not popular and practical. In this paper, for convenience, we use McKeown *et al*.'s OpenFlow as the SDN architecture.

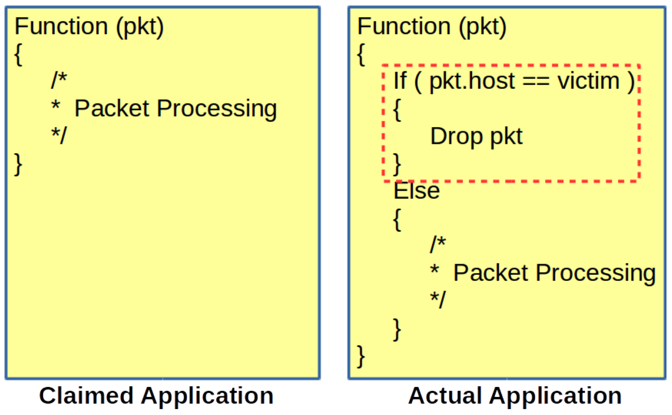
devices process packets with network software on them, SDN provides a possibility that the network administrator can decide how to manage his/her networks. SDN separate the control plane and the data plane from the network devices. The data plane is kept in the network device while the control plane is logically centralized on a controller. So, the network administrator can deploy his/her programs on the controller and modify the network behavior.

Though the SDN architecture is attractive, it also raises another risk: **what if the deployed software is malicious**? The network administrator may download or buy a software module which is claimed to bring some benefits and load the module on the controller. However, the module may include some malicious behavior. For example, the module randomly drops targets for some target user, like presented in Fig. 1. Another example is that a malicious load balance module may forward a VIP user traffics to a weak server instead of a strong one to launch a QoS degrade attack. To solve this problem, we need a defense approach against malicious SDN software[2].

Software testing is a way to detect the malware. There are two kinds of testing, **whitebox testing** and **blackbox testing**. Whitebox testing is a software testing approach that the tester has source codes of the software. However, it is not appropriate for SDN software testing. There are two reasons. First, though there are lots of open source SDN controller projects, like [2–5], most SDN networks are built by network device vendors and their products are close sourced. So, it is not possible for administrators in these networks to run whitebox testing. Another reason is that most SDN controller projects apply module-based development. That is, every function is implemented as a module and this module can be loaded and unloaded dynamically. So, a module may be normal at first until some catalyst module is inserted. This increases the difficulty of whitebox detection. As for the blackbox detection, the tester only focuses on the behavior of the software under test without recognizing its source code. To avoid risking harm to host machines, **sandbox** technology is often applied when blackbox testing. Sandbox is a cybersecurity mechanism that executes a software under testing in an isolated environment. The tester can continuously monitor the sandbox's statuses, like CPU and memory usage, network sockets, system registers *etc*., to check if the software is a malware. In this paper, we apply this idea to test SDN applications.

Since SDN applications are in charge of network packet processing, the isolated environment should be a network instead of a host. We propose **SandboxNet**, an emulated sandbox network for SDN applications. We use mininet [6, 8] to setup an emulated network and periodically check the network status and performance. However, applying mininet directly does not work for two reasons. First, mininet uses Linux kernel network namespace to create virtual links. A packet forwarding through a virtual link can be treated as an inter-process memory copy action. So, when the emulated topology is large, the emulated performance is downgraded very soon. Second, though sandbox technology is widely used for malware detection, attackers may teach their malware to stay inactive when in sandbox [9, 10]. In this way, sandbox-evading malware will bypass detection process. To overcome these two issues, SandboxNet is improved from mininet to support large scale network topology which is copied from a real network. So, the

[2]Here SDN software indicates the software module run on the SDN controller. Undoubtedly the controller itself is also a software, but in this paper, we will not discuss this issue.



```
Function (pkt)
{
    /*
    *  Packet Processing
    */
}
```

```
Function (pkt)
{
    If ( pkt.host == victim )
    {
        Drop pkt
    }
    Else
    {
        /*
        *  Packet Processing
        */
    }
}
```

Fig. 1. An SDN malware example. The left block is the claimed function while the right block is the actual function in process. So, the victim's packets will be dropped.

emulated network will be indistinguishable from the real network. We also implement a monitoring mechanism which can trace a given target's network behavior and see if the SDN application maliciously affects the target.

Though SandboxNet provides an isolated indistinguishable emulated environment, there is a serious problem here: **What is a malicious SDN APP?** In this work, we launch two different approaches to define the malicious SDN APP, the policy-based approach and the machine-learning-based approach. The policy-based approach is a pre-defined criteria detection mechanism according to the user policy, like QoS. Given a user with its policy, our system can emulate the user scenario in SandboxNet and check if the traffic behavior in the emulated environment follows the user policy. Though the policy-based approach is trivial, unfortunately, it is hard to list all checking criteria. To solve this problem, we apply the ML technique to detect undefined malicious behaviors. Based on the popular malicious traffic dataset, we build a prediction model to catch the suspicious behavior of the SDN APP.

1.1 Our Contributions

Our contributions are listed as follows:

- **Large Scale Network Support.** SandboxNet can emulate large scale network. Large scale network implies lots of switches, hosts and network traffics.
- **Online Detection.** SandboxNet supports online detection which means the detection system can coexist with the real network and be controlled by the same SDN controller. This ensures the SDN application under testing is configured with the real setting. Meanwhile, the SDN controller does not need to stop its service for testing setup phase.
- **Indistinguishable to Real Networks.** Since SandboxNet supports large scale network emulation, we can clone the real network into an emulated network, including traffics. So, in the view of SDN controller, the real network and the emulated network are indistinguishable.

- **ML-Integration.** We apply the ML technique to determine if a given SDN APP is malicious or not, that can help the user to catch the unnoticed malicious behavior.

1.2 Organization

This paper is organized as follows. In Section 2, we will introduce some related works about SDN testing and an introduction to mininet. In Section 3, we propose the SandboxNet framework and implementation details. The framework evaluation is in Section 4. Our conclusions are in Section 5.

2. RELATED WORKS

In this section, we review some SDN application testing frameworks. We also describe how mininet works and its limitation.

2.1 SDN Application Testing

There are many SDN application testing frameworks proposed in recent years. Though most works focus on the correctness of the application behavior, we can simply treat the suspicious action as the abnormal case and use the test framework to catch the malicious applications. We also introduce some SDN application debugging tools here since these tools can also trace the SDN application behaviors.

Nick McKeown *et al.* proposed an SDN debugging framework called ndb in [12,13]. They built a packet tracking system to follow packet transmission for checking if the application is correctly developed. They also developed a breakpoint system to interrupt packet transmission and check dynamically. Canini *et al.* integrated model checking and symbolic execution to build an automating testing tool called NICE [14, 15]. This tool can check the SDN application behavior and see if there are any violations about the application properties. Ball *et al.* applied the similar idea with NICE but created their own language to described SDN applications [16]. So, they can handle the case of SDN programs with infinite states by turning the model into First-Order Logics. Yao *et al.* enhanced the NICE work to support the black-box testing [17]. They created a middleware between the SDN application and their state machine, so they did not need to know the source code of the application under testing. Scott *et al.* used a black-box fuzz tester to test a sequence of packet inputs [18]. Durairajan *et al.* used fs-sdn [20] to establish a debugging system [19]. They built an OFf proxy between the SDN controller and fs-sdn simulator and provided many debugging tools in the OFf proxy.

Though the works described above provide lots of approaches to uncover SDN application bugs[3], they all have some limitations. First, most debugging tools focus on only one SDN application. That is, they may ignore the interaction between SDN applications, and some malicious behaviors may be triggered by other applications instead of network events. Second, most debugging tools take the network topology as inputs without considering the network throughput issue. So, the traffic from some host may be directed to

[3]For simplicity, we treat bugs in those works as malicious applications. In our opinion, the difference between a bug and a malicious application is if it is intentionally programmed.

a congested but valid path and it is hard to be detected by existing tools. Third, all debugging tools do not consider the sandbox-evading issue. So, malicious SDN applications can disable themselves when finding the debugging process. For example, the breakpoint-based solution is not suitable since the processing time will lease the debugging fact.

2.2 Mininet

Mininet [6, 8] is the most popular testing platform for SDN applications. Mininet is based on Linux kernel containers to emulate hosts, switches and links in a network. Therefore, mininet can run native userspace applications on each emulated host. As for the SDN switch, mininet uses Open vSwitch [7] as the switch software. Open vSwitch is a virtual switch with production quality and many vendors use Open vSwitch as their OpenFlow protocol implementations. So, the emulated network based on mininet is very close to the real network.

Unfortunately, mininet cannot be used as the sandbox network directly for some reasons. First, mininet uses Linux container technology and each program on a virtual host is directly executed on the mininet host. However, it is difficult to make one computer run so many programs because of resource limitation, especially when considering large scale environment. Second, the virtual link performance in mininet is inverse proportional to the network distance. Lantz *et al*. shown that with longer distance, the end-to-end bandwidth decreases seriously [8]. This is because longer distance implies more memory copy counts and undoubtedly more transmission time. This fact will help the attacker to learn if the network is a real network or an emulated one and the malicious SDN APP can evade the emulated environment.

To overcome these problems, instead of generating real traffics, we modify the Open vSwitch to support virtual traffic generation. We record the network topology and the traffic pattern from the real network. Then we clone the network to the SandboxNet with virtual traffics. We assert that with our SandboxNet, the emulated network is indistinguishable to the real network.

2.3 Machine Learning Based (ML-based) Malicious Traffic Detection

One common technique applied in conventional intrusion detection system (IDS) is rule detection; however, such rule-based algorithms relied on maintaining a list of signatures. When facing new threats (*i.e*., zero-day) or a variant of attack, rule-based methods could not perform well due to lack of knowledge in the threat. Thus, several scholars turned to ML techniques for improving the IDS. In recent years, ML is playing an essential role in addressing different issues. In IDS, several ML studies have been conducted for malicious traffic detection for different application, such as Internet of Things (IoT), SDN, and botnets [28, 35]. Garcia-Teodoro *et al*. compared different IDS based on statistical methods, knowledge about threats, and ML. They listed the strengths and drawbacks of each method [33]. To improve the accuracy and detection speed for IDS, Jyothsna *et al*. reviewed and compared the learning-based approaches with previous studies [34]. Recently, Dushimimana *et al*. proposed a ML-based IDS in IoT [29]. They built an IDS by constructed a Bidirectional Recurrent Neural network (BRNN) technique to detect malicious behavior in IoT. Kasongo *et al*. constructed a deep long short-term memory-based classifier to detect the malicious behavior for wireless IDS [31]. They also compared their method with traditional learning technique. In [30], Lo *et al*. presented an IDS based on GNNs for node classification. They leverage the information in network flow data and transform it to graph, and construct GNNs through feed these features, such as graph structure and edge features.

There are studies related to the NSL-KDD dataset to improve the ML-based IDS. They improve models trained on the NSL-KDD dataset to improve to the higher performance on identifying abnormal traffics. In [40], L. Dhanabal *et al*. proposed a deep network system SAE-LR (Sparse AutoEncoder with Logic Regression) to try to adjust to new patterns of intrusions on the NSL-KDD dataset. Another research [42], trained on NSL-KDD dataset through an AutoEncoder-based approach that uses only the encoder in the autoencoder without a decoder, which helps them compress features, extract key features build faster, and make precise predictions of abnormal traffic.

From the abovementioned works, different ML methods were employed in constructing novel IDS; however, studies applying learning-based IDS on SDN for malicious SDN application detection remain scant. In this study, we constructed a learning-based malicious traffic detection in the proposed emulated network to explore approaches for malicious SDN application detection. Furthermore, we monitored and analyzed the flows from external hosts and the interactions between the SDN applications.

3. OUR PROPOSAL: SANDBOXNET

In this section, we introduce the SandboxNet architecture and its implementation details. Note that in this paper, we only focus on the OpenFlow architecture and its protocol [11]. For simplicity, in this section we use a LAN (local area network) to describe how SandboxNet work, but we think that SandboxNet can be extended to support WAN (wide area network).

3.1 Assumption

We first list assumptions in this work before we describe how SandboxNet works.

- **Trusted SDN switches**. There are some works which focus on how to find out compromised SDN switches, like [21–23]. A compromised SDN switch may not follow the rule from the SDN controller. In this work, we focus on the SDN application, and we assume all SDN switches are trusted. That is, SDN switches honestly execute configured flow rules and the SDN controller can derive correct network information through OpenFlow protocol.

- **Trusted SDN controller**. As described before, our target is the SDN application. So, we assume the SDN controller which is a platform to run SDN applications is trusted. That is, the SDN controller will honestly execute SDN applications. Moreover, we consider that the SDN application can correctly get network information from the SDN controller.

- **Malicious behaviors are only for the network**. In this paper, we assume the malicious SDN application attacks the network only and will not attack other entities like the controller or the computer runs the controller. For example, the application

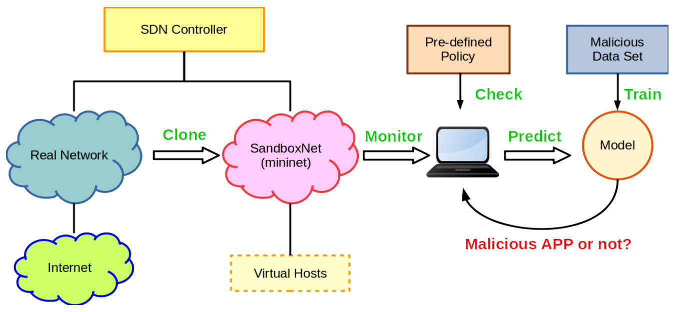


Fig. 2. SandboxNet architecture.

may use out all system's memory and crash the SDN controller. This is an undoubtedly attack scenario, but our work focuses on the network. Moreover, passive attacks are not our concerns in this paper. For example, leaking network information from SDN malware to outsiders is not our concern.

We note that these attacks so serious and an SDN network system need to protect from these attacks. There are many works proposed to discuss and defensive these attacks. SandboxNet can be integrated as a part of protection mechanism.

3.2 SandboxNet Architecture

SandboxNet is an emulated, isolated network environment for testing SDN applications. In order to make the emulated network indistinguishable from the real network, first we clone the network information from the real network, including hosts, network switches, and its topology. Then we build an emulated network in mininet with the given topology. We also add a virtual node to serve as a gateway and data from this node will be treated as data from Internet[4]. We also record the host behaviors and make emulated hosts generate traffics. We describe traffic generation detail in Section 3.3. The architecture is shown in Fig. 2. Note that the real network and the emulated network are both controlled by the same controller at the same time. So, the controller configurations for the real network and the emulated network are definitely the same. For the controller, it controls two isolated and similar networks. Besides, since the emulated network is directly connected to the real controller, all testing cases can be run online.

We call the emulated network **background**, which is the testing condition. Since the testing condition is cloned from the real network, it should be indistinguishable with the real network. The background information is used for the SDN application to do decisions. Then the administrator can add some test traffic to test the SDN application. The test traffic may include host emulation or service flow emulation. For example, a network manager wants to deploy a new SDN application on his/her network. The administrator directly loads the application on the controller. Then the administrator can add a virtual host as a VIP user and start a new service. The administrator monitors this emulated network and see if the network status violates pre-defined criteria. The administrator also collects the traffic attributes and uses the pre-trained model to check the APP. If the answer is yes, which means the traffic introduced by the APP violates the user policy or

[4]Of course, there may be multiple gateways.

Algorithm 1 : Traffic Counter Generation

Input: The set of emulated traffic flows, $F = \{F_1, \ldots, F_n\}$. For a flow F_i, there is a set of OpenFlow rules R_i that F_i matches and follows and a distribution function $D_i(t)$, where t is the time interval. Each OpenFlow rule r has its own counter information c.
Output: The counter of each OpenFlow rule in the network.

```
1: for each $F_i \in F$ do
2:     for each $r \in R_i$ do
3:         $r.c+ = D_i(t)$.
4:     end for
5: end forreturn $E_n$;
```

the prediction result is positive, the monitor system will send an alarm to the administrator and the administrator can remove the application. Otherwise, the SDN application is treated as trusted.

Note that the detection process is continuous and dynamic. The administrator can adjust the emulated environment as it wishes anytime. That is, the administrator is able to try all possible trigger points by adjusting hosts and connections in the background.

Undoubtedly, this testing operation needs the SDN application to be loaded first. Some may doubt that the malicious SDN application may attack the network before we catch it. This is true. However, we think this is unavoidable. SDN is a kind of event-triggered system. Without events, in most cases the applications will not work at all. To ease this problem, we can emulate important users first in SandboxNet before they enter the real network to reduce damage. Besides, since this is an online detection system, the testing procedure can run continuously and parallel with the real network. So, we can test any host or any traffic at any time.

3.3 Mininet Scalability Issue and Host Emulation

As described in Section 2.2, mininet cannot support large scale network emulation. This will be a key point for malware to check if the controlled network is a real network or not. To solve this problem, we back to a simple question: how the SDN controller gets the information of all traffic flows?

In OpenFlow specification [11], the SDN switch uses **counter** to record the status of each flow. The SDN controller uses the OpenFlow protocol to derive the counter information of each flow, table and port. So, we can cheat the SDN controller by replying to artificial values. The design is shown in Fig. 3. The traffic generator module generates counter value of each flow according to the administrator requirement, like constant bit rate or Poisson traffic, and answer the request from the SDN controller directly. In this design, there is no real packet between virtual switches and therefore it is possible to generate traffics with high throughput. The traffic generator module controls all SDN switches in the emulated network at the same time. The traffic generation module's work is described in Algorithm 1.

For the implementation details, we modify OVS as shown in Fig. 4. We modify the function **rule_dpif_credit_stats__** in **ofproto-dpif.c** by adding a customized byte number to the rule. This byte number is generated from a central process called traffic generator,

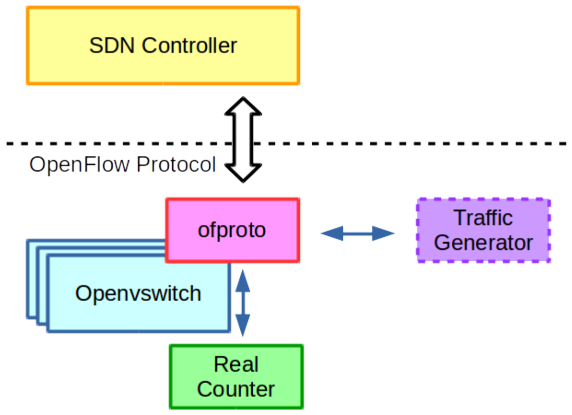
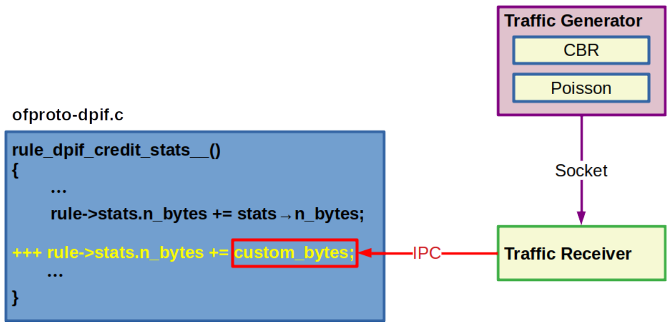


Fig. 3. Artificial cheating traffic generation. The dotted block is our implemented module. The counter information is derived from our traffic generation module except the Packet_In flows.



```
rule_dpif_credit_stats__()
{
      ...
      rule->stats.n_bytes += stats→n_bytes;

+++ rule->stats.n_bytes += custom_bytes;
      ...
}
```

Fig. 4. The implementation of our design.

which can support different traffic patterns, and is passed to the OVS daemon through sockets and the IPC mechanism. The traffic generator pattern is determined by the user so it can setup its own background network for the experiment. The traffic generator is the central architecture because we want to synchronize the customized traffic statuses among all emulated switches. Note that we keep the flow status here because there may be some real packets that pass through the rule and we still count them in.

Note that there are some kinds of counters that cannot be generated. For example, a flow may ask the SDN switch to forward packets with some pattern to the controller. In this case, using artificial counters to cheat the SDN controller is nonsense since the controller can absolutely know how many packets it processes. Therefore, for this kind of Packet_In flows, we leave the openvswitch to derive counter information through normal APIs instead of our traffic generation module. Though the Packet_In flow traffic cannot be generated, we assert that this should be fine in the real use. The reason is that the Packet_In event often happens at the initialization phase and most packets in this phase are control packets only which cost almost nothing in bandwidth. Even the network scale is large, mininet can still afford this task.

In this work, we implement some network service initialization tools, like TCP syn,

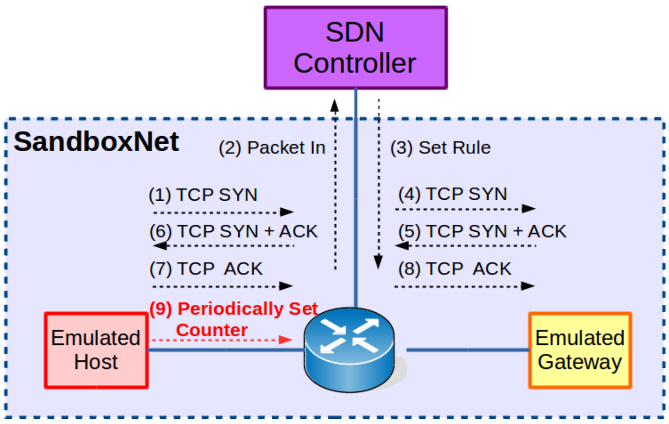


Fig. 5. TCP emulated flow setup. Here we assume the SDN controller sets bi-directional flow rules at the same time. After step (1) to step (8), SandboxNet only needs periodical counter setting to emulate traffics.

UDP packet, HTTP Get and so on. We use these tools to trigger a network service. We also implement related response tools on one host for serving as a gateway or a router to another networks. Note that these tools are only sending the first packets in their protocols to trigger the works of the SDN applications. Once the SDN application setup related packet handling rules on the network switches, we use the traffic generator to modify the flow counter and give a traffic illusion to the SDN controller. For example, if we want to emulate a host watching YouTube, we only send one TCP syn packet and then use constant bit rate to add counters. Some may doubt that even the YouTube video is transmitted in a constant rate, there are other control messages like TLS negotiation[5]. Here we skip these packets and simply add an offset on each counter since most SDN controller monitor applications work periodically and TLS connection time is often short enough in one monitoring interval. If the SDN application requires more packets to the controller, we will use other trigger program which implements more steps in the service setup. The overall process is shown in Fig. 5.

The evaluation result is shown in Section 4. Note that we only use this way to simulate background flows. For the flow that we want to trace, we will make the packets be forwarded in the virtual links.

3.4 Suspicious Behavior Detection Criteria

In SandboxNet, the monitor system will focus on a given host or a given traffic flow and see if there is any violation to pre-defined criteria. If the new generated flow rules violate these criteria, the monitor system will send an alarm to the administrator. The criteria we use to check the SDN applications:

1. **The host cannot reach the destination**. The pass condition is to check if the packet can be correctly forwarded to the destination. The looped path also falls into these criteria.

[5]YouTube video is transmission through HTTPs.

2. **The path is unconsciously duplicated**. The pass condition is to check if a packet is duplicated to unwanted places.
3. **The traffic is directed to an unreasonable path**. If there are multiple candidate paths between the source and the destination[6] and the traffic loads are not even distributed, the monitor system will raise an alarm to the administrator since the network is not fully utilized.

We use **libpcap** to implement a receiver agent for every emulated host to check the forwarded packets.

Note that new criteria can be added to this list, and this is our future work. Of course, sometimes we may program the network with abnormal configurations. So, we just raise an alarm to the administrator to decide if this is a normal case or not.

3.5 Machine Learning Technique

As mentioned in the previous section, we will use the pre-trained model in mininet to check the APP behaviors. The reason that affects us in choosing ML techniques and the structure of ML will be discussed in this section.

3.5.1 Method of machine learning - BRNN

Schuster and Paliwa first introduced Bidirectional Recurrent Neural Networks (BRNN) in 1997 [36], which is an extension of unidirectional RNN. The ordinary RNN only focuses on the previous input, while BRNN focuses on the past and future inputs, and can use more information than RNN to make predictions. Fig. 6 shows how to combine the past and future input connect with the output layer. The structure of the BRNN was constructed by two RNNs that are opposite in direction and connected to the same output layer. The output Y_t can be calculated by:

$$Q(Y_t|\{X_i\}_{i\neq t}) = \sigma(W_Y^f h_t^f + W_Y^b h_t^b + b_Y) \tag{1}$$

where

$$h_t^f = tanh(W_h^f h_{t-1}^f + W_X^f X_t^f + b_h^f) \tag{2}$$

$$h_t^b = tanh(W_h^b h_{t+1}^b + W_X^b X_t^b + b_h^b) \tag{3}$$

In the formula above, the forward layer and the backward layer are represented as superscript f and superscript b. W_Y are the weights that connect the hidden layer to the output layer, W_h are the weights that connect between hidden layers, and W_X are the weights that connect the input layer to the hidden layer. b_Y and b_h are represented as the biases of the output layer and the hidden layer. This structure indicates that BRNN accepts both hidden layer outputs of the previous and the next moments as inputs to achieve the attention of the past and future elements.

Previous studies have leveraged the temporal patterns of network traffic in identifying malicious behaviors and attacks [26, 27]. Based on the observation, we adopt BRNN as a ML model to identify the malicious behaviors in this study.

[6]There may be multiple possible destinations. For example, the load balance service will dispatch the network load into different servers.

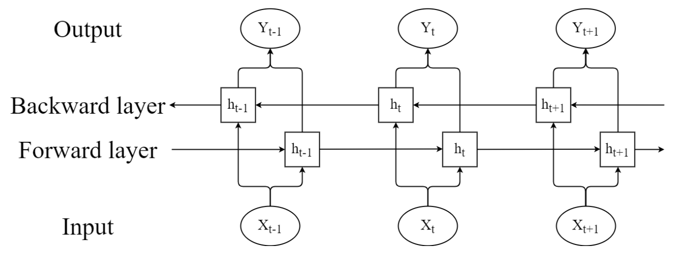


Fig. 6. The architecture of Bidirectional Recurrent Neural Networks (BRNN).

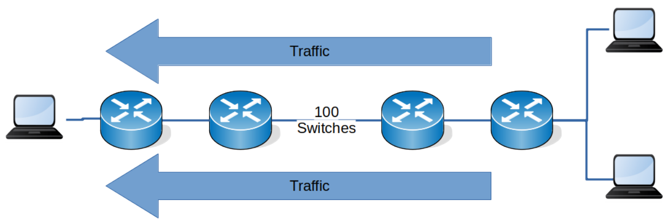


Fig. 7. Testing topology for computational cost comparison.

4. EVALUATION

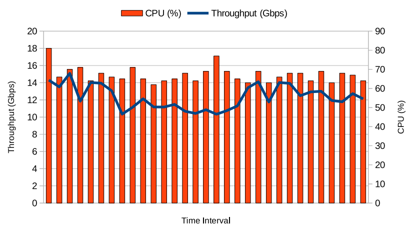
In this section, we evaluate SandboxNet in two ways. First, we evaluate the required computational cost in the emulated networks. Then, we develop some malicious SDN applications and catch them with the help of SandboxNet. The experiments are run on a virtual machine with 2 cores CPU and 4G RAM. The host computer is equipped with INTEL i7-9700K CPU and 64G RAM.

4.1 SandboxNet Computational Cost

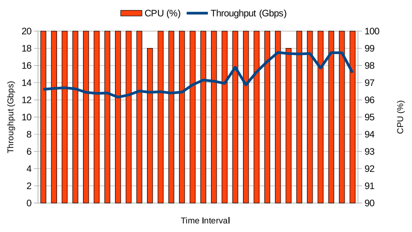
In this subsection, we will evaluate the required CPU resource of SandboxNet. We compare SandboxNet with pure mininet in a chain topology with 100 switches. We attach one virtual host at one end and two hosts at the other end. The testing topology is shown in Fig. 7. We make two hosts generate traffics to the host at the other end independently. In the mininet case, we use **iperf** to generate traffics. In the SanboxNet case, we use our approach to cheat the SDN controller. For simplicity, we use constant bit rate as the generated traffic model. Besides, we make the bit rate in SandboxNet similar to the value we measure in the mininet case. The comparison result is shown in Fig. 8. Since it is hard to isolate each process's CPU resource cost, here the CPU usage is measured at the overall system level[7].

From the mininet result, we can see that iperf takes lots of CPU resource. With two iperf processes can use out the system's CPU resource. More emulated iperf processes implies each iperf can take less CPU resource and this will restrict iperf's performance. So, it is not suitable for mininet to emulate a large network with lots of services. As for the SandboxNet, we can see the CPU usage is much lower than the mininet case. Note that this is overall system CPU usage, including irrelevant processes like Desktop

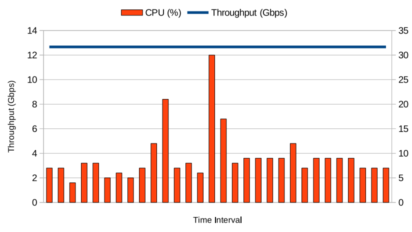
[7]We use the command **vmstate** to record the CPU usage.



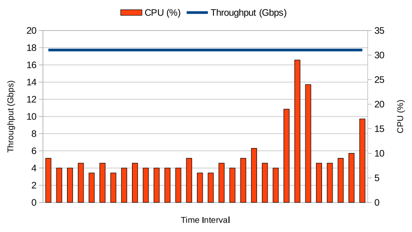
(a) Mininet with one host traffic.



(b) Mininet with two hosts traffics.



(c) SandboxNet with one host traffic.



(d) SandboxNet with two hosts traffics.

Fig. 8. Computational cost between Mininet and SandboxNet.

Environment. The average CPU usage after mininet environment setup is around 7%. So SandboxNet can support real network simulation and therefore can cheat the SDN controller that this is a ”real” network.

4.2 SandboxNet SDN Malware Detection: Pre-defined Policy

In our experiment, we define two attack scenarios as our checking criteria.

4.2.1 Scenario 1: Malicious forwarding issue

Forwarding issue is a problem that the rules for a network flow given by the SDN controller cannot forward the flow packets correctly. For example, packets from some victim host may be dropped at some switch. Another example is that packets from some victim host may be duplicated to other irrelevant hosts.

To catch this issue, first, we prepare a sensitive user list and a sensitive network service list. SandboxNet will focus on these hosts and network services. Then we clone the real network into an emulated network in SandboxNet. The network cloning includes the virtual traffic generation. We randomly pick a host from the user list and trigger a network service from the service list. We check if the receiver can correctly get the packets. When the network service is at the outside of the network, the emulated gateway plays the role of the receiver and runs relative network services. If the receiver cannot get the packet, the SDN applications are suspicious. In the meantime, we also enable the packet capture process on each virtual host other than the receiver. We check if other hosts can get the sensitive users' packets. If yes, there may be a duplication issue.

We write an SDN APP to randomly drop the packets for the sensitive user and treat this APP as a malicious APP. The experiment setting is shown in Fig. 9. Fig. 10 is the monitor result with different drop rates. We can see that it is simple to catch the drop packet event. The administrator can then trace the path, find the problematic rule and locate the APP which sets this malicious rule.

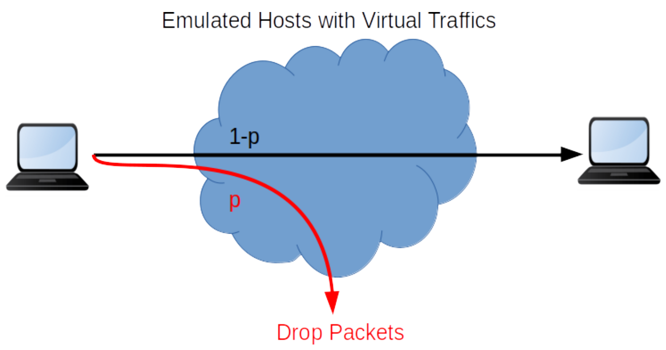
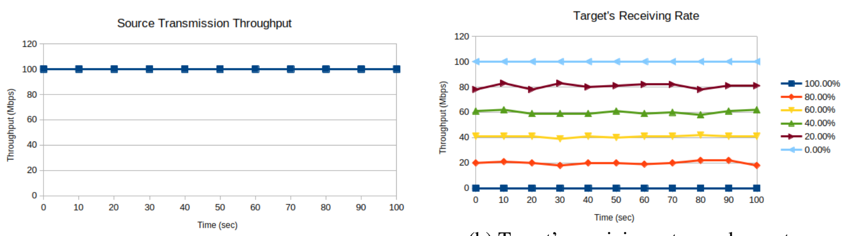


Fig. 9. Malicious forwarding scenario. *p* is the drop rate set by the malicious APP.



(a) Source transmission rate.

(b) Target's receiving rate vs. drop rate.

Fig. 10. Throughput from the unbalanced rates between the source and destination.

4.2.2 Scenario 2: Malicious path selection issue

Generally speaking, when we want to prioritize some network traffic, the most common way to add a tag to the flow, like the VLAN tag or the DSCP field. Then the flow packet will be processed in the high priority queue. Consider the following case. If there is an SDN application that concentrates all VIP flows in a path with low link rate while other unimportant flows use another path with high link rate. In this case, though the packets with higher priority are correctly tagged, the network QoS performance for VIP users will be poor. Fig. 11 is an example of this attack scenario.

To find this kind of attacks, again we first construct an emulated network environment. Again, we focus on the sensitive users and their sensitive network services. For each sensitive flow, according to the network topology, we find all possible paths between the flow source and destination. We record all paths' information, like their link rates and on-going throughput. Then we generate a traffic with a given rate and see if the receiving rate is as expected or not. If not, like Fig. 12, it implies that the SDN application determines a path that will cause an unbalanced network result. So SandboxNet will raise an alarm to the network administrator.

Note that the malicious QoS setting can also be treated as a variant of this attack.

4.3 SandboxNet SDN Malware Detection: Machine Learning Technique

In this section, we will introduce how to integrate the machine learning mechanisms into our design. We also use the NSL-KDD dataset [37] and conduct the experiments of our intrusion classification method using deep learning.

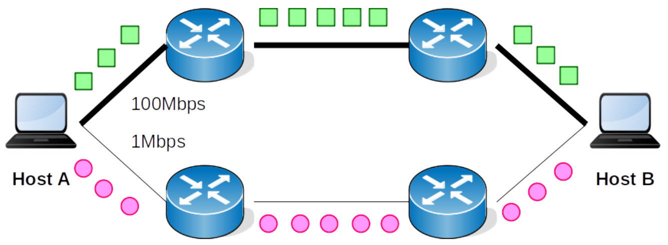


Fig. 11. Malicious path selection. There are two paths between Host A and Host B. The line width implies the link rate. The circle represents the packet with higher priority while the square represents the packet with lower quality. In this case, higher priority packets are forwarded through low link rate. So even these packets are tagged with high priority, this setting does not work and can be treated as an attack.

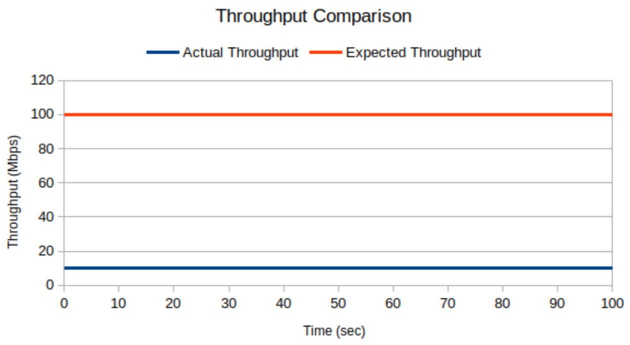


Fig. 12. The expected transmission rate vs. the actual transmission rate.

4.3.1 The centralized controller for behavior monitoring

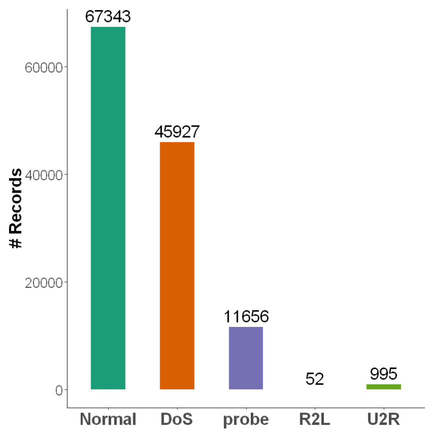
Since SDN is a centralized architecture with a controller to monitor the traffic and behaviors of the whole network, we here adopted the features of a controller that can receive the reports from every end device about the traffic and communication behaviors. Based on this, the controller can communicate with the detection model to complete an application classification.

4.3.2 NSL-KDD dataset

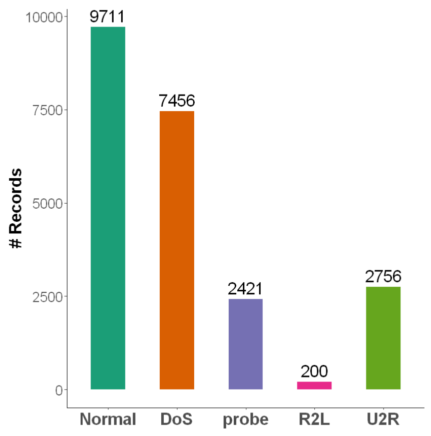
We employ the popular NSL-KDD dataset, which is a reduced and enhanced version of KDD Cup 99 dataset, in our experiment.

Every record in NSL-KDD has a total number of 41 features with a binary value to represent the category of the traffic (normal/ attack type). These features can be subdivided into 4 categories according to different uses. It can also be subdivided into binary values, categorical values, and continuous values based on the different types [38].

Based on the NSL-KDD studies [39], the dataset groups all the attack types into four categories. In each category, sub-labels are indicating detailed attack forms. As shown in Table A5, a total of 23 sub-labels, including 22 attack types and 1 normal label in the training set. In the testing set, there are 38 sub-labels, while 21 of them are also included in the training set. We consider the reason for having different sub-labels between the



(a) The number of records of each category in the training set.



(b) The number of records of each category in the testing set.

Fig. 13. The number of records of each category in the NSL-KDD dataset.

training set and testing set would be including potential attack types which are not shown in the training set.

In Figs. 13 (a) and (b), we show the number of records in each category of training and testing sets in NSL-KDD dataset.

Table 1. Binary classification in all types compared with [40].

Method	Precision	Recall	F1-measure
Our proposal	84.00	99.94	91.28
[40]	84.60	92.87	None

4.3.3 Machine learning experiment

As mentioned in the previous section, we evaluated the proposed work using NSL-KDD dataset. Three metrics used for measuring the performance of our proposal are described as follows:

- Recall= $\frac{TP}{TP+FN} * 100\%$: the ratio (%) of true positive divided by the sum of true positive (TP) and false negative (FN).
- Precision= $\frac{TP}{TP+FP} * 100\%$: the ratio (%) of true positive divided by the sum of true positive and false positive (FP).
- F1-measure= $\frac{2*Recall*Precision}{Recall+Precision}$: the weighted average of Precision and Recall.

What follows illustrates the procedure of our experiment. First of all, we relabel each record into a binary form (0: benign and 1: malicious). After that, we constructed a BRNN machine learning model to identify malicious records. Finally, we conduct experiments in two modes, one is normal versus malicious (binary experiment). The other one is normal versus every attacking category; for each attack category (DoS, Probe, R2L, and U2R), we also develop corresponding classifiers and evaluate them.

Table 2. Binary classification compared with [42].

	Our proposal			[42]		
Attack type	Precision	Recall	F1-measure	Precision	Recall	F1-measure
All	84.00	99.94	91.28	82.22	79.74	76.47
DoS	97.88	98.87	98.37	96.34	83.95	89.72
Probe	93.06	99.67	96	86.37	8038	83.27
R2L	54.85	97.59	70.23	81.86	11.26	18.79
U2R	53.26	87.16	66.12	73.33	32.84	46.36

We also compared the performance of our model with other recent studies. The metrics include Accuracy, Precision, Recall, Error Rate, and F1-measure. Table 1 shows that our proposal receives almost the same performance in Accuracy with [40] but much better performance on Recall. In Table 2, we compared the performance with [42]. In all types, DoS, and Probe attacks, our work significantly outperforms previous studies. Although our Precision is lower in R2L and U2R, we still receive a higher Recall. Finally, we compared the performance of rule-based and ML-based on the NSL-KDD dataset. In Table 3, we compared with the Hoeffding Tree [41]. Our proposal works better in both Accuracy and Error Rates. Based on the results, we assert our model could perform better in identifying DoS and Probe attacks.

Table 3. Binary classification in all types compared with [41].

Method	Accuracy	Error(%)
Our proposal	99.94	0.06
Hoeffding Tree [41]	98.60	1.40

5. CONCLUSIONS

In this paper, we propose SandboxNet, a sandbox concept for SDN applications. We use mininet to build an emulated networks and enhance mininet to support large traffics. SandboxNet provides a possibility to check if loaded SDN applications have any suspicious behaviors. So SandboxNet is a great tool for network administrators to catch the malicious SDN application.

Though SandboxNet provides a testing framework, the suspicious network behaviors are undoubtedly not complete. For example, packet modification problem may be an important issue and need to be discovered. Our next step is to find more attack scenarios and extend SandboxNet for more malicious behavior patterns.

REFERENCES

1. N. McKeown, T. Anderson, H. Balakrishnan, *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, Vol. 38, 2008, pp. 69-74.
2. "The POX network software platform," https://github.com/noxrepo/pox,2019.
3. "ONOS - A new carrier-grade network operating system designed for high availability, performance, scale-out," https://onosproject.org/, 2019.
4. "OpenDaylight," https://www.opendaylight.org/, 2019.

5. “Ryu SDN framework,” https://osrg.github.io/ryu/, 2019.
6. “Mininet: An instant virtual network on your laptop (or other PC),” http://mininet.org/, 2019.
7. “Open vSwitch,” https://www.openvswitch.org/, 2019.
8. B. Lantz, H. Bob, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 19:1-19:6.
9. T. Vidas and N. Christin, “Evading Android runtime analysis via sandbox detection,” in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, 2014, pp. 447-458.
10. N. Miramirkhani, M. P. Appini, and N. Nikiforakis, and M. Polychronakis, , “Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts,” in *Proceedings of IEEE Symposium on Security and Privacy*, 2017, pp. 1009-1024.
11. “The open networking foundation, openflow switch specification,” https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2015.
12. N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, “Where is the debugger for my software-defined network?,” in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*, 2012, pp. 55-60.
13. B. Heller, C. Scott, N. McKeown, *et al.*, “Leveraging SDN layering to systematically troubleshoot networks,” in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013, pp. 37-42.
14. M. Canini, D. Kostic, J. Rexford, and D. Venzano, “Automating the testing of OpenFlow applications,” in *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering*, 2011, pp. 1-6.
15. M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, “A NICE way to test OpenFlow applications,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012, pp. 127-140.
16. T. Ball, N. Bjørner, A. Gember, *et al.*, “VeriCon: Towards verifying controller programs in software-defined networks,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 282-293.
17. J. Yao, Z. Wang, X. Yin, X. Shi, Y. Li, and C. Li, , “Testing black-box SDN applications with formal behavior models,” in *Proceedings of IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2017, pp. 110-120.
18. C. Scott, A. Wundsam, B. Raghavan, *et al.*, “Troubleshooting blackbox SDN control software with minimal causal sequences,” in *Proceedings of ACM Conference on SIGCOMM*, 2014, pp. 395-406.
19. R. Durairajan, J. Sommers, and P. Barford, “Controller-agnostic SDN debugging,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 227-234.
20. M. Gupta, J. Sommers, and P. Barford, “Fast, accurate simulation for SDN prototyping,” in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013, pp. 31-36.
21. P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, “How to detect a compromised SDN switch,” in *Proceedings of the 1st IEEE Conference on Network Softwarization*, 2015, pp. 1-6.

22. Y. Chiu and P. Lin, "Rapid detection of disobedient forwarding on compromised OpenFlow switches," in *Proceedings of International Conference on Computing, Networking and Communications*, 2017.
23. P. Perešíni, M. Kuźniar, and D. Kostić, "Dynamic, fine-grained data plane monitoring with monocle," *IEEE/ACM Transactions on Networking*, Vol. 26, 2018, pp. 534-547.
24. J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 Darpa intrusion detection system evaluations as performed by Lincoln laboratory," *ACM Transactions on Information and System Security*, Vol. 3, 2000, pp. 262-294.
25. M. Z. Shafiq, L. Ji, A. X. Liu, and J. Wang, "Characterizing and modeling internet traffic dynamics of cellular devices," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 39, 2011, pp. 265-276.
26. F. Soldo, A. Le, and A. Markopoulou, "Blacklisting recommendation system: Using spatio-temporal patterns to predict future attacks," *IEEE Journal on Selected Areas in Communications*, Vol. 29, 2011, pp. 1423-1437.
27. A. Lakhina, M. Crovella, and C. Diot, "Detecting distributed attacks using network-wide flow traffic," in *Proceedings of FloCon 2005 Analysis Workshop*, 2005, pp. 1-3.
28. M. S. Elsayed, N.-A. LeKhac, S. Dev and A. D. Jurcut, "Machine-learning techniques for detecting attacks in SDN," *IEEE 7th International Conference on Computer Science and Network Technology*, 2019, pp. 277-281.
29. A. Dushimimana, T. Tao, R. Kindong, and A. Nishyirimbere, "Bi-directional Recurrent Neural network for Intrusion Detection System (IDS) in the internet of things (IoT)," *arXiv Preprint*, Vol. 7, 2020, pp. 524-539.
30. W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "E-GraphSAGE: A graph neural network based intrusion detection system," *arXiv Preprint*, 2021, arXiv:2103.16329.
31. S. M. Kasongo and Y. Sun, "A deep long short-term memory based classifier for wireless intrusion detection system," *ICT Express*, Vol. 6, 2020, pp. 98-103.
32. M. Aljanabi, M. A. Ismail, and A. H. Ali, "Intrusion detection systems, issues, challenges, and needs," *International Journal of Computational Intelligence Systems*, Vol. 14, 2021, pp. 560-571.
33. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, Vol. 28, 2009, pp. 18-28.
34. VVRPV. Jyothsna, R. Prasad, and K. M. Prasad, "A review of anomaly based intrusion detection systems," *International Journal of Computer Applications*, Vol. 28, 2011, pp. 26-35.
35. Y. N. Soe, Y. Feng, P. I. Santosa, R. Hartanto, and K. Sakurai, "Machine learning-based IoT-botnet attack detection with sequential architecture," *Sensors*, Vol. 20, 2020, pp. 4372.
36. M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, Vol. 45, 1997, pp. 2673-2681.
37. "NSL-KDD|Datasets|Research|Canadian Institute for Cybersecurity|UNB," https://www.unb.ca/cic/datasets/nsl.html, 2021.
38. L. Dhanabal and S. P. 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*, Vol. 4, 2015, pp. 446-452.

39. M. V. Mahoney, Matthew and P. K. Chan, "Learning nonstationary models of normal network traffic for detecting novel attacks," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 376-385.
40. S. Gurung, M. K. Ghose, and A. Subedi, "Deep learning approach on network intrusion detection system using NSL-KDD dataset," *International Journal of Computer Network and Information Security*, Vol. 11, 2019, pp. 8-14.
41. S. Geetha, U. N. Dulhare, and S. S. S. Sindhu, "Intrusion detection using NBHoeffding rule based decision tree for wireless sensor networks," in *Proceedings of IEEE 2nd International Conference on Advances in Electronics, Computers and Communications*, 2018, pp. 1-5.
42. C. Zhang, F. Ruan, L. Yin, X. Chen, L. Zhai, and F. Liu, "A deep learning approach for network intrusion detection based on NSL-KDD dataset," in *Proceedings of IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification*, 2019, pp. 41-45.

Po-Wen Chi received his BS, MS and Ph.D. in Electrical Engineering from National Taiwan University in 2003, 2005 and 2016. From 2005 to 2016, he was an Engineer in Institute for Information Industry, Taiwan. From 2016 to 2018, he was a Senior Engineer in Arcadyan Technology Corporation, Taiwan. He joined Department of Computer Science and Information Engineering, National Taiwan Normal University, as an Assistant Professor in 2018. He is currently an Associate Professor. His research interests include network security, applied cryptography, software-defined networking, and telecommunications.

Yu Zheng received his BS degree in the Department of Computer Science and Information Engineering in National Taiwan Normal University. He is currently a Software Engineer at eWay Network Corporation.

Wei-Yang Chang is currently working as a student towards his MS degree in the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. He received his BS degree in Information and Telecommunications Engineering from Ming Chuan University in 2020.

Ming-Hung Wang is an Assistant Professor in the Department of Computer Science and Information Engineering of National Chung Cheng University. He received his Ph.D. in Electrical Engineering at National Taiwan University in 2017. Dr. Wang focuses on addressing security issues about online information, Internet behaviors, and network infrastructures.

A. APPENDIX

Table A1. Denial of Service (DoS): Attacks that make computing or memory too busy or too full to handle normal requests or deny normal users access to machine or server [39].

Attack type	Description
back	The attacker used the wrong IP addresses in the source IP address of the IP header. As a result, the receiver fails to determine the real attacking IP to stop the attacker from sending illegitimate packets.
land	In this kind of attack, a large number of TCP/SYN packets are sent to the target machine. In these attacks, the spoofed IP packets have both the source as well as the destination IP addresses as the target machine's IP address.
neptune	A large number of SYN packets are sent to the target machine to exhaust its buffer.
pod	It uses oversized IP packets to crash, freeze, or reboot the system.
smurf	Use ICMP echo request packets directed to IP broadcast addresses from remote locations to create a DoS attack.
teardrop	It uses overlapping IP fragments. It causes machines to reboot. This attack affects systems that are still using old versions of Windows and Linux operating systems.
mailbomb	Flood SMTP server
processtable	Server flood exhausts UNIX process
udpstorm	Echo / chargen loop flood
apache2	Crash web server with long request
worm	Attacker attack in different by using by storm worm botnet

Table A2. Probe: Trying to avoid network security control by gathering information about a network of computers [39].

Attack type	Description
satan	Automate the process of testing systems for known vulnerabilities that can be exploited via the network
ipsweep	Stealthy slow-scan those exists IP address
nmap	Discover hosts and services on a computer network by sending packets and analyzing the responses.
portsweep	Stealthy slow-scan the status of ports
mscan	Test multiple vulnerabilities
saint	It screens every live system on a network for TCP and UDP services to detect anything that could allow an attacker to gain unauthorized access, create a denial-of-service, or gain sensitive information about the network.

Table A3. Remote to Local (R2L): Unauthorized access from a remote machine, it will intrudes into a remote machine and gains local access to the victim machine [39].

Attack type	Description
guess_password	Dictionary password guessing
phf	Exploited bad Apache CGI script
imap	It makes the mailbox server buffer overflow.
ftp_write	Upload "+ +" to .rhosts to bypass the firewall
multihop	The attacker attacks the victim's remote machine by other victim machines
warezmaster	Access the victim machine through the warez application that user has downloaded
sendmail	SMTP mail server buffer overflow
xsnoop	keystrokes intercepted on open X server
xlock	Steals password through the fake screensaver
snmpgetattack	The attacker steals the community string which is transmitted by clear-text in the early SNMP version to access the machine.
named	DNS nameserver buffer overflow
httptunnel	Backdoor disguised as web traffic
snmpguess	Guess the SNMP community string to access the remote machine in an unauthorized process

Table A4. User to Root (U2R): Unauthorized access to root privileges is an attack type, which login into a victim system and tries to gain root privileges [39].

Attack type	Description
buffer_overflow	In the core of Unix, it was found that the highest level of authority of the system can be obtained by buffer_overflow
xterm	UNIX suid root buffer overflow
loadmodule	UNIX trojan shared library
rootkit	Is a collection of computer software, typically malicious, designed to enable access to a computer or an area of its software that is not allowed
perl	UNIX bug exploit
sqlattack	Database app bug, escape to user shell
ps	UNIX bug exploit

Table A5. Attacks in NSL-KDD dataset.

Category	Sub-labels in training set	Sub-labels in testing set
DoS	back, neptune, smurf, teardrop, land, pod	back, land, neptune, pod smurf, teardrop, mailbomb, processtable, udpstorm, apache2, worm
Probe	satan, nmap, portsweep, ipsweep	satan, ipsweep, nmap, portsweep, mscan, saint
R2L	warezmaster, phf, imap, ftp_write, spy, warezclient, guess_password, multihop	guess_password, phf, imap, ftp_write, multihop, warezmaster, sendmail, xsnoop, xlock, snmpgetattack, named, httptunnel, snmpguess
U2R	rootkit, buffer_overflow, perl, loadmodule	buffer_overflow, xterm, loadmodule, rootkit, perl, sqlattack, ps