Revisiting Traffic Anomaly Detection using Software Defined Networking

Syed Akbar Mehdi[1], Junaid Khalid[1], and Syed Ali Khayam[12]

[1] School of Electrical Engineering and Computer Science
National University of Sciences and Technology (NUST)
Islamabad, Pakistan.

[2] XFlow Research
Santa Clara, CA 95051, USA.
{akbar.mehdi,junaid.khalid,ali.khayam}@seecs.nust.edu.pk

Abstract. Despite their exponential growth, home and small office/home office networks continue to be poorly managed. Consequently, security of hosts in most home networks is easily compromised and these hosts are in turn used for largescale malicious activities without the home users' knowledge. We argue that the advent of Software Defined Networking (SDN) provides a unique opportunity to effectively detect and contain network security problems in home and home office networks. We show how four prominent traffic anomaly detection algorithms can be implemented in an SDN context using Openflow compliant switches and NOX as a controller. Our experiments indicate that these algorithms are significantly more accurate in identifying malicious activities in the home networks as compared to the ISP. Furthermore, the efficiency analysis of our SDN implementations on a programmable home network router indicates that the anomaly detectors can operate at line rates without introducing any performance penalties for the home network traffic.

Keywords: Anomaly detection, Network Security, Software Defined Networking, Programmable Networks, Openflow

1 Introduction

Over the last decade, widespread penetration of broadband Internet in the home market has resulted in an explosive growth of SOHO (Small Office/Home Office) and purely home networks. Since users operating such networks are not well versed in computer networking, security of these networks is generally unmanaged or poorly managed. Computers in such networks are often compromised with malware, mostly without the knowledge of the user.[3] On a local level,

[3] A recent report released by the Anti-Phishing Working Group (APWG) [11] shows that malware infections have reached an alarming coverage of around 48% (among 21.5 million scanned computers) for the last two quarters of 2009, and that around 24% of all the detected malware can be linked to financial crimes.

such malware creates problems for the home users, like hogging their network bandwidth, leaking confidential information, or tampering financial transactions. More importantly, criminal elements often employ these compromised machines as zombies to carry out Internet-wide malicious activities. With the highly diverse hardware and software deployed in home networks, ISPs have little visibility into or control over home networks' traffic. Consequently, instead of resolving the problem at its roots (i.e., the home network), ISPs resort to traffic monitoring and security policing in the network core.

We argue that the emerging concept of Software Defined Networking (SDN)[4] offers a natural opportunity to delegate the task of network security to the home network while sparing the home user from complex security management tasks. Implementation of such a network can be easily achieved using switches supporting the OpenFlow protocol [27], which allows a network controller (e.g. NOX [20]) to programatically control the forwarding behavior of the switch. While benefits of SDN are thus far being envisioned for core network environments, we advocate the use of this technology in the home network where it offers the flexibility to achieve highly accurate security policing, line rate operation (due to low traffic rates), and, most importantly, delegation of security policy implementation to downstream networks. *Our hypothesis is that a programmable home network router provides the ideal platform and location in the network for detecting security problems.*

To put theory to practice, we revisit a security solution which has been explored (rather unsuccessfully) in the past: Deployment of an Anomaly Detection System (ADS) in the network core. ADSs model the normal traffic behavior of a network and flag significant deviations from this behavioral model as anomalies. Many ADSs have been proposed during the last few years to detect traffic anomalies in the network core [19, 21, 28, 30, 31]. However, deployment of ADSs in the network core has been plagued by two problems: 1) *Low detection rates*: While some systems can provide high detection rates [12], they are not usable in a practical setting because they generate a large number of false positives; 2) *Inability to run ADS algorithms at line rates in the network core*: Packet and flow sampling [2–4, 22, 24] is being used to mitigate this problem, but sampling further degrades anomaly detection accuracy by distorting important traffic features [13, 26].

Both of the above problems can be mitigated if anomaly detection is performed close to the anomalous sources, i.e. the endpoints in SOHO or home networks. Thus far, a main obstacle in deployment of security systems in home networks is that switches and routers run proprietary software and are closed to outside control. Moreover, most home users either cannot or do not want to be burdened with the task of configuring and installing such systems. We believe that Software Defined Networking can provide a viable solution to this problem. The primary benefit provided by SDN is *standardized programmability* i.e. once a solution is developed it can be deployed to a diverse range of networking

[4] The SDN concept was originally proposed in [20] and was recently defined more formally in [23].

Table 1: Header fields in an Openflow table entry [7].

Ingress Port	Ether source	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	src port	dst port

hardware which complies with the SDN technology (e.g. Openflow). Furthermore any improvements in the switching hardware or any changes to the SDN implementations are easy to integrate because of the standardized abstraction layer.

In order to test our hypothesis, we ask the following questions: 1) Can existing anomaly detection algorithms be ported faithfully to a Software Defined Network (SDN)? 2) How easy is it to implement these algorithms in a SDN and what benefits do we get from doing so? 3) How much accuracy degradation (if any) is caused when an anomaly detector is run in the ISP versus running the same detector in a home/SOHO network? 4) Is there any difference in accuracy between the SDN and standard implementations running in a home/SOHO network? 5) Are there any performance problems when running the SDN implementations at line rate, without any traffic sampling, in a home/SOHO network?

This paper makes the following contributions to answer these questions:

– We show how four prominent anomaly detection algorithms can be implemented in the NOX SDN controller.[5]
– We perform a detailed accuracy evaluation of our implementations on real-world traffic datasets collected at three different network deployment points: the edge router of an ISP, the switch of a research lab (simulating a small office), and a home network router. Traffic anomalies (including portscans and DoS attacks) are injected at varying rates into the collected datasets. We show that these algorithms fail to mine anomalies with satisfactory accuracy in the ISP level dataset but are able to provide *highly accurate detection* in the home network and small office datasets.
– We provide efficiency evaluation of our SDN implementations on the home and SOHO network datasets showing that, in addition to providing better accuracy, our approach allows line rate anomaly detection.

2 Background and Related Work

2.1 Background: Software Defined Networking

Software Defined Networking (SDN) has recently emerged as a powerful new paradigm for enabling innovation in networking research and development. The

[5] Open-source NOX implementations of the anomaly detectors and sanitized traffic datasets are released publicly [www.wisnet.seecs.nust.edu.pk/downloads.php] to facilitate future research in this area.

basic idea is to separate the control plane of the network from the data plane. While there has been significant previous work in this domain [14, 16, 17], recently Openflow [7, 27] has become the standard bearer for the SDN paradigm.

Openflow is a protocol that allows switches and routers containing internal *flow tables* to be managed by an external *controller*. Each flow table inside a switch contains a set of flow entries. Each flow entry contains a header (against which incoming packets are matched) as well as a set of zero or more actions to apply to matching packets. All packets processed by the switch are compared against its flow tables. If a matching flow entry is found, any actions from that entry are performed on the packet. If no matching entry is found, the packet is forwarded to the controller. The controller may decide to install flows in the switch at this point based on the packet's header. It can also forward the packet through the switch without setting flows.

Table 1 shows the header fields that are present in a flow. Each of these header fields can be wildcarded (i.e. it is ignored while matching packets). This allows flexibility in specifying the exact group of packets on which a certain set of actions is to be performed. For instance, assume a flow entry which has all fields wildcarded except *Ethertype* (set to IP), *IP proto* (set to TCP) and *dst port* (set to 80). If we specify the action for this flow entry as "forward out of port 1", then all traffic destined towards any http server will be forwarded out of port 1 of the switch. Readers interested in further details are referred to the Openflow switch specification v1.0 [7].[6]

Many SDN controllers are now being developed to manage Openflow compliant switches [20, 23, 33, 29]. The basic idea in all of these controllers is to centralize the observation of network state, decide appropriate policies based on this state observation and then enforce these policies by installing flow entries in the switches. NOX [20] and Maestro [33] both provide the concept of a "Network Operating System". Other control platforms like Onix [23] focus primarily on building the control plane as a distributed platform in order to enhance scalability and reliability.

2.2 Related Work

To the best of our knowledge, this work represents the first effort to implement and evaluate existing anomaly detection algorithms in the SDN context. Recently, however, there has been interest in building better solutions for effective management of home networks while hiding the complexity from the home users [1]. Part of this focus has been on improving security and privacy in home networks. Yang *et al.* [32] performed a user study and found that non-expert users mostly relied on OS-based firewall software as opposed to expert users who relied on firewalls built into the router. The primary reason was the inability of non-expert users to configure the router based firewalls. Calvert *et al.* [15] point

[6] At the time of the writing of this paper, Openflow Specification v1.1 [8] has also been released. However, since it has not been widely implemented, our work is based on OpenFlow 1.0.

out many of the difficulties in home network management due to the end-to-end nature of the Internet architecture. They identify security issues, like bot infections, as one of the major problems and attribute them to inability of users to properly set up and integrate networking gear into their home networks. They propose a “smart middle, smart ends” approach which utilizes an intelligent network, as opposed to the “dumb middle, smart ends” approach that forms the basis of the Internet architecture today. Feamster [18] proposes an architecture for home network security which outsources the management and operations of these networks to a third party (e.g. ISP) which has a broader view of network traffic. He suggests the use of programmable network switches which are managed by a centralized controller (located at the ISP). The controller applies distributed inference to detect performance and security problems in the local networks.

3 Anomaly Detection in Software Defined Networks

In this section, we describe implementations of four prominent traffic anomaly detection algorithms in the context of a Software Defined Network (SDN). The aim is to answer the first question asked at the end of Section 1, i.e. is it possible to faithfully implement diverse anomaly detectors in a SDN? Although we assume a setup having a switch running the Openflow protocol with NOX as an external controller, the ideas presented here are easily applicable to any similar SDN paradigm. Similarly, the algorithms ported in this work [19, 25, 28, 31] are simply proofs-of-concept to illustrate the flexibility and ease of anomaly detection algorithm implementation on an SDN platform. We show that the OpenFlow protocol allows us to implement these algorithms within NOX to attain the same accuracy achievable by inspecting every packet, while in fact processing only a small fraction of the total traffic. The main idea is to install flows in the switch whenever a connection attempt succeeds. As a result, only the relevant packets (e.g. TCP SYNs and SYNACKs) within a connection are sent to the controller while all other packets are processed at line rate in the switch hardware.

3.1 Threshold Random Walk with Credit Based Rate Limiting

The TRW-CB algorithm [28] detects scanning worm infections on a host by noting that the probability of a connection attempt being a success should be much higher for a benign host than a malicious one. TRW-CB leverages this observation using sequential hypothesis testing (i.e. likelihood ratio test) to classify whether or not the internal host has a scanning infection. For each internal host, the algorithm maintains a queue of new connection initiations (i.e. TCP SYNs) which have yet to receive a response (i.e. a SYNACK). Whenever one of these connections times out without any reply or receives a TCP RST, the algorithm dequeues it from the queue and increases the likelihood ratio of the host which initiated the connection (i.e. closer to being declared as infected). On the other hand, when a successful reply is received, the likelihood ratio is decreased (i.e.

closer to being declared benign). Whenever the likelihood ratio for a host exceeds a certain threshold η_1, it is declared as infected.

In order to implement TRW-CB in NOX, we employ the fact that the packets this algorithm uses are either connection initiations or replies to them. Thus there is no need to check every packet. The OpenFlow Switch Specification v1.0 [7] indicates that a packet not matching any flow entry in the switch is sent to the controller. We leverage this facility to allow packets related to connection establishment to be sent to the controller. When a connection is established successfully, we install flows in the switch to handle the rest of the packets in the session. The following example explains our implementation:

1. Suppose that internal host **A** sends a TCP SYN to a new external host **B**. Since there are no flows in the switch matching this packet, it will be sent to the NOX controller.

2. The TRW-CB instance running at the NOX controller simply forwards this packet through the switch, without setting any flows. At the same time, the algorithm also does its normal processing (i.e. adds **B** to a list of hosts previously contacted by **A** and adds the connection request to **A**'s queue).

3. The two possible responses from **B** are:

   (a) If a TCP SYNACK from **B** to **A** is received, the switch again forwards this to the NOX controller (since it still does not match any flows). Upon receiving the SYNACK, the TRW-CB instance at the controller installs two flows in the switch. The first flow matches all packets sent from **A** to **B**. It contains **A**'s IP address in the *IP src* field and **B**'s IP address in the *IP dst* field. Except for *Ether type* (which is set to IP), all other fields in the flow are wildcarded. The second flow is similar to the first, but matches all packets sent from **B** to **A**. Each flow contains an action to forward matching packets out of the relevant port of the switch. Additionally, TRW-CB also does its normal processing (i.e. removing this connection request from **A**'s queue and decreasing **A**'s likelihood ratio).

   (b) If the connection times out, then TRW-CB does its regular processing (for the connection failure case) without interacting with the switch. Thus no flows are installed.

Our decision to wildcard all fields in the flow other than *IP src*, *IP dst* and *Ethernet Type*, is a carefully considered one. TRW-CB detects "horizontal" portscans (i.e. across different external hosts as opposed to different ports on the same external host) and is interested only if an internal host is sending new connection requests to *different* external hosts. Typically a benign host can open multiple TCP connections (each with a different source port) to the same server. The destination port may be the same for all requests (e.g. an http server) or it may be different for each request. If each successful connection to the same external host had a seperate flow in the switch it would wastefully increase the size of the flow tables and consequently the switch's processing overhead. Our scheme ensures that only two flows are installed between two communicating hosts. Any new connection requests to the *same external host*, after the first request, will be forwarded at line rate by the switch without passing through the controller. However, any connection initiations to a *new external host* will still be sent first to the controller. This allows TRW-CB to attain the same accuracy achievable by inspecting every packet while actually processing only a fraction of the total traffic.

3.2 Rate-Limiting

Rate Limiting [30, 31] uses the observation that during virus propagation an infected machine attempts to connect to many different machines in a short span of time. On the other hand, an uninfected machine makes connections at a lower rate and is more likely to repeat connection attempts to recently accessed machines. Whenever a new connection request arrives, it is checked against a list of recently contacted hosts called the "working set". If the request is to a host present in the working set, then it is forwarded normally. Otherwise, it is enqueued in another data structure called the "delay queue". Every d seconds a connection is taken out of the delay queue and allowed to proceed forward. If the size of the delay queue increases beyond a threshold T, an alarm is raised.

To implement Rate Limiting in NOX, we again employ the same idea used in TRW-CB that there is no need for the controller to inspect every packet. We implement separate pairs of working sets and delay queues for every internal host. The following rules are applied to packets arriving at the controller:

1. Whenever a new connection request arrives and the remote host is in the working set, we set two flows in either direction between the internal host and the remote host. Rate Limiting like TRW-CB is a host based anomaly detector and does not care about traffic to specific ports. Therefore the flows are also similar (i.e. everything except *IP src*, *IP dst* and *Ether type* is wildcarded).
2. If a new connection request arrives and the remote host is *not* in the working set, we enqueue it into the delay queue. No flows are installed in the switch in this case.
3. Every d seconds a new connection request is moved from the delay queue to the working set. We forward this connection request through the switch without installing any flows.
4. Whenever a positive connection reply (e.g. TCP SYNACK) arrives at the switch, we again install two flows in either direction to handle the rest of the traffic for this connection.

3.3 Maximum Entropy Detector

The Maximum Entropy detector [19] estimates the benign traffic distribution using maximum entropy estimation. Training traffic is divided into 2,348 packet classes and maximum entropy estimation is then used to develop a baseline benign distribution for each class. Packet classes are derived from two dimensions. The first dimension contains four classes (i.e. TCP, UDP, TCP SYN and TCP RST). In the second dimension each of these four classes is split into 587 subclasses based on destination port numbers. Packet class distributions observed in real-time windows (each of duration t secs) are then compared with the baseline distribution using the Kullback-Leibler(KL) divergence measure. An alarm is raised if a packet class' KL divergence exceeds a threshold η_{k}, more than h times in the last W windows.

Unlike TRW-CB and Rate Limiting, Maximum Entropy relies on examining every packet in order to build packet class distributions every t seconds. One approach would be to actually make every packet pass through the algorithm instance running inside the NOX controller. However, this approach will negate Openflow's efficiency benefits. Instead, we use an indirect approach to achieve the same results.

Whenever the switch receives a packet which does not match any flows, it is forwarded to the NOX controller where our algorithm takes one the following actions (based on the packet type):

1. If a TCP SYN or RST packet is received, the count for the relevant packet class, in the current distribution window, is incremented. Next the packet is forwarded through the switch without setting any flows.

2. If a TCP SYNACK packet is received, we install two flows (each handling traffic in one direction) in addition to forwarding the packet through the switch. The flows contain values for the six-tuple consisting of *Ether type* (set to IP), *IP src*, *IP dst*, *IP proto* (set to TCP), *src port*, *dst port*. Other fields are wildcarded.

3. If a UDP packet is received, we install two flows similar to the SYNACK case. Moreover, the count for the relevant UDP packet class is incremented.

As mentioned earlier Maximum Entropy requires the inspection of every packet for building class distributions. The algorithm described above only takes care completely of the TCP SYN and TCP RST classes. The Openflow specification v1.0 [7] states that packet counters are maintained per-flow in the switch. We use this facility to build packet distributions for the rest of the traffic. We schedule a timer function to be called every t seconds, to sequentially perform the following operations:

1. The switch is queried for information on all currently installed flows along with their packet counts.
2. For each flow returned by the switch:
   (a) First the packet class for this flow is determined using its *IP proto* and *dst port* fields.
   (b) Next the packet count for this flow during the last t seconds is determined. Note that the switch returns the cumulative count of all packets

matching this flow since it was installed. In order to calculate count for the last t seconds, we maintain a shadow copy of the switch's flow table along with the last updated packet counts. We subtract each flow's packet count in our shadow copy from the value returned by the switch to get the count for the last t seconds. The shadow copy's packet count is also updated to the latest value at this stage.

(c) Finally the count for packet class determined in the step (a) is incremented with the value determined in step (b).

Note that all the above operations can be easily implemented using OpenFlow. Also, note that in the above case we set a more specific flow than TRW-CB. The reason is that Maximum Entropy builds packet class distributions depending on specific destination ports. We need to treat different connections between two machines separately in order to build correct distributions. In addition, we cannot set flows like “match all packets destined towards TCP port 80”, because that would result in new TCP SYN requests to port 80 on any remote machine being forwarded by the switch without the controller's intervention. This would result in an incorrect packet class distribution for the TCP SYN class.

3.4 NETAD

NETAD [25] operates on rule-based filtered traffic in a modeled subset of common protocols. The filter removes “uninteresting traffic” based on the premise that the first few packets of a connection request are sufficient for traffic anomaly detection. This includes all non-IP packets, all incoming traffic, TCP packets starting after the first 100 bytes and packets to any address/port/protocol combination if more than 16 are received in a minute. It computes a packet score depending on the time and frequency of each byte of packet. Rare and novel header values are assigned high scores. A threshold is applied on a packets score to find anomalous packets.

In order to implement NETAD in NOX, we make use of its filtering stage. The filtering rules imply that typically only the first few packets of a connection will be used by the algorithm. Therefore we base our implementation on the constraint that all packets must pass through the controller unless they satisfy one of the filtering rules. If a rule is satisfied, then we install flows in the switch to forward the rest of the traffic without controller intervention. For instance, since the algorithm specifies that it is not interested in TCP packets starting after the first 100 bytes, we check the sequence number of TCP packets passing through the controller. In case the sequence number of a packet exceeds 100, we install two flows in the switch which correspond to either direction of this packet's connection. The flows in each case contain values for the six-tuple consisting of *Ether type* (set to IP), *IP src*, *IP dst*, *IP proto* (set to TCP), *src port*, *dst port*. Similar to Maximum Entropy, our flows are more specific in this case because the algorithm requires the first few packets of every new connection even if the two hosts already have connections on other ports.

Table 2: Statistics of Benign Traffic Datasets

Dataset Type	Active Hosts	Total Connections	Total Packets	Avg. Connections per sec	Avg. Pkts per sec	Duration
Home	8	3,422	1,042,282	0.21	62.36	21hrs 13mins
SOHO	29	50,082	15,570,794	2.61	320.4	5hrs 28mins
ISP	639	304,914	28,152,467	523	12,210	9 mins

4 Dataset Description

In order to effectively evaluate our hypothesis that home networks provide better accuracy for anomaly detection, we needed network traffic captured at different points in the network for comparison. Furthermore, for comprehensive evaluation, we needed attacks of different types (DoS, portscan, etc.) and different rates for each attack type. We decided to collect our own benign and attack datasets and make them available for repeatable performance evaluations.[7]

4.1 Benign Network Traffic

We collected benign traffic at three different locations in the network. Our aim was to study the accuracy of anomaly detection algorithms in a typical home network, a small-office/home-office network and an Internet Service Provider (ISP). These algorithms have been sanitized using the `tcpmkpub` tool. Moreover, due to privacy concerns, only the first 70 bytes of each packet are available in all the datasets. Some statistics about each dataset are given in Table 2.

Home Network The home network dataset was collected in an actual residential setting. The data was collected over a period of one day (approx. 21 hrs 13 minutes). Eight different hosts were active during this time with varying levels of activity during the day. We collected data from each of these hosts individually before merging it together using `mergepcap`. Various applications including file transfer, web browsing, instant messaging, real-time video streaming were active during this period.

Small Office/Home Office(SOHO) For the SOHO dataset we gathered data from a research lab in our School of EECS. The traffic from the lab is relayed through a 3COM4500G switch. We mirrored all the traffic to one of the ports of the switch where it was captured and saved in pcap format. The data was collected over a period of approximately 5.5 hours from 1340 hrs to 1908 hrs on a working day. During this time, 29 unique hosts were active.

[7] The datasets are available at http://www.wisnet.seecs.nust.edu.pk/downloads.php.

Internet Service Provider The ISP dataset was collected from the edge router of a medium-sized Internet Service Provider for about 10 minutes from 1741 hrs to 1750 hrs on a working day. During this time, 639 hosts were active.

4.2 Attack Traffic

In order to collect the attack traffic, we launched (TCP SYN), DoS (TCP SYN) and fraggle (UDP flood) simultaneously from three end hosts in our research lab. Each host launched attacks at various rates including three low-rate (0.1,1,10 pkts/sec) and two high-rate (100,1000 pkts/sec) instances. Each instance had a period of two minutes. We labeled each attack packet by setting the Reserved bit in the IP header. The TCP portscan attack contains two distinct attacks with the first one targeting port 80 and the second targeting port 135. The TCP SYN Flood consists of attacks on two remote servers at ports 143, 22, 138, 137 and 21. Similarly the UDP flood also attacks two remote servers at ports 22, 80, 135 and 143.

After collecting the attack traffic, we randomly merged it into different hosts in each of the benign datasets using the `mergepcap` tool. In the home dataset, 4 out of 8 hosts were infected, while 8 out of 29 hosts were infected in the SOHO dataset. In the ISP dataset, 24 out of 639 hosts were infected. We used the same attack traffic in all three datasets in order to maintain a uniform evaluation base.

5 Evaluation

This section focuses on answering questions 2-5 asked in Section 1. We investigate the effectiveness and ease of implementation, accuracy comparison between ISP and home/SOHO network datasets and efficiency of the SDN implementations described in Section 3.

5.1 Experimental Setup

We implemented all four algorithms in NOX using C++. For comparison, we also did standard implementations of all algorithms which examine every packet (i.e. no sampling). For the Home Network and SOHO datasets, we performed accuracy evaluation using both the NOX implementations as well as the standard implementations. We observed that there was little or no difference in accuracy between the two evaluations. This was expected, since our objective with the NOX implementations was to ensure that each algorithm attained the same accuracy achievable by inspecting every packet, while in effect most of the packets are forwarded at line rate by the switch without any intervention by the controller. The accuracy results we present for these two datasets can therefore be assumed to represent either implementation. The ISP dataset was tested solely on the standard implementation.

In the case of Maximum Entropy and NETAD which require training on network traffic, we use some traffic from the benign datasets. For instance, before

Table 3: Source Lines of Code(SLOC) for NOX and Standard Implementations

Algorithm	NOX Implementation	Standard Implementation
TRW-CB	741	1060
MaxEnt	510	917
RateLimit	814	991
NETAD	587	944

evaluating on the Home Networks dataset we train each algorithm on the first 40 minutes of data. For the SOHO dataset, the training period is the first 30 minutes while for the ISP dataset we train on the first 1 min of data.

For accuracy evaluations, our testbed consisted of a server with a quad-core Intel Xeon E5420 CPU and 8 GB of RAM running Ubuntu Linux 10.04. It ran Open vSwitch v1.1.0 [6] which implements the Openflow v1.0 protocol. A NOX 0.9 (Zaku) controller was located on the same machine and communicated with the Open vSwitch daemon through a Unix domain socket. Another machine was used to replay traffic from our merged datasets and send it to our server.

For efficiency evaluation of the Home Network data, we decided to use a realistic setting. We assembled a small form-factor wireless home router consisting of PC Engines ALIX 2c3 [9] board with a 500 MHz AMD Geode LX800 processor, 256 MB DDR DRAM and 2GB flash memory. We call this system the "NOX Box" [5]. We installed Voyage Linux [10] on the NOX Box, which is a Debian-derived distribution designed to run on low-end x86 PC platforms. In addition, we installed Open vSwitch v1.1.0 and NOX 0.9 on it, to create a fully equipped home network router.

5.2 Ease of Implementation

Table 3 compares the NOX and Standard implementations of the four algorithms based on Source lines of code (SLOC). We observe that NOX implementations take on average 660 SLOC, while the standard implementations take on average 980 SLOC. Thus in terms of programming effort at least, it is easier to implement in NOX. This is partly due to the fact that NOX already has a significant infrastructure in place for packet processing as well as communicating with the switch. For the standard implementations, however, we had to build our own packet processing infrastructure.

However, the main benefit of the NOX implementations is not immediately evident from these numbers. The standard implementations work completely in userspace and make a system call each time they send and receive a packet. This does not scale well to high data rates. On the other hand, the NOX implementations get the fast datapath provided by the Openflow-compliant switches for free, while actually requiring lesser effort to implement. This datapath may be a hardware TCAM or a linux kernel module depending on the type of switch used. We show in Section 5.4 that our NOX implementations direct more than

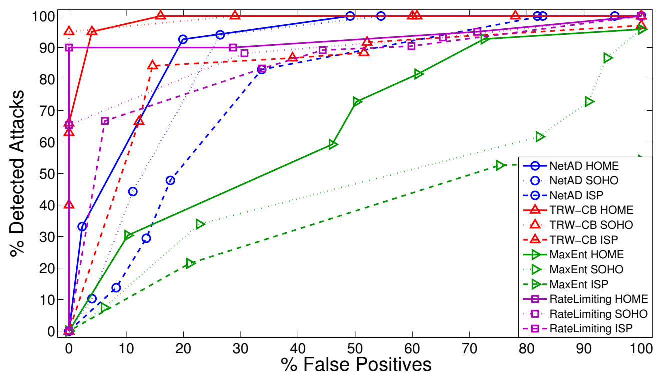


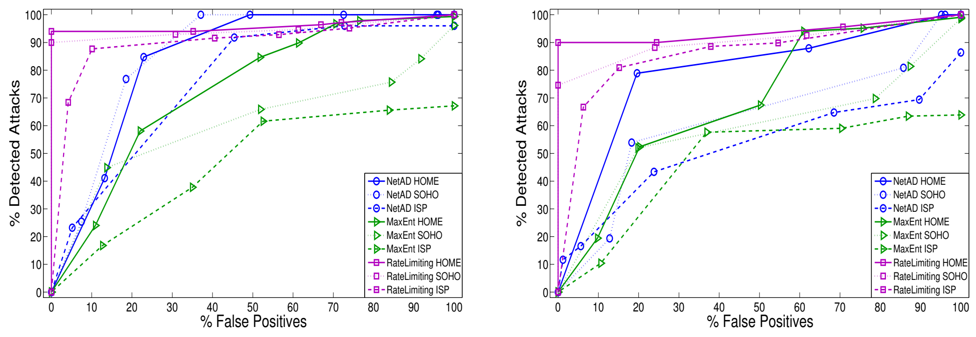
Fig. 1: ROC curves quantifying the accuracy of all four algorithms under TCP portscan attacks; each point is averaged over 5 attack rates.

98% of the network traffic through this fast datapath simply by installing flows in the switch. The standardized Openflow interface allows the use of the same NOX implementations regardless of the underlying switching device used. This makes it easier to update the implementations while maintaining efficiency and portability.

5.3 Accuracy Evaluation

TCP Portscan Attacks Figure 1 shows the Receiver Operating Characteristic (ROC) curves of TCP portscan detection for each of the four algorithms evaluated on all three datasets under varying attack rates. All algorithms perform much better on the Home Network dataset as compared to the ISP dataset. The performance on the SOHO dataset mostly falls in between the performance on these two datasets. This supports our assumption that detection accuracy degrades continuously as we move away from the endpoints towards the network core.

TRW-CB and Rate Limiting perform better than the other two algorithms in almost all situations. Both of these are host-based algorithms and use outgoing connections as the key detection feature, which is specifically suitable for TCP Portscans. Their performance on the home network dataset is excellent: both achieve 90% or better accuracy for a false positive (FP) rate of 0% to 4%. TRW-CB maintains the same accuracy on the SOHO dataset while the accuracy of Rate Limiting degrades significantly and it manages a 90% detection rate for an FP rate of 30%. On the ISP dataset, they both suffer significant accuracy degradation; the best accuracy point is TRW-CB which manages to attain an 85% detection rate for a significant FP rate increase of 11%. The reason for the more rapid accuracy degradation of Rate Limiting is its decline in performance on low-rate scanners. This is apparent from Figures 3a and 3b which show separate ROCs for low and high rate scanners. We observe that both Rate-Limiting and TRW-CB achieve perfect results for high-rate scanners on all datasets. For low-



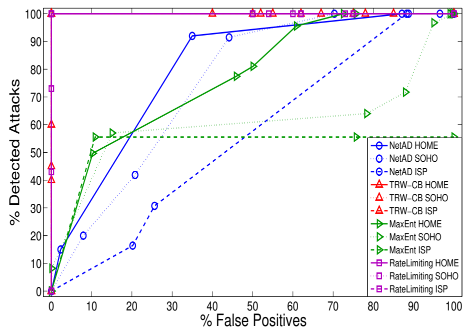
(a) TCP Flood (b) UDP Flood

Fig. 2: ROC curves for evaluating the accuracy of all four algorithms under TCP and UDP Flood attacks; each point is averaged over 5 attack rates.

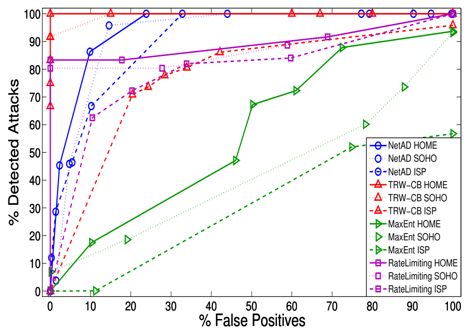
rate scanners evaluated on the Home and SOHO datasets, TRW-CB maintains the same accuracy while Rate Limiting is only able to achieve 80% detection for a 0% FP rate. The reason is that, while Rate Limiting detects port scanners primarily on the basis of rate of new connection attempts, TRW-CB relies on the success or failure of successive connection attempts.

In comparison, both Maximum Entropy and NETAD fail to take advantage of the home network environment. While both algorithms show better performance on the home datasets when compared with their respective accuracies on the ISP datasets, the accuracy improvement is not of the same order as that of TRW-CB and Rate Limiting. As Figure 1 indicates, Maximum Entropy shows the worst performance on all datasets. On the home network dataset, it achieves a maximum detection accuracy of 90% but at a cost of 70% false positives. NETAD performs considerably better as it achieves the same detection accuracy for a 20% false positive rate. Unlike TRW-CB and Rate Limiting, none of these algorithms are host based and do not directly use outgoing connection information. Both Maximum Entropy and NETAD use baseline models of network traffic using predefined attributes and detect deviations from these attributes. They are sensitive to changes in the pattern of benign network traffic. Maximum Entropy is more rigid in this regard because it builds baseline distributions of packet classes. For instance, if a host starts using a P2P application but the algorithm was trained on data containing mostly web traffic, it will result in higher FP rates. This is true both in Home networks as well as ISPs. The reason for the difference in Maximum Entropy's performance between the Home and ISP network is its poor detection of low-rate scanners as evident from Figure 3b. The high background traffic rate of an ISP network means that low rate anomalies do not create a significant enough divergence from the baseline distribution to raise an alarm.

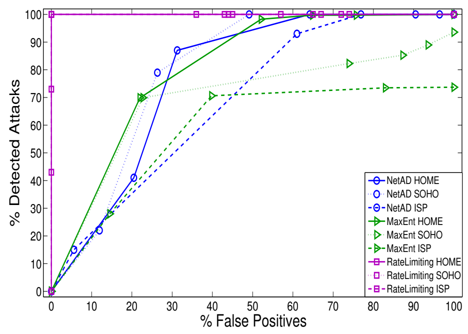
TCP and UDP Flood Attacks Figures 2a and 2b show the ROC curves for TCP and UDP Flood attacks respectively. We did not evaluate TRW-CB for



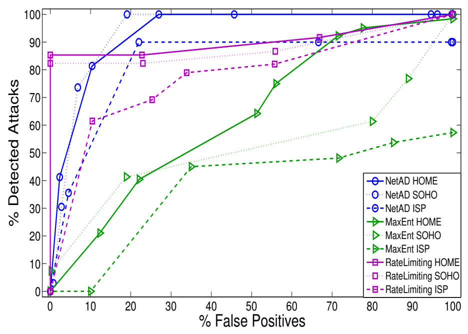
(a) TCP Portscan High Rate



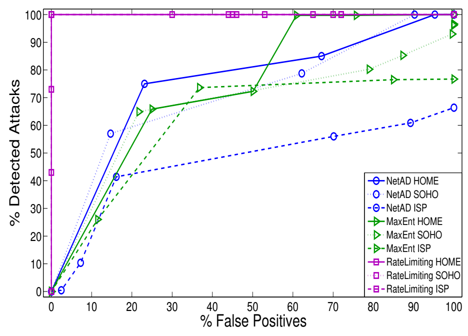
(b) TCP Portscan Low Rate



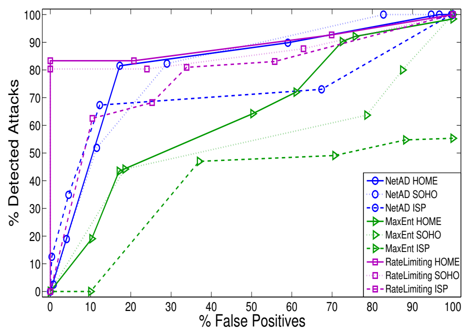
(c) TCP Flood High Rate



(d) TCP Flood Low Rate



(e) UDP Flood High Rate



(f) UDP Flood Low Rate

Fig. 3: Separate ROC results for high rate (100 and 1000 per sec) and low rate (0.1, 1 and 10 per sec) TCP Portscan, TCP Flood and UDP Flood attacks.

these attacks because it is primarily a portscan detector focused on detecting "horizontal scans" of different remote hosts. It is clear from the ROC curves that Rate Limiting provides excellent accuracy on the home and SOHO datasets for both types of flood attacks, while maintaining a low false positive rate. Rate Limiting detects anomalies if the size of its delay queue increases beyond a threshold. Because of the low rate of benign connection requests in a typical home network environment, we give a small value to the size of the working set (of recently made connections). Thus a sudden flood of new TCP SYNs will likely end up in the delay queue and overrun its threshold, raising an alarm. On the other hand, in an ISP network, the rate of benign connection requests is quite high (Table 2). Thus both the working set size and delay queue threshold need to be large in order to reduce unnecessary false positives. This in turn reduces the detection rate.

Maximum Entropy and NETAD show better performance on the home and SOHO networks as compared to the ISP, but their overall accuracy is lower than Rate Limiting. The reason is the same as explained in the TCP portscan case. Both algorithms rely on models of benign network traffic. Since our flood attacks are to well known ports including port 80 and port 22, these algorithms are not able to easily detect divergence from the usual, especially for low rate attacks.

5.4 Efficiency Evaluation

Table 4 shows the efficiency evaluation of our NOX implementations on the Home and SOHO datasets. We compare the algorithms based on four parameters:

1. *Percent of total packets that pass through the controller*: This is an important metric because if a large percentage of traffic is passing through the controller then we do not gain any efficiency benefit from NOX and Openflow. As described earlier, our objective is to ensure that only the relevant packets are handled by the controller and everything else is forwarded at line rate through the switch hardware.
2. *Average rate of packets passing through the controller*: Given that the controller is implemented in software, we need to evaluate whether the rate of packets passing through the controller will cause any performance penalties.
3. *Average number of flows in the switch's flow table*: The switch needs to match every incoming packet against entries in the flow table. We need to measure the average number of flow entries in the switch, at any point in time, for all four algorithms. A larger average flow table size could impact the forwarding efficiency of the switch.
4. *Peak size of the switch's flow table*: We want to know if any of our implementations overflows the switch's flow tables.

As the results clearly show, in most cases, the controller handles less than 2% of the total traffic. The largest value is 3.46 % for NETAD evaluated on the Home Network dataset. This slight increase is due to the fact that, while all other algorithms work on packets related to connection requests, NETAD requires a few more packets to be examined in certain cases. For instance, in the case of TCP it examines all packets for the first 100 bytes of the stream.

The packet rate at the controller is also very manageable in all cases. It lies mostly in the range of 1 to 2 packets per second. Generally, the rate is slightly larger for the SOHO network as compared to the Home network. This is expected

Table 4: Efficiency of NOX implementations on Home and SOHO datasets

Algorithm:Dataset	% of Total Pkts handled by controller	Avg. Pkt rate at controller (pkts/sec)	Avg. No. of entries in Flow Table	Peak No. of entries in Flow Table
TRW-CB : HOME	1.15	0.73	16.11	70
TRW-CB : SOHO	0.37	2.91	42.33	71
MaxEnt : HOME	2.48	1.58	39.72	261
MaxEnt : SOHO	1.26	1.00	172.60	408
RateLimit : HOME	1.00	0.64	16.69	59
RateLimit : SOHO	0.56	4.43	38.28	64
NETAD : HOME	3.46	2.21	24.60	107
NETAD : SOHO	1.07	8.47	74.68	196

since there are more simultaneously active hosts in the SOHO network. Overall, these statistics indicate that there should not be any performance problems at the controller.

The average and peak flow table sizes also show good results for all algorithms. Both TRW-CB and Rate Limiting average about 16 flow entries for the Home Network and about 40 flow entries for the SOHO dataset. The larger value for the SOHO network is due to more simultaneously active hosts. In contrast, Maximum Entropy averages about 40 flow entries on the Home Network and 172 entries on the SOHO network. The larger value for Maximum Entropy is because of the more specific flows we install in the switch. As described in Section 3.3, we set more specific flows for Maximum Entropy because it relies on building distributions of packet classes based on destination port numbers. On the other hand, TRW-CB is a host based detector and does not care about transport layer ports. Thus we can afford to set more general flow entries which can handle a larger number of packets and result in a smaller flow table size.

5.5 CPU Usage

Finally, the results of the performance evaluation of the Home Network Dataset on the NOX BOX (as described in Section-5.1) are shown in Table 5. It shows the average percent CPU Usage of each of the four algorithms for three different data rates. While typical home networks do not have data rates of 50 Mbps or even 10 Mbps, we show the values for comparison and to assess the system's limits. It should be noted that none of these results involved any packet loss.

The results clearly show that for a forwarding rate of 1 Mbps through the NOX BOX, all the algorithms take up a small fraction of the CPU time. Maximum Entropy, which has the highest value, still averages around 3% of CPU time. This increased usage is due to the fact that Maximum Entropy has to fetch all flows in the switch every second, in order to build packet class distributions. As the rate rises to 50 Mbps, the CPU usage of all algorithms increases. However it still remains within very reasonable limits. For instance at 50 Mbps, TRW-CB still takes up only 17.54% of the CPU time. This is quite remarkable because

Table 5: CPU Usage on the NOX BOX for the Home Dataset

Data Rate	Average CPU Usage (%)			
	TRW-CB	Rate Limiting	NETAD	Maximum Entropy
1 Mbps	1.86	2.1	2.94	3.09
10 Mbps	6.70	8.47	10.43	18.43
50 Mbps	17.54	18.87	19.11	28.26

the NOX Box has a low-end 500 MHz processor. We attribute this efficient performance to the fact that the NOX controller running in userspace only handles a small fraction of the traffic at a low rate (as shown in Table 4). Most of the traffic is forwarded by the Open vSwitch kernel module.

6 Conclusions and Future Work

In this paper, we have shown that Software Defined Networks using Openflow and NOX allow flexible, highly accurate, line rate detection of anomalies inside Home and SOHO networks. One of the key benefits of this approach is that the *standardized programmability* of SDN allows these algorithms to exist in the context of a broader framework. We envision a Home Operating System built using SDN, in which our algorithm implementations would co-exist alongside other applications for the home network e.g. QoS and Access Control. The standardized interface provided by a SDN would allow our applications to be updated easily as new security threats emerge while maintaining portability across a broad and diverse range of networking hardware.

This approach could also provide better opportunities for attack mitigation. When an anomaly is detected, our application could communicate it to the ISP where this information could be utilized in a variety of ways. Firstly, it could be used by a human operator to verify the existence of an attack and then inform the home-network owner. Secondly, it could be used to correlate threats from multiple home networks for detecting global network security problems e.g. botnets. Notice that this has several benefits including: 1) The detection is far more accurate in the home networks than the ISP (as shown in Section 5.3). 2) The difficulty of running anomaly detection at high data rates in the ISP's network core is distributed to thousands of home network routers. The efficiency evaluation in Sections 5.4 & 5.5 has already shown that this would be feasible. 3) The programmable nature of SDN allows this sophisticated approach to be implemented in software while still ensuring that almost all the home/SOHO network traffic continues to traverse the fast datapath of the switch possibly in hardware.

7 Acknowledgements

The authors thank Scott Shenker, Murphy McCauley and Kyriakos Zarifis for their continuous feedback throughout the course of this project. We also thank Wahaj-us-Siraj and Jahanzeb Arshad from Micronet Broadband (Pvt) Ltd. for providing the ISP dataset.

References

1. Acm sigcomm workshop on home networks (homenets), `http://conferences.sigcomm.org/sigcomm/2010/HomeNets.php`
2. Arbor networks peakflow-x homepage, `http://www.arbornetworks.com/en/peakflow-x.html`
3. Cisco anomaly guard module homepage, `www.cisco.com/en/US/products/ps6235/`
4. Endace ninjabox homepage, `http://www.endace.com/ninjabox.html`
5. Nox box, `http://noxrepo.org/manual/noxbox.html`
6. Open vswitch, `http://openvswitch.org/`
7. Openflow specification version 1.0.0, `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`
8. Openflow specification version 1.1.0, `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`
9. Pc engines alix 2c3 system board, `http://www.pcengines.ch/alix2c3.htm`
10. Voyage linux, `http://linux.voyage.hk/`
11. Anti-phishingworking group. phishing activity trends report, 4th quarter / 2009 (2010), `http://www.antiphishing.org/reports/apwg_report_Q4_2009.pdf`
12. Ashfaq, A.B., Robert, M.J., Mumtaz, A., Ali, M.Q., Sajjad, A., Khayam, S.A.: A comparative evaluation of anomaly detectors under portscan attacks. In: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection. pp. 351–371. RAID '08, Springer-Verlag, Berlin, Heidelberg (2008)
13. Brauckhoff, D., Tellenbach, B., Wagner, A., May, M., Lakhina, A.: Impact of packet sampling on anomaly detection metrics. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. pp. 159–164. IMC '06, ACM, New York, NY, USA (2006)
14. Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., van der Merwe, J.: Design and implementation of a routing control platform. In: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2. pp. 15–28. NSDI'05, USENIX Association, Berkeley, CA, USA (2005)
15. Calvert, K.L., Keith, W., Rebecca, E., Grinter, E.: Moving toward the middle: The case against the end-to-end argument. In: in Home Networking. Sixth Workshop on Hot Topics in Networks (2007)
16. Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S.: Ethane: taking control of the enterprise. SIGCOMM Comput. Commun. Rev. 37, 1–12 (August 2007)
17. Casado, M., Garfinkel, T., Akella, A., Freedman, M.J., Boneh, D., McKeown, N., Shenker, S.: Sane: a protection architecture for enterprise networks. In: Proceedings of the 15th conference on USENIX Security Symposium - Volume 15. USENIX Association, Berkeley, CA, USA (2006)

18. Feamster, N.: Outsourcing home network security. In: Proceedings of the 2010 ACM SIGCOMM workshop on Home networks. pp. 37–42. HomeNets '10, ACM, New York, NY, USA (2010)
19. Gu, Y., McCallum, A., Towsley, D.: Detecting anomalies in network traffic using maximum entropy estimation. In: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement. pp. 32–32. IMC '05, USENIX Association, Berkeley, CA, USA (2005)
20. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: Nox: towards an operating system for networks. SIGCOMM Comput. Commun. Rev. 38, 105–110 (July 2008)
21. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast portscan detection using sequential hypothesis testing. In: In Proceedings of the IEEE Symposium on Security and Privacy (2004)
22. Kim, M.S., Kong, H.J., Hong, S.C., Chung, S.H., Hong, J.: A flow-based method for abnormal network traffic detection. In: Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP. vol. 1, pp. 599 –612 Vol.1 (2004)
23. Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., Shenker, S.: Onix: a distributed control platform for large-scale production networks. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–6. OSDI'10, USENIX Association, Berkeley, CA, USA (2010)
24. Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions. In: In ACM SIGCOMM. pp. 217–228 (2005)
25. Mahoney, M.V.: Network traffic anomaly detection based on packet bytes. In: Proceedings of the 2003 ACM symposium on Applied computing. pp. 346–350. SAC '03, ACM, New York, NY, USA (2003)
26. Mai, J., Chuah, C.N., Sridharan, A., Ye, T., Zang, H.: Is sampled data sufficient for anomaly detection? In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. pp. 165–176. IMC '06, ACM, New York, NY, USA (2006)
27. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38, 69–74 (March 2008)
28. Schechter, S.E., Jung, J., Berger, A.W.: Fast detection of scanning worm infections. In: RAID. pp. 59–81 (2004)
29. Tootoonchian, A., Ganjali, Y.: Hyperflow: a distributed control plane for openflow. In: Proceedings of the 2010 internet network management conference on Research on enterprise networking. pp. 3–3. INM/WREN'10, USENIX Association, Berkeley, CA, USA (2010)
30. Twycross, J., Williamson, M.M.: Implementing and testing a virus throttle. In: Proceedings of the 12th conference on USENIX Security Symposium - Volume 12. pp. 20–20. USENIX Association, Berkeley, CA, USA (2003)
31. Williamson, M.M.: Throttling viruses: Restricting propagation to defeat malicious mobile code. In: ACSAC (2002)
32. Yang, J., Edwards, W.K.: A study on network management tools of householders. In: Proceedings of the 2010 ACM SIGCOMM workshop on Home networks. pp. 1–6. HomeNets '10, ACM, New York, NY, USA (2010)
33. Zheng Cai, Alan L. Cox, T.S.E.N.: Maestro: A system for scalable openflow control, `http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf`