Journal of Universal Computer Science, vol. 17, no. 11 (2011), 1605-1622
submitted: 14/2/11, accepted: 28/6/11, appeared: 1/7/11 © *J.UCS*

Performance Evaluation of Snort under Windows 7 and Windows Server 2008

Khaled Salah[1]
(Khalifa University of Science, Technology and Research, Sharjah, UAE
khaled.salah@kustar.ac.ae)

Mojeeb-Al-Rhman Al-Khiaty
(King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
alkhiaty@kfupm.edu.sa)

Rashad Ahmed
(King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
rashadyousofi@kfupm.edu.sa)

Adnan Mahdi
(King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
adnanmahdi@kfupm.edu.sa)

Abstract: Snort is the most widely deployed network intrusion detection system (NIDS) worldwide, with millions of downloads to date. PC-based Snort typically runs on either Linux or Windows operating systems. In this paper, we present an experimental evaluation and comparison of the performance of Snort NIDS when running under the two newly released operating systems of Windows 7 and Windows Server 2008. Snort's performance is measured when subjecting a PC host running Snort to both normal and malicious traffic. Snort's performance is evaluated and compared in terms of throughput and packet loss. In order to offer sound interpretations and get a better insight into the behaviour of Snort, we also measure the packet loss encountered at the kernel level. In addition, we study the impact of running Snort under different system configurations which include CPU scheduling priority given to user applications or kernel services, uni and multiprocessor environment, and processor affinity.

Keywords: Network Security, Snort, Operating Systems, Windows 7, Windows 2008, Experimental Performance Evaluation
Categories: C.2.0, C.2.1, C.2.3, C.2.6, C.2.m, D.4.0, D.4.6, D.4.9, D.4.8

1 Introduction

Network intrusions are among the critical security threats that security administrators worry about on a regular basis. Such breaches can potentially expose sensitive and proprietary data to the outside world. Monitoring the network for signs of intrusion is essential for security protection. Network Intrusion Detection System (NIDS) has

[1] Corresponding Author: Prof. K. Salah, PO Box 573, Computer Engineering Department, KUSTAR, Sharjah, UAE, phone: +97165043513 fax: +97165611789

been developed in response to the increasing number of attacks or malware that can misuse the networks. They become vital components of today's network security infrastructure. These NIDSes provide a layer of defense which reads all incoming packets and tries to find suspicious patterns known as signatures or rules. Intrusion detection techniques can be either anomaly detection or misuse detection techniques. Anomaly-detection first establishes a normal behaviour pattern for users, programs or resources in the system, and then looks for deviation from this behaviour [Lan (03)]. On the other hand, misuse (or signature-scan) detection techniques passively monitor traffic seen on a network and detect an attack when patterns within the packet match predefined signatures in a database.

Commercial NIDSes often have high monetary cost (thousands of dollars at minimum, tens or even hundreds of thousands in extreme cases). Snort [Snort (08a)] is a popular lightweight open-source signature-scan based attack detection tool that is publicly available. In fact, it is the most widely deployed intrusion detection technology worldwide, with millions of downloads to date, and it has become the de facto standard for the industry [Snort (08a)]. Snort is typically a PC-based NIDS, but also can be integrated in third-party solutions. Typically, a NIDS is installed on the edge of a network and performs deep packet inspection on every packet that enters the protected network [Lan (03)] against several thousands of attack signatures. The signatures are represented as a set of rules which are frequently updated by the security community. Snort can filter packets based on predefined rules. Each Snort rule operates first on the packet header to check source and destination IP addresses, ports, protocols, etc. If the packet matches a certain header rule, then its payload is examined against a set of predefined patterns.

For a PC-based solution, Snort can run on top of many platforms such as Linux, FreeBSD, and Windows. The preferred choice of the operating system for running Snort NIDS, in today's local area networks of private homes and small- to medium-sized enterprises, is either Linux or Windows. These two platforms are the most widely used for running PC-based high-end servers. In prior work [Salah (05)], the performance of Snort under Windows Server 2003 and Linux has been studied. In this paper, Snort has been empirically evaluated under the two newly released operating systems of Windows 7 and Windows 2008 server. More specifically, the performance of Snort under these two operating systems has been evaluated under UP (uni-processing) and SMP (symmetric multiprocessing) environments, when subjecting Snort to both normal and malicious traffic at different traffic rates.

Under an SMP environment, we investigate the performance of Snort with static and dynamic affinity. The Windows default is dynamic affinity. With dynamic affinity, the affinitzation (or assignment) of Snort to a particular processor is left to the kernel. In this situation, the affinitaizaion is dynamic and there is no guarantee that the execution of Snort will be tied (or affinitized) to a particular processor. With static affinity, the affinitization is fixed, and the execution of Snort will always be tied to a particular processor. For Snort, static affinity can be preferable over dynamic affinity. The reason is that Snort is a single threaded application, and thereby running on the same processor would minimize its context switching which will result in better utilization of the processor's cache or a significant reduction in cache pollution.

The increasing traffic of today's link speeds that go up to tens of Gbps (Gigabits per second) has heightened the need for Snort to be highly effective. The implication

of not doing so can be detrimental, risking the security of the internal network. An intrusion detection system that fails to perform packet inspection at the required rate will allow packets to enter the network undetected. Several studies in the literature addressed the performance limitations of Snort and proposed techniques to boost its overall performance. In [Abbas (02) and Vermeiren (04)], a preliminary study was done to design Snort as a multi-threaded application that takes advantage of today's multi-core architecture. In [Turnbull (07) and Geschke (06)], some improvement in Snort's performance was achieved by offloading some of Snort's essential functions involving alerting and logging, thereby freeing Snort to focus on the primary function of packet inspection. In [Aldwairi (05), Weinsberg (07), Yu (07), Coppens (04), Sourdis (06), Cho (08), Baker (05), Lin (07), Mitra (07)], rule and string matching were substantially improved by using novel optimization techniques coupled with customized FPGA (Field-Programmable Gate Array) and hardware. Speeding up performance was demonstrated using different packet capturing libraries in which Snort can have direct access to the kernel's receiving ring buffer allocated for the NIC (Network Interface Card) [Deri (05) and Snort(08)]. In [Biswas (06)], significant improvement was shown when completely eliminating the traditional network TCP/IP stack and socket interface mechanism. To date, all of these proposed techniques are yet to be adopted by Snort developers. The software architecture and running environment of Snort remain the same. Therefore difficulties and incompatibilities exist in the customization and integration of such solutions, particularly for a typical end user.

The major contribution of this paper is evaluating experimentally and comparing the performance of Snort under the two newly released operating systems of Windows 7 and Windows Server 2008. We consider and investigate the impact of tuning key performance configuration parameters of these two operating systems on the performance of Snort. This will aid in choosing the best configuration parameters to boost and improve Snort's performance. Unlike the work presented in [Abbas (02), Vermeiren (04), Turnbull (07), Geschke (06), Aldwairi (05), Weinsberg (07), Yu (07), Coppens (04), Sourdis (06), Cho (08), Baker (05), Lin (07), Mitra (07), Deri (05) Snort(08), and Biswas (06)] to improve the performance of Snort, our experimental study presented in this paper is focused on determining the best operating system parameters to maximize the performance of Snort. Our improvement involves characterizing the typical execution behaviour and CPU processing requirement of Snort application, and accordingly selecting the best and optimal configuration of those key system parameters. Other contributions of this paper include: (1) analyzing the performance of Snort when running with CPU scheduling with the option of giving preference to user applications or kernel services; (2) evaluating the performance of Snort under UP and SMP environment; (3) studying the impact of setting the processor affinity on the performance of Snort; (4) offering general guidelines and detailed configurations on measuring and evaluating the performance of Snort or any similarly-behaving user application; (5) characterizing the execution behaviour of Snort in order to offer sound interpretation of its performance.

The rest of the paper is organized as follows. Section 2 gives a brief background on Snort's software architecture and running environment. Section 3 describes the experimental setup with configuration details for Windows 7 and Windows Server 2008. The section also describes the installation configuration and setup for Snort.

Section 4 presents performance measurements and comparison for both Windows 7 and Windows Server 2008, with detailed interpretations and analysis. Finally, Section 5 concludes the study and identifies future work.

2 Background

In order to offer sound interpretations of Snort's performance and behaviour, and to recognize key system configurable parameters, we present sufficient background about Snort's software components and its running environment. We also review and discuss the performance limitations of Snort NIDS.

2.1 Snort Software Architecture

To date, Snort is a single threaded application which can be configured to operate in four modes: sniffer, packet logger, network intrusion detection system (NIDS) and intrusion prevention system (IPS). Packet sniffing and logging functions are elementary features of Snort, but Snort's strength and popularity come from its intrusion detection capabilities, and specifically as a NIDS. The IPS is a newly added feature but limited. Itprimarily allows Snort to take preventive actions against malicious traffic such as dropping or re-directing packets to another destination.

As shown in Figure 1, Snort-based NIDS is logically composed of the following major components: Packet Capture Library, Packet Decoder, Preprocessors, Detection Engine, Logging and Alerting System, and Output Modules. These components work together to detect particular attacks and to generate output in a required format from the detection system.

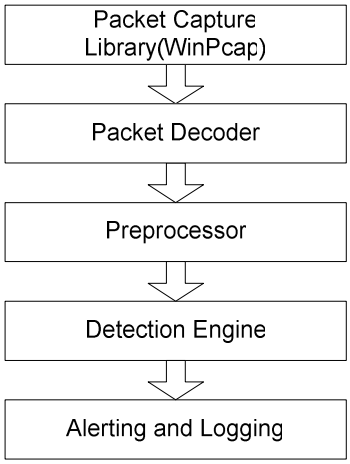

Figure 1: Snort basic software components

Packet Capture Library. The Packet Capture Library (winPcap) is a separate piece of software that reads packets off the network wire and pitches them to Snort. It is the

industry-standard tool for link-layer network access in Windows environments. It allows applications to capture and transmit network packets bypassing the protocol stack. WinPcap is the packet capture and filtering engine of many open source and commercial network tools, including protocol analyzers, network monitors, and network intrusion detection systems.

Packet decoder. The Packet Decoder takes the incoming packet and prepares it to be preprocessed or to be sent to the detection engine. When finished decoding, Snort has all the protocols information in all the right places for further processing.

Preprocessor. Preprocessors are very important components of the intrusion detection system. They are used to protect against different types of attacks which use different techniques to fool NIDS in different ways. For example, NIDS can easily be fooled by a hacker who makes slight modifications to the malicious string to bypass the exact match of Snort rule. Hackers use fragmentation to defeat intrusion detection systems. Preprocessing operates on the decoded packets, performing a variety of transformations, making the data easier for Snort to digest. Preprocessors can alert on, classify, or drop a packet before sending it on to the more CPU-intensive detection engine.

Detection engine. The detection engine is the heart and most important part of Snort. It checks packet headers as well as payloads against several thousands of rules stored in a database of pre-defined attack signatures, as shown in Figure 1. If one rule matches, an action is triggered depending on the rule configuration for the action. There are five possible actions: *Alert* generates an alert using the selected alert method, *Log* logs the packet, *Pass* ignores the packet, *Activate* alerts and then turns on another dynamic rule, and *Dynamic* remains idle until activated by an activate rule, then acts as a log rule.

Alerting and logging. Snort is capable of outputting "alert" and "log" data in a variety of output formats and methods. Output formats include binary (called "unified") and ASCII. Binary format offers speed and flexibility, whereas ASCII format is easier to work with. Output methods include writing to a file, console or screen, syslog, or SQL database plugins. Many users commonly output data to a file in C:\Snort\log\snort or to a MySQL database. The "alert" action in Snort is *hard coded* to perform primarily two actions in sequence: (1) write an event to the alerting facility, and (2) log as much as possible information about the captured packet to the logging facility. The "log" action merely logs the packet to the logging facility without generating an alert.

It is to be noted that all Snort software tasks and components are executed *sequentially*, as shown in Figure 1. After the whole chain is worked through to process one packet, the next network packet can be processed. All packets arriving in between have to be buffered, either by the kernel or the *WinPcap*. Under heavy traffic load conditions, buffers may fill up quickly and many incoming packets may drop. The situation is exacerbated when traffic contains malicious packets which require longer execution times. In this situation, the execution time of Snort will stretch, as Snort blocks repeatedly performing alerting and logging which are typically I/O operations. However, when incoming traffic contains normal or non-malicious packets, Snort will not block repeatedly, resulting in less packets being dropped. In this situation, most of Snort's execution time will be directed towards preprocessing and detection.

The detection engine is the heart of Snort and essentially responsible for analyzing every packet based on thousands of Snort rules that are loaded at runtime. To date, there are close to eight thousand rules [Snort (08a)]. The detection engine is very complex and requires the most CPU processing power [Aldwairi (05), Weinsberg (07), Yu (07), Coppens (04), Sourdis (06), Cho (08), Baker (05), Lin (07), Mitra (07), Deri (05) Snort(08), Biswas (06), and Markatos (02)]. This is mainly due to string matching within the packet payload against these thousands of patterns (or pre-defined signatures). According to [Aldwairi (05)], more than 87% of rules contain a string to match within the packet payload. Generally, finding a single pattern in an input string imposes a computation cost that is proportional to the size of the input string [Rivest (77)]. Most known NIDS implementations use general-purpose string matching algorithms that are known to perform well. However, the computational burden of string matching using those algorithms is significant. Recent measurements on a production network suggest that Snort spends roughly 30% of its total processing time in string matching, while this cost is increased to as much as 80% for Web traffic [Markatos (02)].

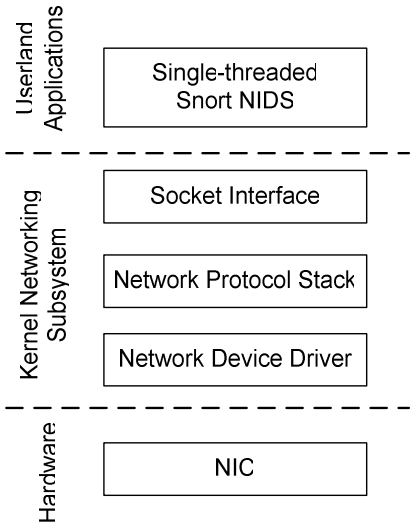

Figure 2: Windows kernel support architecture for Snort

It is important to note that Snort is a single-threaded application that operates at the user level, as shown in Figure 2. Snort uses the WinPcap packet capture library [Carr (07)] to access raw network packets. Figure 2 depicts the underlying supporting building blocks traversed by an incoming packet from the NIC on its way to userland or application-level layer. The WinPcap library offers a userland API to the socket interface of the underlying kernel networking subsystem. In Microsoft Windows the networking subsystem is primarily comprised of the TCP/IP network protocol stack and the NIC device driver. Typically, the kernel networking subsystem inserts packets into the *WinPcap* buffer for Snort to process.

3 Experimental Setup

To empirically measure and evaluate the performance of Snort, we set up a simple testbed (as shown in Figure 3) comprised of two machines, a sender and a receiver, connected with a 1 Gbps Ethernet crossover cable. The basic idea is to overwhelm Snort with high traffic generated from the sender, and then measure the performance exhibited by Snort at the receiver. The sender is a DELL PowerEdge 1800, equipped with two Intel Xeon processors, running at 3.6 GHz with 4 GB of RAM. It has an embedded Intel 82541GI Gigabit Ethernet NIC running with the e1000 driver. The receiver is an HP Compaq DC7600 server equipped with Intel® Pentium® D processor running at 3.2 GHz with 1.5GB of RAM. It has a 3COM Broadcom NetXtreme Gigabit Ethernet card with a BCM5752 controller. The sender uses a Fedora Core 5 Linux 2.6.25, whereas the receiver uses the Windows 7 and Windows 2008 Server.

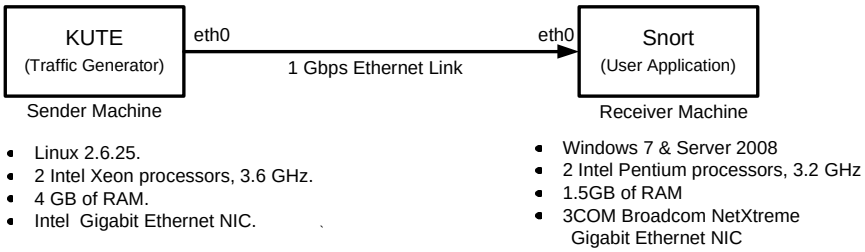

Figure 3: Testbed setup

For the purpose of generating traffic from the sender machine, we used the open-source KUTE 1.4 traffic generator [Zander (05)]. KUTE is a kernel-level generator capable of generating a high traffic rate. KUTE differs form other popular open-source traffic generators such as D-ITG and pktgen [Emma (04) and Olsson (05)], in that it does not require modifying the kernel, and it can be configured without installing a kernel- or user-level receiving component at the receiver machine. Installing a receiving component can be intrusive to experimental measurements, as both Snort and any such receiving component start sharing and competing for CPU cycles. Also such a setup is closer to reality as Snort is typically configured as a standalone application with no other applications running alongside sharing CPU cycles. Finally, unlike pktgen, we determined experimentally that KUTE has the ability to generate more accurate packet rates with finer granularity.

3.1 Windows Configurations

In this section we list important configurations and settings used in our experimental setup and evaluation.

In order to be able to generate high traffic rates, it is important to disable Ethernet flow control on both machines. The intent of Ethernet flow control is to prevent packet loss by providing back pressure to the sending NIC to throttle incoming high traffic. However, our work is based on overwhelming the host machine running Snort with very high traffic, thus not requiring flow control. Disabling Ethernet flow

control was done on the sender machine using the "ethtool" Linux utility. Specifically, to turn off flow control, we issued the command "ethtool –A eth0 rx off tx off autoneg off", and to verify, we issued the command "ethtool –a eth0". For Windows, the receiver machine, this is can be done through Control panel/NetworkProperties/General/Configure/Advanced/Flow Control (from property's list) /Disabled (from the value's list). In fact, for both Windows 7 and Windows Server 2008 flow control is disabled by default. To minimize the impact of other system activities on performance and measurement, we disabled unnecessary windows' services.

For the rest of the configuration parameters for Windows, we mostly used the default values, except for those key system parameters which are used to control the percentage of CPU bandwidth share given to Snort. Specifically, we changed the Windows *Processor Scheduling* configuration options. Two options are available: (1) optimizing for kernel networking or background services, which is the default in Windows Server 2008 and Windows 7; and (2) optimizing for user programs (i.e. user applications). Selecting between these two options can be accomplished by going into System properties/Advanced/Performance/Settings/Advanced and then checking the radio button under Processor Scheduling for "Background Services" or "User Programs".

The Windows configuration to run on UP or SMP can be accomplished through: Run/msconfig/ Boot/Advanced options/Number of processors. Setting the value of Number of processors to 1 would allow Windows to run on UP. Similarly, when the value is set to 2, Windows will run on a dual core or two processors. To set the processor affinity for Snort through the shell we installed the *imagecfg* tool. Then we issued the command: "imagecfg -a 0x2 c:\snort\bin\snort.exe", where 0x2 means CPU 2 (the second CPU). Unlike the task manager, which can be used to set the affinity, *imagecfg* tool has the ability to set the processor affinity permanently.

3.2 Snort Configurations

In this section, we describe important Snort configurations. Our main focus is to measure and compare the performance of the essential and basic functions of Snort, and therefore we chose to install and run the core Snort executable with its built-in command-line interface. We installed and configured Snort version 2.8.5.1 as a standalone NIDS with the default configurations. We set the output methods for both alerting and logging with writing output to files located in the default log directory. The default log directory for Windows is C:\snort\log. We configured the alerting facility for "fast mode" whereby basic information in a simple ASCII format (with a timestamp, alert message, source and destination IPs/ports) is outputted to an alert file. We also set the logging option of writing the captured suspicious packets in ASCII format to a file. The Windows command to run Snort as NIDS is as follows "snort –c snort.conf –i 0". This will enable Snort to be configured as a NIDS according to snort.conf and check all incoming traffic from network interface 0. The file snort.conf keeps the configuration of the internal network, rules, preprocessors, logging, alerting, etc.

The performance of Snort was measured against two types of traffic: normal and malicious. Normal traffic contains back-to-back packets that are recognized by Snort as normal; whereas malicious traffic contains packets that are recognized by Snort as malicious. Malicious traffic imposes more processing on Snort due to the triggering of events and logging. In order to generate malicious traffic, we had to modify and compile the traffic generator KUTE code to insert a string of "malicious.exe" at the end of the payload of the generated packets. Such a feature is not readily available with KUTE commands. The string was inserted at the end of the packet payload in order to allow Snort's detection engine to work harder to match this particular string. For a 64-byte packet size, we inserted 22 spaces ahead of the string "mailicous.exe" in order to meet the minimum payload of Ethernet frame of 46 bytes after counting for UDP and IP headers. For all of our generated traffic, we used UDP packets with constant and minimum 64-byte size packets and constant interarrival times. Using packets with the minimum size of 64 bytes enabled us to generate the highest possible Ethernet traffic rate, which was close to 450 Kpps (packets per second). The KUTE command to generate traffic at a specific rate for a certain period of time is:

./kute_snd.sh –d receiverIP –t timeInSecond –r packetRate –l 64

To measure the performance of Snort under malicious traffic, we added a new rule to Snort's default rule-set. The rule specifically checks every incoming UDP packet for a payload containing the string "malicious.exe". When a match occurs, a message "Malicious packet has been detected" is outputted to the alert file stored in the default log directory with an identity of "44652". The exact format of the rule is as follows:

alert udp any any -> any any (content:"malicious.exe"; msg:"Malicious packet has been detected"; sid:44652;)

The rule was inserted to a new file called malware.rules in the C:\snort\rules directory, and then this file was added at the end of the rule lists in snort.conf, so as to insert it as the bottom and last rule. Bottom or last-matching rules consume considerable processing time compared to other rules, thereby forcing Snort's detection engine to require a considerable higher percentage of CPU cycles or bandwidth.

4 Performance Measurements

To compare and evaluate the performance of Snort, several measurements for two key performance metrics were taken in relation to the generated traffic load of normal and malicious packets. These two key metrics are Snort's average throughput and packet loss. In order to interpret the results and analyze Snort's behaviour, we also measured packet loss seen by the Windows kernel networking subsystem. It is to be noted that the dropping of packets by Snort occurs at *WinPcap* buffer, whereas the dropping of packets by the kernel networking subsystem occurs at Rx DMA Ring. For all experimental results reported and shown in this section, we performed five experimental trials, and the final results are the average of these five trials. For each trial, we recorded the results after the generation of a flow with a specific rate for a sufficient duration of 30 seconds. Longer durations made negligible differences to the

results. As shown in Algorithm 1, we subject Snort to an initial rate of 1 Kpps and then gradually increase the rate by 25 Kpps until reach a maximum rate of 350 Kpps, which was the maximum rate that can be generated by KUTE.

Algorithm 1. Snort Performance Evaluation Methodology

```
Input:  t_type : traffic type {malicious or normal}
        Ps_Option  : processor scheduling option {user or
background}
        SMP : symmetric multiprocessor  {set or not set}
        P_Affinity : processor affinity {set or not set}

Output: st : snort throughput
        sl : snort loss
        kl : kernel loss

1.   Configure Snort for t_type
2.   Configure Processor Scheduling Options for  Ps_Option

3.   If  SMP  then
4.      set symmetric multiprocessor feature
5.      If (P_Affinity) then
6.          Set processor affinity for  snort
7.      end if
8.   end if

9.   time_period←30
10.  rate_increment←25
11.  packet_max_rate←350
12.  packet_current_rate← 1
13.  // Note that SnortThrouput[1..5], SnortLoss[[1..5],
                KernelLoss[1..5] are auxiliary arrays.

14.  repeat
15.    for  trial←1 to 5 do
16.      start Snort
17.      generatePacket(packet_current_rate, time_period)
18.      kill Snort
19.      record(SnortThrouput[trial], SnortLoss[trial],
                  KernelLoss[trial])
20.    end for
21.    st←average (SnortThroughput)
22.    sl←average(SnortLoss)
23.    kl←average(KernelLoss)
24.    packet_current_rate ← packet_current_rate +
       rate_increment
25.  until  packet_current_rate > packet_max_rate
```

For each experimental trial, we launch Snort manually before the start of packet generation by KUTE. Then at the sender machine a shell script is run to have KUTE generate traffic for 30 seconds at a specific rate. When KUTE terminates after packet generation for 30 seconds, it produces various logging information about the generated traffic, particularly the actual total packets that was generated and the actual time to generate them. Also, when Snort is terminated, it prints out statistics of three types of packets: analyzed, dropped, and received. It is to be noted that Snort's total

packets dropped is actually the same as the total packets received minus the total packets analyzed.

From these logs and statistics, key performance metrics can be determined. Snort's actual throughput is the total packets analyzed by Snort by 30 seconds. Snort's packet loss probability can be calculated by dividing the total packets dropped by Snort over the total packets received. The packet loss probability of the kernel networking subsystem can be calculated by the following simple formula: ((the total packets sent by KUTE) – (the total packets received by Snort)) / (the total packets sent by KUTE). Finally, the incoming traffic rate generated by KUTE from the sender machine can be calculated from KUTE's logs by dividing the number of total packets sent over the actual time to send them. Our Snort performance evaluation methodology is summarized in Algorithm 1.

4.1 Snort performance against normal traffic

We measured Snort's performance under both Windows 7 and Windows Server 2008 in terms of throughput and packet loss against incoming traffic. For both Windows 7 and Windows 2008, we measured the performance against the different types of traffic (normal and malicious), for different processor scheduling options giving preference (or priority) to user applications or kernel services, and under UP and SMP environment (with and without affinity). For the SMP environment we measure Snort's performance with and without setting processor affinity. Figure 4 summarizes the results in terms of the average Snort throughput, the average Snort packet loss, and the average kernel packet loss when subjecting Snort host to normal traffic. The Snort throughput is shown in Figures 4(a) and 4(b) with the two available processor scheduling configuration options of (1) being optimization for user application [Application Scheduling Priority], and (2) optimization for kernel background services [Kernel Scheduling Priority].

It is clear from Figures 4(a) through 4(d) that the performance of Snort, when setting processor scheduling option for giving more CPU time to user programs, is slightly better than its performance when setting the processor option for giving more CPU time to kernel background services. In fact, this difference is clear under the SMP environment with affinity not being set. This is expected in the SMP environment, because with affinity set for Snort to run on one processor, the other processor will take care of the kernel background services. Thus, setting the processor scheduling option will not make much difference when the affinity is set, but it does make a difference when the affinity is not set. It is also to be noted that the results in figures 4(a) and 4(b) are consistent with the results shown in Figures 4(e) and 4(f), where the kernel loss is more when the processor scheduling was set for user applications than it is when the processor scheduling was set for the kernel background services. This can be explained as follows. The earlier the packet drop occurs the more the CPU time is given to Snort to process packets, and thus the more Snort's throughput.

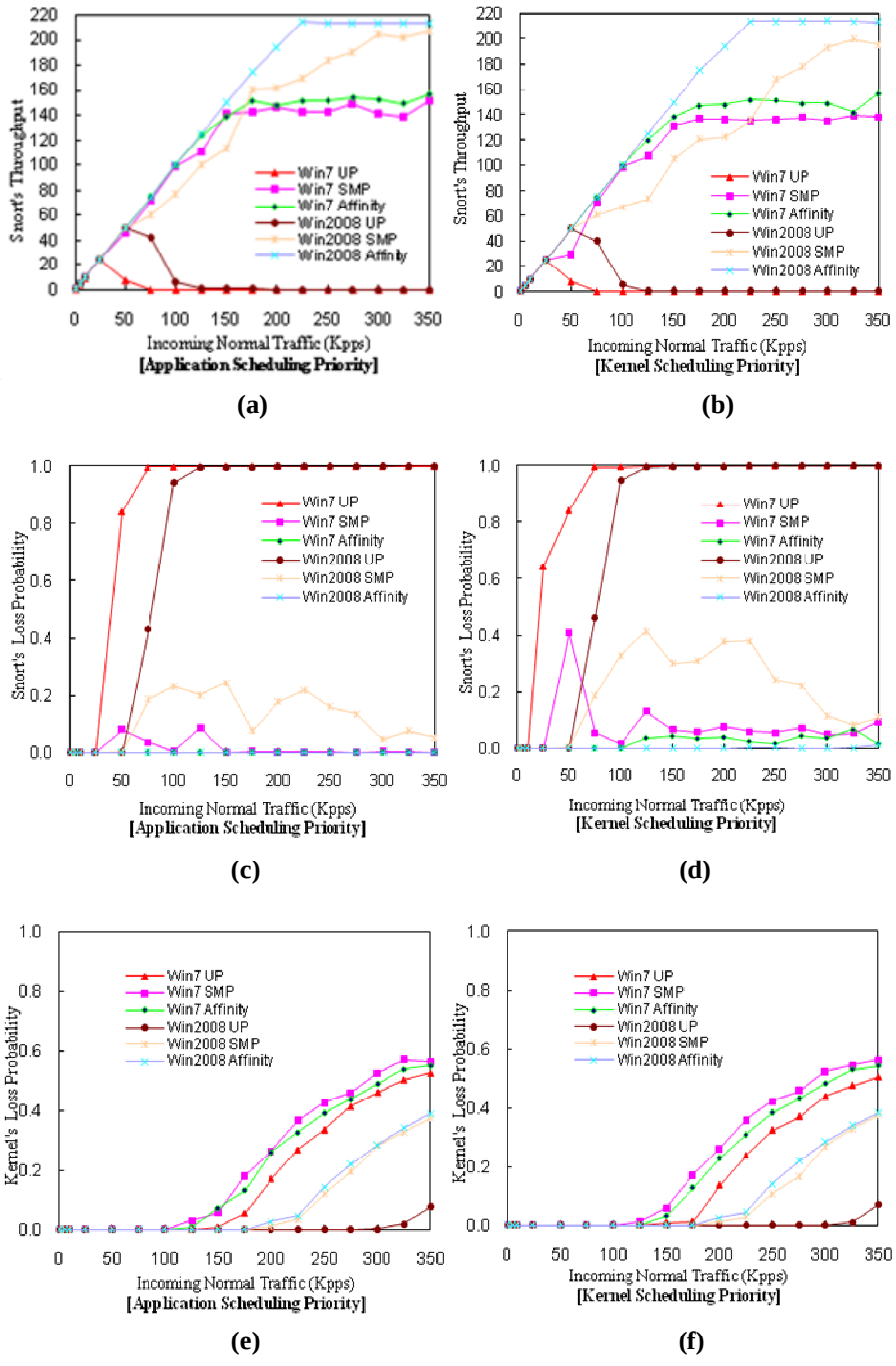

Figure 4: Snort's performance when subjected to normal traffic

As depicted from Figure 4, the highest throughput is achieved at all different rates when setting the processor affinity for Snort under Windows 2008. For Windows 2008 Server (with affinity is set), the Snort throughput increases gradually until it reaches 215 Kpps, where it becomes stable at the incoming rate of 225 Kpps. In addition, it is obvious from Figures 4(a) through 4(d) that either under Windows 7 or Windows 2008 Server, setting the processor affinity for Snort gives higher throughput and lower Snort packet loss compared to running it under SMP environment without setting processor affinity. In other words setting the processor affinity for Snort increases its throughput and decreases the packet loss by Snort. The reason can be attributed to the cache re-use, as explained earlier. In fact, as shown in Figures 4(c) and 4(d), there is no packet loss by Snort at all the different rates when running under both systems with affinity set. However, there is a kernel loss as follows. For windows 2008 Server (with affinity set), the kernel loss occurred when the packet rates exceed 175 Kpps, for both processor scheduling options. Having no packet loss by Snort when the affinity is set, the Snort throughput is bounded by the packets it receives from the kernel. This means that the Snort throughput may increase if it receives more packets from the kernel. For windows 7, there is a kernel's packet loss when the packet rate exceeds 125 Kpps for both processor scheduling options with slight differences, see Figures 4(e) and 4(f).

Measuring the throughput of Snort when running under Windows 2008 SMP against its throughput under Windows 7 SMP without setting processor affinity for Snort, we found that Windows 7 either competes with or outperforms Windows 2008 server under the law packet traffic rate (i.e. less than 175 Kpps, Figure 4(a), or less than 225, Figure 4(b). However, Windows 2008 Server outperforms Windows 7 at the higher packet traffic rate, i.e. greater than 175 Kpps of Figure 4(a), or greater than 175 Kpps of Figure 4(b).

When running Snort under UP environment in both Windows 7 and Windows 2008 Server and at low incoming traffic rates, we obtain better throughput under Windows Server 2008 than running it under Windows 7. However, at high incoming traffic rates, behaviourboth operating systems exhibit a very low throughput. This can be attributed to a low CPU time given to Snort by the operating system.

4.2 Snort performance under malicious traffic

Figures 5 presents the results in terms of the average Snort's throughput, the average Snort's packet loss, and the average kernel's packet loss when subjecting the Snort host to malicious traffic. The Snort throughput is shown in Figures 5(a) and 5(b) when the processor scheduling configuration option was configured for user applications [Application Scheduling Priority] and kernel background services [Kernel Scheduling Priority], respectively.

It is clear from Figures 5(a) through 5(d) that the performance of Snort when setting the processor scheduling option to giving more CPU time to user programs is slightly better than its performance when setting processor option to giving more CPU time to kernel background services. However, the effect of the processor scheduling option is clear under the SMP environment with affinity not set, where setting processor scheduling for user programs shows clearly better throughput. This is expected in the SMP environment, as explained earlier in the case of normal traffic. Thus, setting the processor scheduling option will not make much difference when the

affinity is set, but it does make a difference when the affinity is not set. Similarly, like in the case of normal traffic, the results in figures 5(a) and 5(b) are consistent with the results shown in Figures 5(e) and 5(f), where the kernel's loss is more when the processor scheduling was set for user programs than it is when the processor scheduling was set for the kernel background services. It is clear from Figure 5(b) that the highest throughput was achieved for all traffic rates with Kernel Scheduling Priority under Windows 2008 SMP affinity.

Similar to Snort's performance exhibited under normal traffic, it is obvious from Figure 5 that setting the processor affinity for Snort gives higher throughput and lowers Snort's packet loss compared to running it under SMP environment without setting processor affinity. In other words, setting the processor affinity for Snort increases its throughput and decreases the packet loss by Snort. The reason can be attributed to the cache re-use. (as explained earlier). Unlike the case of normal traffic where there is no packet loss, the packet loss of Snort against the malicious traffic when the affinity is set, increases sharply, as the packet rate sent by KUTE increases till the rate of 100 Kpps, where Snort's packet loss starts to increase very slowly.

Measuring the throughput of Snort when running under Windows 2008 SMP against its throughput under Windows 7 SMP without setting processor affinity for Snort, we found that when the processor scheduling option is set to user programs Windows 2008 Server outperforms Windows 7 at all different rates. However, when the processor scheduling option is set to kernel background services, Windows 2008 server competes with Windows 7 at the law packet traffic rate, i.e. less than 150 Kpps of Figure 5(a), and outperforms it at the high packet rates.

When subjecting Snort to low traffic rates under UP environment in both Windows 7 and Windows 2008 Server, we have obtained better throughput under Windows 2008 than running it under Windows 7. However, at high rates, behaviourboth operating systems exhibit a poor thorughput. This can be attributed to a low CPU time given to Snort by the operating system. The Snort's peak throughput under Windows 7 is around 10 Kpps, achieved at 25 Kpps. However, the Snort's peak throughput under Windows 2008 Server is little less than 14 Kpps, achieved at 50 Kpps. These peaks are much smaller than the Snort's peak throughput under normal traffic which occurs at the same points. This is because Snort needs more time to process malicious traffic due to the extra traffic taken by Snort for alerting and logging.

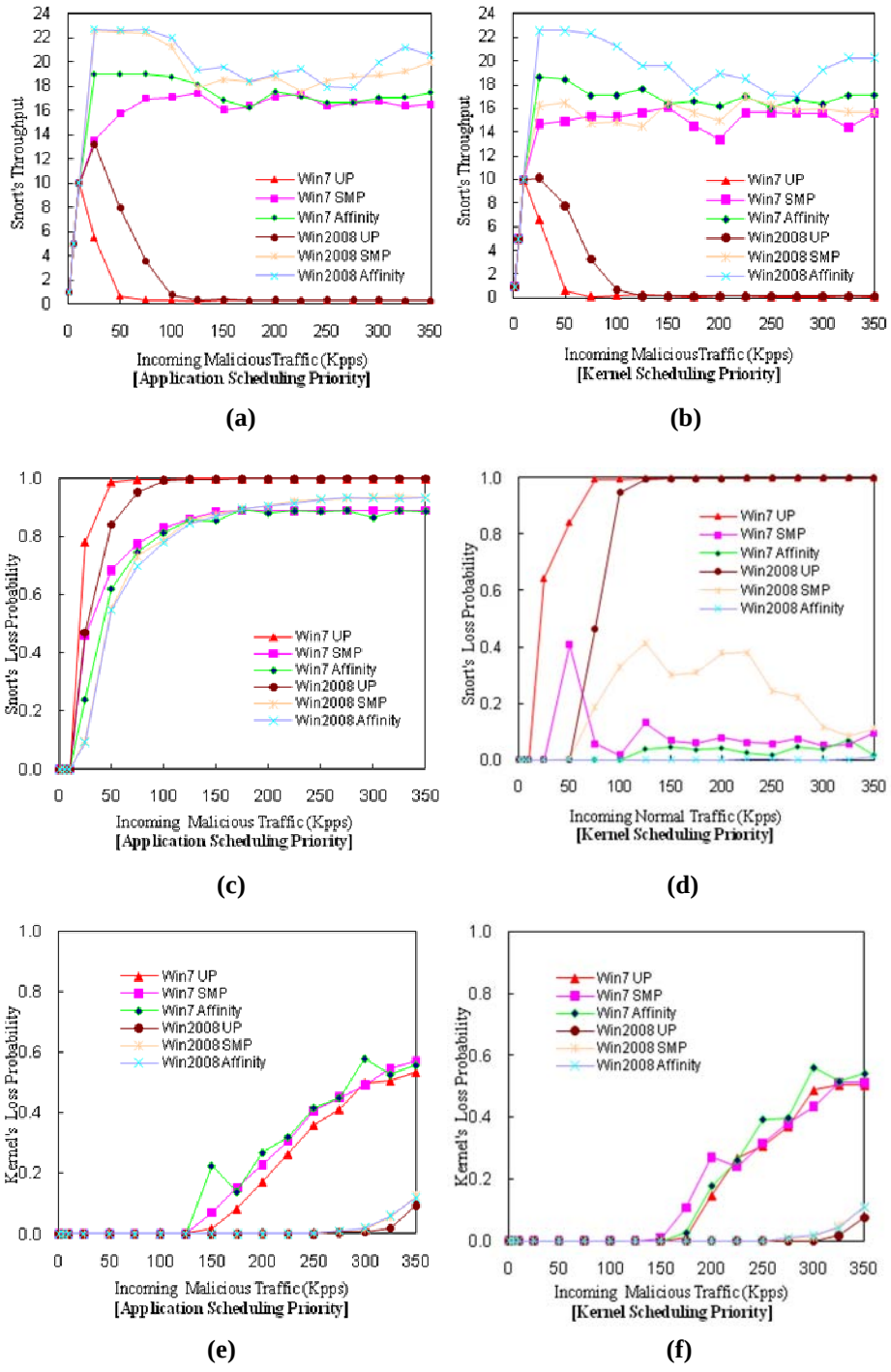

Figure 5: Snort's performance when subjected to malicious traffic

5 Conclusion

In this paper, we have evaluated and compared the performance of Snort NIDS when running under Windows 7 and Windows 2008 Server. The performance was measured and compared in terms of Snort's throughput and packet loss when subjecting a host running Snort to both normal and malicious traffic. We considered key system configurations and options which included CPU scheduling priority, UP and SMP environment, and affinity types. As has been demonstrated in Section 4, setting the scheduling priority to favour either kernel processing or user applications has little or no impact on Snort's performance under both normal and malicious traffic. It has been shown that running Snort under an SMP environment showed significantly better throughput than running it under an UP environment. In addition, it has been demonstrated that under an SMP environment with the option of static affinity, resulted in relatively higher throughput than the default dynamic affinity. Such an observation was true when subjecting Snort to both normal and malicious traffic. As a future work, we plan to implement Snort as a multi-threaded application, and then measure and compare its performance for both Windows and Linux running on hosts with multicore architectures.

References

[Abbas, 02] Abbas, S.: "Introducing Multi-Threaded Solution to Enhance the Efficiency of Snort"; MS Thesis, Department of Computer Science, Florida State University, December (2002)

[Alder, 04] Alder, R., Babbin, J., Doxtater, A., Foster, J. C., Kohlenberg, T., Rash, M.: "Snort 2.1 Intrusion Detection"; 2nd edition, Syngress (2004)

[Aldwairi, 05] Aldwairi, M., Conte, T., Franzon, P.: "Configurable String Matching Hardware for Speeding up Intrusion Detection"; ACM SIGARCH Computer Architecture News, 33, 1 (2005), 99-107.

[Baker, 05] Baker, Z., Prasanna, K.: "High-throughput Linked-Patter Matching for Intrusion Detection Systems"; Proc. ANCS'05, ACM/IEEE Symposium on Architectures for Networking and Communications System, Princeton, New Jersey (2005), 193-202.

[Benvenuti, 05] Benvenuti, C.: "Understanding Linux Network Internals"; O'Rilley Press (2005)

[Biswas, 06] Biswas, A., Sinha, P.: "Efficient Real-Time Linux interface for PCI Devices: A Study on Hardening a Network Intrusion Detection System"; Proc. SANE 2006, the 5th System Administration and Network Engineering Conference, Delft, The Netherlands (2006).

[Bovet, 05] Bovet, D., Cesati, M.: "Understanding the Linux Kernel"; 3rd Edition, O'Rilley Press (2005)

[Carr, 07] Carr, J.: "Snort: Open Source Network Intrusion Prevention"; eSecurityPlanet Article (2007), also appeared as electronic version, http://www.esecurityplanet.com/article.php/11162_3681296_1

[Cho, 08] Cho, Y., Mangione-Smith, W.: "Deep Network Packet Filter Design for Reconfigurable Devices"; ACM Transactions on Embedded Computing Systems, 7, 2 (2008), 452-461.

[Coppens, 04] Coppens, J., De Smet, S., den Berghe, S., De Turck, F., Demeester, P.: "Performance Evaluation of a Probabilistic Packet Filter Optimization Algorithm for High-Speed Network Monitoring"; Proc. HSNMC'04, the 7th IEEE Conference on High Speed Networks and Multimedia Communications, Toulouse, France (2004), 120-131.

[Deri, 05] Deri, L.: "nCap: Wire-Speed Packet Capture and Transmission"; Proc. E2EMON, the 3rd IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, Nice, France (2005).

[Emma, 04] Emma, D., Pescape, A., Ventre, G.: "D-ITG, Distributed Internet Traffic Generator"; 2004. Also available at http://www.grid.unina.it/software/ITG

[Geschke, 06] Geschke, D.: "Fast Logging Project for Snort"; FLoP Report (2006), also appeared as electronic version, http://www.geschke-online.de/doc/index.html

[Lan, 03] Lan, K., Hussain, A., Dutta, D.: "The effect of malicious traffic on the network"; Proc. PAM'03, San Diego, California (2003)

[Lin, 07] Lin, C., Tai, Y., Chang, S.: "Optimization of Pattern Matching Algorithm for Memory Based Architecture"; Proc. ANCS'07, ACM/IEEE Symposium on Architectures for Networking and Communications System, Orlando, Florida (2007), 11-16.

[Markatos, 02] Markatos, E., Antonatos, S., Polychronakis, M., Anagnostakis, K.: "Exclusion-based Signature Matching for Intrusion Detection"; Proc. CCN'02, IASTED International Conference on Communications and Computer Networks, Cambridge, Massachusetts, USA (2002), 146-152.

[Mitra, 07] Mitra, A., Najjar, W., Bhuyan, L.: "Compiling PCRE to FPGA for Accelerating Snort IDS"; Proc. ANCS'07, ACM/IEEE Symposium on Architectures for Networking and Communications System, Orlando, Florida (2007), 127-135.

[Olsson, 05] Olsson, R.: "pktgen the Linux Packet Generator"; Proc. Linux Symposium, Ottawa, Canada (2005)

[Ramakrishnan, 93] Ramakrishnan, K.: "Performance Consideration in Designing Network Interfaces"; IEEE Journal on Selected Areas in Communications, 11, 2 (1993), 203-219.

[Rivest, 77] Rivest, R. L.: "On the Worst-Case Behavior of String-Searching Algorithms"; SIAM Journal on Computing, 6, 4 (1977), 669-674.

[Salah, 10] Salah, K., Kahtani, A.: "Performance evaluation comparison of Snort NIDS under Linux and Windows Server"; Journal of Network and Computer Applications, 33, 1 (2010), 6-15.

[Salim, 01] Salim, J. H.: "Beyond Softnet"; Proc. 5th Annual Linux Showcase and Conference, Oakland, California (2001), 165-172.

[Snort 08a] http://www.snort.org/

[Sort, 08b] The Snort Project, "Snort Users Manual 2.81"; 2008, also appeared as electronic version, http://www.snort.org/

[Sourdis, 06] Sourdis, I., Dimopoulos, V., Pnevmatikatos, D., Vassiliadis, S.: "Packet Pre-filtering for Network Intrusion Detection"; Proc. ANCS'06, ACM/IEEE Symposium on

Architectures for Networking and Communications System, San Jose, California (2006), 183-192.

[Turnbull, 07] Turnbull, J.: "Improving Snort Performance with Barnyard"; EnterpriseLinux.Com Magazine (2007), also appeared as electronic version, http://searchenterpriselinux.techtarget.com/tip/Improving-Snort-performance-with-Barnyard

[Vermeiren, 04] Vermeiren, T., Borghs, E., Haaodorens, B.: "Evaluation of Software Techniques for Parallel Packet Processing on Multi-Core Processors"; Proc. 1st IEEE Consumer Communications and Networking Conference, CCNC, Las Vegas, Nevada (2004), 645-647.

[Weinsberg, 07] Weinsberg, Y., Tzur-David, S., Dolev, D., Anker, T.: "One Algorithm to Match Them All: On a Generic NIPS Pattern Matching Algorithm"; Proc. HPSR'07, Conference on High Performance Switching and Routing, Brooklyn Bridge, New York (2007), 1-6.

[Wu, 07] Wu, W., Crawford, M., Bowden, M.: "The Performance Analysis of Linux Networking – Packet Receiving"; International Journal of Computer Communications, Elsevier Science, 30, 5 (2007), 1044-1057.

[Yu, 07] Yu, J., Xue, Y., Li, J.: "Memory Efficient String Matching Algorithm for Network Intrusion Management System"; Journal of Tsinghua Science and Technology, 12, 7 (2007), 585-593.

[Zander, 05] Zander, S., Kennedy, D.d, Armitage, G.: “KUTE - A High Performance Kernel-based UDP Traffic Engine”; CAIA (Center for Advanced Internet Architectures) Technical Report (2005), also appeared as electronic version, http://caia.swin.edu.au/reports/050118A/CAIA-TR-050118A.pdf