Performance Characterization & Improvement of Snort as an IDS

Report prepared by:

Soumya Sen

Part A:

Snort background and performance measure

1. Introduction to Snort

SNORT is an open source Intrusion Detection System (IDS), which may also be configured as an Intrusion Prevention system (IPS) for monitoring and prevention of security attacks on networks.

When working as an IDS, Snort may detect security attacks and alert the system administrator or take the designated action like logging the malicious packets. If we configure Snort as an IPS, then it tries to prevent attacks by modifying the *iptable* to drop or reject the suspected packets/stream.

Snort was originally developed by Martin Roesch, founder of Sourcefire Inc., Maryland. The work on Snort was started around 1998-1999, but at that time the software was a mere packet sniffing device and had little significance as an Intrusion detection device. Despite of the problems of the earlier versions of Snort, the software steadily developed into an IDS. In 1999 version 0.98 was available, and version 1.0 was released on 4/28/99, but the first stable version with IDS feature was released as version 2.0 on 4/6/2003. Ever since then, there has been a rapid growth in this software and newer versions were released frequently. The current standard version in version 2.4.5. The latest implemental release has been version 2.6.

Snort uses a rule-driven language. It combines the benefits of signature, protocol and anomaly based inspection methods. With more than 3 million of downloads to date, Snort is the most widely deployed intrusion detection and prevention technology worldwide. (Website: www.snort.org)

2. Overview of Snort as an Intrusion Detection System

2.1. Intrusion Detection systems

".. 'Intrusion Detection' addresses a range of technologies that are involved in the detection, reporting, and correlation of system and network security events".
- The Hackers Handbook (Young, Aitel)

Over time, people have proposed several apt definitions for Intrusion detection systems and Intrusion Prevention systems. An IDS is an alerting system that watches data flow at one or more points in the network, providing alerts and forensics on suspect or malicious traffic. While an IDA just alerts the admin on detection of attacks, it may be also required to prevent them from taking place. This leads us to the concept of Intrusion Prevention system. An IPS is a system with a proactive method of detecting and preventing malicious traffic, but allows admin to provide action upon being alerted.
Both IDS & IPS need real-time pattern matching capability at very high speed, working with huge rule sets.
Snort has used a high speed multi-pattern search engine since Version 2.0 release in 2002.Snort has been reportedly tested by OSEC (Open Security Evaluation Criteria) to

provide intrusion detection capability at 750 Mbps under OSEC test conditions. However, it is a challenge to make it work on multi-gigabit applications. Our research is aimed at exploring this possibility of developing an IDS and its implementation on high speed networks, working on large rule sets.

2.2. Signature, Protocol & Anomaly detection capabilities of Snort with a Rule-driven language

Snort is capable of signature, Protocol & Anomaly detection as well. At the heart of all these detection lie the detection engine and the rules for identifying malicious traffic.
The rule set of Snort is very powerful, flexible, and it is easy to understand and construct new rules. The examples below show the rules for working of Snort in various attack scenarios:

2.2.1. Example of Signature Detection Rule:

Snort handles its signatures based detection with the rules created for it. In the second half of 2001 we observed new and powerful worms on the Net, such as Code Red, Code Red II and Nimda.

- Scenario 1: Detecting Nimda

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-IIS cmd.exe access"; flags: A+; content:"cmd.exe"; nocase; classtype:web-application-attack; sid:1002; rev:2;)

- Scenario 2: Detecting Code Red II

alert tcp any any -> any 80 (msg: "CodeRedII root exe"; flags: A+; content:"root.exe"; depth:624; classtype:attempted-admin;)

2.2.2. Example of protocol Detection Rule:

Snort is provided with rule set for detecting protocols and suspicious behavior of these protocols in use. Snort rules have the protocol field that allows the detection of the protocol.

- Scenario 1:

The following rule detects any scan attempt using SYN-FIN TCP packets. The flags keyword is used to find out which flag bits are set inside the TCP header of a packet.

alert tcp any any -> 192.168.1.0/24 any (flags: SF; msg: "SYNC-FIN packet detected";)

- Scenario 2:

Some tools like Nmap may ping a machine by sending a TCP packet to port 80 with ACK flag set and sequence number 0. Since this packet is not acceptable by the receiving side according to TCP rules, it sends back a RST packet. When nmap receives this RST packet, it learns that the host is alive. (This method works on hosts that don't respond to ICMP ECHO REQUEST ping packets). To detect this type of TCP ping, our rule will be like the following that sends an alert message:

alert tcp any any -> 192.168.1.0/24 any (flags: A; ack: 0; msg: "TCP ping detected";)

2.2.3. Example of Anomaly Detection Rule:

Many attacks use buffer overflow vulnerabilities by sending large size packets. Using the ‘dsize’ keyword in the rule, one can find out if a packet contains data of a length larger than, smaller than, or equal to a certain number. The following rule generates an alert if the data size of an *IP* packet is larger than 6000 bytes.

alert ip any any -> 192.168.1.0/24 any (dsize: > 6000; msg: "Large size IP packet detected";)

2.3. The company behind Snort:

Snort is now maintained by developers and researchers at Sourcefire Inc., MD. The company was founded by Martin Roesch in 2001.
Sourcefire has released many versions over the years, and has an active team that provides very frequent version updates. The frequency of releases is graphed below:

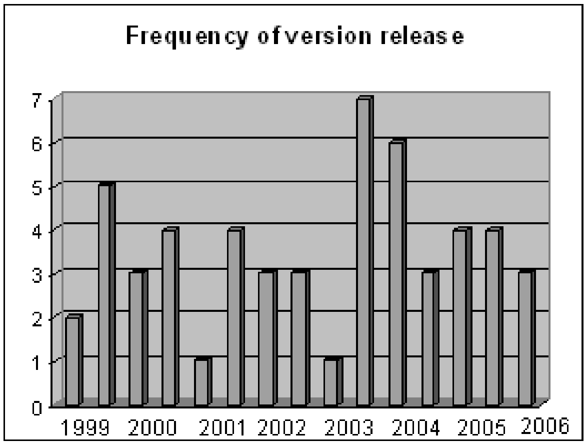


Fig 1: Frequency of version release for Snort

2.3.1. Business model of Sourcefire, Inc.:

The Vulnerability Research Team (VRT) of Sourcefire Inc., provides latest rules at $195/month or $495/quarter or $1795/year to subscribers. However, the older rule sets are made available to the public free of cost, while the latest rule sets are charged for. Sourcefire also organizes training sessions, onsite technical training, and customer support of 3 classes: Standard, Gold, Platinum.

2.3.2. Snort Development program:

- Programs:

The Snort programs are coded and implemented by employees at Sourcefire, Inc. They also identify and implement new algorithm as and when required.

- Rules:

Vulnerability Research Team identifies, tests and certifies new rule updates for snort. Snort users can submit their own rules at the forum, and when the rules are tested by the VRT, they are incorporated into the rule files. Snort provides latest rules certified by the VRT to its subscribers and an older version of the rule files for free to registered and unregistered customers. Individual users can further add their own local rules to the rule files and submit their rules to community rules portal of Snort for testing and certification into a valid rule by the VRT team.

2.4. Need for improving IDS:

In the recent years, Statistics have shown a severe growth in security threats over time, and new software vulnerabilities are being discovered everyday.

Given below is an estimate of the recent trend in the growth of rule sets:
2006-3-31: 5759 rules
2006-4-25: 6182 rules, 2006-6-08: >6471 rules

The alarming fact about the growth in rule set is that larger rule sets implies more severe time constraints on packet handling and pattern matching by Snort, and failing to cope with this growing trend will mean severe performance deterioration and packet loss.

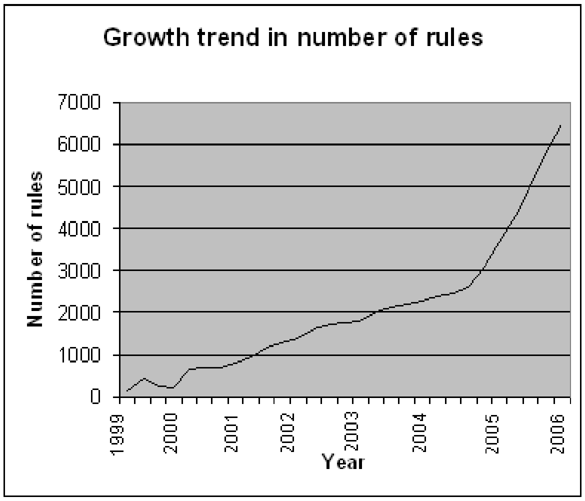


Fig 2: Growing trend in the number of rules over the years

2.5. Operation modes of Snort

- Sniffer Mode- simply reads the packet off the network and displays them in a continuous stream on the console.

Options:

./snort –v : Prints TCP/IP header onto screen (also for UDP/ICMP).

./snort –vd: Prints the application data too.

./snort –vde: Prints the data link layer contents as well.

- Packet Logger Mode- logs the packets to the disk.

Options:

./snort –dev –l ./log: Logs the packets to the directory specified.

./snort –l ./log –b: Binary log, binary file may be read back using –r switch.

- Network Intrusion Detection Mode (NIDS) – most complex and configurable mode. It allows Snort to analyze network traffic against a user defined rule set and performs actions based on detections.

Options:

./snort –A fast –c snort.conf: Fast alert, writes the alert in a simple format with a timestamp, alert message, source and destination IPs/ports.

- Inline mode- obtains packets from iptables instead of the libpcap and then causes iptables to drop or pass packets based on snort rules that use inline-specific rule types. (drop, reject, sdrop options). This is the mode used when Snort is to act as an IPS.

2.6. Improvements in Snort over time:

The combination of Optimized data flow, enhanced rule selection and a new High-Performance Multi-pattern search engine makes Snort 2.0 about eighteen times faster than version 1.9.

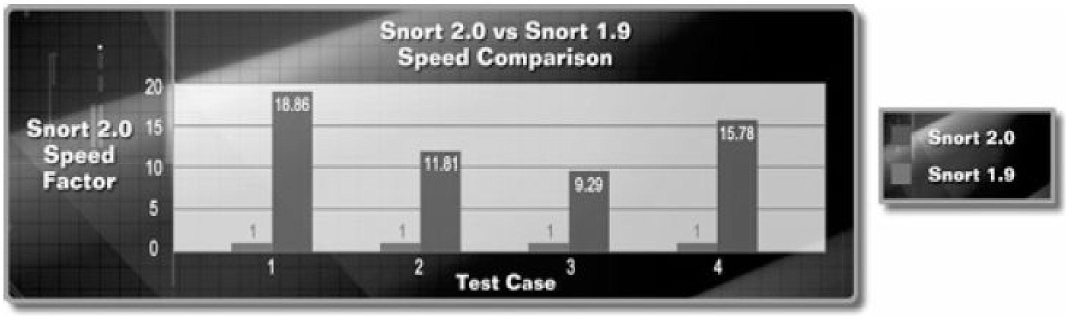


Fig 3: Improvements in the versions of Snort

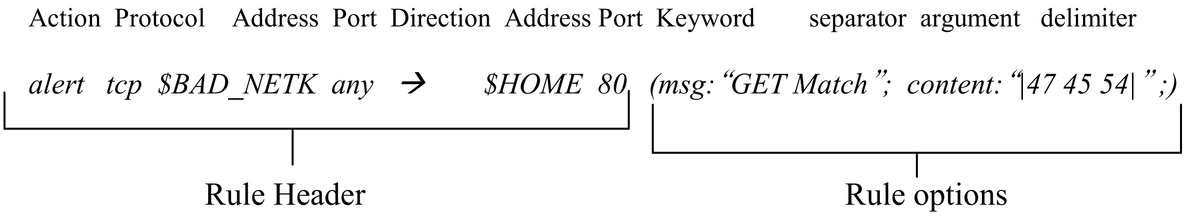
Recent release of Snort 2.6.0 uses Aho-Corasick algorithm as default mode instead of the previously used Wu-Manber algorithm, because Wu-Maber algorithm was found to be sensitive to DoS attacks.

3. Technical details of Snort

3.1. Rule syntax in Snort

The rule syntax in Snort is well-organized. Each rule consists of two parts:

1) Rule Header: This has a static definition and is composed of the 5 tuple as shown. It must be present in every rule.
2) Rule options: This has variable definitions. It is not always present, and it has more than 50 options available to account for different requirements in the description of possible rules.



Action Protocol Address Port Direction Address Port Keyword separator argument delimiter

alert tcp $BAD_NETK any → $HOME 80 (msg: "GET Match"; content: "|47 45 54|";)

3.2. Payload detection rule options- Simple to Complex!

Snort has simple, flexible rule description language to describe how data is to be handled:

- *Example 1: DNS Exploit*

alert tcp $EXTERNAL_NET any->$HOME_NET 53 (msg: "DNS EXPLOIT named 8.2->8.2.1"; flow: to_server, established; content: "../../../"; reference: bugtraq, 788; reference: cve, 1999-0833; classtype: attempted-admin; sid:258; rev: 6;)

- action type: Generate alert using selected method and then log the packet. (other action types: pass, activate, dynamic)

- protocol: TCP (other types are UDP/ICMP/IP)

- address: single IP/CIDR blocks/list of address (other types are: [192.168.1.0/24, 10.1.1.1], negation ! [192.168.1.0/24], or 'any')

- port: 53. (other types: 'any', !443, :1023, 1024:, 6000:6010)

- rule options: msg, flow, content, ref, sid, classid, rev.

('flow' option: used along with TCP stream reassembly to indicate that rules should apply only to certain kind of traffic; in this case match should occur only when client requests the content from the server and the connection is already established)

- *Example 2: SubSeven Trojan*

alert tcp $EXTERNAL_NET 27374 -> $HOME_NET any (msg:"BACKDOOR subseven 22"; flags: A+; content: "|0d0a5b52504c5d3030320d0a|"; reference:arachnids,485; reference:url,www.hackfix.org/subseven/; sid:103; classtype:misc-activity; rev:4;)

- msg:”BACKDOOR subseven 22”; message to appear in logs
- flags: A+; tcp flags; many options, like SA, SA+, !R, SF*
- content: “|0d0…0a|”; binary data to check in packet; content without | (pipe) characters do simple content matches. (Mixed Binary Bytecode and Text in a 'content' keyword, eg. Content: “|5c 00|P|00|I|00|P|00|E|00 5c|”)
- reference…; where to go to look for background on this rule
- sid:103; rule identifier
- classtype: misc-activity; rule type
- rev:4; rule revision number
- other rule options possible, like offset, depth, nocase

- *Example 3: Overflow attempt*

alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg: “WEB-IIS MDAC Content-Type overflow attempt”; flow:to_server, established; uricontent: “/msadcs.dll”; nocase; content:”Content-Type|3A|”; nocase; isdataat 50,relative; content:!”|0A|”; within:50; pcre:”/^POST\s/smi”; reference: bugtraq…, cve…, url…; classtype:web-application-attack; sid 1970; rev:9)

- Rule Option: Text between parentheses is the rule option
- Msg: message to be printed along with packet dump as part of the log or alert
- Flow: specifies the direction of data flow that this rule applies to
- Content: specifies the content to search for in the packet payload. The | symbol represents bytecode for binary data (or the representation of binary data in hexadecimal). Multiple content searches are specified in this one rule – the second content option is preceded by a ! (not).
- Uricontent: searches the normalized request URI field
- Isdataat : tests to ensure that the payload has data at the specified location
- Pcre: allows rules to be written using Perl compatible regular expressions
- Reference : provides a set of security references and documentation for the alert Classtype – categorizes alerts to attack classes
- Sid: uniquely identifies a particular signature ID
- Rev: identifies the revision of a particular rule.

- *Example 4: Bad file attachment*

Snort supports PCRE- Powerful Regular Expression Payload analysis.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 25 (msg:”VIRUS OUTBOUND bad
file attachment”;\
flow:to_server, established;\
content:”Content-Disposition|3A|”;nocase;\
pcre:”/filename\s*=\s*.*?\(?=[abcdehijlmnoprsvwx])(a(d[ep]|s[dfx])|c([ho]m|li|md|pp)|
d(iz|ll|ot)|e(m[fl]|xe)|h(lp|sq|ta)|jse?|m(d[abew]|s[ip])|p(p[st]|if|[lm]|ot)|r(eg|tf)|s(cr|[hy
]s|wf)|v(b[es]?|cf|xd)|w(m[dfsz]|p[dmsz]|s[cfh])|xl[stw]|bat|ini|lnk|nws|ocx)[\x27\x22\n\
r\s]/iR”;\
classtype:suspicious-filename-detect;\
sid:721; rev:7;)
```

The above rule is a complex rule with Perl compatible regular expressions in it, and therefore it needs processing by the Perl library. These kinds of regular expressions can only be handled best by software rather than hardware equipments. Thus the support for regular expression makes Snort rules more flexible and provides advantage to a software based approach for Intrusion Detection Systems.

3.3 Architecture of Snort

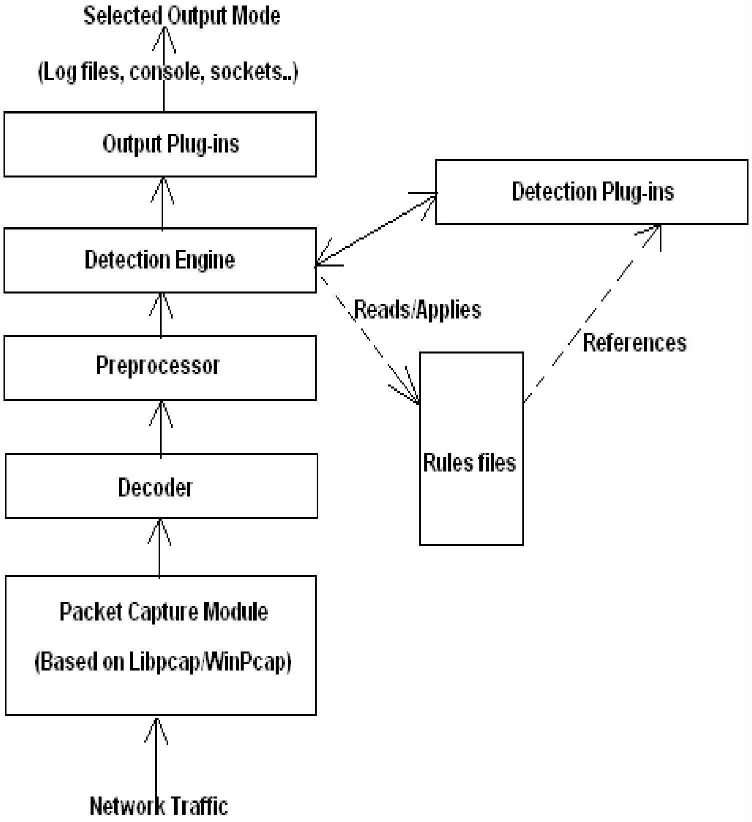


Fig 4: Snort architecture

1. ***Decoder:*** It fits captured packets into data structures and identifies link level protocols. Then it takes the next level, decodes IP, and TCP/UDP to get

information about port addresses. Snort alerts for malformed headers, unusual TCP option.

2. ***Preprocessors:*** They are like filters, which identifies things that should be checked later in Detection Engine module (like suspicious connection attempt to some TCP/UDP port or too many UDP packets received during a port-scan).
3. ***Rule files:*** Text files with rule sets written with a known syntax.
4. ***Detection Plug-ins:*** Those modules referenced from its definition in the rule files, and they are intended to identify patterns whenever a rule is evaluated.
5. ***Detection engine:*** Making use of detection plug-ins, it matches packets against rules previously charged into memory since snort initialization.
6. ***Output plug-ins:*** Alerts, logs, extern files, databases.

3.4. Snort Internals:

Snort 2.0 uses a High Performance Multi-Rule Inspection engine for detecting patterns. Packets are first analyzed by the Rule Optimizer to select the appropriate set of rules for inspection. The Multi-Rule Inspection engine searches for rule matches, builds a queue of detected rule matches, and selects best rule match for logging.

The Process of inspecting Network traffic for matches may be described in these three steps:

1. Rule Optimization to produce efficient rule sets for inspection
2. Set based inspection algorithms that perform high-speed multi-pattern content searches.
3. Parameterized inspection techniques which allow for complicated pattern inspection.

3.5. Rule Optimizer & Snort rule set:

The Rule classifier classifies all Snort rules into rule subsets. This is done prior to any packet streaming. Once this is over, each incoming packet is matched to a corresponding rule set based on the packet’s unique parameters.

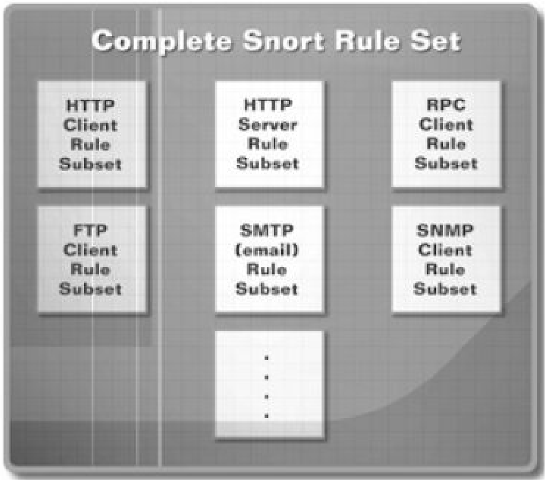


Fig 5: Rule sets

3.6. Other Snort components:

- ***Flow analyzer:***

Protocol Flow analyzer classifies network application protocols into clients and server data flows.

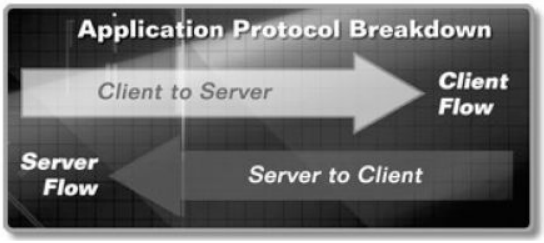


Fig 6: Flow analyzer

- ***Rule Optimizer:***

It utilizes a set-based methodology for managing Snort rules and applying them to Network traffic. Rule subsets are formed on unique rule and packet parameters using a classification scheme based on set criteria. Subsets are predetermined during initialization.

3.7. Multi-Rule search Engine:

Rule & Packet processing flow diagram:

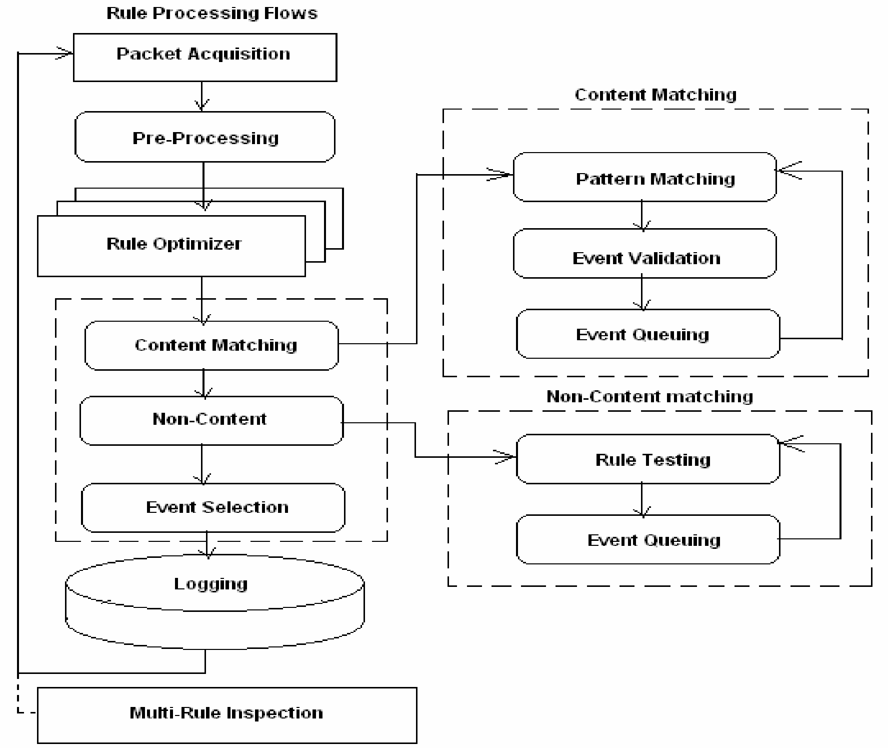


Fig 6: multi-rule search engine

Rule Sets & Search Types used in general by Snort are the four categories of rules for testing are listed below:

1. Protocol Field Rules
2. Generic Content Rules
3. Packet anomaly Rules
4. IP Rules

3.8. Rule files statistics:

Snort version 2.4 has about 45 different rule files. Examples of some of these rule files are given below:
Attack-responses.rules, Smtp.rules, Multimedia.rules, Icmp.rules, Chat.rules, Web-client.rules

Below we have plotted the graphs for the distribution of rule files with the number of rules present in them:

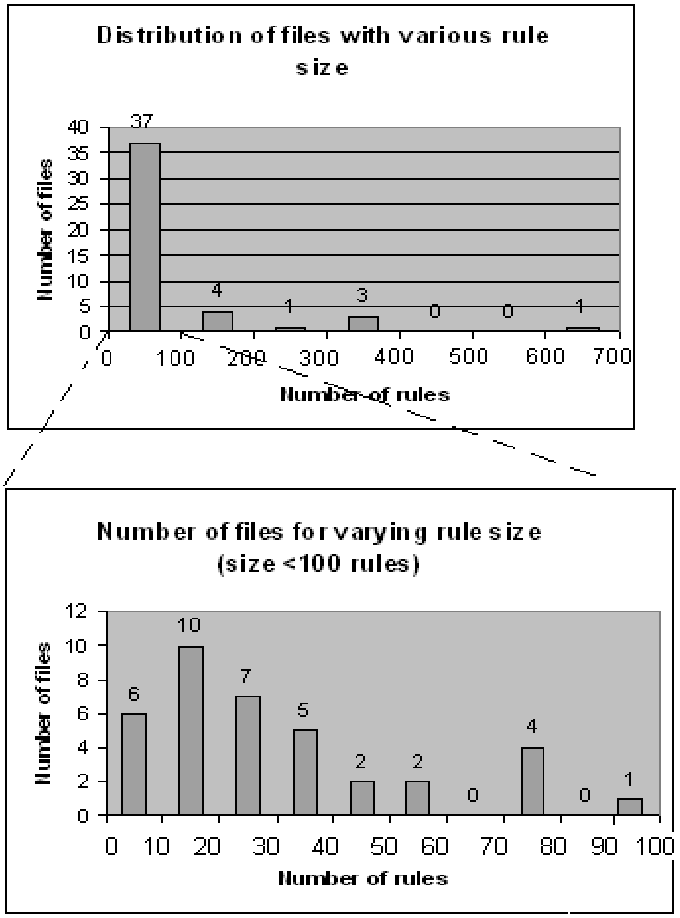


Fig 7: Distribution of rule files against number of rules

Snort rule statistics:
A study in Oct 2003 found that 87% of 1777 Snort rules had content to match. (Ref:http://www.ece.ncsu.edu/erl/faculty/paulfwww/Research/pubs/BeaconSnort.pdf)
Next we show the plot they mentioned in their work.

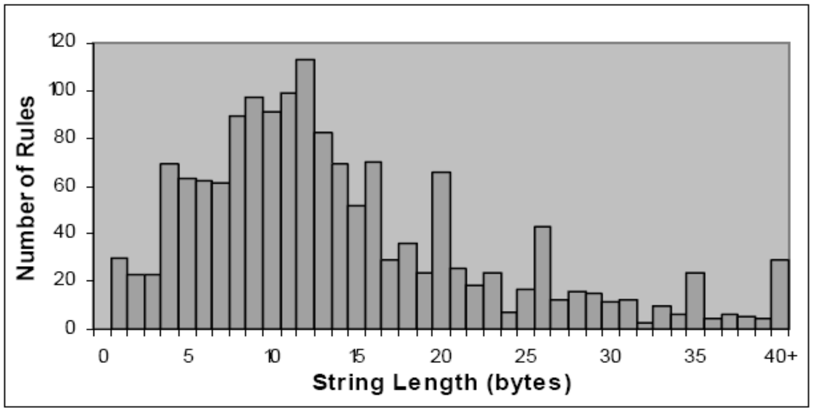
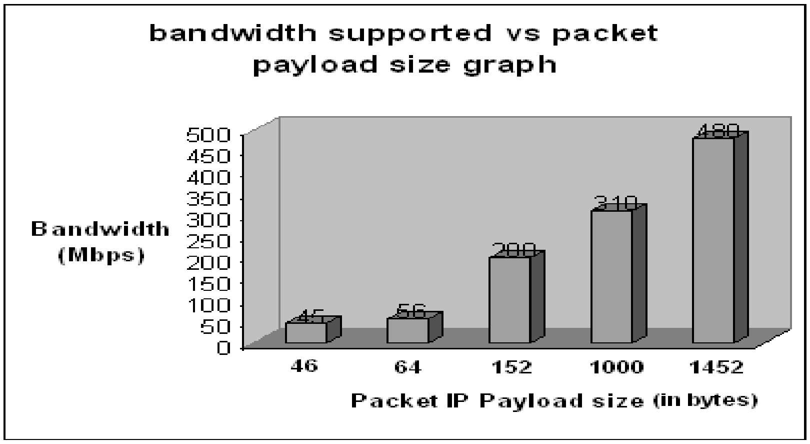


Fig 8: Number of rules vs string length in bytes

4. Performance Analysis of Snort:

We have carried out the experiment to test the performance of Snort by installing Snort on a machine with the following specifications: P4, 2,5 GHz, 533 Mbps bus speed, DDR, 1M cache. Then we pumped in packets from a hardware based packet generation tool, Agilent's hardware based packet generator (N2X). One stream of packets were fed inot the system from the generator at specified rate and with random payload. The allowed loss rate of packets was kept below 0.005%.

The graph showing the bandwidth supported for varying payload sizes is given below: (loss rate<0.005%)



[maximum packet drop
allowed for the runs: 0.005%
Length of each run time: 15 s]

Fig 8: Bandwidth dependence of packet payload size

4.1. Snort's performance for varying rule set sizes: (Case 1)

We now provide the results of Snort's performance when the rule set sizes are varied. We chose to investigate the dependence of bandwidth on the size of the rule set when the packet size is kept constant. The first case we consider is the scenario when the packet size is 1452 bytes. The graph for the same is shown next:

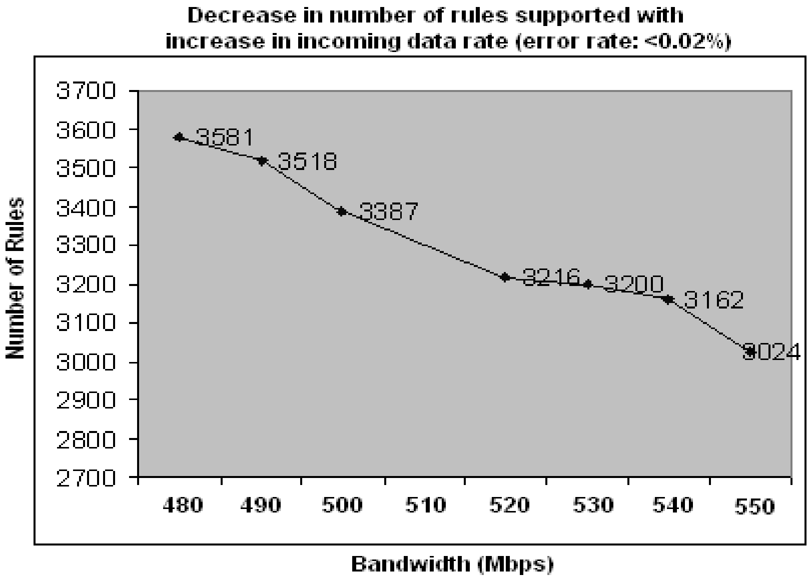


Fig 9: Dependence of bandwidth supported on rule set size (payload size: 1452 bytes)

4.2. Snort's performance for varying rule set sizes: (Case 2)

In this case we study the dependence of the bandwidth supported on the number of rules for packet IP payload size of 46 bytes. The graph for this scenario is shown below:

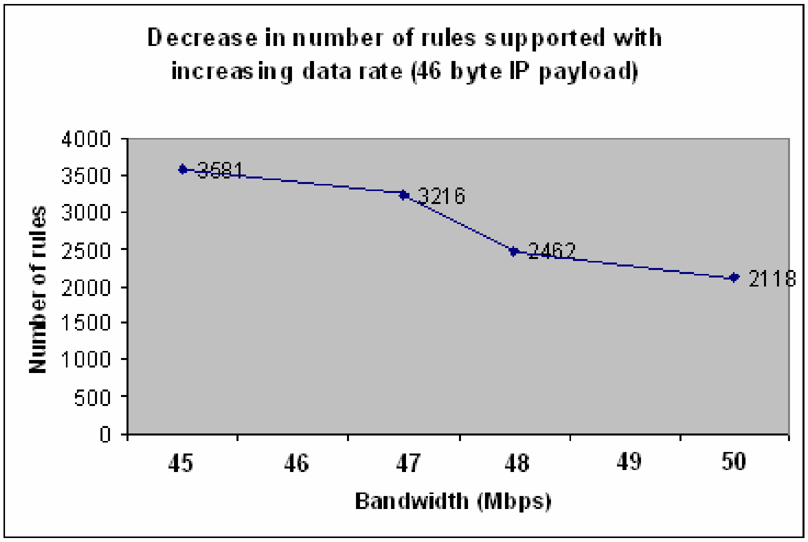


Fig10: Dependence of bandwidth supported on rule set size (payload size: 46 bytes)

The conclusion drawn from the above graphs is that the bottleneck in processing arises from the ability to read packets up from the interface. For a given bandwidth, the number of packets generated will be larger in case of packets with smaller IP payload, than the case where the payload is large. When large numbers of packets are generated, then it becomes difficult to read all of them from the NIC, and therefore packets are dropped. In order to compensate that, and process the packets faster, the numbers of rules have to be drastically decreased in the case when packet sizes are small.

Part B:

Performance Analysis of the Snort Pattern Matching Algorithms

5. Construction Phase diagram for Pattern matching in Snort

The whole process cycle that Snort takes, starts from taking in the rule files, constructing the state machine, and then walking through it and searching for pattern matches with the input strings. It may be represented as shown below:

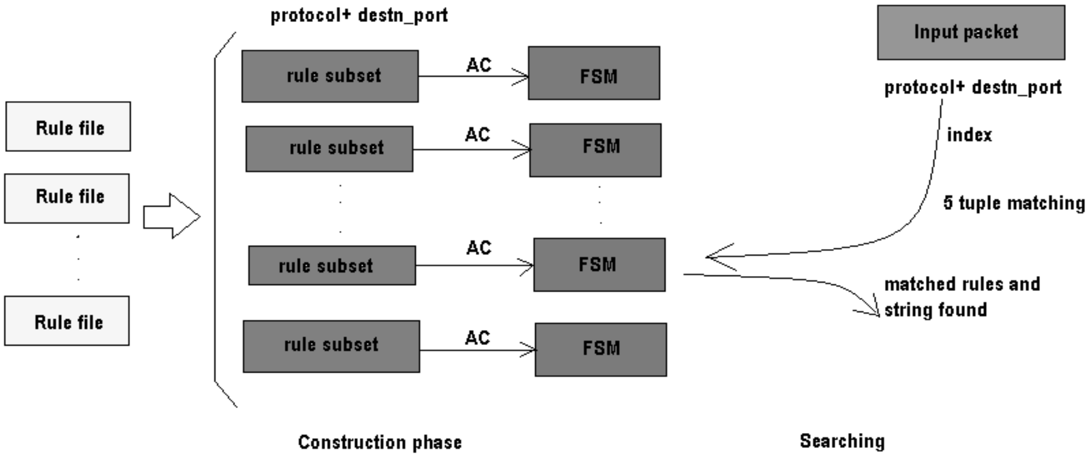


Fig 11: Diagram of the pattern matching module of Snort

At the heart of snort lies the pattern matching algorithm. It has been estimated that almost 70-80% of the time is spent by Snort in pattern detection. Therefore, it is important to improve the efficiency of pattern matching algorithm.

6. Algorithm for Snort: Aho-Corasick Algorithm

Aho-Corasick algorithm involves the construction of a finite state machine from the keywords and then using pattern matching machine to process the text string in a single pass. The construction of a finite State machine (FSM) using Aho-Corasick algorithm may be described using a small example. Let us consider the case where we have four keywords: {he, she, hers, his}.
The state machine will be based on the transition from one state to another, based on the 'go to' function constructed out of the allowed transition into a next state. The 'go to' function can be represented by the following diagram:

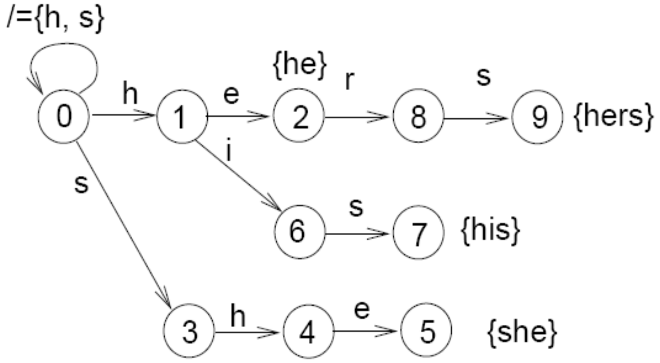


Fig 12: Go-to function

During the transitions from one state to another, if the input value is such that it is not a valid transition, then the next state will be the failure state as governed by the failure function:

Failure function:

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

The output associated with each state is given by the output function:

i	**Output(i)**
2	{he}
5	{she, he}
7	{his}
9	{hers}

6.1. Algorithm for pattern matching:

- Input: A text string $x=a_1a_2...a_n$ where each ai is an input symbol and a pattern matching machine M with go to function ‘g’, failure function ‘f’, and output function as mentioned.
- Output: Locations at which keywords occur in x

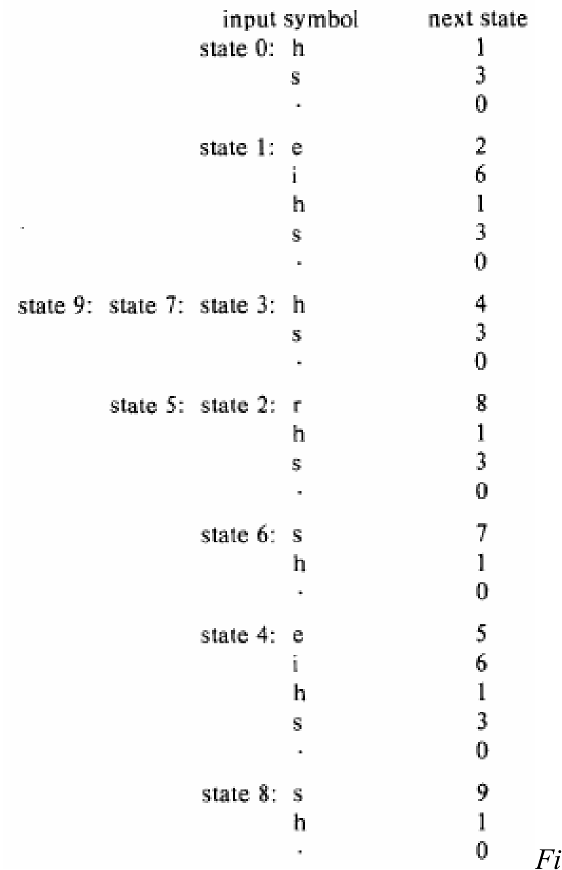
Method:

```
begin
    state←0
    for i←1 until n do
        begin
            while g(state, ai)= fail do state ←f(state)
            state←g(state, ai)
            if output(state) ≠empty then
                begin
                    print i
                end
            end
        end
```

6.2. Getting rid of the Failure function to obtain a DFA:

- Elimination of all failure transitions by using next move function of a deterministic finite automation in place of the ‘goto’ and ‘failure’ functions.
- DFA makes exactly one state transition on each input symbol.
- DFA will explicitly have the transition for each of the states, while for NFA the entries will only correspond to the entries for valid inputs, and the other transitions will be governed by the failure function.

The transition table for the various inputs of the DFA can be written as shown as the one shown below for the example under consideration:



	input symbol	next state
state 0:	h	1
	s	3
	·	0
state 1:	e	2
	i	6
	h	1
	s	3
	·	0
state 9: state 7: state 3:	h	4
	s	3
	·	0
state 5: state 2:	r	8
	h	1
	s	3
	·	0
state 6:	s	7
	h	1
	·	0
state 4:	e	5
	i	6
	h	1
	s	3
	·	0
state 8:	s	9
	h	1
	·	0

Fig 13: Transition Table

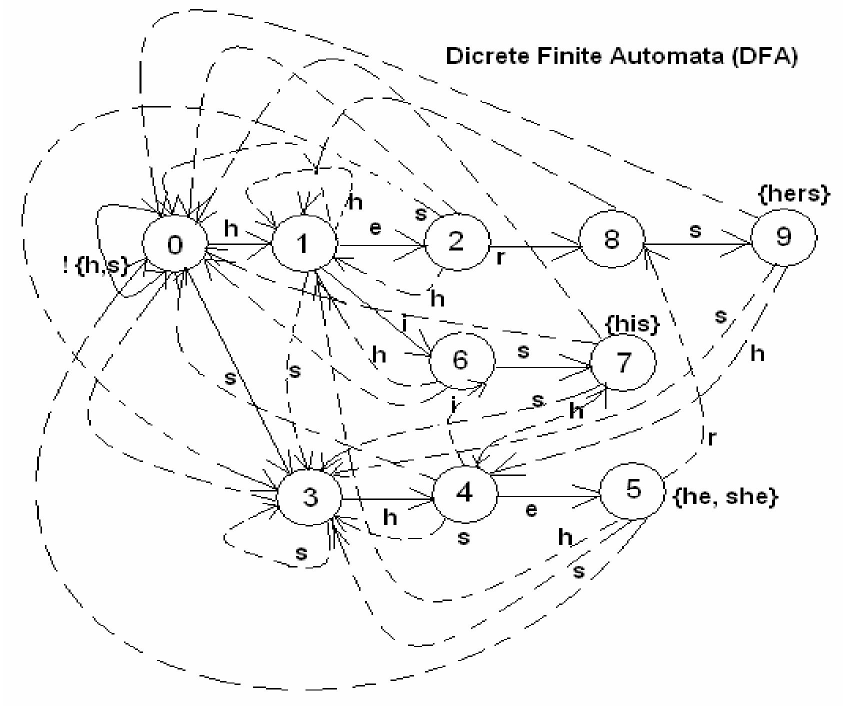


Fig 14: state transition diagram

6.3. Data structures in Snort for AC algorithm

ACSM (Aho-Corasick State Machine) implementation can be based on either of the two implementations as given below:

- NFA (Non-deterministic Finite Automata)
- DFA (Deterministic Finite Automata)

The implementations that we implemented or tested on the code of Snort for the two implementations, NFA and DFA, are listed below:

NFA:

- Full
- Sparse
- Sparsebands
- ***Banded***
- ***Triple Array***
- ***Double Array***

The *Banded, Triple Array* and *Double Array* are the implementations that we have included in the Snort code for performance measure.

DFA:

- Full matrix
- Sparse
- Sparsebands
- Banded

6.4. Data structure representations:

6.4.1. Full Matrix:

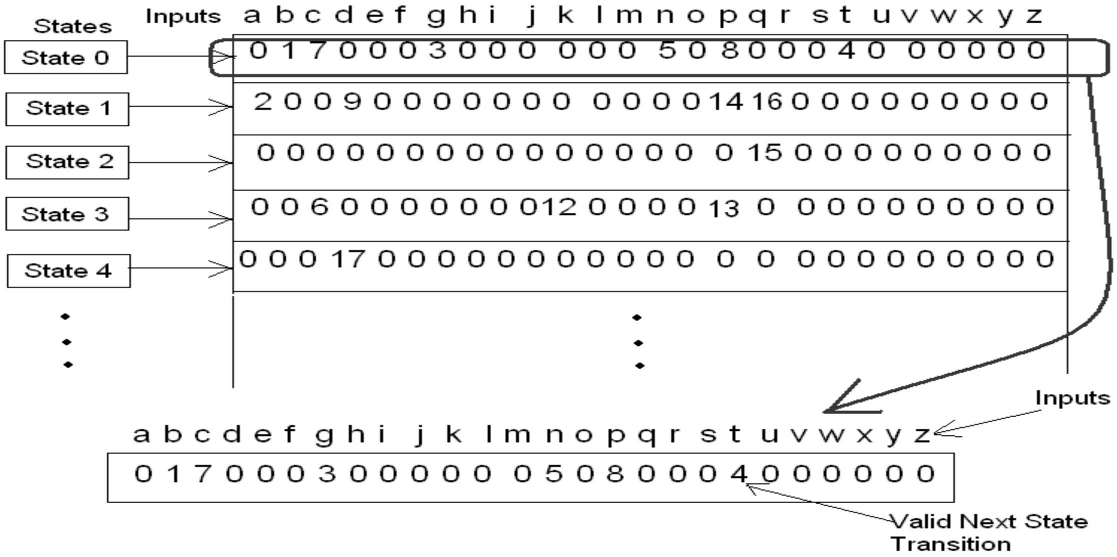
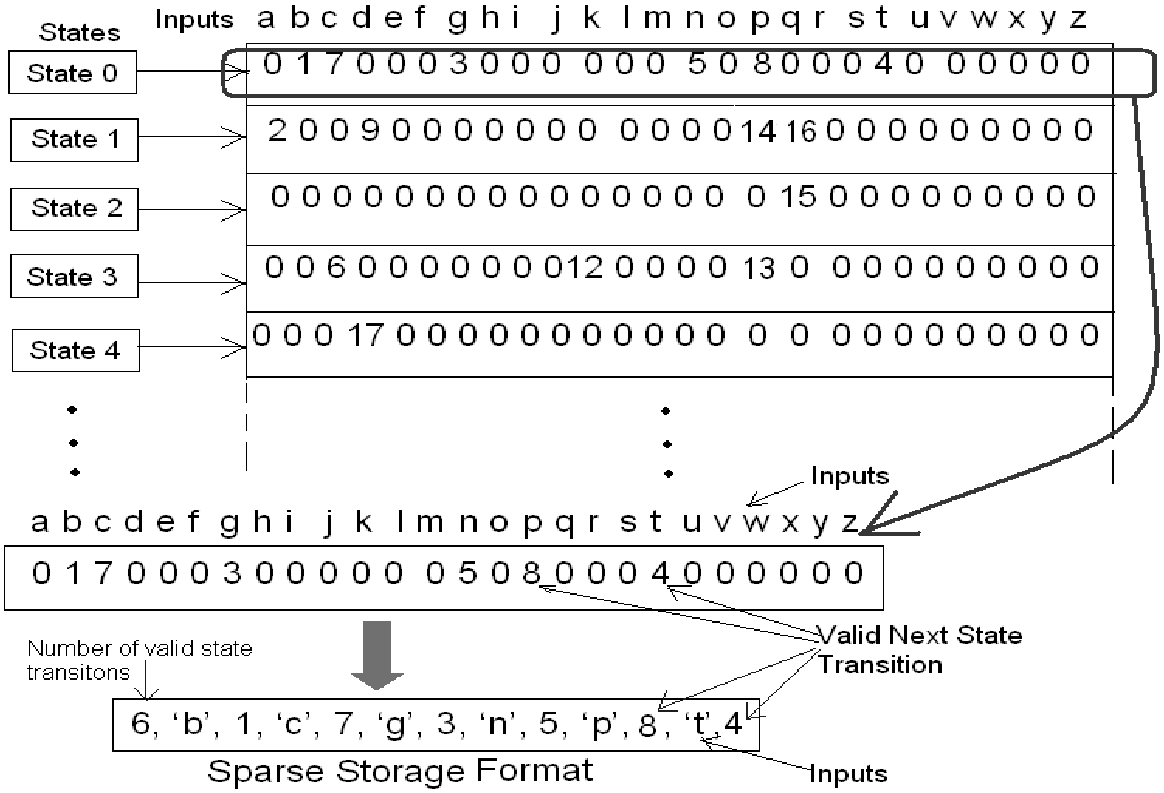


Fig 15: Full matrix representation

The full matrix representation is one in which all the states have a next state entry for each of the possible input values. In the above example, we have the inputs as ‘a’, ‘b’,…, ‘z’ for the various states. The zero entries mean that there is no valid transition for the given input and the state.
However, this kind of a data structure is too memory consuming, and hence not a suitable data structure for storage purposes.

6.4.2. Sparse Storage format:



Inputs a b c d e f g h i j k l m n o p q r s t u v w x y z

State 0: 0 1 7 0 0 0 3 0 0 0 0 0 0 5 0 8 0 0 0 4 0 0 0 0 0 0

State 1: 2 0 0 9 0 0 0 0 0 0 0 0 0 0 14 16 0 0 0 0 0 0 0 0 0

State 2: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15 0 0 0 0 0 0 0 0 0

State 3: 0 0 6 0 0 0 0 0 0 0 12 0 0 0 0 13 0 0 0 0 0 0 0 0 0

State 4: 0 0 0 17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 7 0 0 0 3 0 0 0 0 0 0 5 0 8 0 0 0 4 0 0 0 0 0 0

6, ‘b’, 1, ‘c’, 7, ‘g’, 3, ‘n’, 5, ‘p’, 8, ‘t’,4

Fig 16: Sparse storage format

In the sparse storage format, the data structure is a bit different. The elements of the new storage matrix will be the number of valid state transitions for the given state, the valid transitions and the corresponding next state. This kind of a storage helps us reduce memory, but speed may be compromised because the random access into the matrix is now lost in this modified structure.

6.4.3. Banded matrix:

The banded matrix format is a data structure where the zeros in the front and the end of the row matrix are chopped off, the index of the first non-zero value of the row matrix is stored along with the total number of entries in the band. The banded matrix format is better in terms of speed than the sparse storage format because it allows random access into the elements of the array, thereby improving the speed of operation.

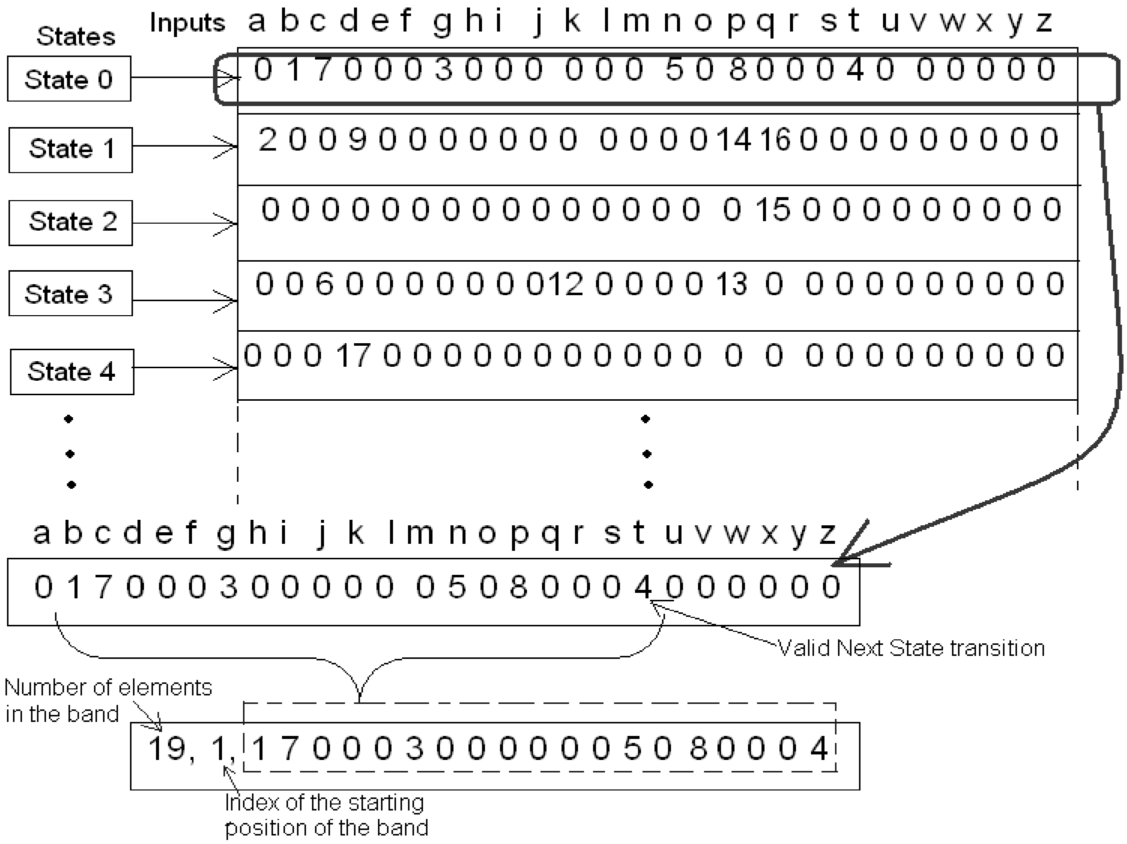
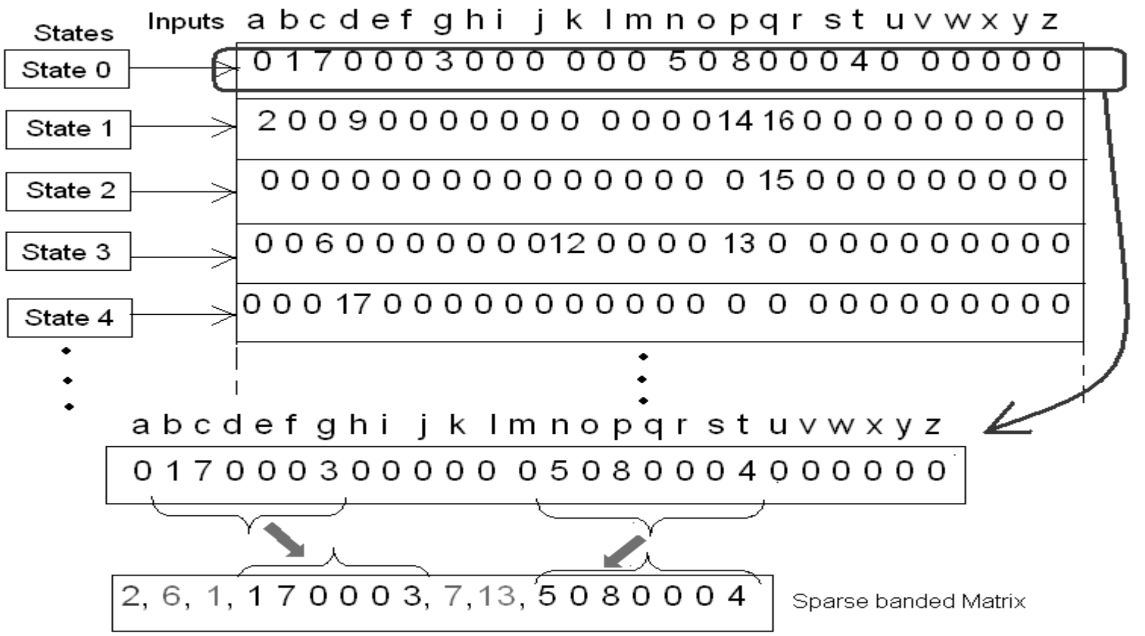


Fig 17: Banded matrix format

6.4.4. Sparse Banded Matrix format:



2 is number of sparsebands, 6 & 7 are number of elements in the two bands resp., 1 & 13 are index of the first value of the bands respectively.

Fig 18: Sparse banded matrix format

The Sparse banded matrix format is the one that maintains the space-saving feature of the Sparse matrix and the random access property of banded matrix. The main idea is that if there is a long row of zeros within a row in the banded matrix format, then that banded matrix may be broken up further into smaller bands. This will lead to a little decrease in performance from that of the banded matrix, but is very efficient in terms of storage.

The elements of interest in the case of sparse banded matrix are the number of the bands present in the structure, the number of elements in each of the bands, the starting index of the bands which gives the position of that band within the row matrix.

6.5. Profile of the Snort pattern matching code:

The gprof profiler run on the acsmx2.c code, taken directly out of the present Snort code, showed that almost 78% time of the pattern matching code is spent in acquiring the next state for the transition. The output of the profiler on this code is shown below:

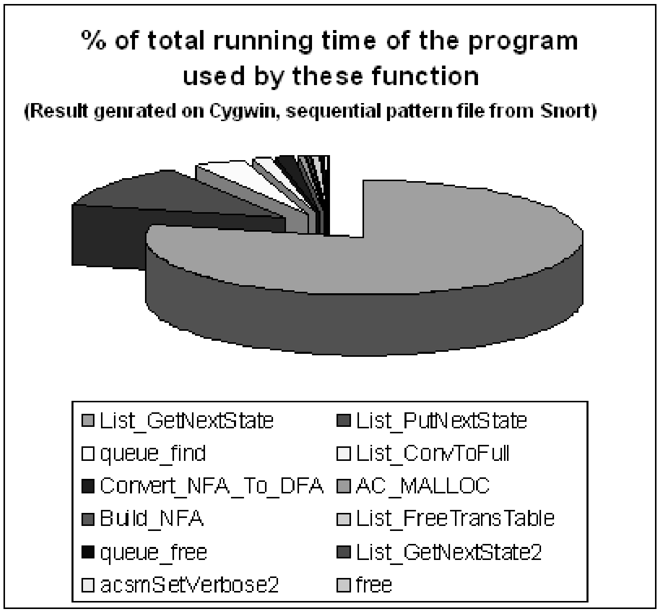


Fig 19: output of the 'gprof' profiler ran on Snort pattern matching code

7. Comparison of the results of various Data Structures:

Comparative Performance Analysis of Data Structures was done to bring out their memory requirements and the speed of execution. These are two most important considerations for the proper choice of the data structure.

Memory requirements by different options:

- DFA full >NFA full >DFA banded >NFA banded

7.1. Performance of ACSM implementation of Snort:

For the purpose of testing of different options, we have used two different types of input files, different types of pattern files and different platforms for testing.
The types of pattern files were obtained from the rule sets based on Snort rule set, Bleeding Snort rule set and worm pattern files.

The input files were of the types: Sequential and Random.

Sequential file is the file obtained by printing out the way Snort stores the rules after parsing. The ASCII value of each character is printed with a blank space in between. This is our input pattern file for sequential input. This arrangement of the rules is the most optimal way of storing the rules for Snort. Thus, it must be noted that each of the input lines of this input data file is a valid pattern and may contain other valid patterns within it. Snort has to match the occurrences of all such valid patterns present in the input data file.

The random input file refers to the randomizing the position of the patterns of the sequential file, thereby destroying the benefits of Snort's optimal arrangement of the rules. This kind of an input file may have repetition in the patterns within the file arising out of the hashing function used for the randomization process. We call this obtained file as the Random data input file. It must be remembered that random input file does not mean that the contents of the input files are random, on the contrary, each line is a valid pattern even for the random file.

The first test result given here is based on the Cygwin platform on an IBM laptop, with a testing file of 2160 Snort patterns. The file is looped over 1000 times to give better precision about the time of execution.

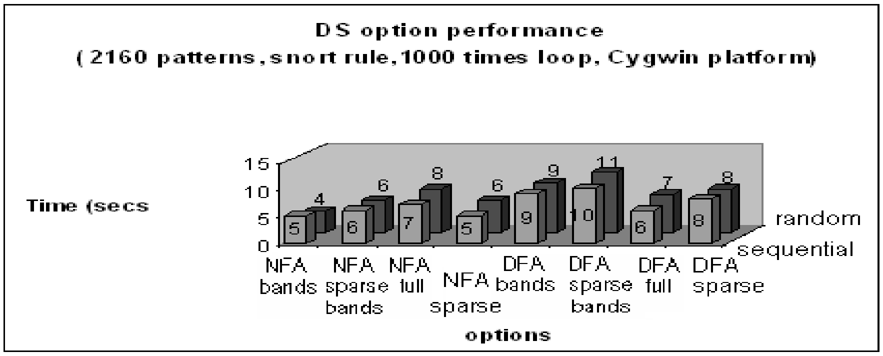


Fig 21: Performance measure of various data structure on Snort rules for Cygwin platform

The comparison of the performance of the various data structures of Snort on linux machine with 2X2 Meg cache, DDR is shown below:

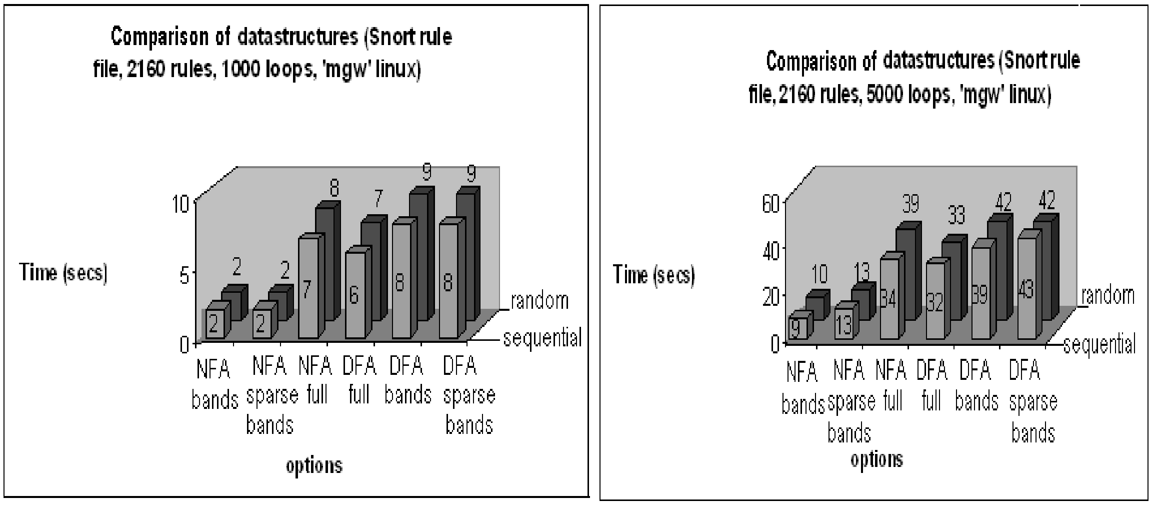
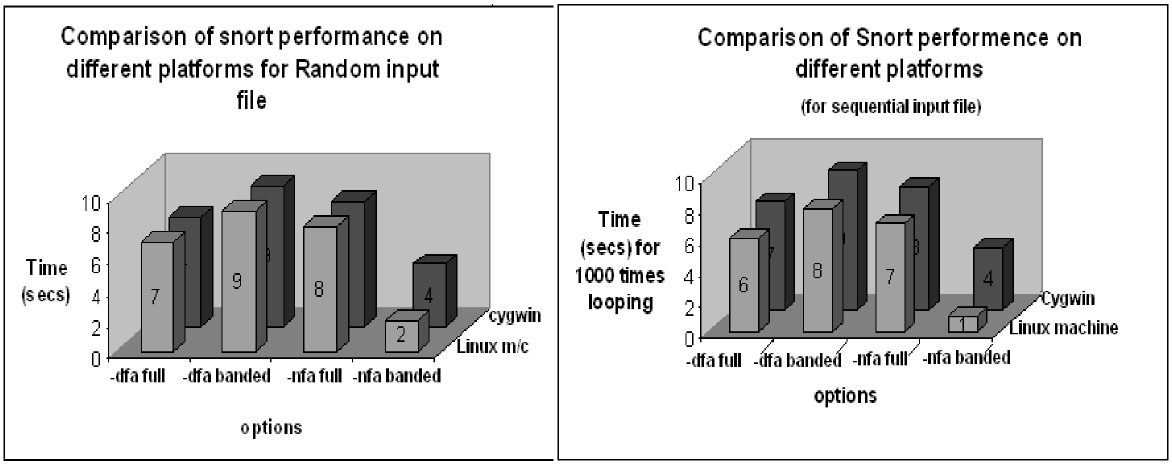


Fig 22: Performance measure of data structure on linux platform

For a 'Sequential' input file, with 2160 snort patterns, looped over 1000 times, a comparative look at the performance dependence of the implementation on platforms:



(Snort rule file with 2160 patterns, looped 1000 times)

(Snort pattern files with 2160 patterns, looped 1000 times)

Fig 23: Performance analysis on different platforms

Time requirement of Snort for pattern file with 355 patterns from worm pattern file: This test was conducted on worm pattern file with 355 patterns, looped 5k times on different platforms for different input files:

ALL GRAPHS REMOVED

Figs: Performance analysis of worm pattern files on different platforms

Time requirement of Snort for Bleeding Snort pattern files with 561 patterns on different platforms for different input files:

ALL GRAPHS REMOVED

Figs: Performance analysis of Bleeding Snort pattern files on different platforms

SECTION REMOVED: ***7.2.*****Improvements in the data structure, its implementation, and performance results.**

8. Conclusions

The growing number of security threats for modern networks, and an increasing speed of transmission of data have made it essential to develop efficient Intrusion Detection Systems, capable of working in Multi Gigabit scenario with large rule sets. Today SNORT stands out as the most widely deployed IDS, and therefore our investigations were focused on Snort's performance as an IDS. We aim at improving the performance of Snort by introducing new data structures that are capable of using the memory efficiently and give fast searching algorithm. The motivation to reduce memory requirements is two folds, the first is to keep the memory size small so that the whole structure fits into the cache and we can avoid cache misses. The second motivation to keep memory requirement small is to allow for the ever-growing number of the rules.

As a part of this work, we have studied the performance analysis of Snort as an Intrusion Detection system by varying packet size and rule set size to observe its effect on allowable bandwidth for Snort's performance. Our implementation of new algorithms into the Snort code and the evaluation of their time and memory requirements have led us to conclude that there is significant room for improving Snort's performance.

We have implemented new data structures and found them to improve both memory and time efficiency. Our future direction, will therefore be towards continued improvement to the data structure studied here as well as explore hardware based pattern matching algorithms have been proposed even in the past, for example, TCAMs. An integration of the best features of hardware and Software may help us to design very efficient Intrusion Detection Systems that will cater to the high speed operation capabilities of the future.

Bibliography

[1] A. Aho and M. Corasick. Fast pattern matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] R. Boyer and J. Moore. A fast string searching algorithm., Communications ACM, 20(10):762–772, October 1977.

[3] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection, Technical Report CS2001-0670 (updated version), University of California – San Diego, 2002.

[4] Marc Norton. Optimizing Pattern Matching for Intrusion Detection, July 2004. http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf.

[5] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, E. P. Markatos. Piranha: Fast and Memory-efficient Pattern Matching for Intrusion Detection, Proc. of Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan.

[6] M. Roesch. Snort: Lightweight intrusion detection for networks, In Proceedings of the 1999 USENIX LISA Systems Administration Conference, November 1999. http://www.snort.org/.

[7] Sourcefire. Snort 2.0 - Detection Revisited, October 2002. http://www.snort.org/docs/Snort_20_v4.pdf.

[8] S. Wu and U. Manber. A fast algorithm for multipattern searching, Technical Report TR-94-17, University of Arizona, 1994.

[9] Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection, Fang Yu, Randy Katx and T.V. Lakshman, Presentation slides, UC Berkeley.

[10] Snort Manual and Whitepapers on Rule Optimization, Detection, High-performance multi rule detection engine, Protocol Flow analyzer. All available at the Snort homepage: http://www.sourcefire.com/products/library.html