

# Traffic Analysis: From Stateful Firewall to Network Intrusion Detection System

Fanglu Guo    Tzi-cker Chiueh  
Computer Science Department  
Stony Brook University, NY 11794  
{fanglu, chiueh}@cs.sunysb.edu

## Abstract

Computer network is already an indispensable part of our modern life. To keep our network run smoothly, we need to know its condition. This calls for the necessity of analyzing the traffic (packets) on the network. In this paper, we investigate traffic analysis techniques need in stateful firewall and network intrusion detection system (NIDS). Stateful firewall analyzes packets up to their layer 4 headers while NIDS analyzes the whole packet.

The key techniques for stateful firewall and NIDS are flow state management and string matching. This paper investigates the design of flow state management and several major string matching algorithms. This paper also suggests some improvement over TCP state management and TCP flow normalization.

## Index Terms

Packet (traffic) analysis, stateful firewall, network intrusion detection system (NIDS), string matching algorithms, traffic normalization.

## I. INTRODUCTION

With the rapid growth of Internet, our life is tied to Internet more and more tightly. Governments, companies, and even individuals rely on Internet to communicate with others. Internet is now an indispensable element of our everyday life.

Given the importance of Internet in human life, it is no surprise that lots of research is invested in it. We need to be confident that it can sustain the normal communication demands of billions of people on the earth, or even some malicious abuses. So being able to identify what is happening on the network becomes an important aspect of network research. Only when we can know what is happening on the network can we control and manage the network. Otherwise, when the network fails, we may not do anything to restore its functionality. On the other hand, if we know the traffic on the network, we can use it for different kinds of applications. For example, we can deny malicious traffic to make our network safer; we can prioritize important traffic with guaranteed bandwidth for the success of our business; we can get a clear picture of the traffic distribution to plan our future network upgrade; and so on. All the above applications depend on what we can know about the traffic. The goal of this paper is to survey the techniques that enable us to understand the traffic, called traffic analysis technology. By using this technology, we will be able to know who is using the network resource, when, where, and why. This paper will also present the applications of traffic analysis technology because applications usually decide how complex the analysis should be.

The first step to analyze network traffic is to have access to packets. There are two ways. The first way is to get packets from inline devices such as routers. Since packets must go through inline devices, it is not a problem for them to get the packets. The second way to get packets is to use a wiretap device that plugs into monitored networks and eavesdrops on the network traffic. This is like to use telephone wiretap for some one to eavesdrop other people's conversation. Brian Laing [13] shows how to use the SPAN port of switches, hub, and tap to get copies of packets. The second way normally only passively monitor the traffic. So we call this type of device as monitoring device.

The second step to analyze network traffic is to set the goal of analysis. We cannot do things without purpose. We need to decide what the analysis is for, i.e., the application of the analysis. The goal will decide what information is useful. Then we can begin to think how to extract the information from the packets. Furthermore, sometimes it is even impractical for us to understand everything on the network for reasons such as the high speed of the traffic,

encryption, etc. Explicit analysis goals allow us to skip information that we don't care. So the device only needs to scrutinize portion of the packets.

Once the goal is set, the third step is to figure out ways to efficiently extract information from the packets. This normally requires decoding the packets, and even keeping state for packets that we are interested in. The key point to decode packets is that we must understand the protocol that the packets use. Otherwise every packet will just look like random bit stream and we will not be able to make any sense of it. Fortunately, most Internet traffic is formatted with stable and public protocols. This simplified the task of interpreting packets. Sometimes we also need to keep state for packets because independent packets themselves cannot give us enough information; but if we group related packets together, we can get more information.

This paper investigates major techniques for traffic analysis. Specifically, the paper mainly investigates techniques used in stateful firewall and network intrusion detection system (NIDS). Other applications of the techniques may also be briefly discussed. The techniques in this paper are limited to raw network data packets analysis. We are not going to go through all aspects of stateful firewall or NIDS. We choose stateful firewall and NIDS because they probably represent the most complex analysis techniques at these two levels: transportation layer and application layer. We will discuss how the techniques work and their rationale.

The goal of this paper is to provide a handy reference on the state of the art of traffic analysis. The paper will summarize the challenging issues in traffic analysis, and some solutions to the problems.

The rest of this paper is organized as follows. Section II presents the design issues of stateful firewall. Section III investigate the issues of network intrusion detection system. It mainly discusses the string matching algorithms and traffic normalization. Section IV discusses some miscellaneous issues related to traffic analysis. Section V concludes the paper with a summary of its content and major contributions.

## II. MATCHING AND MAINTAINING BI-DIRECTIONAL FLOW STATE: STATEFUL FIREWALL

Traditionally, firewall uses stateless techniques to filter packets. As a result, it is called stateless firewall. Stateless firewall doesn't keep any state for packets. It only has a rule set. Whenever the device receives a packet, it will compare this packet against its rule set. The matched rule decides whether a packet is allowed or not. The advantage of this approach is that it will not consume memory resource since it doesn't need memory (state) for packets. The disadvantage is that it is a computation intensive solution. While it saves memory resource, it needs use CPU computation resource to look up the whole rule set for each packet.

When the rule set contains thousands of rules, the overhead to look up rules in the rule set for each packet will become very high. In fact, this is a very hot research topic. The goal of the research is to design faster algorithm to find the matched rule in the rule set. Since the research on this topic is much mature, this paper will not address this problem.

The performance disadvantage of stateless firewall is obvious. But firewall indeed can do better if it can consider the fact that for normal communications, if the rules accept one packet, they are also going to accept all the subsequent packets that are in the same connection of the checked packet. So duplicate checking for subsequent packets of an allowed connection is really a waste. But stateless firewall doesn't take advantage of this fact because it has no memory about if a packet belongs to allowed connections.

Another disadvantage of stateless firewall is that the parameters of its rules are fixed once it is entered into the device. This makes it really ineffective for the cases that parameters of rules cannot be set in advance.

For instance, to allow clients behind a firewall to be able to receive responses from DNS servers on Internet, the firewall must accept incoming UDP packets. But the port number on the clients can be dynamically assigned from anywhere between 1024 and 64K. Stateless firewall cannot accommodate this dynamic information. To enable important UDP-based service, stateless firewall will allow UDP packets for all ports that are between 1024 and 64K. This opens security holes. For example, an attacker can set up a UDP daemon on a compromised host with a port between 1024 and 64K, say 5000. The attacker will be able to connect to this host on port 5000 to remotely control it even if there is a stateless firewall protecting this host.

For TCP connections, the port problem is a little bit better. TCP connections require SYN packet to set up connections. So stateless firewall can just block any incoming TCP packets whose SYN flag is set in order to get rid of the problem shown above for UDP. Unfortunately, sometimes we indeed need incoming TCP connections and its port is dynamically negotiated. For example, when a client behind the firewall initiates an active FTP data

connection, it requires a TCP connections initiated from outside of the firewall to come in. But since the port on the client is dynamically negotiated, the firewall cannot know it in advance. So either the application fails if firewall blocks its connections or the firewall accepts all TCP connections initiated from outside to its internal hosts. If the firewall allows any Internet initiated connection comes in, TCP will also have the same problem as what we discussed above for UDP.

Another danger for a wide allowed port range is that it enables port scanning. NMAP [8] is one of the port scanning tools. They can send out packets to detect if the ports are open or not. To detect a UDP port, it will send UDP packets to a port by watching if there is ICMP port unreachable message coming back. Because stateless firewall doesn't block this packet, the probing packet will reach an internal host. If the port on the host is open, there is no response to NMAP. Otherwise the host will send ICMP port unreachable to NMAP. To detect if TCP port is open, the NMAP tool can send out TCP FIN packet. Closed ports tend to reply with RST packet while opened ports tend to ignore the packet.

The basic guideline on designing firewall rules is to deny everything except to allow what we explicitly need. But stateless firewall must open its gate wider because it cannot handle dynamic ports. This defect greatly decreases the security protection of the firewall.

To address these shortcomings, stateful firewall is introduced. The basic technique is to create state for each connection. For TCP/UDP, the connection is identified by the protocol number, source IP address, source port number, destination IP address, and destination port number in the packet. For most bi-directional ICMP traffic type, the connection can be identified by source IP address, destination IP address and identifier.

When a packet comes in, the device will check its state to see if this packet is the subsequent packet of an existing connection. If it is, it will be accepted or denied based on the policy to this connection. If it is the first packet of a connection, it will be checked against the filter rules. If it is accepted, state will be created for this packet. If it is denied, creating state for this packet is optional. Since most subsequent packets of a connection will match its state in the firewall first, the slow process to match against filter rules is only used for the first packet. The process to match a packet against state in firewall can be implemented as a hash whose lookup speed normally is very fast. So these techniques can greatly enhance the performance of firewall. This is similar as that the cache techniques can improve the CPU access speed to memory.

The second shortcoming of stateless firewall can also be solved by stateful firewall. For the incoming UDP packets of an internally initiated UDP connection, the policy can be set as allowing incoming traffic of internally initiated connections but denying all other traffic. When a UDP packet comes in, the firewall first checks its state. If this packet belongs to an internally initiated connection, it is accepted. Otherwise it is denied. So the firewall can deny all unsolicited incoming UDP packets without blocking the return packets of internally initiated UDP connections. The firewall policy now is as strict as possible.

For externally initiated TCP connections with dynamic destination port like active FTP data connection, the solution is to analyze the FTP control connection. By understanding the information exchanged in the FTP control connection, the dynamic port of the active FTP data connection can be identified. Then the firewall sets up state for the forthcoming externally initiated TCP data connection. When a TCP packet comes in, the firewall will check its state, if it matches existing connections or the temporary state, the packet is accepted. Otherwise, it is dropped. So for unsolicited incoming TCP traffic, it will be denied regardless it has SYN flag or not. This can even block port scanning which may use packet with FIN or ACK flag. While the policy is as strict as possible, externally initiated active FTP data connections can still go through firewall.

Though keeping state for connections entail many advantages, there is some price to pay. The issues are as follows.

**IP fragmentation.** For TCP and UDP packets, we use port number as part of the identifier of the connections. When IP packets are fragmented, the fragments don't have port number any more except for the first fragment. The first fragment may not have all port numbers either if the sender deliberately constructs the IP fragments. Fortunately, for stateful firewall, this is not a problem. The firewall can just accept all fragments except the first fragment. The first fragment still goes through all the scrutiny of firewall if it has all port numbers. If the first fragment doesn't have port number, it can also be accepted. The rationale is that we will be able to control the whole IP packet as long as we can control the first fragment. If we drop the first fragment, the other fragments are no use to the end hosts even if firewall accepts them because IP cannot reassemble the original IP packet and will finally drop all fragments accepted by the firewall. If the first fragment doesn't have all port numbers, that means

the payload of IP packet is less than 4 bytes. Since the IP fragment offset is in units of 8 bytes, the fragments don't have ways to fit the gap between the first fragment and its successive fragment. So the end hosts will finally drop all fragments too.

**TCP state management.** Do we need special treatment for TCP connections? Can we just treat TCP the same way as UDP? Several issues make it more desirable to treat TCP differently. a) TCP connections can last long period of time. If we use short timeout value as UDP, the long lasting TCP connections may be broken by firewall. If we use long time out value, it will make firewall vulnerable to DoS attack. b) If firewall breaks existing TCP connections, unlike UDP sockets, TCP sockets will linger around for a long period of time. c) Unlike UDP which will not respond UDP packet (though it may respond with ICMP) regardless it is open or not, TCP may respond with RST packet for port scanning (as in [11]). If we handle TCP the same way as UDP, we may think that this flow is two-way traffic. As a result, the firewall may incorrectly think that it is a valid flow. But in fact it is just malicious port scanning and may use up our state resource quickly. So it is better to treat TCP differently from UDP. By tracing the state of TCP connections, we can get more benefits than what we lose. The benefits are: a) we can solve all the above problems. b) We can even protect servers from SYN-flooding attacks.

When we treat TCP differently from UDP, we can set up longer timeout value without worrying about state DoS. Longer timeout value will decrease the chance that firewall breaks long lasting TCP connections. Fewer broken TCP connections also help host to reclaim their connection state quickly. By tracing the TCP connection establishment and teardown, we can reclaim state from TCP connections quickly. In the case of UDP, the only way to reclaim state is timeout. That is why UDP must use smaller timeout value. In the case of TCP RST packet, since we know its meaning, we won't create state for it.

Cisco's stateful firewall [9] implement an interesting feature to protect servers from SYN-flooding attacks. A SYN-flooding attack is a kind of attack that an attacker floods a server with TCP SYN packets and forged source IP address. Because the source IP address is forged, the connections cannot be established. But the server will keep the half open connections for a long time. Eventually the server is overwhelmed and cannot accept any TCP connection request regardless of whether it is valid or not, thereby preventing legitimate users from connecting to the server.

There are two modes in Cisco's implementation: intercept mode and watch mode. In intercept mode, the firewall will accept the connection requests from clients on behalf of the server. Only when the client can finish the three-way handshake will the firewall open a second TCP connection to the server on behalf the client. Then the firewall will knit the two connections together. Because SYN-flooding attack cannot finish the three-way handshake, server will not be invoked at all. Firewall itself will timeout half-open connections quickly so as to reclaim the state resource and protect itself from SYN-flooding attacks.

This mode can be implemented as the method proposed in [12]. As in figure 1, the client first send TCP SYN with sequence number CSEQ to firewall. The firewall may reply with sequence number FSEQ. If the connection finish three-way handshake, the firewall will send TCP SYN with sequence number CSEQ to server. This way, when firewall knits the two TCP connections, it needn't change the sequence number of the packets from the client to the server. But the server may reply with sequence number SSEQ in response to the request from the firewall. SSEQ normally will be different from FSEQ as delta. So for the packets from the server to the client, the firewall need to adjust the sequence number by delta. The same thing happens for the ACK number for packets from the client to the server. In this mode, firewall doesn't need to keep track of the sequence number change. But it needs to change the sequence number of all packets.

In watch mode, the firewall just monitors the connections. If a connection fails to get established in a configurable interval, the software intervenes and terminates the connection attempt by sending TCP RST to both servers and clients. So the server will reclaim its socket resources upon receiving the TCP RST. In this mode, firewall need to keep track of the sequence number as its state. But it needn't adjust the sequence number of packets.

Another question for TCP state management is when to create state for TCP connections. For UDP and ICMP, since it is connectionless, we will create state for any packet if we don't have state for it yet. But for TCP, it is connection-oriented. Do we only create state for TCP packets with SYN flag set or any TCP packet without FIN or RST flag?

The advantage to create state for any TCP packet without FIN or RST flag is obvious if we want to keep old TCP connections going. Otherwise, old TCP connections will break when the stateful firewall loses its state or start from clean state because the firewall may not allow the return traffic of the old connections. In [11], Lance Spitzner

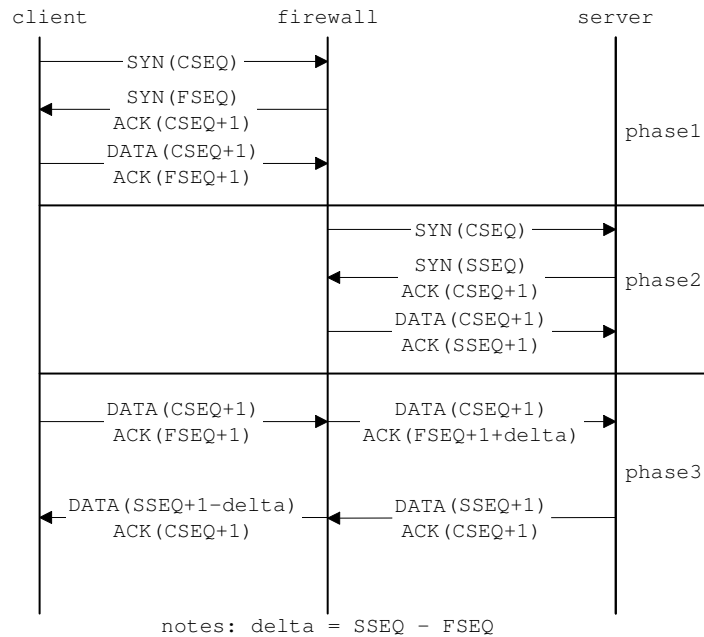


Fig. 1. Intercept mode for SYN flooding protection. In phase 1, firewall will accept the connection from client. Only when client can finish three-way handshake will the firewall to go to phase 2. In phase 2, firewall use the same sequence number CSEQ to connect with server. In phase 3, firewall only need adjust the sequence number for packets from server and the ack number for packets from client. The adjustment is a constant value. So the adjustment is trivial.

raises the Denial of Service (DoS) problem if a person inside the firewall does port scanning. This person may use up all the state resource of the firewall because the firewall will create state for these port scanning packets and set a timeout value as long as 60 minutes. A solution to this problem is to assign this state with shorter timeout value. Then it will be treated the same way as a UDP connection. For state created from TCP SYN packet, we can still assign longer timeout value to it so that we can decrease the chance that connection is broken by the state timeout event of the firewall.

**TCP Sequence number checking.** Do we need to match the sequence number of the packets in the three-way handshake? No. Normally no one can easily forge the SYN/ACK packet from the server for a request. So either there is no SYN/ACK or there is a correct one. For the ACK packet in the three-way handshake, if the source IP address of the SYN packet is forged, the ACK packet normally will not be able to match the sequence number of the SYN/ACK packet. If the client sends the ACK, it indeed helps the server to reclaim its state because server will notice the sequence number in the ACK packet is wrong. That is why SYN-flooding attack does not send ACK to fool the firewall.

The flow state is not only used for stateful firewall, it is also widely used in other applications such as layer-4 switch, NAT, application layer traffic normalization and extraction and so on. But in different applications, the state information maintained may be different. For instance, the application layer traffic extraction will require that the flow state also maintains the sliding window of the receiver. The packet processing requirement may also be different. In stateful firewall, fragments can be handled without reassembly. But port based NAT will require the fragments to be reassembled. Otherwise, the NAT cannot know how to demasquerade the return IP fragments of the flow. Without reassembly, the fragments don't have port number. Then NAT cannot know which flow the fragments belong to. As a result, the NAT cannot demasquerade the return IP fragments.

In summary, the idea of creating flow state for connections can be widely used for different purposes. In different applications, the flow state may need to be tailored to its specific environments.

### III. ANALYZING APPLICATION LAYER TRAFFIC: NIDS

#### A. *The need for application layer traffic analysis*

While in some cases, simple techniques like stateful firewall can already satisfy our needs, in other complex situations, we need to go further to analyze the application layer traffic, i.e., the payload of transport layer. This is also called content-based traffic analysis. Here are some applications that require application layer traffic analysis.

- **Network Intrusion Detection Systems (NIDS).** Paper [20] presents one example to illustrate why NIDS is necessary. The Microsoft IIS Web server [version 1155.0, 5.1] has a bug. If it encounters a series of forward slashes inside of the “Host:” header of an HTTP request, it will crash. A normal stateful firewall will allow traffic to the web server. It does not check the content of the HTTP request to the web server. So an attacker can still crash the web server even if the web server is protected by a firewall. On the other hand, NIDS can check every host field in each incoming HTTP request for pattern: `////////////////////`. So whenever the attackers use this forward slashes attack method, they can be detected or even stopped immediately by NIDS. The attack information, forward slashes, is only available in the application layer of the packets. Without application layer traffic analysis, this kind of attack cannot be detected or stopped.
- **Content-based switch.** Paper [12] presents a scenario why application level information is useful for switching traffic among server clusters. Web server clusters are common for large web providers. In typical configuration, there is a layer-4 switch which distributes connections among the nodes of a cluster. The layer-4 switch essentially uses similar techniques as we described in the stateful firewall. It is content blind. The layer-4 switch requires all nodes in the cluster either to be completely replicated or to share a common file system. Otherwise, some nodes may not be able to service the request blindly dispatched from the layer-4 switch if they don’t have all the content of the web site. Content-Based switch, on the contrary, can check the Request-URI in the HTTP request and partition the content distribution among the nodes of the cluster. This will decrease the overhead of management and increase the performance of web servers because of better cache locality. Same as NIDS, content-based switch also needs the information that is only available from the application layer of the packets.
- **Protocol aware firewall.** In the description of stateful firewall, we mentioned that stateful firewall can create state for a FTP data connection. How does the firewall know the parameters of the forthcoming data connection? In fact, this information is only available from the application layer traffic of the FTP control connection. The string such as “PORT 192,168,0,112,16,71” from the client in the control connection will tell the server that the client is waiting for the data connection on port 4167 (computed from  $16 * 256 + 71$ ). To get this dynamic port 4167, protocol aware firewall must understand the FTP protocol and search for the command “PORT” in the TCP payload.
- **Application (protocol) recognition.** Paper [15] presents the impact of P2P traffic on the WAN network resources. P2P applications like Gnutella and KaZaA can consume large portion of the WAN network resources because they often are used to exchange recreational content like movies which can be 100s Mbytes for a single file. It leaves little bandwidth for business critical applications. Furthermore, P2P applications can use port 80 or masquerade as typical traffic such as HTTP. This makes normal firewall techniques useless to block its port. So to limit the max bandwidth of P2P applications, we need to check the application layer of the packets to distinguish P2P applications from other traffic. Then we can contain or even block the traffic of P2P applications. The work at [18], [40], [16], [17] are some examples on what application recognition can do and how to do it.

#### B. *Techniques to analyze application layer traffic*

The basic techniques to analyze application layer traffic is string matching. The string matching problem is similar as that we need to find a string (pattern, signature) in a text file. The difference is that we may require searching for a set of strings instead of one string. And the text that we are going to check is the application layer byte stream of the packets.

For NIDS, we need search the application layer byte streams for attack patterns such as forward slashes. Normally the NIDS has a set of signatures which have to show up in the application layer byte stream of the packets to succeed an attack. That is to say, the signature is necessary for one to attack a system. But the presence of a signature does not mean that there is definitely an attack or the attack will succeed. For each packet, the NIDS will



<p>Before Shift:</p> <p>one plus two two plus th<b>ree</b> equals five</p> <p>After Shift:</p> <p style="padding-left: 100px;">one plus two two plus three equals five</p> <p>(a) bad character heuristic</p>	<p>Before Shift:</p> <p><b>two</b> plus two count to <b>two</b> hundred thirty</p> <p>After Shift:</p> <p style="padding-left: 100px;"><b>two</b> plus two count to <b>two</b> hundred thirty</p> <p>(b) good suffix heuristic</p>
---	--

Fig. 3. The examples show how bad character heuristic and good suffixes heuristic work. (a) is for bad character heuristic and (b) is for good suffixes heuristic.

signature matching, flow state is also necessary for application recognition.

Although it looks simple to analyze application layer traffic, several issues make the problem challenging. Application layer traffic analysis mainly has the following issues: performance overhead and evasion tactics which can defeat application layer analysis. The following will mainly focus on NIDS. But the problems and solutions may also apply to other application layer traffic analysis scenarios.

### C. Performance issue of application layer traffic analysis

The simple way to match strings in the application layer byte streams is just to use normal string matching algorithms without considering the traffic characteristics. The popular NIDS, snort [22], use this method. Unfortunately, performance is an issue for even the best known string matching algorithms. Mike Fisky, et al [30] show that the string matching cost can be as much as 31% of the total processing time by profiling the snort code.

In another paper [24] presented by Lambert Schaelicke, et al, snort cannot even support moderate speed network. In their setup, the snort only monitors one-way traffic. The traffic speed is kept constant at 100 Mbps. The computer system is Pentium 4 Xeon 2.4 Ghz with 1 Gbytes DDR memory. The authors keep adding rules to snort until snort cannot check all packets. Then they can know how many rules snort can check without missing packets. The result is quite disappointing. Snort can only support 94 rules for packets with 64-byte TCP payload and 217 rules for packets with 1452-byte TCP payload. All the rules are payload string matching rules. So this test showed the performance of the string matching of snort. As a point of comparison, the rule file [23] in snort has 1086 payload string matching rules. And the total number of rules is 1270.

There are two ways to get around of the bottleneck of string matching. One is to design faster string matching algorithm, the other is to decrease the workload of string matching.

1) *Faster string matching algorithm*: We will present the main ideas of the string matching algorithms proposed in the literature. The performance comparison is also presented. The algorithms can be classified to two categories: single pattern match and multi-pattern match. For single pattern match, if there are multiple patterns, the text will be searched multiple times. Each time there is only one pattern is searched. For multi-pattern match, all the patterns are searched simultaneously. We will present single pattern match first. Then we will talk about multi-pattern match. In each category, the order is based on the time they are published.

#### a) Single pattern matching algorithms:

**Boyer-Moore (BM)**. Boyer and Moore [27] proposed this algorithm in 1977. This algorithm is widely regarded as having very good average-case performance for string matching. As a result, it is most often used in text editors. It is also used in Snort for application layer signature searching.

Let's see an example first to understand how it works. Figure 3 (a) is an example from [31]. It shows a pattern P: "one plus two" that needs to be searched in the text T. The left of the pattern is aligned with the left of the text. The comparison start from the right of the pattern. The first comparison is between character 'o' from the pattern P and character 'r' from text T. These two character mismatch. But we find that character 'r' is not in pattern P. So we can move the pattern P to the right by the length of pattern P. If 'r' is in pattern P, P can move to the right by B(r). Let's define some terms to explain the computation.

Pattern P is composed by characters:  $p_1, p_2, \dots, p_n$ .

$$B(r) = \min\{i | p_{n-i} = r\}. \quad i \in [0, n - 1].$$

$B(r)$  is pre-computed from pattern P.

In plain English,  $i$  is the index of the characters of P. The index of the right most character of pattern P is 0 and increases from right to left.  $B(r)$  is the index of character 'r' when it first appears in P if we read P from right to left.

Why can we move P to the right by  $B(r)$  when 'r' is in P? We know that all the characters from  $p_{n-B(r)+1}$  to  $p_n$  are not character 'r'. That is what  $B(r)$  is defined. Now that all characters from  $p_{n-B(r)+1}$  to  $p_n$  cannot match character 'r'. We can safely move P to the right by  $B(r)$ . This is called *bad character heuristic*.

Let's see figure 3 (b) for another heuristic. In figure 3 (b), character 'o' in text T mismatches character 's' in pattern P. From bad character heuristic,  $B(o)$  is 0. It cannot give us any help. But we find our matched suffix "two" in text T appears in P more than once. Then we can move our pattern P to the right to align the suffix "two" in the text T with the second substring "two" in the pattern P. The movement offset is defined by formula  $G(\text{suffix\_of\_P})$ .

$G(\text{suffix\_of\_P}) = \text{index\_of\_second\_appearance}(\text{suffix\_of\_P}) - \text{index\_of\_first\_appearance}(\text{suffix\_of\_P})$   
if suffix\_of\_P appears in P more than once.

$G(\text{suffix\_of\_P}) = n$ . otherwise

$\text{index\_of\_first\_appearance}(\text{suffix\_of\_P}) =$  the index of the first character of  $\text{suffix\_of\_P}$  in P.

$\text{index\_of\_second\_appearance}(\text{suffix\_of\_P}) =$  the index of the first character of  $\text{suffix\_of\_P}$  in P when we find  $\text{suffix\_of\_P}$  again in P. The search order is from right to left.

$G(\text{suffix\_of\_P})$  is also pre-computed for pattern P. The reason why we can safely move by  $G(\text{suffix\_of\_P})$  is that  $\text{suffix\_of\_P}$  in text T cannot match other substring of pattern P until the second appearance of  $\text{suffix\_of\_P}$  in P. In figure 3 (b), the movement is 9.

If  $\text{suffix\_of\_P}$  doesn't show up in P again,  $\text{suffix\_of\_P}$  cannot match P any more. So we can move P all over to the next character of  $\text{suffix\_of\_P}$  in T. This movement is n, the length of P. This scheme is called *good suffixes heuristic*.

When both heuristics are used, the movement value is the bigger one of the two because either one can guarantee the movement is safe. Using the bigger movement won't skip potential match.

**ExB and  $E^2xB$ .** E. P. Markatos, K. G. Anagnostakis, et al. [32], [33] proposed the exclusion based string matching algorithm in 2002 and 2003. The idea is that for pattern P, if any character  $p_i$  doesn't show up in text T, we will not be able to find P in T. This algorithm is mainly designed for NIDS. It has two assumptions. The first assumption is that most traffic will not trigger patterns. So by using this algorithm, we don't need invoke expensive algorithm such as Boyer-Moore algorithm to verify the safety of the traffic. The second assumption is that the text T is not too big. Otherwise the effectiveness of this idea will decrease dramatically because the chance that text T has all the characters of pattern P will increase as text T get big. This forces the NIDS not to accumulate too much data to do pattern matching. Given the trend that NIDS are more and more data stream (e.g. TCP data stream) oriented, this limitation may make this algorithm less valuable.

Figure 4 is the pseudo-code of ExB algorithm. There are two steps to exclude patterns that will not match the packet. In the first step, function `pre_process()` will process the characters in packet (`text_T`) to record their existence in array `exists[256]`. In the second step, function `search()` will check pattern `pattern_P` is in the text `text_T` or not. If any character of pattern `pattern_P` doesn't show up in array `exists[256]`, we can conclude that pattern `pattern_P` will not match text `text_T`.

If there are multiple patterns, the second step will be called repeatedly. But the first step is only called once for all patterns.

To decrease the false positive, the idea can be generalized to use more bits of characters to create array `exists[]`. For common packets with nearly 1500-byte data, the chance that all characters show up in a packet is high. So the 8-bit array (256 elements) will have high false positive. Paper [32] suggests that 13-bit array ( $2^{13}$  elements) will be a good tradeoff between memory usage and false positive.

The major enhancement of  $E^2xB$  to ExB is to use integer for variable `exists[256]`. The reason for this change is that clearing `exists[256]` for each search (each packet in NIDS) is too expensive. When `exists[256]` is integer, we can assign the ID of packets to `exists[256]` in `pre_process()`. In `search()`, we check if `exists[x]` is equal to the ID of the packet. If `exists[x]` is not equal to the ID of the packet, it means character x of pattern P doesn't show up in this packet (text T).

```

boolean exists[256];
pre_process(char * text_T, int len_of_T)
{
    bzero(exists, 256/8); // clear array
    for (int idx = 0 ; idx < len_of_T ; idx++) {
        exists[text_T[idx]] = 1;
    }
}
search(char * pattern_P, char * text_T, int len_of_P, int len_of_T)
{
    for (int idx = 0 ; idx < len_of_P ; idx++) {
        if (exists[ pattern_P[idx] ] == 0)
            return DOES_NOT_EXIST ;
    }
    return boyer_moore(pattern_P, len_of_P, text_T, len_of_T);
}

```

Fig. 4. Pseudo-code of ExB algorithm. In function `pre_process()`, we will create a bit map for all the characters of text T. In function `search()`, we will check the bit map to see if every character of pattern T is marked in the bit map by the function `pre_process()`. If any character of T is not marked in the bit map, it means text T doesn't have some character of P. Then we can conclude immediately that we will not be able to find P in T any more. If all characters of P are in T, we need use other algorithms such as Boyer-Moore algorithm to further check.

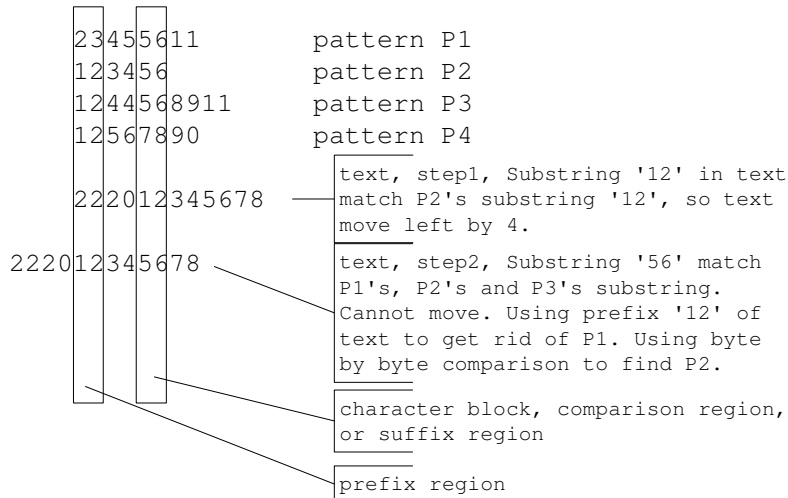


Fig. 5. The example to show how Wu and Manber's algorithm works. Character block is used for bad character heuristic. It is also used to search the hash table of patterns, P1, P2, P3, and P4. Characters in prefix region is used to filter out ineligible patterns quickly.

### b) Multi-pattern matching algorithms:

**Aho-Corasick (AC).** Aho, A. V. and M. J. Corasick [26] proposed the algorithm in 1975 for searching multiple patterns simultaneously in text T. The idea is to create an automaton. The automaton consumes one character from text T each time. The character will drive the automaton to a new state which represents all the partially matched patterns. Multiple regular expressions can also be matched with this algorithm. Algorithms based on Aho-Corasick algorithm are widely used in current compiler technology.

**Wu and Manber's algorithm.** Sun Wu and Udi Manber [28] proposed their multi-pattern matching algorithm in 1994. It mainly uses the bad character heuristic of Boyer-Moore algorithm. But since the large number of patterns will decrease the chance to get bigger movement, this algorithm use a block of characters, say 2 or 3 characters as a block, to find a movement offset.

Let's see an example to understand how it works. Figure 5 shows that we are going to find 4 patterns, P1, P2, P3 and P4 in the text T. The comparison region covers 2 characters. It is also the suffix region and the character block that we are going to use to find patterns in the text.

In step 1, "12" of text T is in the character block. Using precomputed movement table, we know "12" will match P2's substring "12". So the text is moved to left by 4. The precomputed movement table is got by using the same

idea as Boyer-Moore algorithm.

In step 2, “56” of text T is in the character block. Now since “56” of text T match the substring “56” of P1, P2, and P3, the character is good. The bad character heuristic cannot tell us how much to move. The algorithm pre-populated a hash table for the character block of all patterns in the suffix region. Then the algorithm will be able to get all potential patterns from the hash table quickly. In this case, the patterns are P1, P2, and P3.

To eliminate some patterns that will not match the text, the algorithm will get rid of the patterns that don’t match the text in the prefix region. In this case, pattern P1 will be eliminated quickly without checking all characters of pattern P1. Now only P2 and P3 left. The algorithm will check all remaining patterns against the text T. If no pattern matches the text T, the text T will move left by one and restart from step 1.

The key point of this algorithm is that it assumes it can move the text quickly to the left from the bad character heuristic because of its larger character block (2 or 3 characters instead of 1 character). If the substring in the character block of the text match patterns, the bad character heuristic will not work any more. The solution to this problem is to use hash to find potential patterns and use prefix to further eliminate ineligible patterns. Hopefully the number of remaining patterns will be small. Then we can use naive comparison to verify if the remaining patterns can be found in the text.

**Kim’s algorithm.** Sun Kim and Yanggon Kim [29] proposed their encoding and hashing based multi-pattern matching algorithm in 1999. There are two basic ideas. The first idea is that patterns may only have a few distinct characters. Then we can encode the characters with less bits. This is the same idea as compression algorithms. With fewer bits to compare, we can use less number of comparison. The second idea is that we only need compare some patterns with the text instead of all patterns in each comparison. The way to find only potential patterns to compare is via hash. In fact, these two ideas are independent. Either idea can work by itself.

Let’s see an example to understand how the encoding scheme works. In the following example, the pattern string only have 3 different characters: a, b, and c. So 2-bit encoding is enough to represent the pattern. For text string, all characters (e.g. d, e, f, g) other than the characters (a, b, c) in the pattern only need one value to represent. Even if we encode all characters who are not in patterns to one value, the pattern matching will still be correct. In the following example, “abc” of text T and pattern P will match after encoding. But “de” of text T will not match “ac” of pattern P because the encoded value of “de” is not equal to the encoded value of “ac” of pattern P. This encoding scheme is like the ExB algorithm, the special encoded valued for characters not in patterns serve as the role of exclusion.

```
Pattern string: abcac
Text string:   abcdefg
```

```
Encoding scheme: a: 00, b: 01, c: 10, all other characters: 11
```

```
Encoded pattern string: 00 01 10 00 10
Encoded text string:   00 01 10 11 11 11 11
```

To search only one pattern in the text, the algorithm proposed in this paper is the naive one: character by character comparison. In our example, if the comparison fails, the encoded pattern string will move to the right by 2-bit (corresponding to one character of the original strings). But because the encoded pattern string becomes shorter, we may compare all characters of the encoded pattern string with the text in one computer instruction. For instance, the 32-bit computer needs two instructions to compare the original pattern with the text because the pattern is 5 characters long. The encoded pattern string only needs one instruction because it is only 10 bits long.

To search multiple patterns in the text, the authors proposed hash to decrease the number of patterns that need to be compared against the text. All patterns are hashed to a hash table. The key is the first j characters of all patterns. For instance, we add another two patterns to the above example: acbcaaaabb, cccaaabbbccc. The value of j is chosen to the min of the length of patterns. So in this case, the value of j is 5, the length of abcac. The hash table is pre-populated with the patterns.

In each comparison, we use the current j characters of the text as key to search the hash table. Then only potential patterns are found. Other patterns whose hash value is different from the hash value of the j characters of the text will not match the text. So it is safe to use this hash scheme to find the potential patterns.

**Setwise Boyer-Moore-Horspool (Setwise BMH).** M. Fisk and G. Varghese [30] proposed this algorithm in 2002

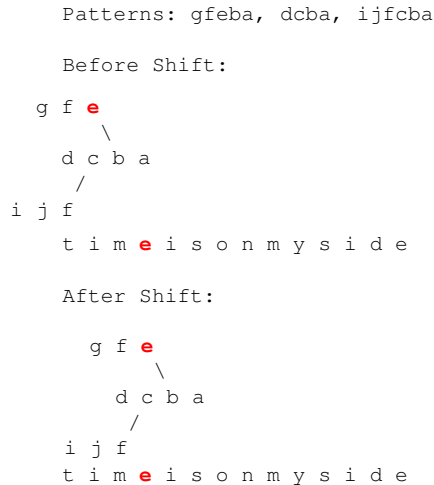


Fig. 6. Example to show how Setwise Boyer-Moore-Horspool algorithm works. In the beginning, the shortest pattern “dcba” is left aligned with text T. Comparison start from right to left. Character ‘e’ of text T mismatch patterns. From bad character heuristic, character ‘e’ of text T is found in pattern “gfeba”. So the movement is 2 and text T is moved left by 2. Then the whole process will be repeated until text T is used up or patterns are found.

based on the Boyer-Moore algorithm. The idea is to construct a trie that stores all patterns,  $P^1, P^2, \dots, P^m$  based on their suffixes. Normal trie store multiple strings based on their prefixes. If we reverse all patterns first before constructing the Setwise Boyer-Moore-Horspool trie, the Setwise Boyer-Moore-Horspool trie will be just a normal trie. Figure 6 is an example of the Setwise Boyer-Moore-Horspool trie for multi-pattern string matching and show how the algorithm works. In the figure, patterns are right aligned. Common suffixes of patterns will be stored in one node of the trie.

To begin pattern matching, the shortest pattern of all patterns is left aligned with the left of text T. The match is tested from right to left the same way as Boyer-Moore algorithm. Because common suffixes are constructed in the trie. One test to the common suffix will serve all tests for patterns who share this common suffix. When the test cannot match any node in the trie for character  $c$  in text T, the algorithm will use the following formula to compute movement offset for moving the trie to the right.

$$B\_Set(c) = \min\{B_i(c), i \in [1, m]\}.$$

Variable  $m$  is the total number of patterns.  $B_i(c)$  is the  $B(c)$  function in Boyer-Moore algorithm for pattern  $P^i$ .

$B\_Set(c)$  is conservative. The minimum of all patterns’ skip will not miss any potential match. So this algorithm is clearly correct.

**AC-BM algorithm.** C. J. Coit, et al. [31] independently proposed almost the same algorithm as Setwise Boyer-Moore-Horspool algorithm in 2002. The idea is to construct a trie that stores all patterns,  $P^1, P^2, \dots, P^m$  based on their prefixes. The common prefixes of patterns will be stored in one node of the trie. Figure 7 is an example on how it works based on bad character heuristic.

To begin test, the shortest pattern is right aligned with the right of text T. The test is from left to right. When the test cannot match any node in the trie for character  $c$  in text T, the algorithm will use  $B\_AC\_BM(c)$  to compute the movement offset to move the whole trie to the left.

$$B\_AC\_BM(c) = \min\{B'_i(c), i \in [1, m]\}.$$

$$B'(c) = \min\{i | p_{i+1} = c\}. i \in [0, n - 1].$$

Variable  $m$  is the total number of patterns.  $B'_i(c)$  is the  $B'(c)$  function for pattern  $P^i$ .  $B\_AC\_BM(c)$  is pre-computed from all patterns. In plain English,  $B'(c)$  is the index of character  $c$  when it first appears in pattern P if we read from left to right. The index starts from zero for the first character and increases from left to right.  $B'(c)$  is just the mirror of  $B(c)$  in Boyer-Moore algorithm.  $B'(c)$  will be equal to  $B(c)$  if the pattern of  $B'(c)$  is P’ and the pattern of  $B(c)$  is P. P’ is the reverse order of P.

$B\_AC\_BM(c)$  is conservative. The minimum of all patterns’ skip will not miss any potential match. So this algorithm is clearly correct.

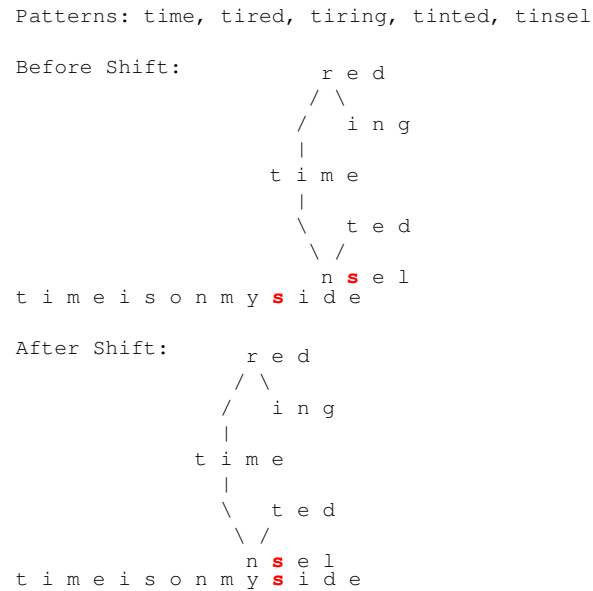


Fig. 7. Example to show how AC-BM algorithm works. In the beginning, the shortest pattern “time” is right aligned with text T. Comparison start from left to right. Character ‘s’ of text T mismatch patterns. From bad character heuristic, character ‘s’ of text T is found in pattern “tinsel”. So the movement is 3 and text T is moved right by 3. Then the whole process will be repeated until text T is used up or patterns are found.

This algorithm also uses good suffixes heuristic of Boyer-Moore algorithm. In fact, in this scheme, the suffixes in Boyer-Moore algorithm now are prefixes of patterns. The reason is that this algorithm searches from the end of the text to the beginning and compares from left to right in each comparison. But the idea is the same.

The movement offset formula is defined as  $G\_AC\_BM(suffix\_of\_P)$ .

$$G\_AC\_BM(suffix\_of\_P) = \min\{G'_i(suffix\_of\_P), i \in [1, m]\}$$

Variable  $m$  is the total number of patterns.  $G'_i(suffix\_of\_P)$  is the mirror of  $G(suffix\_of\_P)$  function in Boyer-Moore algorithm for pattern  $P^i$ . These two have the same relationship as the relationship between  $B'(c)$  and  $B(c)$ .

This formula is also conservative. So it is correct.

### c) Performance results analysis and techniques summary:

From table I, we can only know that Kim’s algorithm is faster than Wu’s. For Setwise BMH, AC-BM, and  $E^2xB$ , it is really hard to draw conclusion on which one is better because their rule set size, running time metric definition, and traffic trace are different.

In paper [25], a modified version of Wu’s algorithm is compared with  $E^2xB$ . The tests showed that the algorithms are sensitive to both packets and pattern contents. No algorithm is always better. These results are not surprising as the algorithms heavily rely on its luck to search for patterns in the text.

$E^2xB$ ’s idea in fact can be combined with any other algorithms. But its effectiveness depends on the traffic. In common case, it will help. But in worst case, say signature DoS, the algorithm indeed will decrease the performance because it takes time to exclude patterns. But since it is signature DoS, finally the algorithm didn’t get rid of many signatures. So care must be taken to avoid this situation. Signature DoS is that attackers send large volume of packets with attack signatures. When NIDS is overwhelmed with these false positive attacks, real attack may go through NIDS undetected.

In the performance report of  $E^2xB$ , the authors report the algorithm can exclude 98.4% packets. But the snort still uses 0.63 of the running time of BM algorithm. This seems suggest other parts of snort also have high overhead. They really need test the string matching part instead of the whole snort for running time comparison.

For Setwise BMH and AC-BM algorithm, it seems that AC-BM is better because it can even decrease the total snort time to be 0.3 of old BM algorithm for 786 string matching rules. While Setwise BMH’s performance will be worse than AC algorithm when the number of string matching rules greater than 100. This means the scalability

TABLE I

THE PERFORMANCE REPORT FROM EACH ALGORITHM. IT IS HARD TO TELL WHICH ONE IS THE BEST BECAUSE THEY ARE TESTED ON DIFFERENT METRICS AND SETUP. BUT GENERALLY, WE FEEL AC-BM IS BETTER THAN OTHERS.

Algorithms	Techniques	Performance test
Wu's	Using bad heuristic to skip and hash to decrease the number of potential patterns	Using 0.5 of the running time of GNU-grep. Running time is total program running time. Patterns are randomly generated from the text. Text is a randomly chosen file.
Kim's	Compact encoding and hash to decrease the number of potential patterns	Using 0.5 of the running time Wu's program. Running time is total program running time. Patterns are randomly generated from dictionary. Text is a randomly chosen file.
Setwise BMH	bad character heuristic and suffix trie	Using 0.2 of the running time of BM algorithm. 0.67 of AC algorithm. Running time is searching algorithm running time. Traffic is captured packets. 310 web string matching rules.
AC-BM	bad character and good prefix heuristic plus prefix trie	Using 0.3 of the running time of BM algorithm. Running time is snort total running time. Traffic is captured packets. 786 string matching rules.
$E^2xB$	Using character map to exclude ineligible packets	Using 0.63 of the running time of BM algorithm. Running time is snort total running time. 98.4% packets can be excluded. Traffic is captured packets. 1575 string matching rules.

of Setwise BMH is not good.

These two algorithms are just mirrors to each other. The AC-BM algorithm also uses good suffixes (to be exact, it is prefixes) heuristic. In our opinion, this should be useful. For thousands of patterns,  $B\_AC\_BM(c)$  could be very small for each comparison because a lot of patterns will increase the chance to find character  $c$  immediately from the beginning of the patterns. So the help of bad character heuristic will decrease. On the other hand, The good prefixes heuristic will become more important because the chance of the appearance of repeated prefixes will increase as the number of prefixes increases. This is also the reason why the performance of Setwise Boyer-Moore-Horspool algorithm gets worse when the number of patterns increases.

Another difference lies in the question of which is more common: common prefixes or common suffixes. We know that the characters in one node of the trie only need one comparison for all patterns that share this prefix or suffix. That is the big saving that will speed up multi-pattern matching. So the effectiveness of the two algorithms will highly depend on the nature of the patterns.

To summarize, there are mainly three ways to decrease the number of potential patterns.  $E^2xB$  uses exclusion to get rid of ineligible patterns by pre-computing a character map for the packets. Wu's algorithm and Kim's algorithm uses hash to get a list of potential list of patterns. And the Setwise BMH algorithm and AC-BM algorithm use a trie to compare multiple patterns for one comparison. In each character comparison, the trie can get rid of all patterns that don't have this prefix or suffix. In terms of effectiveness, the trie is the best for running time. But it requires more memory and pre-processing time. Pre-processing time may not be an issue for NIDS though.

As to the bad character heuristic and good suffix (or prefix) heuristic, the effectiveness of bad character heuristic will decrease as the number of patterns increase. And the effectiveness of good suffix (or prefix) heuristic will increase as the number of patterns increases.

2) *Decreasing the workload of string matching*: The first way to decrease the workload of string matching is to match the packet only for potential signatures. From the report of paper [30], around 310 rules in the snort rule set (over 1000 rules) apply to HTTP traffic. So there is no need to check every packet against all other 700 rules in the rule set if we know the packet is HTTP packet. Snort can do this by filtering the port number of packet. For instance, the 310 HTTP rules only apply to packets whose destination port is 80. Other 700 rules apply to packets whose destination port is not 80. Then the HTTP packet doesn't need to match all signature but only the potential possible signatures. Snort uses the data structure showed in figure 8 to filter out the packets that need to be checked for potential signatures. The nodes in the first row are chain header nodes. The nodes below each chain header are chain option nodes. Only when a packet matches the parameters of one chain header node, will the signatures in the chain option nodes be searched in the packet. For example, packets with destination port 80 will be checked against only with chain option nodes below the chain header node which requires the destination port to be 80. In

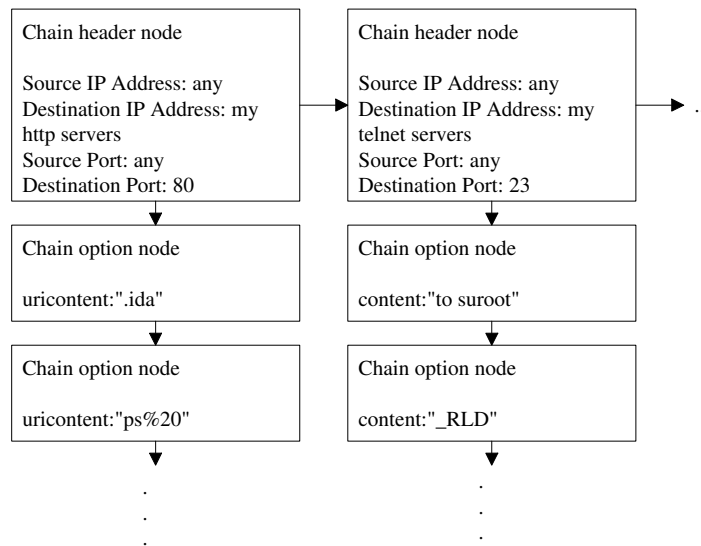


Fig. 8. The snort use chain header node to filter out the packets the patterns cannot match. In each chain option node, the patterns can be stored. The packet will be checked against chain header nodes from left to right. If a match is found, The packet will be checked against the patterns stored in the chain option nodes which belong to that chain headers.

the snort implementation, the chain headers are searched sequentially.

The second way to decrease the workload of string matching is to match only the potential part of the byte stream. For instance, the NIDS for web servers may reasonably assume that intrusion only comes from outside. Then it doesn't need to monitor the outgoing traffic but only the incoming traffic. Even for the incoming traffic, the NIDS don't need to check all the incoming HTTP traffic. For example, a user may use "POST" to upload a file to the web servers. It is not necessary to check the byte stream of the uploaded file. Furthermore, in the HTTP request header, not all fields are of the same importance. Normally we only need to check the URL of the request. To achieve this goal, paper [23] mentioned the idea of protocol aware NIDS. The protocol aware NIDS doesn't search for the whole byte stream for signatures. Instead, it can understand protocols and extract only the byte stream of fields that may contain attacks. Only this small part of data stream is checked for attack signatures. In snort, the http\_decode preprocessor can extract the URL in HTTP traffic and people can use the "uricontent" to design rules to check the content of URL.

In fact, the snort data structure showed in figure 8 may also decrease the volume of byte stream that need to be checked against the signatures. For instance, a NIDS may monitor both web servers and FTP servers. The FTP traffic will not be checked against web attack signatures by using the filtering of chain header nodes. But this is not always true. If the NIDS in our example only monitors web servers, the chain header nodes cannot help to decrease the volume of byte stream to be checked. But if the NIDS is HTTP aware, it can use its HTTP knowledge to skip some data. So generally, the more knowledge the NIDS has, the more efficient and powerful it is.

#### D. Evasion issue of application layer traffic analysis

1) *Evasion at layer 3 and layer 4:* In discussing application layer traffic analysis, we assume we can get the application layer byte stream. Unfortunately it is not easy to get the byte stream, especially the correct one. In paper [36], [37], the issue that people can use tricks to evade the check of application layer traffic analysis software was raised.

For instance, NIDS often uses monitoring device at the edge of the network to detect intrusion for all the hosts inside the internal network. Attackers can fool the NIDS with different byte streams from what the hosts get. Here is an attack mentioned in [20].

The default root directory for Microsoft IIS web server normally is at C:\INETPUB\WWWROOT. An attacker may use the following request to access files and system resources outside C:\INETPUB\WWWROOT. The request can be:

```
GET /USSales/revenues/../../../../../../../../WINNT/SYSTEM32/CMD.EXE?/C+dir+C:\+/S
```

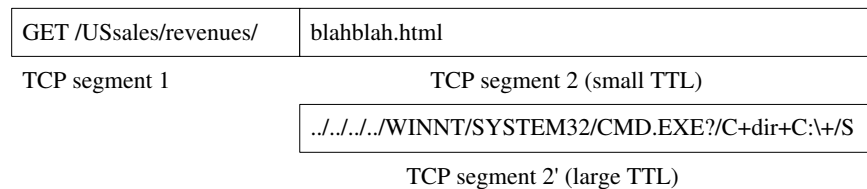


Fig. 9. Example to show how to evade the check of NIDS. First, attacker can break the request to multiple packets. If NIDS doesn't reassemble packets, it cannot get the whole request. Second, if the NIDS cannot tell which packet will arrive end host for overlapped segment 2 and segment 2', it cannot tell the end host is attacked or not.

The IIS will interpret the above request as:

```
\WINNT\SYSTEM32\CMD.EXE /C dir C:\ /S
```

Then CMD.EXE will be loaded. CMD.EXE in turn will execute command "dir C:\ /S" to get the list of all files on drive C:. The file list is finally sent to the attacker via HTTP response.

For a NIDS to detect the above attack, first it must be able to get the whole string:

```
"GET /USsales/revenues/../../../../WINNT/SYSTEM32/CMD.EXE?/C+dir+C:\+/S"
```

The attacker can simply use tools to break the whole string to be carried in several TCP segments. If the NIDS cannot reassemble TCP segments, it will not be able to get the whole string. Second, even if the NIDS can reassemble TCP segments, it may not be able to get the correct string. The attack can be done as figure 9 shown. The attacker sends out both TCP segment 2 and TCP segment 2'. TCP segment 2 is only different from TCP segment 2' with content and TTL. TCP segment 2 has smaller TTL and benign content. When the web server is not on the same network as NIDS, the web server may not receive TCP segment 2. The NIDS will receive both TCP segment 2 and TCP segment 2'. But it may ignore TCP segment 2' because it already received TCP segment 2 and think that TCP segment 2' is a duplicate. As a result, the attacker evades the detection of the NIDS by fooling it with incorrect content.

Application (protocol) recognition software may also has the same evasion problem as NIDS. For example, the Network-Based Application Recognition (NBAR) feature of Cisco IOS [40] can classify packets based on their protocol. After packets are classified, their speed can be controlled. People may want to limit the speed of FTP download speed to get faster web browsing response. If the FTP software uses evasion techniques, the NBAR software may not be able to correctly get the byte stream about the dynamic port information exchanged in the control connection. As a result, it can not get the correct port that the data connection is using. Then it cannot recognize FTP downloads traffic any more.

Not every application layer traffic analysis needs to take care of the evasion issue though. The FTP data connection port tracking problem also exists in stateful firewall. But stateful firewall needs not care about evasion. If some one wants to evade the firewall, his data connection will be blocked because firewall cannot track its data connection port. As long as most FTP software can be tracked correctly, the firewalls don't need to handle the evasion problems.

Among the evasion tricks, some must be handled. For instance, the IP fragmentation and TCP segmentation must be handled. Otherwise, traffic analysis software may not get strings long enough to check signatures. If the traffic analysis software fails to reassemble packets to long strings, the attacker will evade the check of traffic analysis. This is false negative.

Other tricks may just create ambiguity to traffic analysis. In the above example, if the NIDS also checks TCP segment 2', it will be able to detect a possible attack. The issue here is that NIDS cannot be sure what will happen on the end host. If the NIDS reports alarm, it may be false positive. If the NIDS doesn't report alarm, it may be false negative.

The evasion issue theoretically is impossible to be solved if the traffic analysis is solely passive. For example, the NIDS cannot know exactly which TCP segment will be accepted by the end host for two overlapped TCP segments because of network congestion, network topology (TTL), the resource availability at the end host, different TCP implementation (some prefer old segments while others prefer new segments for overlapped segments), etc. These are just some factors that cannot be known by the passive NIDS.

2) *Normalization to solve evasion problem at layer 3 and layer 4:* The solution [34], [35] to this issue is to put an active forwarding node in the path of traffic to a site. The forwarding node (normalizer) will normalize the

packet by removing potential ambiguities. For instance, the attackers may use small TTL to make the NIDS to get a packet without being seen by the host. The normalizer will increase the TTL to make sure that the packet will not be dropped because of TTL. In the following, we will summarize the main tricks that attackers can use and what normalization that one can do.

In the IP layer, the major tricks are the following fields in the IP header: Don't Fragment (DF) flag, IP fragmentation, TTL, IP header checksum, IP options.

- **Don't Fragment (DF) flag.** When the DF flag is set and a router needs to fragment the packet, the router will drop the packet. So the attacker can send benign packets with DF flag set. The attacker also sends attack packets without DF. So the end host will only receive attack packets. But the NIDS may only see the benign packets and overlook the attack packets. This can be achieved by using similar techniques as figure 9. To normalize the packets, one way is to clear the DF flag. But this will break the Path MTU Discovery. So if the operator knows the MTU is the same or larger than the incoming interface for all internal networks, the DF flag can be left alone. Another case is that if DF is set and either More Fragments (MF) or Fragment Offset is not zero, the packet should be dropped because this is a contradictory combination.
- **TTL.** This is similar as DF flag. A simple way is to increase the TTL to a bigger value. The first problem of this approach is that it causes packet to loop forever. Second, traceroute will not work. Third, multicast application which use expanding ring search may see all the internal hosts is adjacent to the normalizer. Another solution to this problem is to drop the packets if its TTL is not big enough. This assumes normal traffic will use large enough TTL. This solution only solves the first problem of the scheme of increasing TTL because traceroute and multicast expanding ring search indeed use small TTL.
- **IP header checksum.** In authors' opinion, this attack will become more and more practical because more and more routers just update the checksum without verifying it for forwarding performance purpose. Routers simply assume the end host will drop it if it is wrong. The simple way to normalize it is to verify its correctness. If it is wrong, just drop it. The issue here is that checksum computation is expensive. Can we just skip this check? No, we cannot totally skip this check. But we may defer the check until we find attack in the packet. If there is no attack in the packet, whether the checksum is correct or not doesn't matter. If there is attack in the packet, we can check the checksum to estimate if the end host will accept it or not. So we can decrease false positive. The saving should be big because most traffic shouldn't have attack signatures. IP header checksum is a little different from TTL and DF because the packet will be dropped if its checksum is wrong while TTL and DF depend on the network topology. This deferred checksum computation scheme requires that we can detect intrusion at the normalizer. This requirement seems not a problem because the NIDS also needs do all the work as what the normalizer does. So why don't just use the NIDS as both a normalizer and NIDS? In fact, the trend to integrate normalizer, firewall and NIDS to one device [20], [21], [10] already emerged. In our following discussion, the normalizer may also be a NIDS.
- **IP fragmentation.** The solution is to reassemble the fragments. If the packet can be reassembled completely, it will be fragmented again to be forwarded if the outgoing interface MTU is smaller than the complete packet. If the packet cannot be reassembled, all fragments will be dropped. In a word, the normalizer never forwards incoming fragments directly. We know reassembly is a big overhead. Can we skip this step? The answer is no. First, without reassembling the fragments, we don't know if the fragments can be reassembled at the end host. This is the same ambiguity problem as TTL, DF. Second, we cannot use similar strategy as deferred checksum check. If we cannot find attack signatures in fragments, we must reassemble them to see if the whole packet has attack or not. If the fragment has attack signatures, we also need to reassemble to eliminate ambiguity to decrease false positive. The net result is that we still need to reassemble all fragments. Third, because reassembly anyway is necessary to get the application layer traffic for NIDS, normalizer can just use the result of NIDS. Fourth, the total length of the reassembled IP packet must not be longer than 64K. Finally, we must also be careful to protect the device itself from Denial of Service (DoS) attack because we keep state for fragments. A good news is that we can easily detect which packet is fragment. When our resource is nearly used up, we can just drop the new incoming fragments. The effect is that only the guys who use fragments are penalized. Most traffic will not use fragments and will continue to work well.
- **IP options.** The solution is to remove them. Most traffic will not use IP options, the danger of removing the options is small.

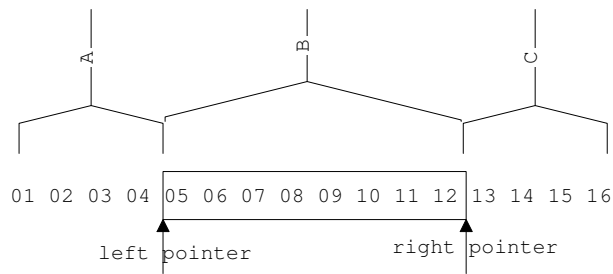


Fig. 10. Example to show how the normalizer can normalize TCP traffic. The basic idea is to trace the TCP sliding window from the receiver by tracing its Ack number and Window size. Then we can tell the Sequence number of the sender is valid or not by checking if it is region A, B or C.

In the UDP layer, the fields are UDP length and checksum.

- **UDP length.** The UDP length is actually redundant. Its check can also easily be done by comparing it with the difference between IP packet length and IP header length.
- **UDP checksum.** This check can also be deferred as IP header checksum. The check will decrease false positive.

In the TCP layer, the fields are Flag fields, sliding windows related fields, TCP checksum, and TCP options.

- **SYN flag.** a) The SYN packet can also contain data. The normalizer needs to remove the data. The data will be sent if the connection can be created successfully or will be dropped if the connection cannot be created. b) SYN can only be combined with ACK flag. Any other combinations will be dropped.
- **ACK flag.** ACK flag must be set unless SYN is set. Otherwise, the packet will be dropped.
- **FIN flag, PSH flag.** These two flags only require ACK flag is set.
- **URG flag and urgent pointer.** Other than requiring ACK flag set, the urgent pointer only valid when URG flag is set. Also the urgent pointer must not larger than the TCP payload length. Notes that this normalization cannot totally eliminate ambiguity because the interpretation of urgent pointer depends on the receiver's socket setting.
- **RST flag.** a) If the packet with RST flag set also contains data. The data needs to be removed. b) To prevent attacker cheating NIDS to clear state, paper [34] proposed a very smart scheme to make sure that end host also indeed closes the connection. The idea is to use TCP keep alive probing packet to check the host's state. If the connection is not closed, the host will respond to the probe of NIDS. So the NIDS can know this connection should be kept even if it received RST packet.
- **Sliding window related fields: Sequence number, Ack number, Window size.** As in figure 10, for each direction, the normalizer needs to keep two variables: the left pointer and right pointer. The normalizer also needs to have a buffer for received data whose sequence number is between the left pointer and the right pointer. The state is controlled by the Sequence number from the sender which controls where to put data in the buffer and Ack number and Window size from the receiver which control the left pointer and right pointer.

For **Sequence number**, it can be in the region A, B, or C regarding to the sliding window. If the Sequence number is in A, the first thought is to drop it. But actually it should be forwarded. The issue is discovered in paper [38]. The rationale is that the normalizer may get the ACK packet from receiver and advance to window to the right side. But the ACK packet is not received by the sender. So sender will retransmit the data. If the normalizer stops the retransmission, the receiver will not retransmit ACK because the retransmitted data packet is dropped by the normalizer. The sender retransmits several times and finally will brake the connections. On the other hand, if we allow the packet pass, the receiver will simply drop it. So it cannot harm the receiver.

If Sequence number is in B, the packet needs to be checked to see if there is any overlapped part with existing data in the window? If yes, the normalizer needs send the old data for overlapped part of the packet (except for checksum verifying, see explanation in "TCP checksum"). This will eliminate the problem shown in figure 9. The overlapped part cannot be just trimmed because the receiver may not receive the data even if the normalizer received it.

If Sequence number is in C, the packet can be safely dropped. The normalizer doesn't need keep state for it. One exception is that sender's persistent timer will send probing packet for window opening if the Window size is zero. As a result, the normalizer needs to allow window opening probing packet. But window opening probing packet contain one byte of data. The normalizer needs to change this packet to be keep-alive probing packet by removing the one-byte data.

For **Ack number**, if it is at A, it can be just ignored. This may be a delayed ACK packet. If it is at B, there are two cases. In the first case, if the data up to Ack number is received, the left pointer of the window can be updated with the Ack number. In the second case, if the data up to Ack number is not all received, the Ack can be ignored or the connection can be reset. When Ack number is at C, it is the same case as the second case of B.

For **Window size**, there are two issues. One is window recision. To eliminate ambiguity, the normalizer should still keep old data if the shrunk window makes them out of the window. The second issue is DoS to the data in the window. The attacker (the receiver) can advertise large window without advancing the Ack number. The sender will be seduced to send large amount of data. The normalizer must buffer all the data. This is similar as IP reassembly. So when the normalizer has stringent CPU resource or memory resource, it can set a smaller value to the Window size. This only decreases the performance. But the whole system will still work.

- **TCP checksum.** Verifying TCP checksum can be done by using similar strategy as IP header checksum. The checksum is only verified when attacks are found or overlapped packets (Duplicate is the special case of overlap: total overlap) in the sliding window are detected. When attacks are found, verifying checksum can decrease false positive. When packets overlap, verifying TCP checksum can help the normalizer pass the correct data. One example is as follows. TCP segment x is corrupted without being verified by the normalizer. Because the receiver will always verify the checksum, TCP segment x will be dropped by the receiver. The sender will retransmit the segment as x'. Now segment x' is overlapped with segment x. The normalizer will now check if x has the correct checksum. If the checksum is correct, the overlapped portion of segment x' will use old data in segment x. Otherwise, it will use new data in segment x'. Because the normalizer can be sure that data in segment x will be dropped if its checksum is wrong, there is no ambiguity even if normalizer allow different version of data be sent. This scheme can handle occasional errors without many performance penalties.
- **TCP options.** The general rule is that if the NIDS cannot handle it, the normalizer can just remove it.
- **Flow state management.** We can see the flow state maintained in the normalizer is the basis of most normalization process. So flow state itself is very important for the correct functionality of the normalizer. In paper [34], a smart scheme is proposed to create state for flows without SYN flag. This can happen for existing flows before the normalizer start or somehow the normalizer lost its state. This scheme assumes the normalizer is between a trusted side and hostile side. For the packets from the trusted side, the normalizer will create state for it if it doesn't do it yet. If the packet is from the hostile side, the normalizer will hold it and send a TCP keep-alive probe packet to the internal host. If the packet is part of a valid connection, the internal host will respond with ACK packet. The ACK packet will trigger the normalizer to set up state. Otherwise, either RST will be generated from the host or nothing happens. In case RST reaches the normalizer, the normalizer will just drop it. This prevents the attacker from getting information about internal hosts. In case that nothing happens, because the normalizer didn't create any state for the request, so there is nothing that needs to be done at the normalizer. The good point of this scheme is that it can create state for requests from hostile side in a stateless fashion. It relies on the state of the host in the trusted side to verify the validity of the requests. But this assumption is not always true. As in [11], people in the trusted side can also do port scanning. So we still need use different timeout value for these special states. In case of flow state Denial of Service such as SYN flooding, it seems that the only solution is to use smaller timeout value for these half-open states.

Another solution to TCP traffic normalization is to use transport layer proxy. A transport layer proxy such as TCP proxy will create two TCP sockets for each TCP connection between a client and a server. For the client, the proxy works like a server. It will accept the connection request from client. When the client requests to create a connection with the server, it actually connects with the proxy. The socket on proxy for the client is called client socket. For the server, it will work as a client. When the proxy accepts a connection request from the client, it will

create another connection to the server. The proxy socket to the server is called server socket. Then the proxy will forward data between the client socket and server socket. Because the sockets of the proxy always work as normal sockets, the server and NIDS will always see unambiguous traffic.

Paper [35] claims that the proxy approach is less scalable than the pure normalization approach discussed above. The proxy approach needs to maintain two sockets for each connection. Each socket needs to schedule several timers, estimate round-trip time, calculate window size and verify checksum of every packet. On the other hand, the pure normalization approach maintains less state for each connection. If the attacks only come from outside, only half of the connection needs to be normalized. There is no retransmission in this approach. Every event is driven by end hosts. So paper [35] claims that this approach can scale to tens of thousands of concurrent connections with throughput performance that is comparable to stateful firewalls.

3) *Evasion at application layer:* Even in the application layer, tricks to evade the check of traffic analysis also exist. Rain Forest Puppy [39] lists some tactics on how to evade the HTTP traffic analysis. For example, the following is an attack request that the NIDS want to find.

```
GET /cgi-bin/some.cgi
```

Some tactics can be used to change the request. Here are some major tricks.

- **URL encoding.** The “cgi-bin” can be encoded as “%63%67%69%2d%62%69%6e”. So the request becomes as “GET /%63%67%69%2d%62%69%6e/some.cgi”. If the NIDS just searches for the string “cgi-bin”, it will not be able to find one in this case.
- **Directory tricks.** The directory “/cgi-bin/” can be played with a lot of ways. For instance, it can be:
  - “/cgi-bin/blahblah/..”
  - “/////cgi-bin////////”
  - “/cgi-bin/./././”
  - “/cgi-bin/.\.\.”
- **Case sensitivity.** The request can be “GET /cGi-Bin/sOmE.CGi”.

4) *Protocol awareness to solve evasion problem at application layer:* Given all these varietyies, directly searching signatures in the raw application layer data stream is impossible. The solution is to convert the data to one common format based on the protocol specification. The signature will be designed against only the common format. In the snort implementation, this is done via the preprocessor HTTP decoder.

From this example, we can see application layer protocol aware traffic analysis is important for not only performance but also correctness. Application layer protocols often allow different representations for the same purpose. This introduces a lot of varieties. Only by understanding the protocol can the NIDS normalize the data stream to a common representation.

Another advantage of protocol awareness is that people can do protocol conformity check. In fact, this traffic analysis technique is totally different from the string matching technique. For protocol conformity analysis, there is no signature at all. The check is based on the protocol specification. The FTP bounce attack in [21] is a good example to show how this technique works.

The FTP protocol has a design flaw which allows an attacker to command a FTP server to create a TCP connection with any host (the victim) on the Internet and send or receive data from the victim via the created TCP connection. In this case, the victim is connected with the FTP server. It knows nothing about the attacker. This makes it very hard to track the attacker from the victim because the attacker is hidden by the FTP server. The FTP server in fact is only a scapegoat. That is why it is called “bounce attack”.

To launch this attack, the attacker sends the FTP server a “PORT” command. We know there are two parameters in the PORT command. One is the IP address of the host that the server will connect to. The other is the port number of the host. Normally, the IP address is the FTP client itself. So the client can use this connection to upload or download data. For FTP bounce attack, the attacker (the FTP client) sends PORT command with the IP address set to be the victim instead of itself. So the FTP server will connect to the victim instead of the attacker. The FTP server doesn’t know it connects to a victim and will happily download or upload data to the victim.

To detect this attack, the NIDS needs to check if the IP address in the PORT command is the same as the source IP address of the TCP packet. If they are different, it can be a possible bounce attack. In this example, the IP address is not a fixed signature but a relationship. So signature matching based analysis is useless for this kind of anomaly.

The evasion can also happen in application recognition. In this paper we mentioned that people may want to limit the bandwidth of P2P applications. The P2P traffic now can be recognized from its signature strings in the application layer. But this check should be able to be evaded. There is no reason that P2P applications cannot simulate the normal HTTP traffic. To find files, the P2P applications can just submit their request via standard HTTP to their servers like people search keywords on google via HTTP. The search results are returned via normal HTTP response and the location of the files is represented as hyper links in HTML files. To download a file, the request and response can also be the same as normal HTTP traffic. To update P2P node's shared files list, the node can use HTTP PUT or POST method to upload the list to their servers. For a P2P node to get into the P2P network, it must keep a dynamic list about the current active servers. This server list can also be exchanged via HTTP. Theoretically, there is no problem that prevents P2P applications from working the same way as normal HTTP traffic. So they can just hide as HTTP traffic and evade the check of application recognition. Moreover, it doesn't seem difficult for people to upgrade the network since people just need to download a new version of their P2P software. Although most protocols are standardized and the change is not easy, P2P applications on the other hand have both incentive and capability to evade the check.

#### IV. SOME NOTES ABOUT TRAFFIC ANALYSIS

**The source address of traffic.** In traffic analysis, we cannot take for granted that the source IP address of the packet is the IP address of the host that sends the packet. We must always be aware that the source IP address of a packet may not be the attacker. For connectionless protocols such as UDP and ICMP, it is very easy for the sender to send packets with spoofed IP address. The attacks can be ICMP PING flooding, PING-of-death, UDP local network broadcast storm, etc. Even for TCP, the sender can use techniques such as SYN flooding to attack the server. Another example is FTP bounce attack. The FTP server is only a scapegoat. If the attacker can monitor the return traffic of the victim, it can even connect with the server directly with spoofed source IP address. So if the software wants to block the attack traffic by blocking its source IP address, it must be careful. One danger mentioned in [36] is that the software may block the IP address of its own servers such as DNS server if the attacker uses the site's DNS servers' IP address as the source IP address of the attack packets.

**Why not analyze traffic on the hosts?** From our investigation, we conclude that it is really not trivial to analyze traffic on the network. The logical question is why we don't analyze the traffic on each host. On the host, the traffic analysis can get unambiguous information on the data. Also it has the advantage to get more information about how the host processes the data. Furthermore, the resource requirement such as CPU, memory is low because it only needs handle its own traffic. The problems with host-based traffic analysis are as follows. a) The host's environment can be heterogeneous. The host based traffic analysis must be implemented for all kinds of common used operating systems. b) The management of hosts often is decentralized. This makes it hard to update the traffic analysis software on the hosts. For instance, host based intrusion detection system (HIDS) will be harder to manage than NIDS. c) More fundamentally, host based traffic analysis doesn't have good view of the global traffic nor control on global traffic. This makes it impossible for some functionalities such port scanning, traffic classification for QoS management, etc. On the other hand, network based traffic analysis can use one box to monitor and/or control the whole network. It is cheap to deploy and easy to manage.

**Encrypted traffic.** Now more and more complex traffic analysis techniques need to understand the application layer traffic. Also more and more content need to be securely transmitted on the network. But encryption will prevent traffic analysis from understanding the content of application layer traffic. So it will be difficult for current network based traffic analysis to do its job. One solution is to use proxy which is similar as techniques used in the man-in-the-middle attack. But practically it will be hard to deploy. The good news is that most protocol and most traffic are not encrypted. Although it is not clear how long this status can last.

**Evading the traffic analysis.** In discussing application layer traffic analysis, we presented the evasion tactics and their solutions. But application layer traffic analysis may not solve all the evasion problems. For instance, it seems impossible to recognize P2P traffic if it masquerades it as HTTP traffic.

#### **Issues that make traffic analysis difficult.**

- Dynamic port. FTP data connection is one example. To get to know the protocol of packets, we must trace the FTP control connection to know about the dynamic port of the FTP data connection. Otherwise, we cannot know what protocol the FTP data connection is. RTSP and H.323 are another two examples of this kind of

protocols. Paper [19] explained their experiences on building a tool just to trace the dynamic port of RTSP and H.323.

- Non-self-explained connections. For protocols such as HTTP, even if people use non-standard port such as 1000, traffic analysis can still recognize the protocol of the packets. The reason is that HTTP connections are self-explained. Each connection also contains the protocol control information such as request or response. On the other hand, some connections only transfer raw data. FTP data connection is one example. So other than trace the dynamic port of the connection, there is no other way to identify the protocol of the FTP data connection.
- Time. The slow port scanning is one of the examples to show how time make traffic analysis difficult. To detect port scanning, most traffic analysis techniques keep records on how many incoming connection requests come from Internet. For a given period of time, if the requests number exceeds a limit, the requests will be treated as port scanning. The traffic analysis computer cannot record the requests of all Internet hosts. So it only records the Internet hosts whose request is much faster than normal hosts. The slow port scanning can just slow down its scanning speed to evade the detection. Time plays an important role here. If the event interval is long enough, traffic analysis will not be able to track it. Paper [41] provide a solution to detect this slow scanning by detecting the anomaly of high frequency of closed port or hosts.
- Different representations. HTTP request representation is an example of this issue. To convey the same information, there may be several different ways to express it. Different representations require traffic analysis to understand all the possible ways. In the case of HTTP request, normalization is a feasible way. Unfortunately we cannot normalize everything. Human language is another example. For Spam email filtering software and web content filtering software, the ability to understand different ways of expression is critical.
- ASCII style protocol. For IP or TCP protocol, it is easy to analyze them because their information fields are fixed. The data in each field is in binary format. For protocols such as HTTP, things get more difficult because their information fields will not always show in the same position of the packets. We need to parse the string to get the information out.

## V. CONCLUSION

This paper investigates the stateful firewall design and the traffic analysis part of NIDS. In the stateful firewall design, we discussed the following design issues.

- Why do we trace connection state?
- How to handle IP fragments?
- Why do we treat TCP differently from UDP?
- How to manage TCP state?
- And why is it not necessary to track the sequence number?

In the application layer traffic analysis, we mainly focus on the traffic analysis part of NIDS. We investigate the following questions.

- What are the major techniques in analyzing application layer traffic?
- What are the major string matching algorithms and their performance?
- How to prevent evasion tactics from evading application layer traffic analysis?

Our main contributions in this paper are as follows.

- For TCP flow state management, when we create state for internally sent TCP packet without SYN flag, we propose to mark the flow in a special state with a shorter time out value. This way we will be able to prevent flow state from exhaustion caused by internal hosts port scanning as found in [11].
- We investigated the major string matching algorithm and analyzed their techniques and performance. In our opinion, the AC-BM algorithm should be a very good algorithm. The  $E^2xB$  algorithm can also be combined with AC-BM algorithm. But care must be taken for signature DoS.
- To normalize TCP, UDP and IP packets, we suggest to combine NIDS with normalization together on one device. This way we can defer the IP, UDP, TCP checksum computation until attacks are detected. For common case, we will in fact skip checksum verification for most packets.
- In TCP normalization, we suggest the following scheme. a) We need to send packet whose Sequence number falls in region A in figure 10. b) We need to send the whole packet if its Sequence number falls in region

B and overlaps with existing packets. Our scheme addresses one important problem missed in [34]: when normalizer receives packet, it doesn't mean the receiver will receive it too.

- In TCP normalization, we suggest to check the Window size to prevent the normalizer from DoS attack.

We suggest the following topics for future research based on our investigation.

- A detailed design paper on IP, UDP, and TCP normalization with application layer traffic extraction. This work can be used in NIDS, application (protocol) recognition, etc. And it will be a foundation for application layer traffic analysis.
- A paper that compares the major string matching algorithms with different traffic traces and different signature sets. The comparison should be the running time of the string matching algorithms. Then we can have a good idea on which algorithm is better.

## REFERENCES

- [1] K. G. Anagnostakis, et al. Efficient packet monitoring for network management. In Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS), April 2002.
- [2] A. Feldmann, "BLT: Bi-layer tracing of HTTP and TCP/IP," WWW9 / Computer Networks, vol. 33, no. 1-6, pp. 321–335, 2000.
- [3] K. McCloghrie et al., "Management Information Base for Network Management of TCP/IP-based internets: MIB-II," RFC 1213
- [4] S. Waldbusser, "Remote Network Monitoring Management Information Base," RFC 1757
- [5] S. Waldbusser, "Remote Network Monitoring Management Information Base Version 2 using SMIv2," RFC 2021
- [6] Cisco "NetFlow Switching Overview," [http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fswtch\\_c/swprt2/xcfnfov.pdf](http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fswtch_c/swprt2/xcfnfov.pdf)
- [7] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in Proceedings of the 2001 ACM SIGCOMM Internet Measurement Workshop, pp. 75–80, (San Francisco, CA), Nov. 2001.
- [8] nmap, [http://www.insecure.org/nmap/nmap\\_documentation.html](http://www.insecure.org/nmap/nmap_documentation.html)
- [9] Cisco IOS Security Configuration Guide, Release 12.3. [http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123cgcr/sec\\_vcg.pdf](http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123cgcr/sec_vcg.pdf)
- [10] Configuring Cisco IOS Firewall Intrusion Detection System, [http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fsecur\\_c/ftafwl/scfids.pdf](http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fsecur_c/ftafwl/scfids.pdf)
- [11] Lance Spitzner, Understanding the FW-1 State Table. <http://www.spitzner.net/fwtable.html>
- [12] G. Apostolopoulos et al. Design, Implementation and Performance of a Content-Based Switch, in Proceedings of IEEE INFOCOM, March 2000
- [13] Brian Laing, How to guide: implementing a network based intrusion detection system. <http://www.snort.org/docs/iss-placement.pdf>
- [14] R. Fielding, et al. "Hypertext Transfer Protocol – HTTP/1.1," RFC2616
- [15] Packeteer, Best Practices - Peer-to-Peer (P2P) Applications, <http://www.webtorials.com/main/resource/papers/packeteer/paper10.htm>
- [16] Packeteer, <http://www.packeteer.com/>
- [17] Rether Network Inc., <http://www.rether.com/>
- [18] Justin Levandoski, Ethan Sommer, et al. "Application Layer Packet Classifier for Linux" <http://l7-filter.sourceforge.net/>
- [19] J. van der Merwe, et al. Mmdump - A Tool for Monitoring Internet Multimedia Traffic. ACM Computer Communication Review, 30(4), October 2000
- [20] NetScreen, NetScreen's Deep Inspection Firewall, <http://www.webtorials.com/main/resource/papers/netscreen/paper14.htm>
- [21] NetScreen, Intrusion Detection and Prevention <http://www.netscreen.com>
- [22] M. Roesch, Snort: Lightweight intrusion detection for networks, In Proceedings of the 1999 USENIX LISA Systems Administration Conference, November 1999.
- [23] Neil Desai, Increasing Performance in High Speed NIDS, [http://www.snort.org/docs/Increasing\\_Performance\\_in\\_High\\_Speed\\_NIDS.pdf](http://www.snort.org/docs/Increasing_Performance_in_High_Speed_NIDS.pdf)
- [24] Lambert Schaelicke, et al, "Characterizing the Performance of Network Intrusion Detection Sensors," Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003),
- [25] S. Antonatos, et al, Performance Analysis of Content Matching Intrusion Detection Systems, Proceedings of the International Symposium on Applications and the Internet (SAINT2004), January 2004
- [26] Aho, A. V., and M. J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (June 1975), pp. 333-340.
- [27] Boyer R. S., and J. S. Moore, A fast string searching algorithm, Communications of the ACM 20 (October 1977), pp. 762-772.
- [28] Wu S., and U. Manber, A Fast Algorithm for Multi-Pattern Searching, Technical Report TR-94-17, Department of Computer Science, University of Arizona,
- [29] Sun Kim and Yanggon Kim, "A Fast Multiple String Pattern Matching Algorithm," Proc. of 17th AoM/IAoM Conference on Computer Science, August 1999
- [30] 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.
- [31] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster pattern matching for intrusion detection, or exceeding the speed of snort. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II), June 2002.
- [32] E. P. Markatos, et al., Anagnostakis. ExB: Exclusion-based signature matching for intrusion detection. In Proceedings of CCN'02, November 2002.
- [33] K. G. Anagnostakis, et al., E2xB: A domain-specific string matching algorithm for intrusion detection. In Proceedings of the 18th IFIP International Information Security Conference (SEC2003), May 2003.

- [34] M. Handley, et al, Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics, Proc. USENIX Security Symposium 2001.
- [35] Malan, G., et al, Transport and Application Protocol Scrubbing, INFOCOM 2000.
- [36] Thomas H. Ptacek and Timothy N. Newsham, "Insertion, Evasion, And Denial Of Service: Eluding Network Intrusion Detection," Technical Report, Secure Networks, Inc., January 1998. <http://www.snort.org/docs/idspaper/>
- [37] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," In 7th Annual USENIX Security Symposium, January 1998.
- [38] Guido van Rooij, Real Stateful TCP Packet Filtering in IP Filter [http://www.iae.nl/users/guido/papers/tcp\\_filtering.ps.gz](http://www.iae.nl/users/guido/papers/tcp_filtering.ps.gz)
- [39] Rain Forest Puppy, A look at whisker's anti-IDS tactics, <http://www.wiretrip.net/rfp/txt/whiskerids.html>
- [40] Cisco, Network-Based Application Recognition, <http://www.cisco.com/univercd/cc/td/doc/product/software/ios121/121newft/121limit/121e/121e2/nbar2e.pdf>
- [41] S. Staniford, et al. Practical automated detection of stealthy portscans. Journal of Computer Security, 2002.