PARA-SNORT : A MULTI-THREAD SNORT ON MULTI-CORE IA PLATFORM

Xinming Chen[1,2], Yiyao Wu[1,2], Lianghong Xu[1,2], Yibo Xue[2,3] and Jun Li[2,3]
[1]Dept. Automation, Tsinghua University, Beijing, China
[2]Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
[3]Tsinghua National Lab for Information Science and Technology, Beijing, China
chen-xm09@mails.tsinghua.edu.cn, {wuyiyao, xulianghong05}@gmail.com, {yiboxue, junl}@tsinghua.edu.cn

ABSTRACT
As security threats and network bandwidth increase in a very fast pace, there is a growing interest in designing high-performance network intrusion detection system (NIDS). This paper presents a parallelization strategy for the popular open-source Snort to build a high performance NIDS on multi-core IA platform. A modular design of parallel NIDS based on Snort is proposed in this paper. Named Para-Snort, it enables flexible and easy module design.

This paper also analyzed the performance impact of load balancing and multi-pattern matching. Modified-JSQ and AC-WM algorithms are implemented in order to resolve the bottlenecks and improve the performance of the system.

Experimental results show that Para-Snort achieves significant speedup of 4 to 6 times for various traces with a 7-thread parallelizing test setup.

KEY WORDS
Network Security, Snort, Multi-core

1 Introduction

Network intrusion detection system (NIDS) analyzes network traffic and reports malicious activities that intend to compromise computer and network security. Nowadays the species and quantities of attacks on Internet have increased exponentially, and the ruleset of NIDS is becoming larger and larger [1], which makes the computational burden of NIDS continuously increasing. At the same time, proliferation of the Internet coverage and applications fuels rapid bandwidth increase that demands high-performance NIDS an urgent need.

Although some NIDS products based on ASICs or FPGA can achieve quite high processing speed, their flexibility can hardly meet the evolving structure complexity and the increasing ruleset size of NIDS. Besides, they are usually quite expensive for the edge network. IA (Intel Architecture, often generically called x86, or x64 for IA-64) platform is more compatible to enterprise application, has the advantage of low price and high flexibility compare to ASIC or FPGA solutions. On the other hand, performance improvement on single processor platform has been significant, but insufficient to fulfill the performance requirements of a high speed edge NIDS. This paper studies the strategy of parallel processing implementation for a high-performance NIDS on IA platform.

Multi-core IA platforms is leading the trends of higher processor computation power, but to take advantage of the multi-core platform, software must have a parallel structure to fully utilize the processing capability of the multi-core architecture. This multi-core parallelism has been rarely leveraged in existing NIDS, mainly because of the complexity of NIDS. Some functionality in NIDS, such as multi-pattern matching and preprocessing, can cost majority of overall processing time. Parallel implementation of NIDS may also introduce new bottlenecks, such as the load balancing and replicated memory usage. Modular design is also required for better scalability, which presents another major challenge to a parallel processing NIDS.

In this paper, a new parallel structure of NIDS is proposed. Compared to the previous work on parallelization, this structure has a clean-cut modular design, along with optimized core algorithms. Section 2 describes the application background of this work, and summarizes related work. In Section 3, the modular design of the proposed parallel structure is described in details. Section 4 describes optimization methods to resolve the performance bottlenecks, where Modified-JSQ algorithm is used to increase the performance of the load balancer, and AC-WM algorithm is used to break the bottleneck of multi-pattern matching. A prototype is developed for performance evaluation; experimental results are discussed in Section 5. As a summary, in Section 6, we state our conclusion.

2 Background and Related Work

2.1 Overview of NIDS

Unlike firewalls, which inspect packet headers according to user policy to make network access decisions, a NIDS looks into both header and payload of packets to identify intrusion, and this full packet content inspection requires much more computational ability. NIDS can be implemented based on ASIC, FPGA, network processors, or IA platform, which is discussed in this paper. There are a lot of popular NIDSes on IA platform, such as Snort [1] and Bro [2]. They all have similar structures. Here we take

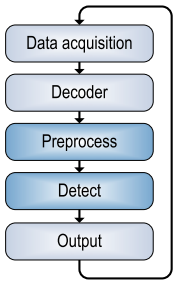

Figure 1. Snort 2.x processing loop [4]

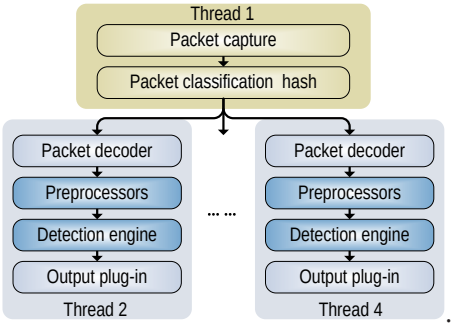

Figure 2. Structure of Supra-linear Packet Processing [6]

Snort as an example to explain the common structure of NIDSes, and how we design a parallel NIDS based on this single-thread structure.

Snort is so far the most popular open source NIDS on IA platform, currently maintained by Sourcefire Inc. The latest stable release is Snort 2.8.4. Figure 1 depicts the processing flow used by Snort 2.x. Firstly, Snort performs data acquisition through libpcap [3] or other libraries, and then it decodes the packets to fill the data structure used by Snort. After that, Snort uses preprocessors to modify or analyze packets before detection, and the detection engine is employed to do pattern matching for packet payload with compiled ruleset. If attacks are found during the procedure, Snort sends alert to the administrator.

The Snort 2.x uses single-thread processing. There is only one CPU core used during the process on multi-core processors. It is unable to utilize the full computer ability of multi-core platforms.

To build a parallel structure for a system, we must first find the bottlenecks in its serial implementation. Derek L. Schuff from Purdue University has studied the execution time of each components of Snort. He proved that detection component occupies an average of 51.90 % of execution time, and preprocessing occupies an average of 18.02 % [5]. Therefore, this work focuses on the parallelization of the preprocessor and detection engine.

It is worth mentioning that Sourcefire Inc. released a new platform named SnortSP in 2008, the current version is 3.0.0 beta 3, and no stable release version now. SnortSP's approach is similar to ours and its code base meets our demands well, so we leveraged on the code of SnortSP during the development of our prototype system.

2.2 Previous Work on NIDS Parallelization

2.2.1 Supra-linear Packet Processing

Supra-linear Packet Processing [6] is an achievement made by Intel Corporation in 2006. It is based on Snort 2.x. Figure 2 shows its structure.

The data acquisition component of Supra-linear is separated and other components are duplicated. A Packet classification hash module is added to dispatch the packets into processing threads. Data acquisition and dispatch component is in one thread, and each processing component is a separated thread. Each processing thread executes the same code to go through a flow from decode to output, and they don't communicate with each other.

Supra-linear is a simple implementation. But the processing threads have a problem of replicated memory usage. Some data which is read-only during the processing should be shared, such as compiled ruleset. When compiling the whole ruleset with AC algorithm, the size of compiled ruleset in one thread is about 1.5 GB. On most of IA platforms, it is difficult to allocate so much memory for several threads.

2.2.2 MultiSnort

MultiSnort [5] is a multi-thread Snort presented by Derek L. Schuff, Purdue University. It is based on Snort 2.6. Compared with Supra-linear Packet Processing, MultiSnort only executes multiple instants of original Snort in parallel, and it proposed a strategy of memory sharing. Figure 3 shows its structure.

MultiSnort has a load balancer that uses distributed task queues to dispatch traffics. After the load balancer, each processing thread does the job such as decode, preprocess, and detect. Alert is then generated by a unified output module.

MultiSnort is modified from original Snort 2.6. Each component of Snort 2.x, such as data acquisition component or decoder, is tightly coupled with others. It is not easy to insert new data sources or processing modules. As an experimental system, MultiSnort reads packets from a large in-memory buffer instead of network interfaces, so it cannot be deployed in real network environment yet.

3 Para-Snort: A Modular Design of Parallel NIDS

An ideal parallel NIDS needs the following features. First of all, a clean-cut modular structure is needed for better extension ability. It should be easy to insert new modules, and

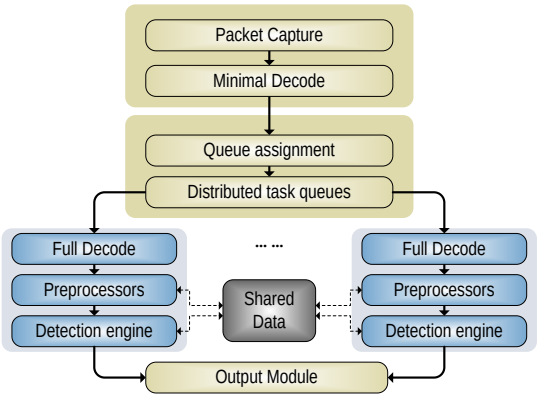

Figure 3. Structure of MultiSnort [5]

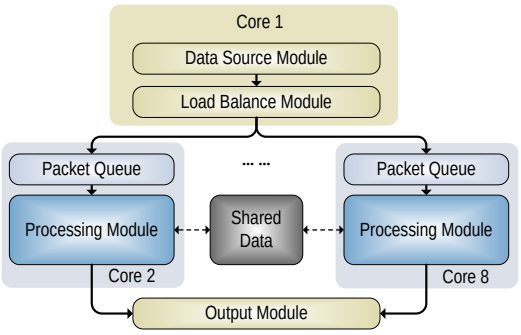

Figure 4. The structure of Para-Snort

even allow replacing some modules with hardware acceleration modules. The bottlenecks, such as the pattern matching and the load balancing, should be cleared, too. At last, an ideal parallel NIDS should be easy to be deployed and tested.

Based on the above demands, we designed a parallel structure for high-performance NIDS. As shown in Figure 4, the new structure named Para-Snort contains a data source module, one or more load balance module, multiple processing modules, and an output module.

3.1 Data Source Module

As illustrated in Figure 4, data acquisition and decoder are merged into a data source module. This module ensures the processing modules to receive packet data in a unified data structure even if they are from different interfaces (Ethernet, 802.11x, files, and so on), and thus it is not coupled with the other modules. Several methods can be used to build a special data source module, such as NFQueue [7] that implements inline mode, or even a hardware capture.

3.2 Load Balance Module

After the data source module gets packets, the load balance module dispatches them to processing modules. The

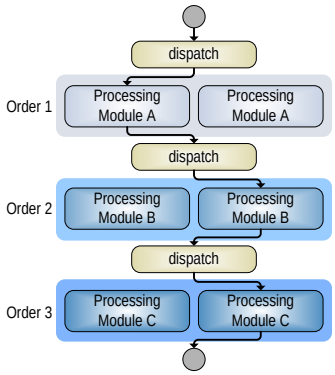

Figure 5. Multi-staged load balancer

load balance module has a multi-staged feature that can dispatch packets more than one time. So it is possible for the packets to be processed several times by different types of processing modules. For example, some applications requires its traffic to pass through both intrusion detection and anti-virus processing modules. These two kinds of processing modules can be arranged in two stages. Figure 5 represents a situation of multi-staged processing modules arrangement. Processing modules with the same functionality are grouped in the same satge for parallel processing. Every time before packets enter a stage of processing modules, a load balance module dispatches the packets again to make sure every processing module at the same stage gets balanced load. This kind of load balanceing can meet the dynamic security filtering and inspection demand flexibly. The numbers of processing modules at different stages can be different, such as a stage of 7 intrusion detection modules followed by a stage of 4 anti-virus modules. The stages of processing modules implement parallelization in pipeline.

Load balance algorithm can be optimized according to the traffic composition, which will be analyzed in section 4.1.

3.3 Processing Module

Each processing module is implemented as a single thread, while data source module and load balance module share one thread. These threads are mapped to certain CPU cores to avoid costs on thread scheduling. Processing modules get packets from the load balance module via several queues. Each processing module has a dedicated queue. Load balance module fetches packets from data source module actively. It is possible that the data source may miss some packets during the capture if it is not inline, and the packet lose rate depends on the NIDS processing speed.

Processing modules is built with the preprocessors and detection engine from Snort 2.x. It is made easy to develop processing modules with other functions. For example, Para-Snort has a processing module built with the anti-virus engine from ClamAV [8], and this scales Para-

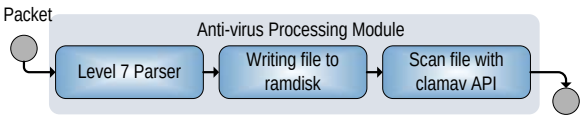

Figure 6. The design of anti-virus processing module

Snort into an integrated solution of NIDS and anti-virus. As shown in Figure 6, after getting packets from a load balance module, the anti-virus processing module uses a 7-layer parser to analyze packets with HTTP and SMTP protocol, rebuilds files on a RAMDisk, then calls ClamAV API to scan for virus, and alarms, if any, are sent to output module to report viruses found.

4 Breaking the Bottlenecks

As described in Section 1, there are performance bottlenecks about load balancing, multi-pattern matching, and replicated memory usage. Para-Snort addressed each of these issues carefully to better leverage the advantage of parallel processing power of multi-core platform without introducing new show stoppers.

4.1 Optimize Load Balancing

Generally, there are two kinds of load balancing schemes: packet-level and flow-level. However, for NIDS, due to some special requirement (for example, detecting inter-packet attack signatures), packets from the same flow should be assigned to the same processing module. As a result, Para-Snort requires flow-level load balance.

The current load balancing scheme provided by SnortSP 3.0 is an IP hash algorithm, which generates a hash key from source and destination IP addresses. Although this algorithm efficiently reduces the inter-communications among processing modules, the network load assigned to each of the modules are not very uniform and thus results poor system performnace.

5-tuple hash is a dispatching algorithm naturally derived from IP hash. Fully exploiting the 5-tuple information in packet headers (source IP, destination IP, source port, destination port, protocol), this algorithm is expected to achieve better distributing performance than IP hash in most cases. However, 5-tuple hash is unable to detect port scan attack, because it dispatches packets of different port to different processing modules. But detecting port scan can be done with front-end devices, this disadvantage is acceptable.

An inherent limitation of the two hash algorithms, however, is that they are both static distributing schemes, so that the destination module of each and every packet under processing is fixed without being adjusted according to the real-time situation. A dynamic load balancing scheme named JSQ (Join-the-Shortest-Queue) is proposed in this paper for Para-Snort, which assigns every new flow to the least busy processing module, that is, the processing module with the shortest buffer queue. JSQ is a local optimum yet not perfect method. The possibility exists that one processing module has too many active flows while others have very few, because the binding relationship between a flow and the corresponding processing module is permanent once established. Therefore, a more effective mechanism is desired to allow flow reassignment with minimum loss in security.

Modified-JSQ (M-JSQ) is proposed in this paper as an improvement of JSQ in terms of dispatching granularity, in which a flow is able to be reassigned when the number of interval packets between two adjacent packets belonging to this flow is larger than a given threshold. When the threshold is positive infinity, M-JSQ reduce to JSQ. The introduction of reassignment threshold seldom decreases detection accuracy because its value is usually much larger than the length of the flow reassembly buffer in Snort. Implemented in Para-Snort, M-JSQ is a more fine-grained load balancing scheme than JSQ, however, we should note that the former is not always better than the latter in terms of system throughput due to the overhead of counting the number of every incoming packet and reassigning flows to different analyzers.

4.2 Optimize Multi-pattern Matching

As mentioned in Section 2.1, multi-pattern matching costs majority of NIDS computational capacity. In this section, AC-WM, an algorithm combines the advantages of both classic AC and WM algorithms, is proposed to improve multi-pattern matching performance.

AC algorithm is based on deterministic finite automata (DFA) where all the key patterns are compiled into a specific DFA table. During the scanning process, DFA records the current state. As soon as an input character is received, AC finds the current state in the DFA table, and index to the next state it should jump to according to the ASCII code value of the input character. Generally speaking, the size of a DFA is in direct proportion to the number of characters of all the patterns (about 1 KB per character), indicating a huge memory cost.

WM is a shift algorithm based on suffix matching. In the pre-compiling, all the patterns are left aligned and the prefix of each pattern is obtained with a length that is the length of the shortest pattern. We take the prefixes as the useful patterns. Then, for each 2 character sub-string in the useful patterns, a shift value (number of characters to skip) is calculated according to the shortest distance between the 2 character sub-string and the end of the useful patterns among all the patterns that the 2 character sub-string appears. All the shift values are saved in a shift table. For every 2 character sub-string whose shift value is zero, the prefix of the pattern whose suffix is the 2 character sub-string will be linked to the 2 characters in another table. In the scanning process, when 2 input characters are received,

WM can usually skip several characters according to the value in the shift table. If the 2 character array doesnt appear in any of the key patterns, the shift value will be the length of the shortest key pattern. If the input string is a suffix of a useful pattern, the whole pattern linked to the suffix would be examined. From the pre-compiling process we can see that the length of the shortest pattern greatly influence the shift length and the scanning speed.

In normal network flow, few states in DFA of AC algorithm are frequently visited. So the data structure that AC generates is utilized with a very high cache locality. That is to say, at most of the time, further memory access is not necessary. However, when the rate of states that having been visited during the scan process raise to a certain level due to the change of network flow, we come to a relative low cache locality. So the scanning speed slows down rapidly. At the same time, the performance of WM algorithm is limited by the length of the shortest pattern.

Based on the above observation, AC-WM algorithm is designed to employ AC algorithm to compile and check against the short patterns, while utilize WM algorithms take care of the long ones. On one hand, we minimize the size of data structure that AC algorithm compiles by minimizing the total state numbers to increase cache hit rate providing fixed cache size; On the other hand, the WM algorithm performs gains higher average shift value by increasing the shortest pattern length. As the results of the AC-WM, firstly, because DFA of AC algorithm cost most of the memory resource, the total size would be reduced greatly by 50% – 70% when the size of DFA reduces to 20% – 40% of the original one; secondly, AC-WM keeps the system oscillating in a small bound when the network flows changes violently. In another word, we build a robust system which is insensitive to the content of network flows. It should be noted that in the worst case which is unlikely to happen, the scanning speed of the AC-WM algorithm will be a bit slower than AC algorithm.

4.3 Reduce Replicated Memory Usage

Processing modules at the same stage are usually instants of the same program, and thus they usually share the same ruleset while performing the same processing on different data, a typical SPMD (Single Program Multiple Data) case of parallel processing. The same ruleset can be shared rather than replicated, as it is read-only during the processing. A mechanism is developed in this paper and deployed in Para-Snort to share ruleset in processing modules. Now the Snort compiled intrusion fingerprint for intrusion detection and the ClamAV compiled virus signatures for antivirus are shared by their associate processing modules at each of the two stages, respectively. This reduces redundant memory storage significantly. While one-thread allocates 1.5 GB of memory, seven-thread requires only 1.7 GB of memory. If the memory is not shared, 10.5 GB of memory is needed, which IA platforms can seldom afford nowadays.

Table 1. Property of packet traces

Trace Name	LL1	LL2	CERNET	http
Source	Lincoln Lab	Lincoln Lab	NSLab RIIT	Random generated
Date	4/01/99	4/09/99	5/05/08	-
Size (MB)	519	970	700	-
Packets	2356503	2651589	889957	-
Average Size(B)	159	272	664	792
HTTP packets	51.5%	88.4%	5.8%	100%
Alerts	308	623	1505	random

In addition, the share of ruleset can lead to higher cache locality. According to our experiments, only less than 0.5% of states or about 100KB state information are frequently accessed. With the memory share for SPMD, it is more likely for them to stay in L2 cache of the CPU. On IA platform, it is much faster to access L2 cache than to access DRAM on board, so sharing memory can make the processing speed faster.

5 Performance Evaluation

5.1 Development Platform and Test Environments

The following experiments are running on a 1U server with two quad-core Xeons E5335 at 2.00GHz (8 processors in total). The system has 4 GB of DRAM and runs Linux kernel version 2.6.24 in Ubuntu 8.04 32-bit distribution. The NIDS platform and the testing machines are networked with gigabit network interface cards.

The prototype system developed in this work is based on SnortSP 3.0 beta, downloadable from www.snort.org. The configuration of Snort includes all important preprocessors such as Stream 5, HTTP Inspect. The multi-pattern matching algorithm is AC-full. All the 10000 released rules are used. We have both NIDS and anti-virus processing modules working here. One NIDS processing module is in one-to-one correspondence with one anti-virus processing module.

We have three types of testing flow here. The first type of packet traces come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab [9]. The Lincoln Lab (LL) traces are widely used for NIDS testing and are downloadable from Internet [10]. Here we use two traces from fourth and fifth week in the 1999 test. The second type of packet traces is captured from CERNET by our lab in May 2008. These packets contain full payload, so they can reflect the real condition of network flow. These

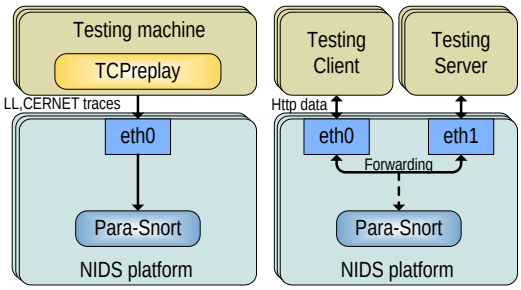

Figure 7. The testing environments

two types of traces are sent to the network card of NIDS platform using tcpreplay [11]. The third type of trace is generated by two testing machines; NIDS platform bridges them together with its two network card interfaces. The two testing machines create several sessions to translate random data with HTTP protocol at the speed of about 600Mbps. Our NIDS captures and analyzes these flows. The source and destination IP addresses of the sessions are arranged to meet the IP hash algorithm, so the flows are ensured to dispatched equally to the processing modules when using IP hash load balancer.

Every test will be taken three times for a reasonable result. The final result is the mean of three values. In fact, the testing environment is almost fixed, so the difference between the three values is less than 10 Mbps for most of the tests.

5.2 Performance of Para-Snort

Figure 8 shows the performance of different threads on different traces. Figure 9 shows the parallel speedup achieved by the scheme we mentioned above. We choose M-JSQ as the load balancer algorithm and the flow-splitting threshold is 2000. Processing speed is stated from one thread to seven. We choose 7 as the maximum number of processing threads because there are 8 CPU cores and load balancer needs to take up one thread.

The processing speed of each trace is affected by many factors. Higher percentage of HTTP packets can leads to slower speed, because Snort has more HTTP rules than any else protocols. Unknown TCP/UDP-based Protocols make up 93.1 % in CERNET trace, and Snort does not have many rules for these packets. So CERNET trace gets a top speed of 843 Mbps. The packet length can also affect the processing speed. Too short payloads do not have much contribution for throughput. LL1 and LL2 trace have shorter average packet size, so the processing speed for them is much slower.

Some curves, such as LL1 and CERNET, have a flatten top. This is because some bottlenecks other than processing modules have limited the system performance, such as network card performance, memory access time and cache size. When the processing speed reaches a limitation, increasing the number of processing modules can no longer increase the processing speed. So the top of the curves turns flat.

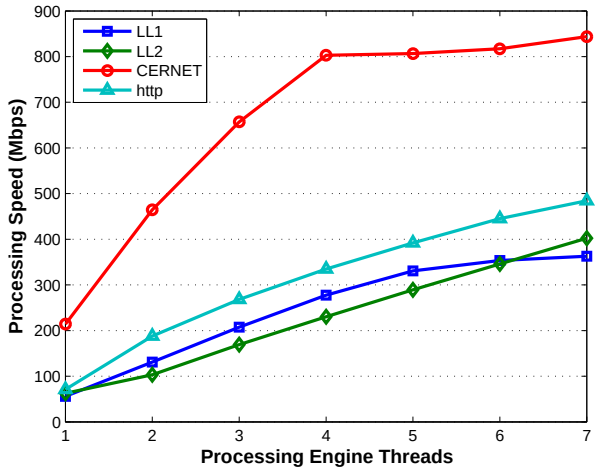

Figure 8. The processing speed

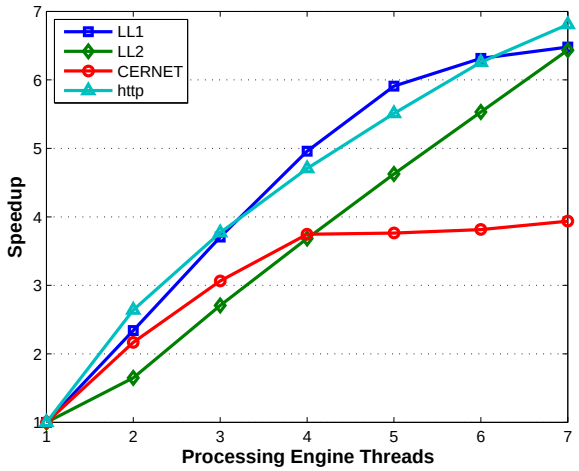

Figure 9. The speedup under different traces

Test on Lincoln Lab traces have a speedup of 5.6 and 6.4 with 7 threads, and are almost linear. In contrast, MultiSnort only has about a speed up of 3.5 under LL traces. From the result we can see that parallel NIDS has a good speed up. CERNET traffic has a lower speedup of about 3.6 and 4.0. It is because CERNET has faster processing speed for single thread, and reaches some bottlenecks such as memory access for 7 threads.

5.3 Performance of Different Load Balancers

We compared four load balancer algorithms — IP hash, 5-tuple hash , JSQ and Modified-JSQ (MJSQ), and stat their processing speed with different traces. The threshold for

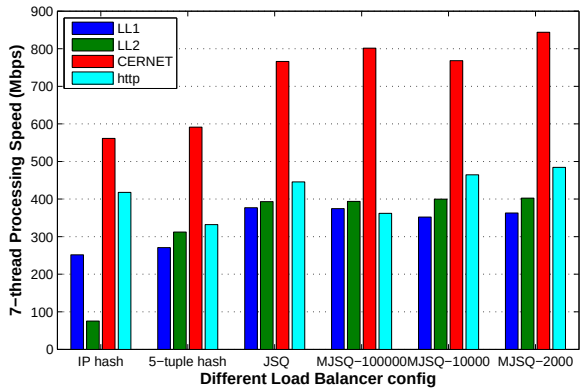

Figure 10. The performance of different load balancers

Modified-JSQ algorithm is 100000, 10000 and 2000. Figure 10 shows that they are mostly increasing from left to right. Modified-JSQ with threshold 2000 has the best performance for most cases, nearly 1.5 times as much as IP hash. From the result we can see Modified-JSQ algorithm has the smallest disparity, which means packets are well-distributed and each processing thread has a balanced load. As mentioned above, the http traces are generated with a equal IP hash value for the 7 processing modules. So for the http traces, IP hash has almost the same performance as JSQ and M-JSQ, and better than 5-tuple hash. Under these special traces, the throughput using IP hash can be seemed as the maximum throughput.

5.4 Performance of Different Pattern Matching

We divide the ruleset into several groups according to the port of each rule. Snort compiles each group and the memory usage of DFA is shown in Figure 11. AC-WM algorithm has much smaller size of DFA than AC algorithm. This will effectively avoid memory explosion on some special rules. We also compared the performance of the two algorithms. Figure 12 shows the performance comparison between AC and AC-WM algorithms. The testing environment is 7-thread processing. In most cases, the performance of AC-WM is a bit worse than AC algorithm, for LL traces the performance turns much worse. But AC-WM algorithm decreases memory usage a lot. In our test we use only AC algorithm. It is because AC algorithm has better performance in most of situations, and our platform is not sensitive about memory usage. However, if users need Para-Snort running on some embedded devices, which have limited memory, then AC-WM algorithm is an efficient choice. Users can choose the pattern matching algorithm according to their platforms.

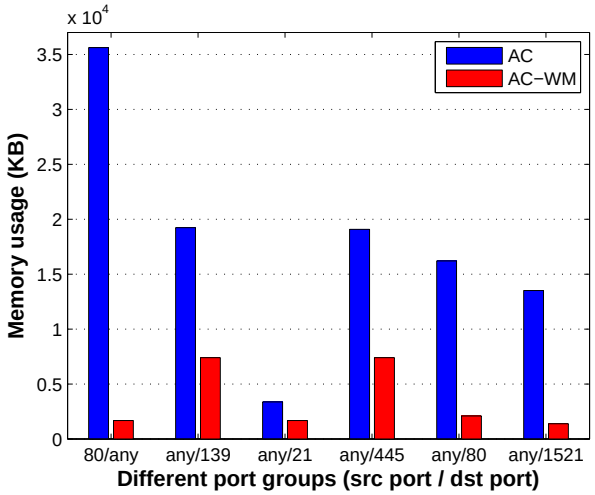

Figure 11. Memory usage of AC and AC-WM

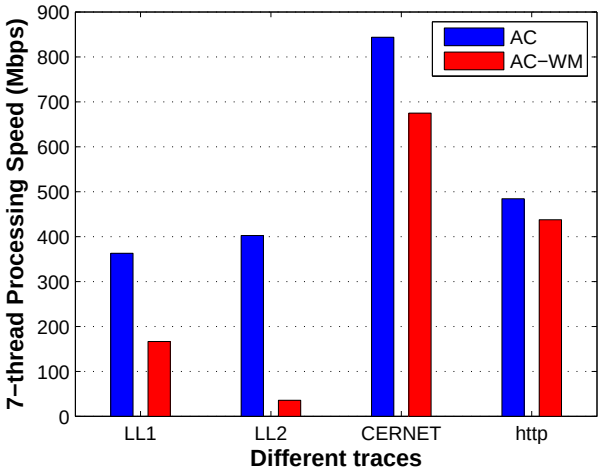

Figure 12. The performance of AC and AC-WM

6 Conclusions

In this paper, we present Para-Snort, a structure of multi-thread Snort for high-performance NIDS and anti-virus on multi-core IA.

A modular design of parallel Snort is proposed, it's flexible and easy to scale with new modules, even enables to replace some modules with hardware acceleration.

Efforts are made to break the bottlenecks of parallel Snort in load balancing and multi-pattern matching. Experimental results show that there is a significant speedup of about 4 to 6 for various traces.

In future work, we plan to further improve the performance of Para-Snort processing engine; there is still much head room because the CPU occupancy rate is only on the average of 70 %. We expect the gain will mostly from fine-tuning algorithms and optimize the code.

Future work also includes the implementation of inline mode data source module, based on NFQueue, so that the system can act as NIPS (Network Intrusion Prevention System), an inline device, without the trouble of dropping packets.

7 Acknowledgements

This work is supported by the National High-Tech R&D Program (863 Program) of China under grant No.2007AA01Z468. The authors would like to thanks Yaxuan Qi, Baohua Yang, and other colleagues in Network Security Lab of Tsinghua University, for their suggestions.

References

[1] M. Roesch. Snort – lightweight intrusion detection for networks. In *the 13th USENIX Conference on System Administration*, 1999.

[2] V. Paxon. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, USA, January 1988.

[3] V. Jacobson, C. Leres, and S. McCanne. Libpcap. *HTTP://www.tcpdump.org/*, June 1994.

[4] M. Roesch. Snort 3.0. In *CanSecWest 2009*, 2009.

[5] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.

[6] Supra-linear packet processing performance with intel multi-core processors white paper. Intel Corporation, 2006.

[7] H. Welte. libnetfilter_queue project. *HTTP://www.netfilter.org/projects/libnetfilter_queue/index.html*, June 2008.

[8] Sourcefire. Clam antivirus 0.95.2 user manual. *HTTP://www.clamav.com/*, 2008.

[9] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 darpa off−line intrusion detection evaluation, 2001.

[10] L. Laboratory. Darpa intrusion detection data sets. *HTTP://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html*.

[11] A. Turner. Tcpreplay. *HTTP://tcpreplay.synfin.net/trac/*.