YACA: Yet Another Cluster-based Architecture for Network Intrusion Prevention

Fei He[1,2], Yaxuan Qi[2,3], Yibo Xue[2,3] and Jun Li[2,3]
[1]Department of Automation, Tsinghua University, Beijing, China
[2]Research Institute of Information Technology, Tsinghua University, Beijing, China
[3]Tsinghua National Lab for Information Science and Technology, Beijing, China
hefei06@mails.tsinghua.edu.cn
{yaxuan, yiboxue, junl}@tsinghua.edu.cn

Abstract—**Inline stateful and deep inspection for network intrusion prevention system (NIPS) is progressively challenging to cope with the fast growing volume and ever increasing complexity of network traffic. Traditional cluster-based architectures provide a solution for scalable and high performance NIPS, but with some common limitations. This paper proposed yet another cluster-based architecture (YACA) with a stateful traffic splitter. As an architectural approach for building a high performance NIPS, we present a novel design of stateful traffic splitter. The performance of its network processor implemented prototype demonstrates that such a design is suitable for the proposed architecture.**

***Keywords-** intrusion prevention, session table, network processor*

I. Introduction And Related Works

Stateful and inline deep inspection for intrusion prevention is growing increasingly difficult, due to the growth in both the volume of network traffic and the complexity of the analysis required to perform. Traditional network equipments, such as routers and firewalls, are relatively easy to scale up for high-speed network links because their operations focus on packet headers only and the functions of these systems are relatively static. The difficulties of building high performance network intrusion prevention systems stem mainly from the fact that NIPS needs to analyze not only packet headers but also packet payloads, and that the operations to be performed on each packet is growing with the continuously increase of network attacks.

Both complexity and the need for scalability and flexibility make it hard to design a high performance NIDS or NIPS. Application Specific Integrated Circuits (ASICs) lack the needed scalability and flexibility. New generation multi-core network processors, such as Cavium OCTEON [1], and NetLogic XLR [2], can reach over 10Gpbs line speed, when used to implement packet header based network equipments. However, network processors are not suitable to implement NIPS for two reasons. First, NIPS are both memory-usage and memory-bandwidth intensive. Network processors use multi-core architecture to exploit the inherent parallelism in network traffic, and use specific chips to offload some of the important operations of packet processing, such as packet receiving and transmitting, header parsing and workload distribution between processing engines. But the overall memory bandwidth of network processors is limited by the development of memory technology [10]. Second, while over 90% of L4-L7 services software is deployed using standard programming model on general purpose processors [3], it is sophisticated to porting existing software to network processors and exploit the maximum hardware performance.

One design that offers both performance and flexibility is cluster-based NIDS, which deploys multiple commodity PC platforms as detection engines behind a traffic splitter [4] [5] [6]. Traditional cluster-based architectures usually use simple traffic splitters, which distribute traffic to backend engines on packet basis or flow basis, due to performance consideration. However, these architectures have some common limitations. First, it is complicated to implement the prevention functionality in this kind of architectures. Second, since some correlation information is lost at the time of traffic distributing, traditional cluster-base NIDS/NIPS needs a communication scheme between backend detection engines, which brings in a substantial amount of overhead. In addition, traffic analysis software deployed on the backend nodes needs to be modified to support this kind of coordination.

In this work we pursue a different architectural approach: yet another cluster-based architecture, YACA, which uses a Network Processor as a stateful in-line traffic splitter, and several commodity PC platforms as backend nodes to perform deep inspection. The goal is to take advantage of both the inline line-speed processing power of new generation network processors and the flexibility of general purpose processor platform. There are two main differences between YACA and traditional cluster-based architecture. Firstly, the traffic splitter in YACA maintains per-flow state which offers not only the feasibility of the prevention functionality, but also provides several other architectural advantages (Section II). The second difference is the separation of I/O intensive inline forwarding engines and deep inspection engines which are memory-bandwidth intensive. Multi-core network processors are used to implement the forwarding engines, in order to guarantee line speed packet forwarding performance. Commodity PCs are used for deep inspection, which provides the needed flexibility of intrusion prevention system, and are cost-effective to scale up to line speed deep inspection.

As the critical component of a YACA system, the stateful traffic splitter should be both flexible and high performance. We present a novel session table design which separates the control and data plane data structures. The control plane session table is like ordinary hash table with some modification to support some specific functions for YACA-based NIPS

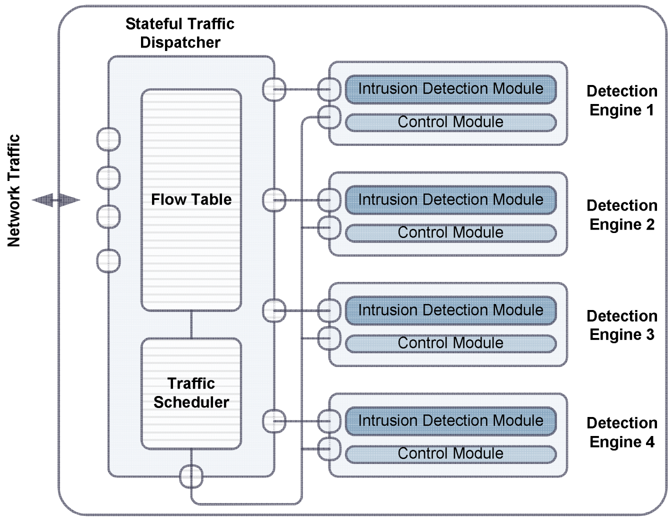


Figure 1. YACA Architecture

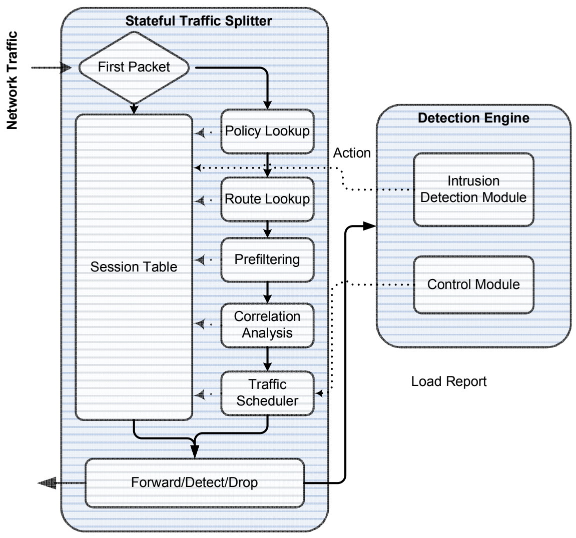


Figure 2. Block Diagram of Packet Processing Subsystem

including correlation analysis [7], prefiltering, and adaptive load balancing. Since data plane session table only needs to provide per-flow state lookup, it can be tailored to store in fast/on-chip memory which guarantees high forwarding performance.

The contributions of this paper are as follows:

1) Architecture: Based on the analysis of existing NIPS architectures, a new cluster-based architecture is proposed: YACA. This architecture provides several advantages over traditional cluster-based architectures.

2) Design: We propse a session table design concerning both flexibility and high performance for the stateful traffic splitter which is the key component of a YACA system.

3) Implementation: We implement our design of the traffic splitter on a Cavium OCTEON 5860 multi-core network processor. The performance results illustrates the feasiblity of our design and the prototype traffic splitter can achieve over 10Gpbs stateful forwarding performance with minimum packets.

The rest of this paper is organized as follows: In Section II, we present the YACA architecture and the functionalities and optimization mechanisms it can support. In Section III, we discuss the design of the stateful traffic splitter in the YACA architecture. The performance of the proposed system is evaluated in Section IV. We discuss future works and conclude in Section V.

II. The YACA Architecture

In this section, we present the YACA architecture. We begin with an overview of the architecture and the motivation behind it, and then discuss in detail the stateful traffic splitter, which acts as the front-end of the platform.

A. Overview

The YACA based NIPS can be divided into two types of components, as showed in Figure 1: a stateful traffic splitter as the frontend and several backend detection engines. The YACA architecture can be characterized by the following properties:

- The I/O intensive forwarding function of an inline device and the memory-bandwidth intensive traffic analysis function of IDS/IPS are implemented separately on different hardware platforms.
- The frontend traffic splitter in the platform maintains per-flow state to support flexible traffic dispatching and other optimization to be described in the next subsection.

B. The Frontend

The stateful packet forwarding engine is the key component in YACA architecture. Figure 2 shows the block diagram of forwarding engine. All incoming traffic that arrives at the system enters the stateful traffic splitter. When a packet enters the traffic splitter, the engine looks up the session table with the usual 5-tuple of source and destination IP address, source and destination ports, and transport protocol (TCP or UDP). The packet is processed according to the action stored in its session state. The action includes forward, drop and inspect. The inspect action points to the deep inspection, and it can even be extend to a sequence of detections, called detection chain.

If the packet is the first packet in its flow, a series of operations, including policy lookup, route lookup, correlation analysis, initial load balancing decision, need to be performed. The results of all these operations are resulted in the session state of the corresponding session. After examining a packet, detection engines will update the corresponding session state based on the detection result.

Session table is the most important component for achieving high performance in frontend, as it enables fine-grained, per-session decision-making instead of per-packet decision-making. Based on per-session state, several important functionalities can be implemented on the stateful frontend:

1) Policy enforcement: Policy enforcement module provides the capability of determining what kind of operations need to perform on a group of traffic specified by some fields in packet header, typically 5-tuple.

2) *Correlation analysis*: Each session of application protocols like FTP, H.323 usually contains several flows. Flows belongs to the same session contains information correlated for intrusion detection. Besides, detection and prevention of some intrusions, such as worm, port scanning, also require a single detection engine to analyze correlated flows of the attacks. In cluster-based NIDS, simple flow-based load balancing schemes, such as dispatching flows according to hash value of several fields in packet header, are usually used to distributed traffic to backend. When using these schemes, some correlation information is lost. The loss of correlation information severely affects the detection accuracy. Inter-backend communication mechanisms can be designed to share correlation information between detection engines to solve the problem. However, this solution can incur a substantial amount of communication traffic. Moreover, detection software deployed on the backend nodes need to be modified to adapt to the coordination between engines. When correlation analysis is implemented in the frontend, the loss of correlation information can be reduced, as correlation analysis groups the correlated flows into sessions. Adaptive session-level traffic distributing schemes can be used instead of flow-level schemes.

3) *Adaptive load balancing*: The goal of load balancing is to distribute the network traffic among the backend nodes so as to keep them as evenly loaded as possible. However, the characteristics of intrusion detection and network traffic place certain constrains on the design of such a load balancer. For instance, traffic belongs to the same flow or even same attack context should be placed on the same detection engine, but the length of the flow and the workload needed to process the flow are not known at the time that the initial load balancing decision are made. Placing such restrictions on the mapping of the packets to particular engines may easily lead to load imbalance among sensors, so adaptive load balancing schemes are needed to avoid such situations. Without per-session state, load balancing may not perform fine-grained load adjusting when some detection engines cannot cope with their workload.

C. *The Backend*

Unlike traditional cluster-based architecture, existing open source NIDS system, such as Snort and Bro, can be deployed on the backend detection engines with little modification. In traditional cluster-base NIDS, most of which employ simple hash-based load distribution, packets that are part of a given attack context can be distributed over several different detection engines, making it difficult, and computationally expensive, to combine information from the different engines and recognize the attack. Given the stateful frontend traffic dispatcher in YACA, most of the correlation analysis can be done at the frontend.

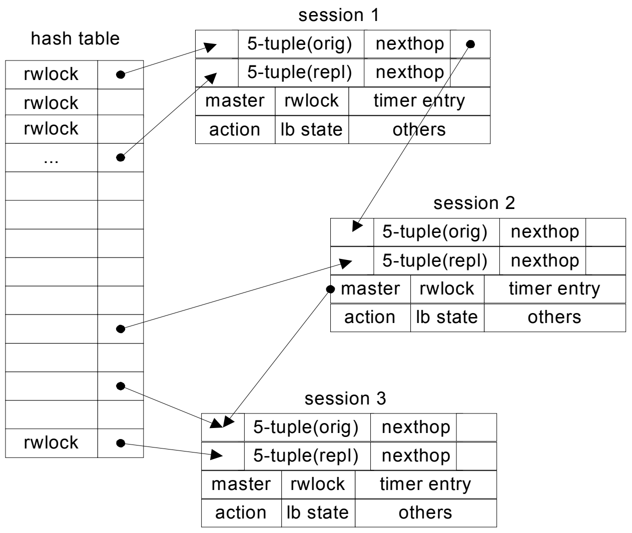
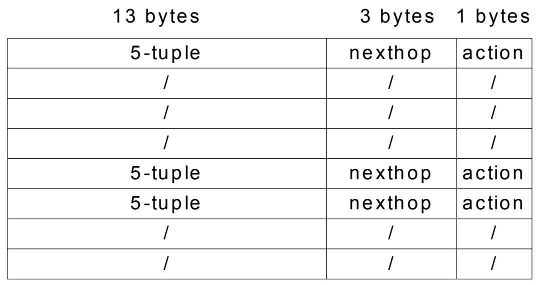


Figure 3. Data structure of Basic Session Table



13 bytes	3 bytes	1 bytes
5-tuple	nexthop	action
/	/	/
/	/	/
/	/	/
5-tuple	nexthop	action
5-tuple	nexthop	action
/	/	/
/	/	/

Figure 4. Data structure of Active Flow Buffer

As shows in Figure 2, despite intrusion detection module, there are two modules needed in backend detection engines. Control Module on each detection engine checks the load of the detection engine and reports to the frontend load balancer periodically. Intrusion Detection Module determines which action to perform on the packet or the following packets in the same session, and update the session state in the frontend.

Furthermore, a sophisticated load balancing scheme can support not only homogeneous backend detection engines, but heterogeneous engines. Thus, the configurations of backend detection engines are much more flexible.

III. BUILDING A STATEFUL TRAFFIC SPLITTER

This section details our experience in building a stateful traffic splitter for intrusion prevention system based on the YACA architecture.

A. *Hardware Platform*

We have implemented a prototype stateful traffic splitter for YACA using Cavium OCTEON 5860 network processor. The OCTEON 5860 has 16 MIPS cores running at 750 MHz with 2MB on-chip shared L2 cache and 4GB DDR2 SDRAM. More detail about Cavium OCTEON 5860 can be found at [1]. There are two basic programming choices with Cavium OCTEON Software Developer Kit: programming in Linux mode (with Linux OS) and Simple Executive mode (no OS).

Our design is implemented in simple executive mode to eliminate the performance cost due to running an OS.

B. Session Table Design

Session table is the most important building block in the stateful frontend, as every packet that enters the system is processed based on the action in the session entry. We propose our design of fast session table suitable for the YACA architecture in this subsection.

1) Design Principle

In network systems, control plane and data plane often operate the same data structure in shared memory. However, we argue that using two different data structures for session table in control and data plane can achieve much better performance while not losing any flexibility. The reason is that the operation of most packets is forwarding based on the per-flow state without changing the session table, so a lookup-optimized session table or even an incomplete table can be used in data plane.

2) Basic Session Table

We now present our session table design. First we present the basic form of our design which we call Basic Session Table (BST). BST uses the hash table algorithm in which the collisions are resolved by chaining. A BST consists of an array of *m* buckets with each bucket pointing to the list of items hashed into it.

A session entry contains two parts: the first part is direction-related state (or flow-level information), which is called wing in this work. A wing, which consists of 5-tuple of one direction of the session and direction-related information, such as routing information, is the item actually inserted into the hash table. The second part contains session-level information which is not direction-related. Each session entry has a master session pointer, pointing to its correlated session. This design enables the load balancing module, correlation analysis module and other modules access correlated flows through one session table lookup.

In our design, we utilize some important acceleration hardware embedded in OCTEON 5860 network processors. The first one is called Free Pool Allocator (FPA) [8], a hardware buffer management component, which services malloc/free requests very quickly to accelerate the session set-up and tear-down speed. The second hardware accelerator is timer unit. Since every active session needs a timer, implementing software-based timer consumes CPU time to traverse the timer data structure. The OCTEON hardware timer support can minimize timer maintenance consumption [8]. The hardware traverses up to 80 million timer entries per second.

Another important design consideration is how to provide mutual exclusion effectively in the multi-core environment. Instead of using a single lock to synchronize every access to the session table, which is not scalable, we split the session table into independently synchronized components. Every bucket and its corresponding chain is an independently synchronized component for query, insert and delete operations. Before performing these operations, the lock in the bucket should be acquired. Since two wings of a session are often hashed into different buckets, two packets belong to different direction of the same session may be processed in two cores at the same time. In addition, there are other threads performing load adjusting and detection decision making may access session entry, as shown in Figure 2. Therefore, we need an additional lock in every session entry to cope with these situations.

To achieve further finer-grained synchronization, readers-writers locks are used instead of spinlocks. The session table in the YACA architecture has the property that most operations, called readers, return information about the flow state without modifying the state, while only a small number of operations, called writers, actually modify the flow state. There is no need for readers to synchronize with one another. On the other hand, writers must lock out readers as well as other writers. Therefore, we employ readers-writers lock to allow multiple readers or a single writer to enter the critical section concurrently.

3) AFB-based Session Table

The BST design uses fine-grained synchronization which doesn't have scalability problem comparing to coarse-grained synchronization. However, even if a critical section is not blocked by other threads, acquiring a lock usually requires hundreds of cycles on typical network processor platforms. In BST, two locks need to be acquired which consumes too much processing time.

Based on the design principle described previously, we present an alternative design which separates the control plane and data plane session tables. Since the operations of most packets involve just lookup the session table, get the session state, and process based on the action field and route information stored in session entry, a tailored data structure called Active Flow Buffer (AFB) is employed as data plane session table. We continue to use the same data structure in BST as the control plane session table. This design is named as AFB-based Session Table (AST). AFB is a compact hash table without collision resolving mechanism as a buffer of active flows, as shown in Figure 4. Entries stored in AFB contain all necessary information needed to process packets. Entries stored in the active flow buffer contain all necessary information needed to process packets. The following pseudo-code describes the lookup procedure of AST.

```
AST-Lookup (Tuple)
key1 ← hash(Tuple, BUFFER_SIZE)
if Buffer[key1].tuple = Tuple then
  return Buffer[key1].state
else
  state ← BST-Lookup(Tuple)
  ReplacePolicy (Buffer[key1], state)
  return state
```

The advantages of AST are as follows: 1) the memory footprint of AFB is much smaller than that of BST, so it can be stored in cache or other kind of on-chip memory; 2) both lookup and modification of AFB are atomic, so no locking is needed. Thus, the performance of AST is determined by two factors:

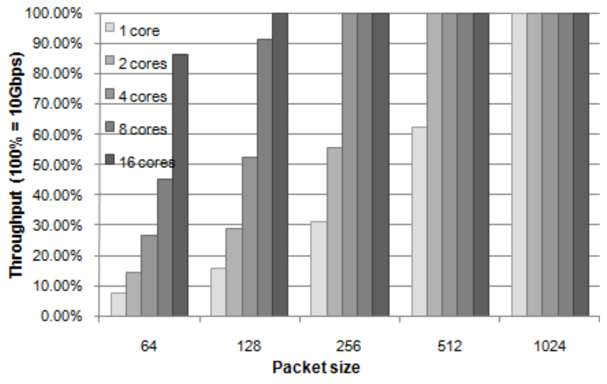


Figure 5. Throughput of Stateful Traffic Splitter using BST

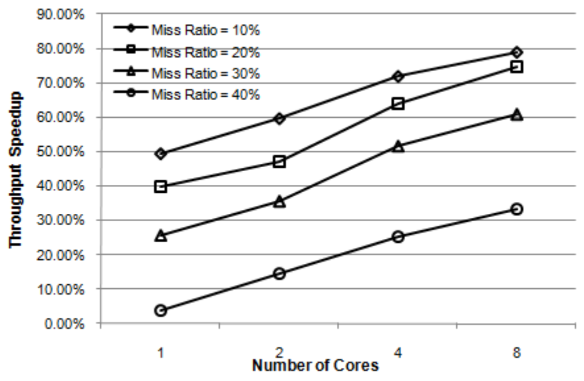


Figure 6. Throughput Speedup of AST over BST

1) The hit rate of AFB, i.e. the proportion of packets that do not need a BST lookup. We define the load factor of AFB as $\alpha = n / m$, n is the concurrent flows to support, which is twice the number of concurrent sessions. m is the size of AFB.
2) The relative performance penalty of an AFB miss comparing to per-packet locking operation needed BST.

IV. Performance Evaluation

To verify the efficiency of the stateful traffic splitter, we have done several hardware tests. The performance of the BST and AST is evaluated on a Cavium OCTEON 5860 multi-core network processor. A SmartBits packet generator is used to generate 10Gpbs traffic.

Figure 5 show that the traffic splitter using BST reaches 7.8 Gbps throughput for 64-byte packets using 16 cores. When the packet size is over 128 bytes, the forwarding performance is over 10Gbps. It is clear that there are enough resources to implement several NIPS optimizations on the stateful traffic splitter while keeping the 10Gbps throughput.

Figure 6 shows that the traffic splitter using AST achieves about 33~86 percent speedup over BST using 8 cores for the AFB miss ratio from 40% to 10%. It is worth noticing that the traffic splitter using AST can reach over 10Gbps with minimum packet size when using 16 cores. Also, we can see that the speedup of AST over BST increases when more cores are used. The reason is that the performance degradation caused by mutual exclusion in BST increases with the number of cores in use.

The extra memory needed by the AFB for a session table supporting one million concurrent sessions is only 2.5 MB for a load factor of 0.125, 20 MB for a load factor of 1. Experiments using LBNL enterprise traces [9] show that when the load factors (size of AFB divided by number of concurrent flows) are 1, 0.5, 0.25, and 0.125, the miss ratios of AFB are 7.85%, 11.60%, 17.02%, and 24.84% respectively. This means for real life traffic AST can achieve about over 70% performance speedup with only 2.5MB ~ 5MB extra memory consumption.

V. Conclusion and Future works

In this paper, the proposed YACA employs a stateful traffic splitter as the frontend of a cluster-based architecture to provide a more flexible and scalable way to build a NIPS system, and also to remove some limitations of traditional cluster-base architecture. A design principle of separating control plane and data plane session table is proposed to build a high performance stateful traffic splitter while not losing the flexibility. Our prototype implementation and performance results illustrate the feasibility of YACA.

Acknowledgment

This work was supported by National High-Tech R&D 863 Program of China under grant No. 2007AA01Z468.

References

[1] http://www.caviumnetworks.com/OCTEON-Plus_CN58XX.html.

[2] http://www.netlogicmicro.com/Products/MultiCore/MultiCore.htm.

[3] M. R. Hussain, "Multi-Core Processors for Networking and Communication Equipment", Keynote of ANCS, 2006.

[4] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware," Proc. of 10th International Symposium on Recent Advances in Intrusion Detection, 2007.

[5] K. Xinidis, K. G. Anagnostakis, and E. P. Markatos, "Design and Implementation of a High-Performance Network Intrusion Prevention System," Proc. of 20th International Information Security Conference, 2005.

[6] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "An Active Splitter Architecture for Intrusion Detection and Prevention," IEEE Transactions on Dependable and Secure Computing, Vol. 3, No. 1, 2006.

[7] A. Le, R. Boutaba, and E. AI-Shaer, "Correlation-Based Load Balancing for Network Intrusion Detection and Prevention Systems," Proc. of 4th International Conference on Security and Privacy in Communication Networks, 2008.

[8] Cavium Networks, "OCTEON Plus CN58XX Hardware Reference Manual," 2008.

[9] LBNL/ICSI Enterprise Tracing Project, http://www.icir.org/enterprise-tracing/index.html.

[10] J. Mudigonda, H. M. Vin, R. Yavatkar, "Overcoming the Memory Wall in Packet Processing: Hammers or Ladders," Proc. of the 2005 ACM Symposium on Architecture and Networking and Communication Systems, 2005.