Kargus: A Highly-scalable Software-based Intrusion Detection System

Muhammad Jamshed
Department of Electrical Engineering, KAIST
ajamshed@kaist.ac.kr

Jihyung Lee
Department of Electrical Engineering, KAIST
lee.jh@kaist.ac.kr

Sangwoo Moon
Department of Electrical Engineering, KAIST
mununum@kaist.ac.kr

Insu Yun
Department of Computer Science, KAIST
wuninsu@kaist.ac.kr

Deokjin Kim
Cyber R&D Division, NSRI
djkim@ensec.re.kr

Sungryoul Lee
Cyber R&D Division, NSRI
srlee0525@ensec.re.kr

Yung Yi
Department of Electrical Engineering, KAIST
yiyung@ee.kaist.ac.kr

KyoungSoo Park
Department of Electrical Engineering, KAIST
kyoungsoo@ee.kaist.ac.kr

ABSTRACT

As high-speed networks are becoming commonplace, it is increasingly challenging to prevent the attack attempts at the edge of the Internet. While many high-performance intrusion detection systems (IDSes) employ dedicated network processors or special memory to meet the demanding performance requirements, it often increases the cost and limits functional flexibility. In contrast, existing software-based IDS stacks fail to achieve a high throughput despite modern hardware innovations such as multicore CPUs, manycore GPUs, and 10 Gbps network cards that support multiple hardware queues.

We present Kargus, a highly-scalable software-based IDS that exploits the full potential of commodity computing hardware. First, Kargus batch processes incoming packets at network cards and achieves up to 40 Gbps input rate even for minimum-sized packets. Second, it exploits high processing parallelism by balancing the pattern matching workloads with multicore CPUs and heterogeneous GPUs, and benefits from extensive batch processing of multiple packets per each IDS function call. Third, Kargus adapts its resource usage depending on the input rate, significantly saving the power in a normal situation. Our evaluation shows that Kargus on a 12-core machine with two GPUs handles up to 33 Gbps of normal traffic and achieves 9 to 10 Gbps even when all packets contain attack signatures, a factor of 1.9 to 4.3 performance improvements over the existing state-of-the-art software IDS. We design Kargus to be compatible with the most popular software IDS, Snort.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and Protection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

General Terms

Design, Security, Performance

Keywords

Intrusion Detection, Pattern Matching, Batch Processing, GPU

1. INTRODUCTION

The demand for a high-speed intrusion detection system (IDS) is increasing as high-bandwidth networks become commonplace. The traffic aggregation points in the regional ISPs are already handling tens of Gbps of Internet traffic, and many large enterprise and campus networks are adopting 10 Gbps links for Internet connection. 4G mobile communication technologies such as Long-Term Evolution (LTE) employ IP packets for internal routing while the traffic is aggregated into a few central points whose bandwidth requirements often exceed a multiple of 10 Gbps. Securing the internal networks has become a common and crucial task that constantly deals with flash crowds and external attacks.

Detecting malicious attack patterns in the high-speed networks entails a number of performance challenges. The detection engine is required to monitor the network traffic at line rate to identify potential intrusion attempts, for which it should execute efficient pattern matching algorithms to detect thousands of known attack patterns in real time. Reassembling segmented packets, flow-level payload reconstruction, and handling a large number of concurrent flows should also be conducted fast and efficiently, while it should guard against any denial-of-service (DoS) attacks on the IDS itself.

Today's high-performance IDS engines often meet these challenges with dedicated network processors [1, 3], special pattern matching memory [30], or regular expression matching on FPGAs [12]. However, these hardware-based approaches limit the operational flexibility as well as increase the cost. In contrast, software-based IDSes on commodity PCs lessen the burden of cost and can extend the functionalities and adopt new matching algorithms easily. However, the system architecture of the existing software stacks does not guarantee a high performance. For example, Snort [40], the most widely-used software IDS, is unable to read the network packets at the rate of more than a few Gbps, and is designed to utilize only a single CPU core for attack pattern matching.

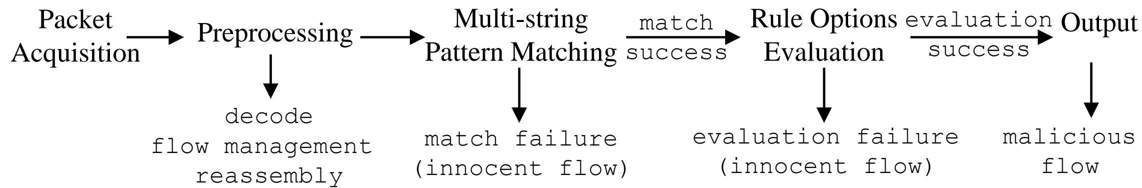

Figure 1: Typical Intrusion Detection Process in a Signature-based IDS

More recent IDS solutions like Suricata [9] and SnortSP [8] employ multiple CPU cores, but the performance improvement is marginal mainly due to suboptimal usage of modern commodity hardware advancements. To the best of our knowledge, there is no software IDS that can handle 10 Gbps traffic for minimum-sized IP packets. This implies that the software solution itself can be the target of a DDoS attack with a massive stream of small packets.

In this paper, we present Kargus, a highly-scalable software IDS architecture that fully utilizes modern hardware innovation, e.g., multiple CPU cores, non-uniform memory access (NUMA) architecture, multiqueue 10 Gbps network interface cards (NICs), and heterogeneous processors like graphics processing units (GPUs). Two basic techniques that we employ for high performance are *(i) batch processing* and *(ii) parallel execution with an intelligent load balancing algorithm,* as elaborated more next. First, we extensively apply batch processing from packet reception, flow management, all the way to pattern matching. Fetching multiple packets from a NIC at a time significantly reduces per-packet reception (RX) overhead and allows a high input rate up to 40 Gbps even for minimum-sized packets. In addition, in Kargus, each pattern matching function handles a batch of packets at a time. This function call batching allows efficient usage of CPU cache and reduces repeated function call overheads as well as improves GPU utilization for pattern matching. Second, we seek high processing parallelism by balancing the load of flow processing and pattern matching across multiple CPU and GPU cores. We configure NICs to divide the incoming packets by their corresponding flows into the queues affinitized to different CPU cores, and have each CPU core process the packets in the same flow without any lock contention. We also devise a load balancing algorithm that selectively offloads pattern matching tasks to GPU only if CPU is under heavy computation stress. Processing with CPU is preferable in terms of latency and power usage, but excessive workloads could swamp the CPU, which would miss the chance to inspect the packets at a high load. Our load balancing algorithm dynamically adjusts the offloading threshold, enabling us to achieve both low latency and low power usage at a reasonable input rate, but quickly switch to using GPUs when the workload exceeds the CPU capacity.

Our contributions are summarized as follows:

1) We present an IDS design that scales to multiple tens of Gbps regardless of the packet size. On a dual-CPU Intel X5680 platform with 12 processing cores and two NVIDIA GTX 580 GPUs, Kargus achieves 25 Gbps for 64B packets and reaches 33 Gbps for 1518B packets from packet capture to pattern matching. Compared with MIDeA [47], a GPU-accelerated multicore IDS engine, Kargus shows a factor of 1.9 to 4.3 performance improvements by the per CPU-cycle metric.
2) We analyze the performance bottleneck at each stage of packet handling: packet capture, pattern matching, and load balancing across CPU and GPU. Moreover, we present an intelligent load balancing algorithm that dynamically adjusts the resource consumption according to the input workload and reduces both detection latency and power usage when the input rate can be handled by the CPU alone. GPUs are utilized only when the aggregate throughput can be further improved at the cost of increased power consumption and latency. For easy deployment, we design Kargus to be functionally compatible with Snort and its configuration files.

2. BACKGROUND

We provide the brief background of a signature-based IDS that is widely deployed in practice. We focus on the design of a typical software-based IDS and popular pattern matching algorithms that are currently in use, and how we exploit GPU for attack string pattern matching.

2.1 Signature-based IDS Architecture

Figure 1 shows the functional components of a typical IDS through which each incoming packet is processed. The IDS reads packets, prepares them for pattern matching, runs a first-pass multi-string pattern matching for potential attacks, and finally evaluates various rule options and confirms if a packet or a flow contains one or more known attack signatures.

2.1.1 Packet Acquisition and Preprocessing

The first step that an IDS takes is to read incoming packets using a packet capture library. Existing packet capture libraries include pcap [45] and ipfirewall [25]. While these libraries provide a common interface, their performance is often suboptimal in the high-speed networks with multi-10 Gbps input rates. This is mainly because they process the packets one by one, i.e., read one packet at a time from a NIC, allocate a memory chunk for each packet, wrap the packet with a heavy kernel structure (called `sk_buff` in Linux), and pass it to the user-level IDS. It is reported by [22] that 50-60% of the CPU time is spent on per-packet memory allocation and deallocation in this approach.

Recent packet capture libraries such as PF_RING [19], netmap [38], and PacketShader I/O engine [22] avoid these problems by batch-processing multiple packets at a time. To remove per-packet memory allocation and deallocation overheads, they allocate large buffers for packet payloads and metadata, and recycle them for subsequent packet reading. They also use receive-side scaling (RSS) [31] by affinitizing each RX queue to a CPU core, removing the lock contention from concurrent queue access from multiple CPU cores. RSS distributes the incoming packets into multiple RX queues by hashing the five tuples of IP packets (src/destination IP/port and the protocol). This allows the packets in the same flow to be enqueued into the same NIC queue, ensuring in-order processing of the packets in the same flow while having multiple CPU cores process them in parallel without serial access to the same RX queue.

```
alert tcp $EXTERNAL_NET any →
$HTTP_SERVERS 80 (content: "attack10";
                  pcre: "/a[0-9]z/";
                  msg: "ATTACK10 DETECTED")
```

Figure 2: An attack rule of a typical signature-based IDS. The detection engine categorizes the attack rules based on the port numbers.

After reading the packets, the IDS prepares for matching against attack patterns. It reassembles IP packet fragments, verifies the checksum of a TCP packet, and manages the flow content for each TCP connection. It then identifies the application protocol that each packet belongs to, and finally extracts the pattern rules to match against the packet payload.

2.1.2 Attack Signature Matching

Once flow reassembly and content normalization is completed, each packet payload is forwarded to the attack signature detection engine. The detection engine performs two-phase pattern matching. The first phase scans the entire payload to match simple attack strings from the signature database of the IDS. If a packet includes one or more potential attack strings, the second phase matches against a full attack signature with a regular expression and various options that are associated with the matched strings. This two-phase matching substantially reduces the matching overhead in the second phase, bypassing most of innocent packets in the first phase.

Phase 1: Multi-string pattern matching. The first phase matches a set of simple strings that potentially lead to a real attack signature. In Snort, the attack signatures are organized in *port groups* based on the source and destination port numbers of the packet. Only the attack signatures associated with the relevant port group are matched against the packet content, reducing the search space.

Most IDS engines adopt the Aho-Corasick algorithm [11] for multi-string pattern searching. The Aho-Corasick algorithm represents multiple pattern strings in a single deterministic finite automata (DFA) table. Its DFA follows the transition function, $T : S \times \Sigma \rightarrow S$ where S and Σ are the sets of states and alphabets, respectively. Since the computation consists of a series of state machine transitions for an input character stream, the Aho-Corasick algorithm provides $O(n)$ time complexity regardless of the number of string patterns, where n is the size of the input text. The Aho-Corasick algorithm constructs a DFA in a form of 'trie'. The trie manages the normal links that follow the pattern string as well as the failure links that lead to a state where the next input character resumes after failing to match the pattern. The failure links are coalesced by those patterns that share the same string prefixes. This allows the Aho-Corasick algorithm to perform multi-pattern matching without backtracking. There are several alternative multi-pattern matching algorithms such as Boyer-Moore [39] or Wu-Manber [42], but only the Aho-Corasick algorithm ensures the equal performance for the worst and average cases. This makes the algorithm highly robust to various attacks, which is the main reason why the Aho-Corasick algorithm is one of the most widely-used algorithms for intrusion detection [33].

In practice, the majority of normal packets are filtered in the multi-string matching phase. Therefore, the performance of an IDS in a normal situation is determined by the execution time of the Aho-Corasick algorithm. Under attacks, however, the second phase is likely to become the performance bottleneck.

Phase 2: Rule option evaluation. If packets are caught in the string matching phase, they are evaluated further against a full attack signature relevant to the matched string rule. Full attack signatures are described in terms of the rule options in Snort, as exemplified in Figure 2. Snort-2.9.2.1 supports 36 rule options other than '`content`', which specifies a string pattern matched in the first phase. Other popular rule options include '`distance`', '`within`', '`offset`', '`nocase`' and '`pcre`'. `pcre` stands for Perl-Compatible Regular Expression [6], which allows a flexible attack pattern in a regular expression supported by Perl 5. The `pcre` option is typically evaluated last since it requires a full payload scan. Unlike multi-string pattern matching, each PCRE option is implemented as a separate DFA table, so it requires payload scanning twice to confirm a match (string and PCRE matching).

To reduce the waste in memory bandwidth, it would be desirable to combine string and PCRE matching into one scan. One problem with merging multiple PCRE options into one DFA is that it would lead to state explosion. While there have been many works that address the DFA state explosion problem [13,35,41], they are rarely used in practice because (a) it is difficult to integrate other rule options into a table, (b) some PCRE options (such as a back link) cannot be represented by a DFA, and (c) it often becomes a barrier to further performance improvement by GPU, since the proposed solutions involve state management that requires frequent branching.

2.2 Employing GPU for Pattern Matching

High-speed pattern matching in an IDS often makes the CPU cycles and memory bandwidth the bottlenecks. With a large number of processing cores and high memory bandwidth, GPU can further improve the performance of pattern matching. For example, NVIDIA's GTX 580 has 512 processing cores and 192 GB/s of the peak memory bandwidth, which has 39.9x more raw processing capacity and 6x more memory bandwidth compared with a 3.33 GHz hexacore Intel CPU.

Modern GPU achieves high throughputs by parallel execution of many concurrent threads. Its execution model is called Single Instruction Multiple Threads (SIMT), where the same instruction stream is shared by a unit of multiple threads (called a *warp,* which is 32 threads in NVIDIA's CUDA programming library) while each thread works on different data in parallel. Toward high performance, it is important for the threads to take the same execution path since any deviation (even by one thread) would serialize the execution of all the other threads. Also, the memory access by the threads should be contiguous enough to maximally benefit from the high memory bandwidth. For this reason, a complex algorithm that requires high execution-flexibility could produce suboptimal performance, often much lower than that of the CPU version.

Both Aho-Corasick and PCRE matching algorithms run DFA transition with an input text, and one can exploit high parallelism by launching multiple threads that share the same execution path. However, employing GPU is not free since CPU has to spend its cycles to offload the workload to GPU and to get the result back from it. Preparing the metadata for GPU pattern matching consumes both CPU cycles and host memory bandwidth. Also, performing DMA copy of packet payloads and metadata to GPU memory would increase memory access contention, which could slow down other memory-intensive CPU tasks such as reading packets from NICs, preprocessing, and CPU-based pattern matching. We analyze these factors and leverage the offloading cost to ensure both CPU and GPU cycles are well-spent for high pattern matching performance.

3. SYSTEM PLATFORM

Throughout this paper, we base our measurements on a machine with two 3.33 GHz Intel X5680 CPUs and two NVIDIA GTX 580 GPUs. Our IDS platform has 24 GB physical memory with

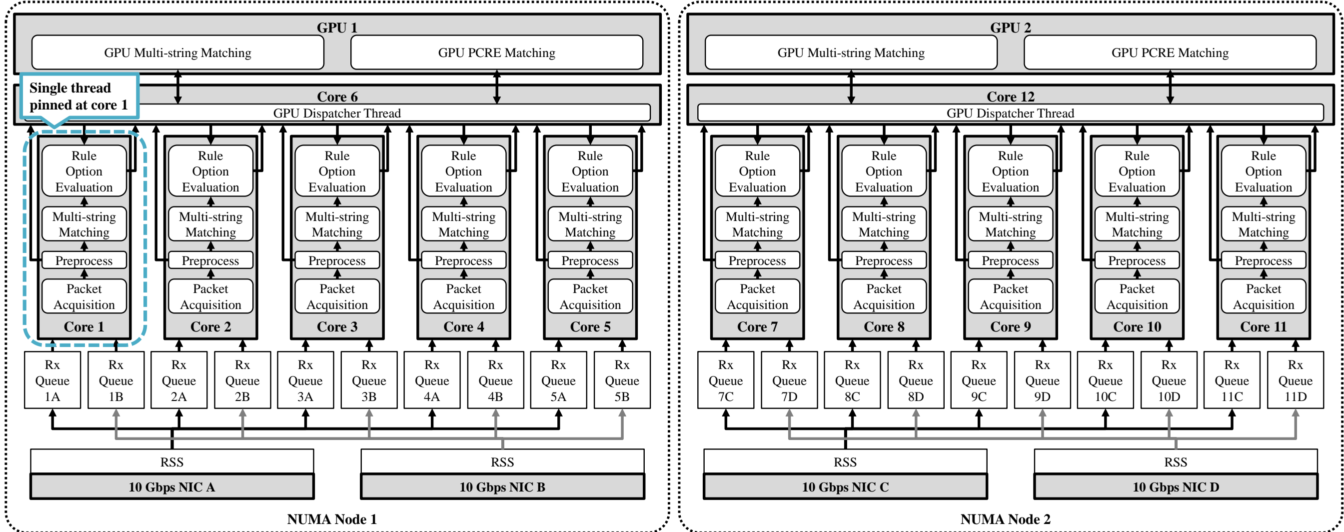

Figure 3: Overall architecture of Kargus on a dual-NUMA machine with two hexacore CPUs and two GPUs

FUNCTION	TIME %	MODULE
`acsmSearchSparseDFA_Full`	51.56	multi-string matching
`List_GetNextState`	13.91	multi-string matching
`mSearch`	9.18	multi-string matching
`in_chksum_tcp`	2.63	preprocessing

Table 1: A profiling result of CPU-intensive functions in Snort-2.9.2.1 monitoring 1518B TCP packets at 20 Gbps input rate

12 GB local memory per each NUMA domain and has two dual-port 10 Gbps Intel NICs with the 82599 chipsets. We use our own IDS workload generator that we have developed for this work from another machine with the same hardware specification except the GPUs. Our IDS workload generator can produce TCP packets with a random payload at 40 Gbps regardless of the packet size. Unless specified otherwise, our throughput numbers include the 14B Ethernet frame header size and the 20B frame overhead (8B preamble and 12B interframe gap), and a packet size in this paper refers to the Ethernet frame size. Including the Ethernet frame header and overhead is necessary to show the line rate for small packet sizes.

4. ARCHITECTURE

In this section, we explain the basic architecture of Kargus. Kargus achieves a high performance in pattern matching by applying the principle of batch processing and parallel execution at each stage of packet processing. It consists of multiple CPU threads where each thread reads and processes packets from NIC queues affinitized to it by RSS. The thread mainly uses its affinitized CPU core for string and PCRE matching, and additionally it offloads packets to GPU when its own CPU core is overloaded. We present the fast packet capture library, function call batching and NUMA-aware data placement in this section, and a load balancing algorithm that adaptively harnesses both CPU and GPU cores in the next section.

4.1 Process Model

Figure 3 illustrates the overall layout of Kargus. Kargus adopts a single-process multi-thread architecture. It launches as many threads as the number of CPU cores, and affinitizes each thread to a CPU core. Threads are divided into IDS engine threads and GPU-dispatcher threads. An IDS engine thread reads the incoming packets from multiple NIC RX queues, and performs the entire IDS tasks, e.g., preprocessing, multi-string matching, and rule option evaluation in its thread. When it detects that its own CPU core is overloaded, it offloads the pattern matching workloads to a GPU-dispatcher thread.

The multi-thread model provides a number of benefits over the multi-process model adopted by earlier works [37, 47]. First, it allows efficient sharing of attack signature information among the threads. The entire ruleset of Snort-2.9.2.1 amounts to 1 GB when it is loaded, and having a separate copy per process would be wasteful, otherwise they need to set up a shared memory or access it via IPC. Second, employing GPUs in multiple processes would require each process to reserve a portion of GPU memory for its own DFA tables. For this reason, MIDeA [47] uses a compact version of Aho-Corasick DFA (called *AC-Compact*), which incurs a performance hit. Third, multiple threads would collect a bunch of packets for short time, and would saturate the GPU more quickly. This would reduce the service latency and allow better utilization of the GPU.

An alternative to the one-thread-do-all model is pipelining [4,5,7]. In the pipelining model, threads are divided into I/O and analyzer threads. That is, some threads are dedicated to packet capture and flow management, while the others focus on pattern matching of the received packets. While this architecture is suited for the existing packet capture libraries that do not exploit multiple RX queues, it tends to produce a suboptimal performance due to inefficient CPU utilization. Pipelining suffers from heavy cache bouncing due to the frequent passing of packets between I/O and analyzer threads. Also, the CPU cores of the I/O threads are often underutilized since the pattern matching speed of the analyzer threads is typically slower than the input rate of the I/O threads at high-speed networks. Table 4.1 shows the top four functions that consume most CPU cycles in a single-threaded Snort-2.9.2.1 with a 3,111 HTTP attack rules under 20 Gbps input rate of 1518B TCP packets. The top three functions are called at multi-string pattern matching, which

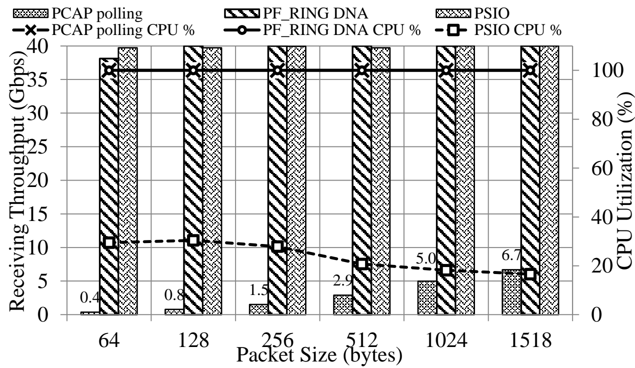

Figure 4: Packet reception performance and CPU utilization of various packet I/O libraries

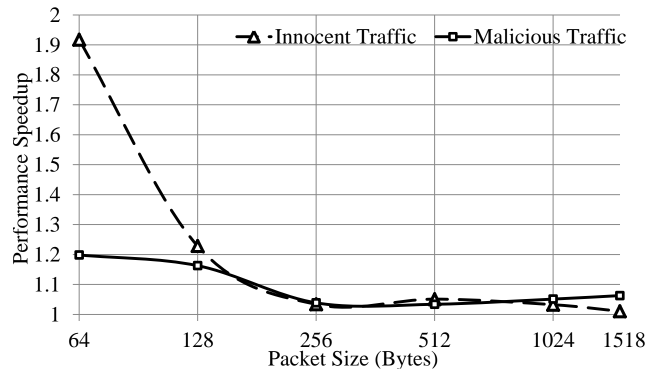

Figure 5: Speedups with function call batching. The input rate is 40 Gbps with innocent and malicious traffic

is the primary bottleneck in this workload. This implies that the pipelining model would eventually slow down the I/O threads if the analyzer threads cannot catch up with the input rate. On the contrary, the one-thread-do-all model avoids this problem since it naturally adjusts the pattern matching throughput to its own input rate.

4.2 High-Performance Packet Acquisition

A high-performance IDS requires the capability of capturing incoming packets fast with few packet drops. Since existing packet capture libraries (e.g., pcap and ipfirewall) are not designed for multi-10 Gbps networks, they become the first obstacle to high performance regardless of the performance of the pattern matching.

To boost the packet capture performance, we use the I/O Engine (PSIO) from the PacketShader software router [22]. PSIO drastically improves the packet RX performance by batch processing multiple packets from NICs. It allocates a large contiguous kernel buffer per each RX queue (e.g., 8 MB) at driver initialization and moves multiple packets from a RX queue to the corresponding buffer in a batch. When the packets are consumed by a user application, it reuses the buffer for the next incoming packets, obviating the need to deallocate the buffer. This effectively reduces memory management and I/O MMU lookup overheads per packet, which is the main bottleneck of other packet capture libraries. Moreover, PSIO removes most of redundant fields of the `sk_buff` structure tailored to IDS packet processing needs, and prefetches the packet payload to the CPU cache while moving the packets to a user-level buffer, allowing the applications to benefit from the CPU cache when processing the received packets. We modify the original PSIO to map the flows in one connection to the same RX queue regardless of their direction. The basic idea is to modify the initial seeds of RSS such that the RSS hashes of the packets in the same TCP connection take on the same value regardless of the direction [48]. We add a callback function to feed the received packets to the payload inspection part.

PF_RING provides the similar batching benefit, and its Direct NIC Access (DNA) extension allows the NIC to move the packets directly to the kernel buffer without CPU intervention. However, PF_RING DNA limits each thread to tap on at most one RX queue, reducing the load balancing opportunity when an IDS monitors multiple 10 Gbps ports. Distributing the packets in the user-level threads/processes could mitigate the problem. However, it would not only contaminate the CPU cache but also increase the IPC overhead. In addition, PF_RING DNA depends on application-level polling, which wastes CPU cycles and increases power consumption regardless of the input load. In contrast, PSIO allows a NAPI-like functionality in the user level and it blocks when there is no packet to process.

Figure 4 compares the RX performance of various packet capture libraries, with a 40 Gbps input rate. PCAP shows less than 7 Gbps regardless of the packet size since it suffers from high per-packet memory management and copy overheads. PF_RING and PSIO mostly show the line rate over the various packet sizes, but the CPU consumption of PF_RING is 100% due to polling. In comparison, the CPU consumption of PSIO ranges from 29.6% (64B) to 16.6% (1518B).

4.3 Function Call Batching

The function call overhead becomes significant when an IDS deals with a massive number of RX packets. For example, to achieve 10 Gbps with 64B packets, the IDS has to handle 14.88 million packets per second. Calling a series of pattern matching functions for each packet would increase the function call overhead and would reduce the overall throughput.

To remedy this problem, Kargus ensures that packets are passed from one function to another in a batch, where only when a packet needs to take a different path, it diverges on the function call. In a typical IDS, each packet travels through common functions like `decode(),preprocess(),manage_flow(),ahocorasick_match()` after it is read. When an IDS engine thread in Kargus reads a batch of packets from RX queues, it passes the batch to these functions as an argument instead of calling them repeatedly for each packet. This effectively reduces the per-packet function call overhead from the packet acquisition module to the rule option evaluation module. The advantage of the function call batching becomes pronounced especially for small packets. Figure 5 shows the performance gain of function call batching in Kargus over the iterative function calls from packet reception to rule option evaluation. We generate innocent and attack traffic at the rate of 40 Gbps. We observe that the function call batching is most effective at 64B packets, producing a 20% to 90% speedup over the iterative counterpart. As expected, as the packet size becomes larger, the bottleneck shifts to pattern matching from function call overheads, which reduces the benefit from batched function calls.

4.4 NUMA-aware Data Placement

When the packet reception is no longer a bottleneck, memory access becomes the key factor that dominates the IDS performance. The NUMA architecture is becoming popular with multi-CPU sys-

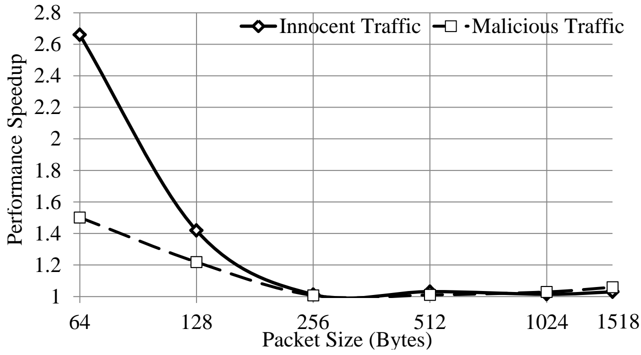

Figure 6: Speedups from removing shared globally variables. The input rate is 40 Gbps with innocent and malicious traffic

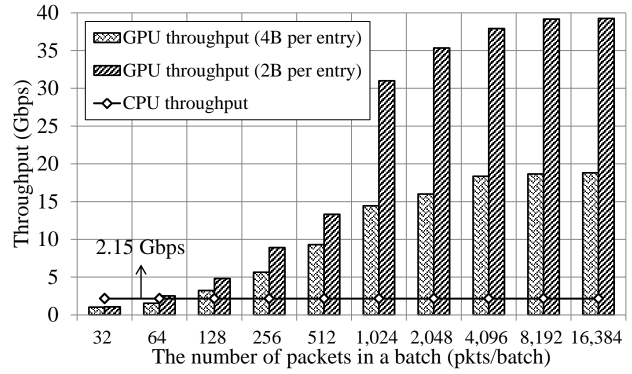

Figure 7: Multi-string pattern matching performance. We use one NVIDIA GTX 580 card for GPU and one Intel X5680 CPU core for CPU

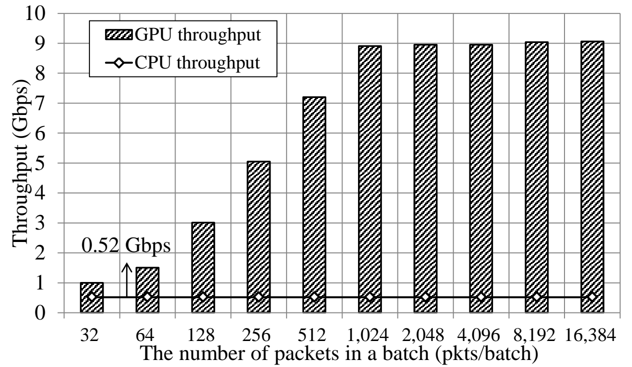

Figure 8: PCRE pattern matching performance in CPU and GPU. We use one NVIDIA GTX 580 card for GPU and one Intel X5680 CPU core for CPU

tems, which helps reduce memory contention by dividing the physical memory among the CPUs. However, achieving high memory-access performance on a NUMA system requires the application to carefully place the data to minimize the cross-domain memory access, because remote memory access tends to incur 40-50% more latency and 20-30% lower bandwidth compared to local memory access [22].

Kargus adopts a symmetric architecture per each NUMA domain. As shown in Figure 3, an IDS engine thread reads and handles the packets only from the NICs attached to the same NUMA domain, and offloads pattern matching workloads to the in-node GPU device. This eliminates expensive NUMA domain crossing of the packet workloads, which allows s linear performance growth with the number of NUMA domains. We have also removed all global variables that are shared by the IDS engine threads, and aligned the packet metadata boundary to the 64B CPU cache line. Figure 6 shows the effect of removing shared global variables, which gives 1.5 to 2.7x speedups with 64B packets. As with function call batching, we can see that the remote memory access overhead is significant for small packet sizes.

4.5 GPU-based Pattern Matching

Pattern matching typically consumes the most processing cycles in an IDS. It is heavily memory-intensive since every byte of the received packets is scanned at least once. For example, one CPU core in our platform produces only about 2 Gbps for multi-string pattern matching even if it is fully devoted to it. With 12 CPU cores, it is challenging to achieve more than 20 Gbps since they have to spend cycles on other tasks such as packet capture and flow management as well as pattern matching.

GPU is known to be well-suited for compute or memory-intensive workloads with a large array of processing cores and high memory bandwidth. It is particularly effective in processing multiple flows/packets at a batch due to its SIMT execution model. To exploit the massively-parallel execution capacity, Kargus collects incoming packets in parallel from multiple IDS engine threads, and offloads them to GPU together via a single GPU-dispatcher thread per each NUMA domain. The GPU-dispatcher thread is responsible for handling all GPU workloads from the CPU in the same NUMA domain, and communicates only with the in-node GPU device.

We implement GPU-based pattern matching in Kargus as follows. For Aho-Corasick multi-string pattern matching, we port the DFA tables in each port group to GPU and implement state transition functions as simple array operations on the table. At initialization, Kargus stores the DFA tables at the global memory of a GPU device and shares them with all pattern matching GPU threads. Each GPU thread takes one packet at a time, and scans each byte in the payload by following the DFA transition function. Since accessing the global memory is expensive, we store the packet payloads at *texture memory*, which prefetches and caches the sequentially-accessed content from the global memory. This improves the pattern matching performance by about 20%. Also, a GPU thread fetches 16 bytes of the payload at a time (via the `uint4` data type supported by the NVIDIA CUDA library), as suggested by [47]. For PCRE pattern matching, we take the standard approach of converting each PCRE to a DFA table. We first transform each PCRE to an NFA form by Thompson's algorithm [27] and apply the subset construction algorithm to convert an NFA to a DFA table. We find that there exist complex PCREs that generate too many DFA states (called *state explosion*), but over 90% of the Snort-2.9.2.1 PCREs can be converted to a table with less than 5,000 states. In Kargus, GPU handles PCREs with less than 5,000 states while CPU handles the rest. For both GPU pattern matching, we use the *concurrent copy and execution* technique [22, 26, 47] that overlaps the DMA data transfers between the host and device memory with GPU code execution itself. We use multiple CUDA streams in the GPU-interfacing thread to implement this.

Figure 7 compares the performance of multi-string pattern matching for a CPU core and a GPU device, with 1518B packets. The

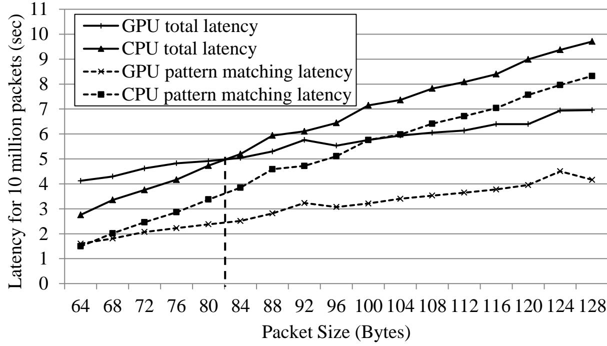

Figure 9: Latency for total packet processing and for only pattern matching with CPU and GPU

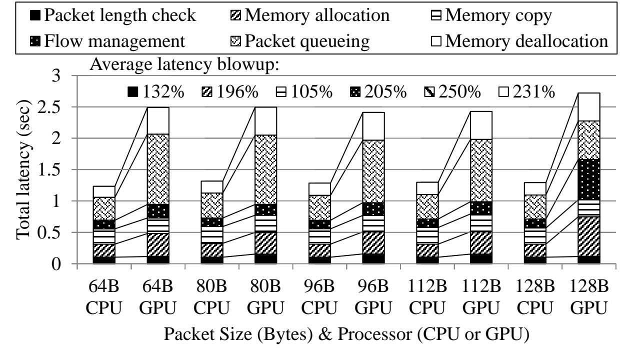

Figure 10: Breakdown of the packet processing latency

GPU performance increases as the batch size grows since more GPU processing cores are utilized with a larger batch size. We find that reducing the DFA table entry to 2 bytes from 4 bytes almost doubles the throughput, which confirms that the GPU memory access is the main bottleneck. The GPU performance is saturated at 8,192 packets with 39.1 Gbps, which would require 18.2 CPU cores at 3.33 GHz to reach that. Figure 8 shows the performance of PCRE matching. The performance of GPU saturates at 1,024 packets with 8.9 Gbps, showing a factor of 17.1 improvement over a CPU core. Interestingly, the performance of GPU PCRE matching is not comparable to that of multi-string matching even if they implement the same DFA transitions. This is because PCRE matching requires each packet to follow a different PCRE DFA table in most cases and the similar performance degradation is seen with the CPU implementation. In contrast, multi-string matching shares the same Aho-Corasick DFA table across all threads in most cases, benefiting from lower memory contention.

5. LOAD BALANCING

Kargus employs load balancing for two different cases. First, it is applied to incoming flows across CPU cores with help of RSS, where the incoming packets are classified into different NIC queues by hashing the five tuples (src/dst IPs, port numbers, and protocol) and affinitizing each NIC queue to a CPU core. Second, load balancing between CPU and GPU is enforced to utilize extra processing capacity of GPU when Kargus cannot handle the high input rates with CPU alone. We have elaborated the load balancing across CPU cores in Section 4.2, so we focus only on the load balancing between CPU and GPU here.

5.1 Only CPU for Small Packets

Our experiments show that unconditional GPU-offloading is not always beneficial for the following reasons. First, employing GPU requires the additional usage of CPU and memory resources. GPU offloading necessitates metadata preparation for pattern matching and copying the packet payload into a page-locked memory that is DMA'ed to GPU memory. Preparing the metadata and payload copying consumes CPU cycles, and DMA operations would increase memory access contention. The increased memory contention interferes with other memory-intensive CPU tasks, e.g., packet acquisition from NICs, preprocessing, and CPU-based pattern matching. Second, GPU typically consumes more energy than CPU. For example, one NVIDIA GTX 580 device consumes 244 Watts [34] while an Intel Xeon 5680 CPU consumes 130 Watts [24] at peak. This implies that we can save power if we use only CPU for the input rates that can be handled by CPU alone.

GPU-offloading sacrifices extra CPU cycles and memory access that could be used to do pattern matching with CPU otherwise. It implies that there exists a threshold of a packet size under which utilizing only CPU is always beneficial (irrespective of the input load). For example, CPU produces a better multi-string matching performance for any packets smaller than 82B on our platform. One might attempt to determine the threshold by comparing GPU and CPU pattern matching latencies, but it turns out that a naïve approach will not successfully find the accurate threshold.

Figure 9 shows the latencies for various packet sizes for multi-string pattern matching. Given a packet size, we measure the latency for 10 million packets. GPU pattern matching latency refers to the entire time for GPU offloading. It includes the time for preparing metadata, copying the payloads to a page-locked memory, DMA copying to GPU, executing the GPU kernel, and receiving the results from GPU. CPU pattern matching time refers to the time to run the Aho-Corasick algorithm in CPU. These two lines cross only at 64B, and one might think that offloading packets of any size to GPU is always beneficial. However, total latencies suggest that offloading any packets smaller than 82B is detrimental. Total latency includes the time for the common CPU operations such as flow management, memory allocation and copying of received packets, etc.

To figure out why we have the discrepancy, we analyze the time for the common CPU operations in more detail. Figure 10 shows the latency breakdown for the common operations. It shows that when GPU is used for pattern matching, the latencies for the common operations increase by a factor of 1.8 to 2.0 compared with those of the CPU-only version. This is because the host memory access contention increases, where GPU-offloading would involve constant DMA copying between the host memory and the GPU memory.

5.2 Dynamic GPU Offloading

We now move on to the case when the packet size is larger than the threshold, where dynamic GPU offloading becomes important due to energy efficiency. The goal of our algorithm is to use GPU only if CPU is unable to handle the input rates, and use CPU for low latency and better power utilization otherwise.

We introduce the notion of *workload queues*, each of which is associated with one IDS engine thread. The state of a workload queue evolves over cycles, where during each cycle the following operations are performed: (a) Each IDS engine thread reads a batch of packets from its RX queues (the number of packets read at a time is called RX batch size), (b) enqueues them into the associated work-

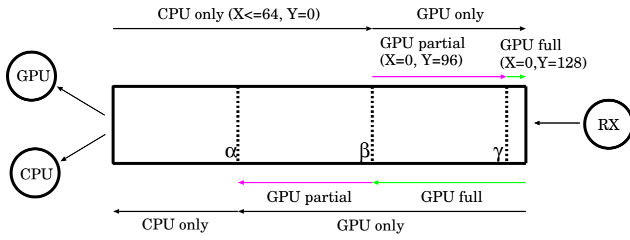

Figure 11: Dynamic GPU offloading algorithm

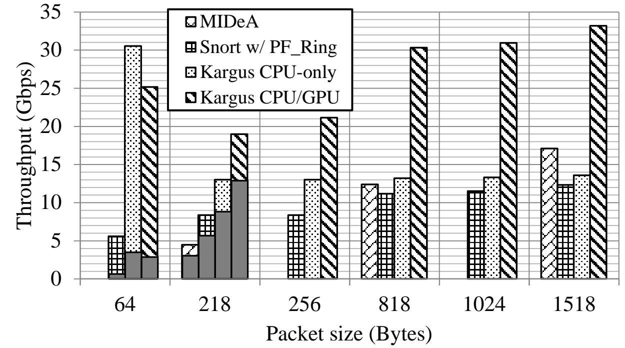

Figure 12: IDS throughputs for innocent content at a 40 Gbps input rate

load queue, and (c) dequeues a unit of packets from the workload queue for pattern matching. The maximum RX batch size is set to 128 packets in our implementation to maximize the CPU cache efficiency. For a larger value than 128, the CPU cache would overflow and often lead to noticeable performance degradation. Note that the numbers of enqueued and dequeued packets at a time can differ, and consequently the input and output rates of the queue differ over cycles. Especially, the output rate is controlled by the number of dequeued packets at a time in our algorithm.

As depicted in Figure 11, at each cycle, the algorithm detects the overloading status of a CPU core by examining the queue size based on three thresholds, α, β, and γ. Given a cycle, either CPU or GPU (but not both) is used for pattern matching depending on the status of the work queue. X and Y refer to the numbers of packets dequeued at one cycle for CPU and GPU pattern matching, respectively. First, when the queue is in the interval $[0,\beta]$, only CPU is utilized for pattern matching, where the engine thread dequeues 64 packets at maximum, a much lower output rate than the maximum input rate. The rationale is that CPU can handle low input rates whose average RX batch size is smaller than 64 packets over time. We have β to absorb temporary fluctuation in the input rates. The queue size reaching β implies that CPU alone is not able to satisfy the input rates. Spending CPU cycles on pattern matching in this state is detrimental since it should spend more cycles to read the packets from NICs to avoid packet drops. Thus, we start using GPU, but we set Y (96) to slightly smaller than the maximum RX batch size (128). If we set Y to 128, the queue will be drained quickly if the average RX batch size is close but smaller than 128. This would create frequent oscillations between the two modes even if the input rates cannot be handled by CPU. When the input rate stays very high for an extended period, the queue size eventually reaches γ, and we set Y to the maximum RX batch size. If the dequeued packets are fully processed by GPU, the queue would start to decrease and reach β, and we set Y to 96 again. If the input rate is close to 128, we keep on using GPU while the queue size is in the interval $[\beta,\gamma]$. If the input rate goes down, so that the queue size reaches α, then pattern matching only on CPU is conducted.

6. IMPLEMENTATION

We have implemented Kargus using Snort-2.9.2.1 as our code base. Kargus reuses the existing Snort code such as Aho-Corasick and PCRE matching and rule option evaluation modules, and reads unmodified Snort attack signature rule files. We have modified single-process Snort to launch a thread for each IDS engine, and have replaced Snort's packet acquisition model with a PSIO-based callback function. Both IDS engine and GPU-dispatcher threads can be configured with the numbers of available CPU cores and GPU devices on the platform, by a LUA [10] start-up script supported by SnortSP. We have also applied function call batching by rendering each function to take a batch of packets as an argument.

The most invasive modification has been made to the Snort's flow metadata structure, where we reduce its size from 2,976 to 192 bytes by removing most redundant fields that include (1) header values that can be retrieved from the raw packet, and (2) fields that are used to gather bookkeeping statistics. We align the size to the CPU cache line boundary and eliminate the chance of false sharing with other CPU cores. We do this because we find that the large flow metadata reduces the CPU cache efficiency. We implement our own flow management module that detects attack patterns for merged TCP segments that are delivered out of order. Currently, Kargus provides analyzing support for HTTP and TCP rules, and integration with other Snort modules is in progress.

To test the Kargus' performance under various scenarios, we have developed a simple IDS workload generator that produces TCP packets at any transmission rate up to 40 Gbps regardless of the packet size. The current version also extracts the `content` and `pcre` options from the Snort rules and generates an attack packet that contains the signature. We use PSIO for efficient packet transmission and implement the pattern string generation based on the content and PCRE options. Kargus consists of 72,221 lines of code in total: 14,004 lines of our code and 58,217 lines of ported Snort code. Our code includes 4,131 lines of packet generator code, 2,940 lines of Snort-rule-to-DFA-table converter code for GPU, and 530 lines of GPU kernel code for PCRE/multi-string matching.

7. EVALUATION

We evaluate the performance of Kargus under various conditions using the synthetic TCP traffic generated by our workload generator. Also, we measure the performance with the real HTTP traffic traces collected at one LTE backbone link at a large mobile ISP in South Korea. Finally, we gauge the power saving from the CPU/GPU load balancing at various input rates.

7.1 Synthetic TCP Workloads

Figure 12 shows the throughputs of Kargus over different packet sizes with synthetic TCP workloads. We use the same platforms mentioned in Section 3, and generate TCP packets with a destination port 80 that contain random content without any attack signatures. This experiment would measure the performance of packet reception, preprocessing, and multi-string pattern matching in an signature-based IDS. We compare the performances of Kargus, Kargus-CPU-Only, Snort-PF_RING, and MIDeA with all HTTP rules in Snort-2.9.2.1. Kargus-CPU-Only uses all 12 CPU cores for IDS engine threads while Kargus uses 10 cores for the engine threads and the

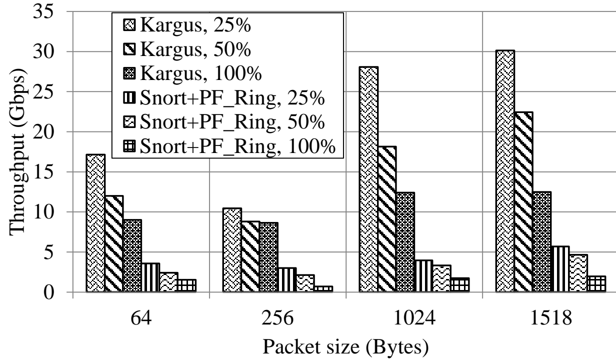

Figure 13: IDS throughputs for malicious content at a 40 Gbps input rate

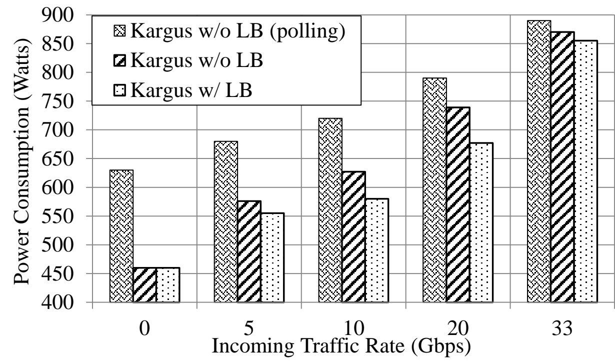

Figure 14: Power consumption by Kargus when 1518B packets are monitored at various input rates

remaining two cores for GPU-dispatcher threads. We use the default setup for Snort, but disable the flow management module since the incoming TCP packets have random sequence numbers without connection establishment. For MIDeA, we draw the numbers for a similar test from their paper, and normalize them to reflect the available CPU cycles of our platform and to include the Ethernet frame overhead and header size. We use 128 as the maximum RX batch size, and offload 1K packets per engine in a batch to GPU for pattern matching.

Kargus shows 19.0 to 33.2 Gbps over the packet sizes. It improves the performance of MIDeA by a factor of 1.9 to 4.3. Compared with PF_RING-based Snort, it shows 2.7x to 4.5x performance improvement. Except for 64B, Kargus presents increasing performance as the packet size grows. The performance of 64B packets is dominated by non-payload overheads since the analyzed TCP payload is only 6 bytes per packet while the overhead is 78 bytes (40B for IP/TCP headers, 18B for Ethernet header and CRC and 20B frame overhead). The shaded portion at 64B represents the performance of analyzed TCP payloads, and we can see that the portion increases at 128B. The reason why Kargus shows lower performance than the CPU-only version at 64B is that the latter uses extra two cores for pattern matching. For the packet sizes larger than 64B, the throughput of Kargus-CPU-Only stays at 13 Gbps, which is the maximum performance achievable by only CPU cores. In comparison, Kargus shows 1.4x to 2.4x performance improvement with GPU offloading.

The reasons why Kargus achieves better performance are as follows. First, it uses an improved packet capture library that does not incur polling overhead and the remaining CPU cycles can be converted for other tasks that gear towards performance improvement. Second, it effectively converts the performance improvements from both GPUs without data crossing overhead over different NUMA domains. Also, the memory footprint of Kargus is much smaller with its multi-threaded model than MIDeA with a multi-process model. Third, it extensively applies function call batching through the packet analysis path and fixes some of core data structures to be more CPU-cache friendly.

Figure 13 show the throughputs of Kargus when we control the portion of attack packets by 25%, 50%, and 100%, for various packet sizes. In this case, the PCRE rules are triggered as part of rule option evaluation, and it significantly increases the pattern matching overheads. As expected, the performance degrades as the packet size becomes smaller as well as the attack portion increases. For example, for 64B packet size, the performance ranges from 9.0 Gbps to 17.1 Gbps, depending on the attack portion. Even when we have 100% attack traffic, Kargus shows over 10 Gbps when the packet size is larger than 1024 B. It shows 12.4 Gbps and 12.5 Gbps at 1024B and 1518B, respectively.

We also measure the per-packet latency in Kargus. At 10 Gbps input rate, Kargus shows 13 microseconds from packet reception to multi-string pattern matching for 1518B packets, but it increases to 640 microseconds at 33 Gbps input rate. This is because Kargus uses only CPU at 10 Gbps input rate, but it incurs a buffering overhead for GPU offloading at 33 Gbps input rate.

Total number of packets	$109,263,657$
Total number of TCP sessions	$845,687$
Average packet size	808 bytes
Performance of Kargus	25.2 Gbps

Table 2: Statistics on the real traffic traces

7.2 Performance Under Real Traffic

We measure the performance of Kargus by replaying the real traffic traces gathered at a 10 Gbps LTE backbone link at one of the largest mobile ISPs in South Korea. We have three traces with different time frames, and each trace covers 30 minutes to 1 hour of full IPv4 packets captured at the link. The total number of packets we replay is 109.3 million with 845K TCP sessions, which takes up 84 GB of physical memory. Before replaying the traces, we load the entire packets to the memory and make sure that the packets in the same TCP connection are transmitted to the same 10 Gbps NIC port in the Kargus machine while evenly distributing the flows among all four NIC interfaces.

To gauge the peak performance of Kargus, we have our workload generator replay the packets as fast as possible, generating the traffic at the speed of 40 Gbps. Kargus achieves 25.2 Gbps, which is 17% lower than that of the synthetic workload with 800B packets in Figure 12. Our profiling results reveal that the performance degradation is mainly attributed to the inflated flow management module. That is, additional memory allocation, packet copying, and multi-string pattern matching in the assembled flow buffer to detect attack signatures that span multiple packets in the same flow consume extra CPU cycles. We suspected that GPU kernel execution with diverse packet sizes would reduce the throughput since GPU threads in a CUDA warp have to wait until the execution of a thread with the largest packet size. But we find that GPU pattern matching increases the latency only by 1.8% with the real traffic, implying

that the overall performance degradation due to GPU is marginal even with diverse packet sizes.

7.3 Power Consumption

Figure 14 shows the energy efficiency of Kargus contributed by dynamic GPU offloading. We plot the power consumption for different incoming traffic rates. The packet size is set to 1518B and we tested Kargus with dynamic GPU-offloading, Kargus without load balancing (i.e., unconditional GPU offloading, Kargus w/o LB) that either uses polling or not. Kargus with polling and no load balancing is a version of Kargus which polls the NIC queues for packet acquisition. We show the polling version to understand the power usage of the IDS that uses a I/O library like PF_RING.

Overall, Kargus with polling consumes the largest power because it uses full CPU cycles regardless of input rates. The gaps from other schemes decrease as the input rates increase, as expected. The power efficiency gain of Kargus over Kargus w/o LB increases with decreasing input rates, because for the high input rates, Kargus offloads most of pattern matching workloads to GPU to maximize the analysis throughput. For example, the energy gain for 10 and 20 Gbps is about 14% and 19% compared with the polling version of Kargus, 7.5% to 8.7% compared with Kargus w/o LB.

8. RELATED WORK

We discuss previous works that are related to Kargus, but due to the large volume, we do not attempt to cover a complete set.

Pattern matching on specialized hardware: There are many works that improve the performance of an IDS through specialized hardware such as FPGAs, network processors, ASICs, and TCAMs. Clark *et al.* store the entire 1,500 rules of Snort 2.0 into a single one-million-gate FPGA [18], and extend the work by pre-filtering packets with a network processor before pattern matching on FPGA for higher performance [17]. Baker *et al.* demonstrate an FPGA implementation of the Knuth-Morris-Pratt algorithm for string matching [12], and Lee *et al.* extract the common prefixes that are shared by many patterns to efficiently utilize the limited FPGA resources, and use multiple FPGAs in parallel for higher performance [29]. Mitra *et al.* use direct compilation of PCREs into VHDL blocks and onto the FPGA to enhance the performance while maintaining the compatibility with Snort rule sets [32]. Tan *et al.* implement a high-throughput Aho-Corasick engine on an ASIC [44]. Similar related works [15, 16] implement hardware-based string matching co-processors for Snort running at the speed of 7 Gbps. Some works employ TCAMs for parallel pattern matching. TCAM compares a given input string against all the occupied pattern entries in parallel, and returns the matched entry. In [49], only string matching is performed in TCAM, which is extended to regular expressions in [30]. While TCAM enables a memory lookup to take constant time for any input, regardless of the number of entries, it has some drawbacks: (i) the limited size of TCAM restricts the maximum DFA table size, (ii) TCAM's performance becomes restricted when packet capturing and preprocessing consumes a large portion of CPU cycles as in high-speed networks, because TCAM does not provide additional computation resources as FPGA or ASIC processors do, and (iii) it is challenging to avoid contentions in a multicore system, which could degrade overall system utilization and performance. In comparison to these hardware approaches, Kargus shows that one can build a multi-10Gbps software IDS even on commodity hardware if we exploit parallelism and batching with multiple CPU cores and GPUs.

Pattern matching algorithm enhancement: Since regular expressions are the de-facto language that specifies modern attack signatures, many algorithms have tackled improving the performance of regular expression evaluation. Most works focus on addressing DFA state explosion as well as reducing the number of state transitions. D^2FA [41] compresses the number of DFA state transitions by postponing input processing at certain DFA states and removing redundant state transitions. Hybrid-FA [13] proposes a hybrid DFA that combines the desirable properties of NFA and DFA for small states and fast transitions. XFA [35] reduces the number of DFA states by running per-state computation with a small set of variables that characterize the state. The evaluation of XFA on a G80 GPU is promising [36], but direct performance comparison with Kargus is difficult due to hardware and rule set difference. In comparison, Kargus adopts the Snort approach that uses Aho-Corasick DFAs for multi-string pattern matching, which acts as a first-pass filter, and maintains a separate DFA for each PCRE. This significantly reduces the need for PCRE matching as most innocent flows bypass this stage. While this approach requires multiple payload scans for attack attempts, it allows simpler GPU implementations with high throughputs, and it becomes easy to evaluate advanced rule options.

Clustered IDS engines: To cope with the growing network link capacities, IDSes (e.g. Bro [46]) are now being deployed in a cluster. In a typical setup, one server acts as a front-end node that captures ingress traffic while the remainders serve as distributed packet processing nodes. While clustering provides scalable performance, it may incur additional processing overheads due to inter-node communication and lock contentions between shared packet queues across multiple servers. Another central issue lies in load balancing among the back-end IDS nodes. Kruegel *et al.* adopt a static load balancing algorithm to scatter packets across IDS nodes based on flow properties [28]. The SPANIDS [43] load balancer employs multiple levels of hashing and incorporates feedbacks from the IDS nodes to distribute packets over the servers using FPGA-based switches. We believe that Kargus can be easily extended to support clustering.

Software-based IDS on commodity hardware: Modern multicore architectures have enabled development of high-performance software-based IDSes. SnortSP [8], multi-core Suricata [9] are some of the initial attempts that utilize multiple CPU cores for pattern matching. Para-Snort [14] extends SnortSP for better utilization of the multicore architecture. It has a pipelining structure, where a data source module and processing modules are dedicated to each CPU core. While the design is intuitional, it presents a few performance problems as mentioned earlier: (i) it could incur considerable CPU cache bouncing due to frequent traversal of packets among the cores and (ii) it leads to unbalanced CPU utilization due to different processing speeds across the modules.

GPUs have been used to improve the performance of IDS pattern matching. Gnort [21] is a modification of Snort to utilize GPU for string matching using Aho-Corasick algorithm. Huang *et al.* modify the Wu-Manber algorithm in order to run string matching on GPU [23]. However, it is reported that the Aho-Corasick algorithm outperforms the Wu-Manber algorithm in terms of worst-case performance [33]. MIDeA [47] is the closest work to Kargus. It improves the packet reception performance with PF_RING on a 10 Gbps NIC and scales the performance of Aho-Corasick multi-string pattern matching with GPUs. While MIDeA reports 70 Gbps multi-string pattern matching performance with two GPUs, their overall IDS performance numbers are still limited to less than 10 Gbps. To further scale the overall performance, Kargus makes many different design choices. It takes the multi-thread model instead of the multi-process model, which eliminates GPU memory waste for redundant DFA tables, and could allow load balancing among CPU cores. We find function call batching and NUMA-aware packet processing greatly help in processing small packets, and present power-efficient load

balancing among CPU cores and GPUs. Kargus also implements PCRE matching and compares the IDS performances with varying levels of attack traffic.

Kargus benefits from recent works on high-performance software routers. RouteBricks [20] and PacketShader [22] exploit multiple CPU cores for parallel execution. PacketShader scales the router performance with GPUs, and applies batching to packet I/O and router applications. Kargus takes the similar approach, but it applies batching and parallelism in the context of an IDS, and effectively uses computation resources for power efficiency.

Packet acquisition enhancement: Fast packet acquisition is a crucial component in high-performance IDSes. Some of recent research efforts include PF_RING [19], PSIO [22], and netmap [38]. PF_RING exports a shared memory of packet rings in kernel to user-level applications, and its direct NIC access (DNA) extension allows the NIC to move packets to ring buffers directly without CPU intervention. However, each thread can bind on no more than one RX queue with DNA. Libzero for DNA [2] distributes the packets read from one RX queue to other applications with the zero copy mechanism. In comparison to PF_RING, PSIO allows threads to read packets directly from multiple RX queues in different NICs, thus it does not need to distribute the packets among the threads. It also bypasses heavy kernel structures, and provides a blocking packet input mode. Netmap is similar to PSIO in terms of techniques (linear, fixed size packet buffers, multiple hardware queues, lightweight metadata), and it supports traditional event system calls such as `select()`/`poll()`.

9. CONCLUSION

We have presented Kargus, a high-performance software IDS on a commodity server machine. Kargus dramatically improves the performance by realizing two key principles: *batching* and *parallelism*. We have shown that batching in receiving packets allows a high input rate by reducing the per-packet CPU cycle and memory bandwidth cost. Function call batching and pattern matching with GPUs also lead to efficient usage of the computing resources, minimizing the waste in processing cycles. Kargus exploits high parallelism in modern computing hardware by efficiently balancing the load of flows across multiple CPU cores and by employing a large array of GPU processing cores. The end result is impressive, summarized as: Kargus achieves 33 Gbps for normal traffic, and 9 to 10 Gbps even when all traffic is malicious.

Our analysis of the cost for GPU offloading suggests that one needs to be careful about the usage of GPU for pattern matching. We find that blind offloading often produces a poor performance if the offloading cost exceeds the CPU cycles required for the workload, which frequently occurs for small size packets. Even when the offloading cost is small for large size packets, it is beneficial to use CPU due to power saving and latency reduction when the workload level is satisfied by the CPU capacity. We have developed a simple, yet energy-efficient GPU offloading algorithm and have shown that it nicely adapts to the input rate, and effectively reduces the power consumption accordingly.

ACKNOWLEDGEMENT

We thank anonymous CCS'12 reviewers and our shepherd, Salvatore Stolfo, for their insightful comments and feedback on the draft version. We also thank Yongdae Kim for discussion on potential attacks on RSS, and Byungchul Bae for his continued support for the project. We thank Bumsoo Kang for his help on the early development of the IDS packet generator. This project is supported in part by the National Security Research Institute of Korea (NSRI) grants #N03100069 and #N02120042, and the National Research Foundation of Korea (NRF) grant #2012R1A1A1015222.

10. REFERENCES

[1] Intelligent Networks Powered by Cavium Octeon and Nitrox Processors: IDS/IPS Software Toolkit. `http://www.cavium.com/css_ids_ips_stk.html`.

[2] Libzero for DNA. `http://www.ntop.org/products/pf_ring/libzero-for-dna/`.

[3] McAfee Network Security Platform. `http://www.mcafee.com/us/products/network-security-platform.aspx`.

[4] More about Suricata multithread performance. `https://home.regit.org/2011/02/more-about-suricata-multithread-performance/`.

[5] Optimizing Suricata on multicore CPUs. `http://home.regit.org/?p=438`.

[6] PCRE (Perl Compatible Regular Expressions). `http://pcre.org`.

[7] Single Threaded Data Processing Pipelines and the Intel Architecture. `http://vrt-blog.snort.org/2010/06/single-threaded-data-processing.html`.

[8] SnortSP (Security Platform). `http://www.snort.org/snort-downloads/snortsp/`.

[9] Suricata Intrusion Detection System. `http://www.openinfosecfoundation.org/index.php/download-suricata`.

[10] The Programming Language Lua. `http://www.lua.org/`.

[11] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18:333–340, June 1975.

[12] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2004.

[13] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of CoNEXT*, 2007.

[14] X. Chen, Y. Wu, L. Xu, Y. Xue, and J. Li. Para-Snort: A Multi-thread Snort on Multi-core IA Platform. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, 2009.

[15] Y. H. Cho and W. H. Mangione-Smith. A Pattern Matching Co-processor for Network Security. In *Proceedings of the 42nd annual Design Automation Conference (DAC)*, 2005.

[16] Y. H. Cho, S. Navab, and W. H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, 2002.

[17] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Koné, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the Workshop on Network Processors and Applications (NP3)*, 2004.

[18] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *In Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[19] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of the International System Administration and Network Engineering Conference (SANE)*, 2004.

[20] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[21] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[22] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated Software Router. In *Proceedings of ACM SIGCOMM*, 2010.

[23] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications Workshops*, 2008.

[24] Intel, Inc. Intel Xeon Processor X5680 Specifications. `http://ark.intel.com/products/47916/Intel-Xeon-Processor-X5680-(12M-Cache-3_33-GHz-6_40-GTs-Intel-QPI)`.

[25] IPFW: FreeBSD Handbook. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-ipfw.html`.

[26] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[27] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[28] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2002.

[29] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A High Performance NIDS using FPGA-based Regular Expression Matching. In *Proceedings of the ACM symposium on Applied computing*, 2007.

[30] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the USENIX Security Symposium*, 2010.

[31] Microsoft. Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS. In *WinHEC (White paper)*, 2004.

[32] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2007.

[33] M. Norton. Optimizing Pattern Matching for Intrusion Detection, 2004. `http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf`.

[34] NVIDIA, Inc. GeForce GTX 580 Specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications`.

[35] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakload)*, 2008.

[36] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[37] Ricciulli, Livio and Covel, Timothy. Inline Snort multiprocessing with PF_RING. `http://www.snort.org/assets/186/PF_RING_Snort_Inline_Instructions.pdf`.

[38] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[39] Robert S. Boyer, and J Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20:762–772, October 1977.

[40] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Systems Administration Conference (LISA)*, 1999.

[41] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proceedings of ACM SIGCOMM*, 2006.

[42] San Wu, and Udi Manber. A Fast Algorithm for Multi-Pattern Searching. Technical report, 1994.

[43] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 315–322, 2005.

[44] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[45] TCPDUMP. `http://www.tcpdump.org/`.

[46] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 107–126, 2007.

[47] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[48] S. Woo and K. Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Technical report, KAIST, 2012. `http://www.ndsl.kaist.edu/~shinae/papers/TR-symRSS.pdf`.

[49] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)*, 2004.