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

M. Asim Jamshed*, Jihyung Lee†, Sangwoo Moon†, Insu Yun*, Deokjin Kim‡, Sungryoul Lee‡, Yung Yi†, KyoungSoo Park*

* Networked & Distributed Computing Systems Lab, KAIST
† Laboratory of Network Architecture Design & Analysis, KAIST
‡ Cyber R&D Division, NSRI
Network Intrusion Detection Systems (NIDS)

- Detect known malicious activities
  - Port scans, SQL injections, buffer overflows, etc.
- Deep packet inspection
  - Detect malicious signatures (rules) in each packet
- Desirable features
  - High performance (> 10Gbps) with precision
  - Easy maintenance
    - Frequent ruleset updates
Hardware vs. Software

• H/W-based NIDS
  – Specialized hardware
    • ASIC, TCAM, etc.
  – High performance
  – Expensive
    • Annual servicing costs
  – Low flexibility

• S/W-based NIDS
  – Commodity machines
  – High flexibility
  – Low performance
    • DDoS/packet drops

IDS/IPS Sensors (10s of Gbps)
~ US$ 20,000 - 60,000

IDS/IPS M8000 (10s of Gbps)
~ US$ 10,000 - 24,000

Open-source S/W
≤ ~2 Gbps
Goals

- High performance

• S/W-based NIDS
  - Commodity machines
  - High flexibility
Typical Signature-based NIDS Architecture

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
    (msg:"possible attack attempt BACKDOOR optix runtime detection";
     content:"/whitepages/page_me/100.html";
     pcre:"/body=\x2521\x2521\x2521Optix\s+Pro\s+v\d+\x252E\d+\S+sErver\s+Online\x2521\x2521\x2521/"
```

**Bottlenecks**

* PCRE: Perl Compatible Regular Expression
Contributions

**Goal**
A highly-scalable software-based NIDS for high-speed network

**Slow software NIDS**
- Inefficient packet acquisition
- Expensive string & PCRE pattern matching

**Fast software NIDS**
- Multi-core packet acquisition
- Parallel processing & GPU offloading

**Bottlenecks**

**Solutions**

**Outcome**
Fastest S/W signature-based IDS: **33 Gbps**
100% malicious traffic: **10 Gbps**
Real network traffic: **~24 Gbps**
Challenge 1: Packet Acquisition

- Default packet module: Packet CAPture (PCAP) library
  - Unsuitable for multi-core environment
  - Low performing
  - More power consumption
- Multi-core packet capture library is required

Packet RX bandwidth*  
0.4-6.7 Gbps

CPU utilization  
100 %

* Intel Xeon X5680, 3.33 GHz, 12 MB L3 Cache
Solution: PacketShader I/O

- PacketShader I/O
  - Uniformly distributes packets based on flow info by RSS hashing
  - Source/destination IP addresses, port numbers, protocol-id
  - 1 core can read packets from RSS queues of multiple NICs
  - Reads packets in batches (32 ~ 4096)
- Symmetric Receive-Side Scaling (RSS)
  - Passes packets of 1 connection to the same queue

Packet RX bandwidth
- 0.4 ~ 6.7 Gbps
- 40 Gbps

CPU utilization
- 100%
- 16-29%

* S. Han et al., “PacketShader: a GPU-accelerated software router”, ACM SIGCOMM 2010
Challenge 2: Pattern Matching

- CPU intensive tasks for serial packet scanning

- Major bottlenecks
  - Multi-string matching (Aho-Corasick phase)
  - PCRE evaluation (if ‘pcre’ rule option exists in rule)

- On an Intel Xeon X5680, 3.33 GHz, 12 MB L3 Cache
  - Aho-Corasick analyzing bandwidth per core: 2.15 Gbps
  - PCRE analyzing bandwidth per core: 0.52 Gbps
Solution: GPU for Pattern Matching

• GPUs
  – Containing 100s of SIMD processors
    • 512 cores for NVIDIA GTX 580
  – Ideal for parallel data processing without branches
• DFA-based pattern matching on GPUs
  – Multi-string matching using Aho-Corasick algorithm
  – PCRE matching
• Pipelined execution in CPU/GPU
  – Concurrent copy and execution

Aho-Corasick bandwidth
2.15 Gbps
39 Gbps

PCRE bandwidth
0.52 Gbps
8.9 Gbps
## Optimization 1: IDS Architecture

- How to best utilize the multi-core architecture?
- Pattern matching is the eventual bottleneck

<table>
<thead>
<tr>
<th>Function</th>
<th>Time %</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>acsmSearchSparseDFA_Full</td>
<td>51.56</td>
<td>multi-string matching</td>
</tr>
<tr>
<td>List_GetNextState</td>
<td>13.91</td>
<td>multi-string matching</td>
</tr>
<tr>
<td>mSearch</td>
<td>9.18</td>
<td>multi-string matching</td>
</tr>
<tr>
<td>in_chksum_tcp</td>
<td>2.63</td>
<td>preprocessing</td>
</tr>
</tbody>
</table>

* GNU gprof profiling results

- Run entire engine on each core
Solution: Single-process Multi-thread

- Runs multiple IDS engine threads & GPU dispatcher threads concurrently
  - Shared address space
  - Less GPU memory consumption
  - Higher GPU utilization & shorter service latency

**GPU memory usage:** 1/6
Architecture

- Non Uniform Memory Access (NUMA)-aware
- Core framework as deployed in dual hexa-core system
- Can be configured to various NUMA set-ups accordingly

▲ Kargus configuration on a dual NUMA hexanode machine having 4 NICs, and 2 GPUs
Optimization 2: GPU Usage

• Caveats
  – Long per-packet processing latency:
    • Buffering in GPU dispatcher
  – More power consumption
    • NVIDIA GTX 580: 512 cores

• Use:
  – CPU when ingress rate is low (idle GPU)
  – GPU when ingress rate is high
Solution: Dynamic Load Balancing

- Load balancing between CPU & GPU
  - Reads packets from NIC queues per cycle
  - Analyzes smaller # of packets at each cycle ($a < b < c$)
  - Increases analyzing rate if queue length increases
  - Activates GPU if queue length increases

Packet latency with
- GPU: 640 μsecs
- CPU: 13 μsecs
Optimization 3: Batched Processing

- Huge per-packet processing overhead
  - > 10 million packets per second for small-sized packets at 10 Gbps
  - reduces overall processing throughput
- Function call batching
  - Reads group of packets from RX queues at once
  - Pass the batch of packets to each function

\[ \text{Decode}(p) \rightarrow \text{Preprocess}(p) \rightarrow \text{Multistring\_match}(p) \]

\[ \text{Decode}(\text{list-}p) \rightarrow \text{Preprocess}(\text{list-}p) \rightarrow \text{Multistring\_match}(\text{list-}p) \]

2X faster processing rate
Kargus Specifications

NUMA node 1:
- 12 GB DRAM (3GB x 4) - $100
- Intel X5680 3.33 GHz (hexacore) 12 MB L3 NUMA-Shared Cache - $1,210
- NVIDIA GTX 580 GPU - $370
- Intel 82599 Gigabit Ethernet Adapter (dual port) - $512

NUMA node 2:

Total Cost (incl. serverboard) = ~$7,000
IDS Benchmarking Tool

- Generates packets at line rate (40 Gbps)
  - Random TCP packets (innocent)
  - Attack packets are generated by attack rule-set
- Support packet replay using PCAP files
- Useful for performance evaluation
Kargus Performance Evaluation

• Micro-benchmarks
  – Input traffic rate: 40 Gbps
  – Evaluate Kargus (~3,000 HTTP rules) against:
    • Kargus-CPU-only (12 engines)
    • Snort with PF_RING
    • MIDeA*

• Refer to the paper for more results

Innocent Traffic Performance

- 2.7-4.5x faster than Snort
- 1.9-4.3x faster than MIDEA

Actual payload analyzing bandwidth
Malicious Traffic Performance

- 5x faster than Snort
Real Network Traffic

• Three 10Gbps LTE backbone traces of a major ISP in Korea:
  – Time duration of each trace: 30 mins ~ 1 hour
  – TCP/IPv4 traffic:
    • 84 GB of PCAP traces
    • 109.3 million packets
    • 845K TCP sessions

• Total analyzing rate: **25.2 Gbps**
  – Bottleneck: Flow Management (preprocessing)
Effects of Dynamic GPU Load Balancing

- Varying incoming traffic rates
  - Packet size = 1518 B

![Bar chart showing power consumption for varying offered incoming traffic](image)
• Software-based NIDS:
  – Based on commodity hardware
    • Competes with hardware-based counterparts
  – 5x faster than previous S/W-based NIDS
  – Power efficient
  – Cost effective

> 25 Gbps (real traffic)
> 33 Gbps (synthetic traffic)
US $~7,000/-
Thank You

fast-ids@list.ndsl.kaist.edu

https://shader.kaist.edu/kargus/
Backup Slides
Kargus vs. MIDeA

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDeA</th>
<th>KARGUS</th>
<th>OUTCOME</th>
</tr>
</thead>
</table>

### Kargus vs. MIDEA

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDEA</th>
<th>KARGUS</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet acquisition</td>
<td>PF_RING</td>
<td>PacketShader I/O</td>
<td>70% lower CPU utilization</td>
</tr>
</tbody>
</table>

**Kargus vs. MIDeA**

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDeA</th>
<th>Kargus</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet acquisition</td>
<td>PF_RING</td>
<td>PacketShader I/O</td>
<td>70% lower CPU utilization</td>
</tr>
<tr>
<td>Detection engine</td>
<td>GPU-support for Aho-Corasick</td>
<td>GPU-support for Aho-Corasick &amp; PCRE</td>
<td>65% faster detection rate</td>
</tr>
</tbody>
</table>

## Kargus vs. MIDEA

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDEA</th>
<th>KARGUS</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet acquisition</td>
<td>PF_RING</td>
<td>PacketShader I/O</td>
<td>70% lower CPU utilization</td>
</tr>
<tr>
<td>Detection engine</td>
<td>GPU-support for Aho-Corasick</td>
<td>GPU-support for Aho-Corasick &amp; PCRE</td>
<td>65% faster detection rate</td>
</tr>
<tr>
<td>Architecture</td>
<td>Process-based</td>
<td>Thread-based</td>
<td>1/6 GPU memory usage</td>
</tr>
</tbody>
</table>

## Kargus vs. MIDEA

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDEA</th>
<th>KARGUS</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet acquisition</td>
<td>PF_RING</td>
<td>PacketShader I/O</td>
<td>70% lower CPU utilization</td>
</tr>
<tr>
<td>Detection engine</td>
<td>GPU-support for Aho-Corasick</td>
<td>GPU-support for Aho-Corasick &amp; PCRE</td>
<td>65% faster detection rate</td>
</tr>
<tr>
<td>Architecture</td>
<td>Process-based</td>
<td>Thread-based</td>
<td>1/6 GPU memory usage</td>
</tr>
<tr>
<td>Batch processing</td>
<td>Batching only for detection engine (GPU)</td>
<td>Batching from packet acquisition to output</td>
<td>1.9x higher throughput</td>
</tr>
</tbody>
</table>

## Kargus vs. MIDEA

<table>
<thead>
<tr>
<th>UPDATE</th>
<th>MIDEA</th>
<th>KARGUS</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet acquisition</td>
<td>PF_RING</td>
<td>PacketShader I/O</td>
<td>70% lower CPU utilization</td>
</tr>
<tr>
<td>Detection engine</td>
<td>GPU-support for Aho-Corasick</td>
<td>GPU-support for Aho-Corasick &amp; PCRE</td>
<td>65% faster detection rate</td>
</tr>
<tr>
<td>Architecture</td>
<td>Process-based</td>
<td>Thread-based</td>
<td>1/6 GPU memory usage</td>
</tr>
<tr>
<td>Batch processing</td>
<td>Batching only for detection engine (GPU)</td>
<td>Batching from packet acquisition to output</td>
<td>1.9x higher throughput</td>
</tr>
<tr>
<td>Power-efficient</td>
<td>Always GPU (does not offload only when packet size is too small)</td>
<td>Opportunistic offloading to GPUs (Ingress traffic rate)</td>
<td>15% power saving</td>
</tr>
</tbody>
</table>

Receive-Side Scaling (RSS)

- RSS uses Toeplitz hash function (with a random secret key)

**Algorithm: RSS Hash Computation**

```plaintext
function ComputeRSSHash(Input[], RSK)
    ret = 0;
    for each bit b in Input[] do
        if b == 1 then
            ret ^= (left-most 32 bits of RSK);
        endif
        shift RSK left 1 bit position;
    end for
end function
```
Symmetric Receive-Side Scaling

• Update RSK (Shinae et al.)

<table>
<thead>
<tr>
<th>ox6d5a</th>
<th>ox56da</th>
<th>ox255b</th>
<th>ox0ec2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ox4167</td>
<td>ox253d</td>
<td>ox43a3</td>
<td>ox8fbo</td>
</tr>
<tr>
<td>oxdoca</td>
<td>ox2bcb</td>
<td>oxae7b</td>
<td>ox30b4</td>
</tr>
<tr>
<td>ox77cb</td>
<td>ox2d3a</td>
<td>ox8030</td>
<td>oxf20c</td>
</tr>
<tr>
<td>ox6a42</td>
<td>oxb73b</td>
<td>oxbeac</td>
<td>ox01fa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ox6d5a</th>
<th>ox6d5a</th>
<th>ox6d5a</th>
<th>ox6d5a</th>
</tr>
</thead>
<tbody>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
<tr>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
<td>ox6d5a</td>
</tr>
</tbody>
</table>
Why use a GPU?

Xeon X5680: 6 cores
GTX 580: 512 cores

*Slide adapted from NVIDIA CUDA C A Programming Guide Version 4.2 (Figure 1-2)
GPU Microbenchmarks – Aho-Corasick

The number of packets in a batch (pkts/batch)

Throughput (Gbps)

GPU throughput

CPU throughput

39 Gbps

2.15 Gbps
GPU Microbenchmarks – PCRE

The number of packets in a batch (pkts/batch)

Throughput (Gbps)

- GPU throughput
- CPU throughput

0.52 Gbps

8.9 Gbps
Effects of NUMA-aware Data Placement

- Use of global variables minimal
  - Avoids compulsory cache misses
  - Eliminates cross-NUMA cache bouncing effects

![Graph showing the effects of packet size on performance speedup]

- Performance Speedup
- Packet Size (Bytes)

Innocent Traffic vs. Malicious Traffic
CPU-only analysis for small-sized packets

- Offloading small-sized packets to the GPU is expensive
  - Contention across page-locked DMA accessible memory with GPU
  - GPU operational cost of packet metadata increases

![Graph showing latency versus packet size for CPU and GPU]
Challenge 1: Packet Acquisition

- Default packet module: Packet CAPture (PCAP) library
  - Unsuitable for multi-core environment
  - Low Performing

![Graph showing receiving throughput and CPU utilization vs packet size]
Solution: PacketShader* I/O

![Graph showing receiving throughput and CPU utilization for different packet sizes.

- PCAP polling
- PSIO
- PCAP polling CPU %
- PSIO CPU %

Packet Size (bytes): 64, 128, 256, 512, 1024, 1518

Receiving Throughput (Gbps): 0.4, 0.8, 1.5, 2.9, 5.0, 6.7

CPU Utilization (%): 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]