Scalable TCP Session Monitoring with Symmetric Receive-side Scaling

Shinae Woo
Department of Electrical Engineering
KAIST
shinae@ndsl.kaist.edu

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

Abstract

Receive-side scaling (RSS) is a technique that stores the arriving IP packets in the same flow into the same hardware queue of a modern network interface card (NIC). It allows scalable processing of the received packets by allowing exclusive access to the NIC queues by each CPU core. This removes the lock contention when accessing the NIC queue, and it allows concurrent access to different queues by multiple CPU cores.

One problem with the existing RSS mechanism, however, is that it maps the IP packets in the same TCP connection to different NIC queues depending on the direction of the packets. In this paper, we present symmetric RSS, which allows mapping the packets in the same TCP connection to the same NIC queue regardless of whether they are upstream or downstream. The basic idea is to manipulate the RSS seeds such that the RSS hashes of the IP packets in the reverse direction to take on the same values with those in the forward direction. Since RSS hash calculation is done in the NIC hardware, it does not consume extra CPU cycles and promises high performance.

1. INTRODUCTION

Receive-side scaling (RSS) is one of the widely-used NIC features that utilizes multiple CPU cores concurrently for packet processing in high-speed networks. It stores the arriving IP packets in the same flow into the same hardware RX queue of a modern NIC, and provides exclusive access to the queue by each CPU core on a multicore system. This allows scalable processing of the received IP packets by eliminating the lock contention by preventing the access to the same RX queue from multiple CPU cores.

RSS ensures storing of the packets in the same flow into the same reception (RX) queue by hashing the five tuples in the TCP/UDP packet header (source/destination IPs and port numbers, and the protocol). That is, the packets with the same RSS hash values will end up in the same RX queue. Typically, the CPU core responsible for the RX queue processes the packets in the received order, and one does not need to worry about out-of-order processing of the packets, which could reduce the throughput of a TCP connection unintentionally by the packet processing middleboxes or software routers in the middle of the network.

One problem with RSS, however, is that it could map the packets in the same TCP connection to different NIC RX queues depending on the direction of the packets. This is because the RSS hash of the packets in the reverse direction could take on a different value from that of the forward direction. It is especially problematic for high-performance TCP-session processing systems such as stateful firewalls, intrusion detection systems [3, 6, 7], and transparent WAN accelerators [1, 2] that need to monitor and process the TCP packets in both directions. Having two CPU cores process the TCP packets in the same

connection requires sharing the data structures across different threads/processes, which often need to be protected by a lock, undermining the original benefit of RSS.

In this paper, we present symmetric RSS, which allows mapping the packets in the same TCP connection to the same NIC queue regardless of whether they are upstream or downstream. The basic idea is to manipulate the RSS seeds such that the RSS hash of the IP packets in the reverse direction to take on the same value with that in the forward direction. Since our symmetric RSS scheme does not change the RSS algorithm itself, we can avoid re-hashing of the IP packets outside the NIC hardware. This allows high-performance TCP session processing in the same CPU core without extra CPU cycles. We also show that our symmetric RSS scheme does impair load balancing among the available CPU cores. We provide theoretical analysis as well as experiment results in this paper. We believe that our symmetric RSS scheme will be useful for any TCP packet processing systems that requires high scalability on a multicore system.

2. SYMMETRIC RECEIVE-SIDE SCALING

We first describe the original RSS algorithm and propose an efficient way to map the TCP packets in both directions into the same RX queue.

2.1 Original Receive-side Scaling Algorithm

Receive-side scaling (RSS) is a NIC feature that provides load balancing in processing received packets[5]. It first calculates the hash value based on the packet header using the Toeplitz hash function [4], and decides which RX queue to store the packet by a modular operation. RSS can be applied to any byte ranges in the IP/TCP/UDP header, but typically, it hashes the {source IP, destination IP, source port, destination port, protocol} tuple. While we focus on IPv4 packets in this paper, the idea can be easily applied to IPv6 packets.

Algorithm 1 RSS Hash Computation Algorithm

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

Algorithm 1 shows the pseudocode of the RSS hash function. The *INPUT* is the 5-tuple of IPv4 UDP or TCP packets, which is 12 bytes in total. Random Secret Key (RSK), which is 40 bytes (320 bits), is used as seed values that get mixed with the input values. Since the RSS algorithm is implemented in recent NIC hardware for fast processing, modification of the RSS algorithm to support symmetric mapping is difficult. However, *RSK* is part of the device driver and set into the NIC when the driver is first loaded into memory. We focus on updating *RSK* to allow symmetric mapping.

2.2 Symmetric Receive-side Scaling Algorithm

The symmetric flow mapping needs to produce the same RSS hash value even if the source and destination IPs and port numbers are reversed.

$$ComputeRSSHash(srcIP :: dstIP :: srcPort :: dstPort, RSK)$$
$$= ComputeRSSHash(dstIP :: srcIP :: dstPort :: srcPort, RSK) \quad (1)$$

Equation (1) shows the basic condition of the symmetric RSS queue mapping. We need to find the condition of *RSK* that satisfies the equation.

RULE 1. *During calculation of n^{th} bit of INPUT, only the n^{th} to $(n+31)^{th}$ bit range of RSK is used.*

We observe that not every bit of *RSK* is used to get the hash value. Given an input bit, only 32 bits of RSK are used for the XOR operation. That is, for n-bit *INPUT*, only $(n-1)+32$ bits from *RSK* are used to produce the hash value. If *INPUT* is larger than or equal to 290 bits, *RSK* will wrap around to the first bit. For a specific bit of *INPUT*, a fixed bit range of *RSK* is used as explained in Rule 1.

INPUT	Length	Input Bit Range	RSK Bit Range
Source IP	32 bit	1 - 32	1 - 63
Destination IP	32 bit	33 - 64	33 -95
Source port	16 bit	65 - 80	65 -111
Destination port	16 bit	81 - 96	81 - 127

Table 1: *RSK* bit ranges used in the calculation of IPv4/TCP packet fields

Using Rule 1, we can derive the bit ranges of *RSK* used in the calculation of the hash value for an IPv4/TCP (or UDP) packet. For example, the first byte (first eight bits) of *INPUT* uses the first 39 bits in *RSK*. The second input byte (from 9th to 16th bit) needs a *RSK* bit range of 9th to 47th. In this way, we calculate the bit ranges of each element in *INPUT* in Table 1.

Let $RSK[A:B]$ be a bit range of RSK from A^{th} bit to B^{th} bit.

$$i)\ RSK[1:63] = RSK[33:95]$$
$$ii)\ RSK[65:111] = RSK[81:127] \quad (2)$$

Using Table 1, we can convert Equation (1) to Equation (2). This equation requires that the RSK bit ranges for source and destination IPs take on the same values, as well as for the source and destination ports. If RSK bit ranges satisfy this condition, the RSS hash values will be the same for TCP packets in both directions.

$$i)\ RSK[1:15] = RSK[17:31] = RSK[33:47] = RSK[49:63] = RSK[65:79]$$
$$= RSK[81:95] = RSK[97:111] = RSK[113:127]$$
$$ii)\ RSK[16] = RSK[48] = RSK[80] = RSK[96] = RSK[112]$$
$$iii)\ RSK[32] = RSK[64] \quad (3)$$

We note that Equation (2) has overlapping regions, so the condition breaks into three non-overlapping parts as shown in Equation (3). One example of *RSK* satisfying Equation (3) is in Figure 1. 0x6d5a shows a group of 16 bits (in 2 bytes). Every group except the second and fourth group has exactly same bit sequence. The second and fourth group have a different bit in 8th bits, and other bits are the same as the other groups.

0x6d5a **0x6d5b** 0x6d5a **0x6d5b**
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a

Figure 1: A sample *RSK* that satisfies Equation(3)

RULE 2. *During calculation of n^{th} byte of INPUT, only the n^{th} to $(n+5)^{th}$ byte range of RSK is used.*

We can also re-phrase Equation (1) in the byte level by changing Rule 1 to Rule 2, which is simpler than the bit-level condition.

Let $RSK[[i]]$ be i^{th} bytes, $1 \leq i \leq 80$

$$\begin{aligned} &i)\ RSK[[1]] = RSK[[3]] = RSK[[5]] = RSK[[7]] \\ &\quad = RSK[[9]] = RSK[[11]] = RSK[[13]] = RSK[[15]] \\ &ii)\ RSK[[2]] = RSK[[4]] = RSK[[6]] = RSK[[8]] \\ &\quad = RSK[[10]] = RSK[[12]] = RSK[[14]] = RSK[[16]] \end{aligned} \tag{4}$$

We note that the set of RSK that satisfies Equation (4) is a subset of the set satisfying Equation (3) . One example of RSK that satisfies Equation (4) is in Figure 2.

0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a

Figure 2: A sample *RSK* that satisfies Equation(4)

2.3 Analysis

We have identified two RSK independent conditions that satisfy the symmetric RSS algorithm. In this section, we analyze whether the new RSK affects the level of load balancing compared with the original RSS.

Let $K_i(1 \leq i \leq 320)$ be the i^{th} bit in the new RSK, $P_j(1 \leq j \leq 96)$ be the j^{th} bit in input bit sequence, and $R_{i,j}(1 \leq j \leq 32)$ be the j^{th} bit in the result of the RSS function after i iterations in the for loop in Algorithm 1. After the first iteration in Algorithm 1, each bit of the result value becomes as follows:

$$R_{1,j} = P_1 \& K_j, (1 \leq j \leq 32)$$

After the second iteration,

$$R_{2,j} = R_1 \wedge P_2 \& K_{(j+1)}, (1 \leq j \leq 32)$$

So, after n iterations in the loop,

$$
\begin{aligned}
R_{n,j} &= R_{n-1,j} \wedge P_n \& K_{(j+n-1)} (1 \le j \le 32) \\
&= (\sum_{i=1}^{n} P_i \, K_{(i+j-1)}) \mod 2
\end{aligned} \tag{5}
$$

Since our input is 96 bits and the new *RSK* repeats in a unit of 16 bits by condition 4, the final hash value (*RET*) returned by the function is,

$$
\begin{aligned}
RET_j &= R_{96,j} \\
&= (\sum_{i=1}^{96} P_i \, K_{(i+j-1)}) \mod 2 \\
&= (\sum_{i=1}^{16} K_{(i+j-1)} (\sum_{n=0}^{6} P_{i+16n})) \mod 2 \\
&= (\sum_{i=1}^{16} K_{(i+j-1)} Q_i) \mod 2, \text{ where } Q_i = \sum_{n=0}^{6} P_{i+16n}
\end{aligned} \tag{6}
$$

From Equation 6, we find two corner cases for RSK that would produce undesirable results.

RULE 3. *If all bits in RSK are zero, the hash value is zero.*

RULE 4. *If all bits in RSK are one, all bits in the hash value are either 0 or 1.*

It is easy to see that Rule 3 holds. If K_i is one, every RET_j ($RET_j = \sum_{i=1}^{96} P_i$) takes on either 0 or 1 regardless of j. These two corner cases of *RSK* limits the output value to a small space, which makes it unsuitable for load balancing among multiple queues. From (5), we also find that if RSK is an n-bit sequence, the result is also repeated as an n-bit sequence. As our result needs a 16-bit sequence, the returned hash value has a 16-bit pattern.

$$
\begin{aligned}
RET &= RET_1 RET_2 RET_3 ... RET_{16} RET_1 RET_2 RET_3 ... RET_{16} \\
&= \sum_{i=1}^{16} 2^{16-i} * RET_i + \sum_{i=1}^{16} 2^{32-i} * RET_i \\
&= (2^{16} + 1) * \sum_{i=1}^{16} (2^{16-i} * RET_i)
\end{aligned} \tag{7}
$$

Since $\sum_{i=1}^{16} (2^{16-i} * RET_i)$ takes on a random value in the 16-bit value space if RET_i is random, we find that ($RET \mod$ (number of RX queues)) would take on a random value if the number of RX queues is smaller than 65,536.

2.4 Experiments

We conduct experiments that show even load balancing of our symmetric RSS. We create 100,000 TCP packets with random source and destination IPs and port numbers, and calculate the hash values for each packet using the RSS algorithm.. We test with various numbers of RX queues and use two RSK sets as shown in Figure 3.

For comparison with the original RSK, we plot the coefficient of variance (CV) in Figure 4. If CV is close to zero, it implies that the variance between the queue size is small and load is balanced among the queues. As shown in the Figure, both symmetric RSS and original RSS have CV values smaller than

0x6d5a 0x6d5a 0x6d5a 0x6d5a	0x6d5a 0x56da 0x255b 0x0ec2
0x6d5a 0x6d5a 0x6d5a 0x6d5a	0x4167 0x253d 0x43a3 0x8fb0
0x6d5a 0x6d5a 0x6d5a 0x6d5a	0xd0ca 0x2bcb 0xae7b 0x30b4
0x6d5a 0x6d5a 0x6d5a 0x6d5a	0x77cb 0x2da3 0x8030 0xf20c
0x6d5a 0x6d5a 0x6d5a 0x6d5a	0x6a42 0xb73b 0xbeac 0x01fa
<Symmetric RSS>	<Original RSS>

Figure 3: *RSK* sets used in the experiments

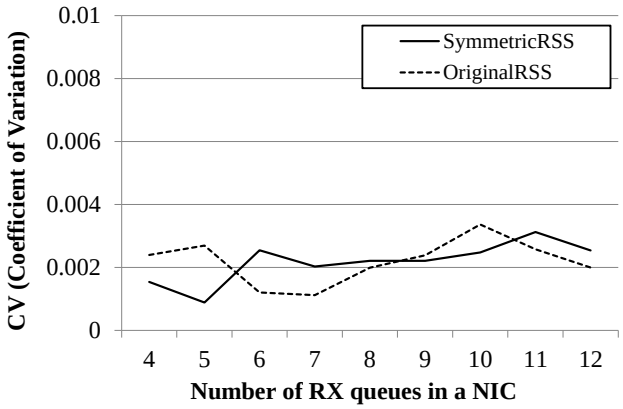

Figure 4: CV of the numbers of packets distributed to RX queues

0.005. From this, we find that our symmetric RSS algorithm provides a similar level of load balacing with that of the original RSS algorithm.

3. CONCLUSION

Load balancing of the received packets into multiple cores is the key to high performance TCP session processing. Receive-side scaling is a simple and efficient solution for load balancing of the packets into multiple RX queues; however, it cannot map the packets in the same TCP connection to the same RX queue, which creates an extra overhead in managing the packets in the same TCP connection. We find that preparing RSK to have a repeating 16-bit pattern makes the original RSS algorithm symmetric for the packets in the same TCP connection. While this new RSK set returns a 32-bit hash value with a 16-bit pattern, we show that the new seeds do not affect the level of load balancing compared with that of the original algorithm if we avoid a few corner cases.

4. REFERENCES

[1] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *Proceeding of the USENIX Symposium on Networked Systems Design and Implementation*, pages 419–432, 2010.

[2] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 Conference On Data Communication*, SIGCOMM '09, pages 87–98, New York, NY, USA, 2009. ACM.

[3] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the ACM Conference on Computer and*

Communications Security (CCS), 2012.

[4] H. Krawczyk. Lfsr-based hashing and authentication. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '94, pages 129–139, London, UK, UK, 1994. Springer-Verlag.

[5] Microsoft. MSDN: Introduction to Receive-Side Scaling. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff556942%28v=vs.85%29.aspx`, 2012. [Online; accessed 24-April-2012].

[6] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238, 1999.

[7] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Midea: a multi-parallel intrusion detection architecture. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 297–308. ACM, 2011.