SEE
TEXT
ONLY

NETWORK TRAFFIC ANALYSIS

Lynx

BY PAWEL JAKUB DAWIDEK
AND MILOSZ KANIEWSKI

Over the last few years, a number of TLS connections inside corporate networks increased significantly. The percentage of encrypted Internet traffic passed 50% last year and it doesn't look like this trend is going to slow down anytime soon. While it is very good to observe that encryption is getting widespread adoption, it is worth remembering that security has many faces.

In corporate networks, it is crucial for security teams to control the traffic exchanged with the outside world. Incoming traffic may contain harmful software such as viruses or ransomware. Malware designers have already started to use TLS to hide traffic that should never be detected by security software. Moreover, as it is now possible to get trusted certificates for free, assumptions such as "a padlock in the browser address field means the webpage is secure" are no longer valid. There is also a huge demand to control traffic from a local network to the Internet. It may contain confidential data that should never leave internal company infrastructure.

There is a large market of specialized tools in traffic analysis that helps network administrators provide necessary safety and guidance for network traffic control. However, IDS (Intrusion Detection Systems), IPS (Intrusion Prevention Systems), and DLP (Data Leak Prevention) tools may become powerless when the traffic they are monitoring is encrypted, e.g., with the TLS protocol. One way to deal with this problem would be to block entire TLS traffic, but, of course, that is totally impractical nowadays. The only way to get insight into it is to use a controlled MiTM technique which we describe below.

Lynx, developed by our company, is an enterprise class product which performs security-driven TLS interception. Using FreeBSD as our base system, we were able to create a product that can greatly outperform competing products from leading vendors in this market. In this article, we would like to share some of our experiences. For the last two years, we have tested and experimented with a number of approaches to fast packet processing. We hope that our story will be interesting for everyone interested in a network system architecture.

TLS Interception

As we have mentioned before, to properly intercept TLS connections, we have to use a controlled MiTM technique (Figure 1). First, pretending to be a destination server, we accept a TCP connection from a client. Next, we create a second TCP connection to the destination server. After that, we can start to exchange packets between client and server and, as a result, see the content of the payload. Next, we start to analyze packets sent by the client and check if there is a TLS Client Hello packet that would indicate the beginning of a TLS session. When we detect such packet, we check the content of the Server Name Indication (SNI) extension, and if it exists, we save it. This extension is used to allow servers to serve more than one certificate on a single IP address. For example, if there are two servers, example1.org and example2.org, and they both are hosted on the same IP address, then SNI allows the server to choose which certificate should be presented to the client. It is also possible that server will have one certificate that matches both SNIs. However, this is not required and a server may have two certificates, as in our example.

After reading the SNI, we suspend communication with the client and we establish a new TLS session with the server (of course using same SNI that we have stored just before). After establishing this TLS connection, we know what the server certificate looks like. Having that knowledge, we can next generate an almost identical certificate and sign it using the custom certificate authority. We can then resume communication with the client, complete the TLS handshake and present the just generated certificate. If we want the TLS session to be successful, the client needs to trust this certificate. Usually, in corporate networks, TLS clients (e.g. web browsers) are already configured to trust the additional CA certificate used to sign certificates of internal resources. If Lynx uses the same CA (or sub-CA) to sign certificates that it generates, then the client will automatically trust them.

This step finishes the negotiation phase and leaves us with two TLS channels: one with the client, and one with the server.

As we can see for every intercepted TLS session, Lynx has to handle two independent TLS connections: one with a client and one with a server. Being a part of both connections, Lynx has the ability to see decrypted traffic transmitted in two directions. This decrypted data can be next sent to other tools that can analyze its content looking for malware or data leaks.

Beginning of the Lynx Architecture

The very first version of the architecture we experimented with used one process per connection. Of course, that is a very naive approach performance-wise, but has some great security properties—we could close each process in a very tight sandbox using Capsicum, thus isolating every connection from one other. With such an approach, we could handle a relatively small number of connections: we were limited by a number of processes in the system, which, by default, is 100,000, and we needed a lot

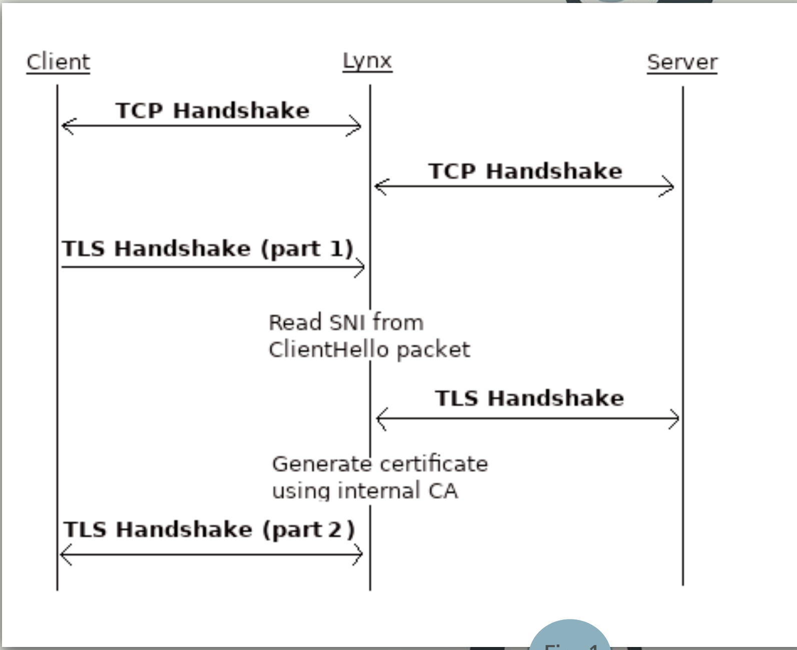


Fig. 1

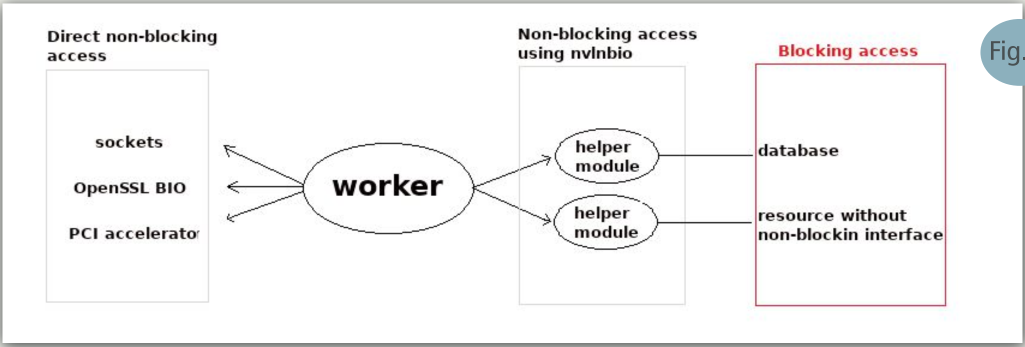
of memory. We also put huge pressure on the scheduler to switch between all those processes as the packets were arriving.

We even didn't try to optimize this architecture, as it didn't allowed us to handle hundreds of thousands of simultaneous connections which was already our goal. Using threads was also not an option, as it resulted in the same problems (mostly a scheduler overhead). Instead we moved to a fully non-blocking and event-driven architecture. This approach to packets processing was, at this time, already implemented in some network applications like Nginx. As the name suggests, it is based on two main concepts:

Non-blocking:
In non-blocking programming, a user program never performs any operation that would block its execution. If some resource is not immediately available, then the function returns and it is the responsibility of the user to call this function again later.

Event-driven:
The user program needs to know when it should repeat a non-blocking operation. In the classical approach, the user program regularly polls a specified resource to check if its state has changed. However, polling may be expensive. Generally a better solution is to let the operating system inform the user program about resource availability. This kind of information is called an event.

Fig. 2


Direct non-blocking access
sockets
OpenSSL BIO
PCI accelerato
worker
Non-blocking access using nvlnbio
helper module
helper module
Blocking access
database
resource without non-blockin interface

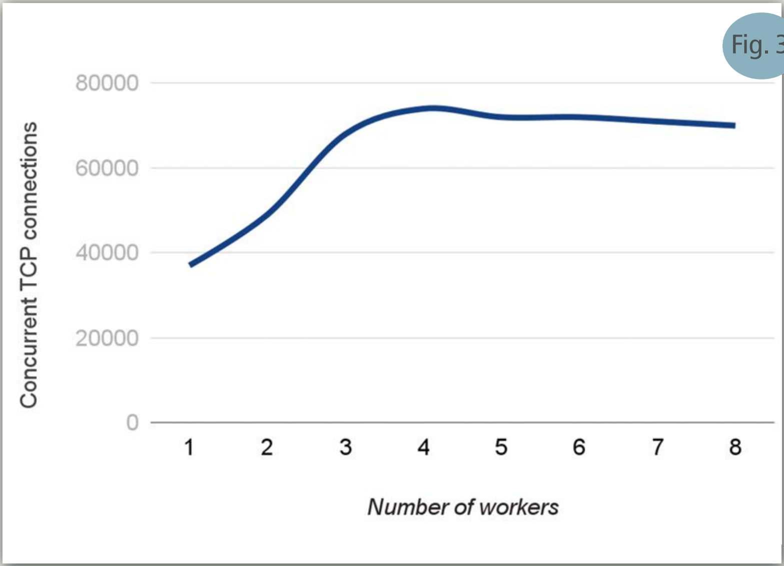
The result of this approach is a finite and small number of worker processes. All workers are listening on the same accept socket and it is the responsibility of the kernel to share incoming connections fairly between them. By binding workers to particular CPUs, we guarantee that scheduler overhead will be completely unnoticeable.

To adapt Lynx to this new model, we had to localize all blocking operations and rewrite them to non-blocking versions. First, we needed to modify all network I/O operations, which wasn't too hard, as non-blocking operations are supported by BSD sockets for a very long time. Also, OpenSSL modification was very straightforward. One of the resources that we had to treat specially was the PostgreSQL database. Due to the nature of DBMS, the database can execute only a single query at one given time. One way to execute more than one query is to have multiple database connections. However, their number is limited (in the case of PostgreSQL, it should not be bigger than several hundred), and using too many of them may affect performance. Therefore, the only way to handle more queries is to queue them at the level of the user program. To give workers the illusion that they communicate with the database in a fully non-blocking manner, we decided to delegate direct database operations to a separate module. To make communication between a worker and a module non-blocking, we have prepared a modified version of an nv(3) interface (we have named this new IPC "nvlnbio"). We also started using separated modules in all situations where workers needed to communicate with some resource that did not offer a fully non-blocking API (Figure 2).

Some time later during the tests, we discovered that in the case of a large number of TCP connections per second, our worker model does not scale very well. It turned out that it didn't scale linearly—every additional worker resulted in worse "per-worker" performance, and after four workers, we stopped scaling at all, even though each worker had a dedicated CPU core (Figure 3).

To investigate this problem we decided to look for locks used by the network stack during heavy

Fig. 3



load (we were using FreeBSD 10.2). For this purpose we used a LOCK_PROFILING(9) interface provided by the kernel.

First we checked lock usage with one worker running (Figure 4).

Next we repeated the same situation, but with two simultaneously running workers (Figure 5).

```
max  wait_max      total  wait_total      count     avg wait_avgcnt_hold  cnt_lock    name
 18        68     455629     2835614    3157952       0        0        0    515451   /src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
 19        74     580341       33734    8447282       0        0        0    101461   /src/sys/kern/kern_timeout.c:535 (spin mutex:callout)
  2      3198        202       31632       1410       0       22        0        14   /src/sys/kern/kern_exit.c:469 (sx:allproc)
  0     30284          0       30284          4       0     7571        0         1   /src/sys/dev/coretemp/coretemp.c:327 (spin mutex:sched lock 0)
 60        38     699727       17510    5413739       0        0        Q     48641   /src/sys/net/route.c:439 (rw:rtentry)
```

Fig. 4

When two workers were running, the new type of locks related to TCP processing started to be hot. When more than two workers were launched, those locks became more and more active, and we suspected that they may be a source of the performance decrease we observed.

```
max  wait_max      total  wait_total      count     avg wait_avgcnt_hold  cnt_lock    name
 18        43     572746     3814587    4077828       0        0        0    572552   /src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
310       573    1046182     3237040     582523      17        5        0    196106   /src/sys/netinet/tcp_usrreq.c:1142 (rw:pcbinfohash)
552       538   10662226     2522505     582523      18        4        0    241451   /src/sys/netinet/tcp_usrreq.c:296 (rw:pcbinfohash)
 17       103      81477     1970702    1137267       0        1        0    250167   /src/sys/netinet/in_pcb.c:1322 (rw:pcbinfohash)
```

Fig. 5

We had also used dtrace to detect all situations in which blocking could occur. Our goal was to have a program that blocks only if there is no work to do. We wanted to eliminate blocking on socket operations, IPC communication or reading data from the disk. The only place that we expected to be blocking was on invocation of the kevent(2) function on the main event loop.

Therefore, we created a dtrace script that informs us whenever blocking occurs in any other function (next page).

The sleepq_add(9) is the best place to check if the process is blocking somewhere in the kernel.

```
#!/usr/sbin/dtrace -s

  syscall::kevent:entry
  /execname == "lynxd"/
  {
      self->inkevent = 1;
  }

  fbt::sleepq_add:entry
  /!self->inkevent && execname == "lynxd"/
  {
      printf("%s(%d)\n", execname, pid);
      stack();
      ustack();
  }

  syscall::kevent:return
  /execname == "lynxd"/
  {
      self->inkevent = 0;
  }
```

Network Stack Virtualization (VNET)

To overcome locking problems, we decided to give VNET(9) a try. We hoped that locks detected during our test would be virtualized, and as a result workers would no longer need to compete for them. The test results showed that our assumptions were right (Figure 6).

After using VNETs, hot locks on the TCP stack were no longer present. Performance tests also showed that worker scalability after using VNETs gets much better.

To easily adapt VNET technology into our architecture, we had to make a couple of changes in the kernel code. By design, VNETs are tightly integrated with jails. For example, when an interface is attached to the VNET, it disappears from the global system view and is only accessible from within the appropriate jail. Therefore, to use interfaces attached to VNETs, we would have to launch our worker programs in many separate jails, but we didn't want that. Instead, we have modified the

Fig. 6

```
max  wait_max     total  wait_total      count     avg wait_avgcnt_holdcnt_lock name
 19       135    700557     3763943    4556117       0         0        0   480993  /src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
 19        56   1067986       93090   12345133       0         0        0   347169  /src/sys/kern/kern_timeout.c:535 (spin mutex:callout)
 33        19   1169609       39617    7805677       0         0        0    95923  /src/sys/net/route.c:439 (rw:rtentry)
 25        30    220474       32546    1952683       0         0        0    86194  /src/sys/kern/kern_event.c:1895 (sleep mutex:kqueue)
  0      3253       251       18068       1664       0        10        0        7  /src/sys/kern/kern_exit.c:469 (sx:allproc)
```

socket(2) interface to allow us to define VNET numbers during socket creation (Figure 7):

```
/* Create socket bounded to VNET (jail) number 100. */
int sock = socket(PF_INET, SOCK_STREAM | VNET_TO_SOCKTYPE(100));
```

Fig. 7

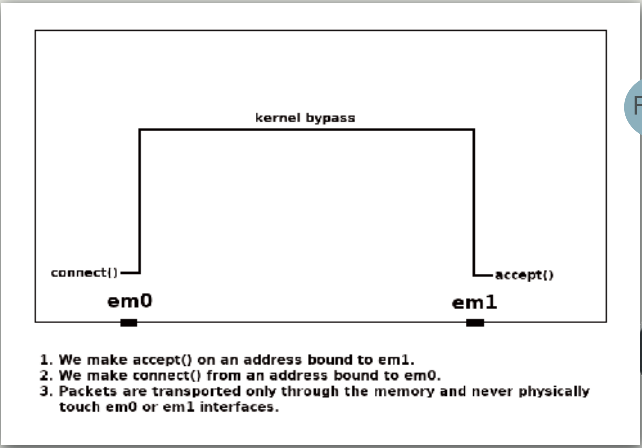
Choosing VNET technology allows us to resolve one other problem that we were facing at the time. As mentioned before, Lynx only decrypts traffic and doesn't do any kind of traffic analysis. Other monitoring devices (e.g., IDS or IPS) are used for this purpose. Such devices can usually operate in two modes. The first is when they fetch a copy of the traffic and analyze it without accessing the main data path—this mode is called "out-of-band." In the second mode, a monitoring device needs to have access to the real traffic and thus to the main data path—this mode is called "inline." The out-of-band mode was supported in Lynx from the beginning, but the inline mode was still to be implemented. As Lynx itself stands in the main data path, there was a need to create a kind of sub-path to which a monitoring device could be connected. From the hardware view, such a sub-path would be just a wire loop between two NIC interfaces. The question we were facing was how we should create a TCP connection on such a loop. One possibility was to craft all TCP packets in our code (we were already doing it for out-of-band mode). However in out-of-band mode, we only send packets, so there is no possibility that TCP problems like packets reordering or packet drops would happen. In the inline mode such situations could take place (e.g., the monitoring device would drop packets) and so it was mandatory to handle them properly. Therefore, we quickly became convinced that the only possible solution was to use a full TCP stack implementation.

The most obvious solution implied usage of the FreeBSD kernel network stack. The plan was to call accept(2) on one NIC port, connect(2) on the second port and as a result establish a TCP connection where packets would appear on the wire linking two interfaces. However, it turned out that in such a situation, the network stack detects that it controls both sides of the transmission and therefore forwards all the traffic through the kernel. As a result, the TCP connection is properly established, but no packets are ever transferred over the physical wire (Figure.8).

Using VNETs solved this problem because it put two NIC interfaces into two separate virtualized stacks that now did not see one other (Figure 9). The result was that packets were forced to be transferred over the wire. VNETs also allow us to use the same TCP/IP addressing on multiple interfaces, which is not possible with a single TCP/IP stack.

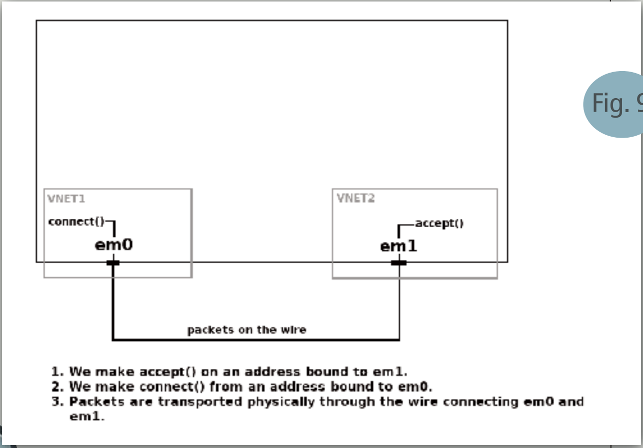
Full Scalability

Up to this point, we have been talking about performance mostly in terms of the maximum number of TCP connections per second. However, another very important benchmark in the case of network devices is a traffic throughput. The nice thing about using modern CPUs is that there are no major differences between transmitting plain and encrypted traffic. With support for AES-NI acceleration, processing of encrypted packets is almost free. One CPU core (tests were performed on Intel Xeon E5-2697A v4) can process about 10.4 gigabits of encrypted data per second. This means that in theory, two cores are enough to decrypt/encrypt TLS traffic on a full-duplex 10Gb NIC port. During our test it turned out that logical cores that were available when Hyper-Threading was enabled were per-



1. We make accept() on an address bound to em1.
2. We make connect() from an address bound to em0.
3. Packets are transported only through the memory and never physically touch em0 or em1 interfaces.

Fig. 8



1. We make accept() on an address bound to em1.
2. We make connect() from an address bound to em0.
3. Packets are transported physically through the wire connecting em0 and em1.

Fig. 9

forming AES as well as physical cores.

However at such high throughput, other factors start to have significant impact on performance. Most of them are related to memory:

- number of cache misses,
- memory locality in case of NUMA architecture,
- speed of RAM memory (both throughput and latency),
- number of memory channels on the motherboard,
- speed of the QPI bus.

A general approach to handling all of these cases is to guarantee that data processed by the device is read/write as infrequently as possible, and when it is happening, then as many operations as possible are performed on it (run-to-completion architecture). In other words, we need to have full control of the place where packets are processed (i.e. on which CPU) and when it happens.

The first place where we can control packets is the network card. In modern NICs, packets are processed in separate queues, and it is possible to steer how they are distributed across those queues. Usually RSS (Receive Side Scaling) is used, and packets are classified by IP addresses (and sometimes by port numbers). As a result, packets from one connection are always processed by the same NIC queue. If we want the connection in both directions to be hashed to the same RSS queue, then we need to be sure that the RSS key used for hashing is symmetric. Such a key can be easily found on the Internet.

Going forward, we would like to be able to process packets from one queue always on the same CPU core. To achieve this we have to bind all workers and corresponding IRQ threads (one per receive NIC queue) to specified cores.

All of this hardware optimization is possible using the kernel stack. However, even using VNETs, there is still a possibility that multiple workers would compete for some locks. Switching between kernel and userspace context may also decrease performance. Every read-and-write operation causes execution of a syscall which triggers a copy of memory from a user buffer to a socket buffer. To mitigate those problems, we have decided to try a solution that is recently getting widespread attention—user space TCP stack. There are many implementations of such stacks available, but we decided to try libuinet developed by Patrick Kelsey. It is a user space implementation of a FreeBSD kernel stack and, therefore, it inherits most of its attributes like stability and efficiency. It works on top of netmap, which is another excellent software that allows user space apps to get direct access to NIC buffers. Both of these solutions guarantee that after receiving a packet on a NIC it will be delivered directly to a user space program without the need to use kernel context (besides IRQ threads for communication) and with as few as possible memory operations.

Summarizing, at this point incoming packets would be processed in this order:

1.Received on NIC, which forwards it to an appropriate queue.
2. DMA packets to a memory region mapped by netmap.
3. Packets processing by libuinet stack.
4. Read packet data into worker code.

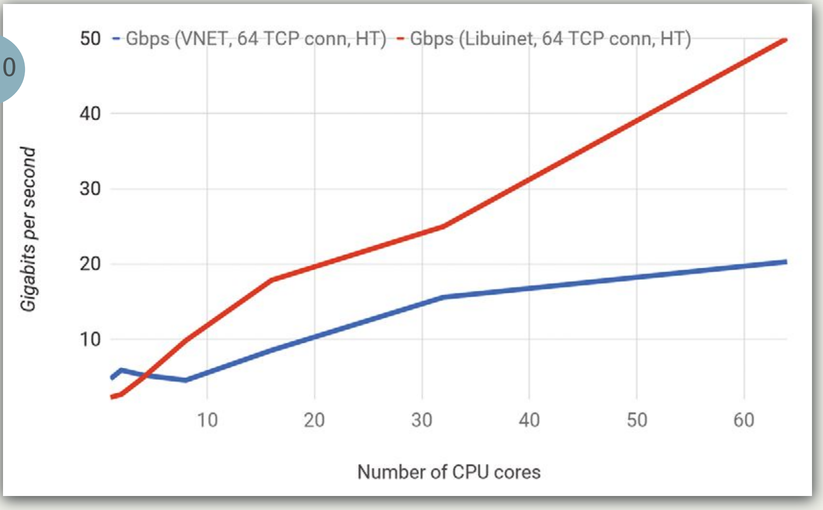
All work required in this path would be done on the same CPU core. The output route would look very similar (in our case one worker process is responsible for both receiving and sending packets, so output processing would still take place on the same CPU core).

Using libuinet and netmap also allow us to optimize a couple of things. We could, for example, do zero-copy of packets other than TCP, as they don't contain TLS and, therefore, are not interesting to us. This feature is already implemented in libuinet, so using it is as simple as setting an L2 bridge.

After implementing all those pieces of new architecture, we began performance tests. As we expected, the new architecture scales much better compared to the previous (VNET) solution (Figure 10)

To generate TCP traffic, we were using our own software, which worked similarly to the iperf utility. We have a client machine and a server machine that exchange random data using 64 TCP connections. Between those two machines, we put Lynx, which exchanges data between connections on both sides.

Fig. 10



Lynx is equipped with the following hardware:

- 2 x Intel Xeon E5-2697A v4 CPU (16 physical cores with HT enabled). In total, it gives us 64 logical CPU cores.
- 256 GB of 2400 RDIMM RAM placed in 8 channels (4 channels per CPU socket).
- 10 Gbps T540-CR Chelsio NIC cards.

We run our tests by increasing the number of CPU cores processing packets starting from 1 and going to 64. For one CPU, VIMAGE architecture gave us very good result (about 5 Gbps). With 4 cores processing packets, we have seen similar results for both VIMAGE and libuinet architecture. But for a higher number of cores, libuinet architecture showed its potential. While its performance didn't scale linearly, it still offered a very good ratio between processed traffic and number of cores used. With a maximum number of 64 cores, it allows Lynx to process over 50 Gbps of traffic (this is still related to TCP traffic, but with some other optimizations not described in this article, we were also able to process 50 Gbps of TLS traffic). In the same time, VIMAGE architecture lets us process only 20 Gbps of traffic.

What's Next

While libuinet architecture did a great job, there are still a lot of possible optimizations that we would like to explore in the future. We are sure that our biggest current bottleneck is the memory bandwidth. Using PCM tools, we were able to measure that the saturation of RAM throughput almost reaches its limit. We need to investigate whether all packet copying operations are required and to look for solutions that would increase the possibility that processed data is already in the CPU cache.

During our test we didn't have NUMA or COD enabled. Both of these technologies should help us achieve better memory locality per CPU socket. Without it, both processors use each other's RAM banks, which causes increase of memory access latency. Furthermore, it causes high utilization of the QPI bus whose throughput is also limited.

There is also a broad range of optimizations that can be done on NIC cards. We use Chelsio cards that offer a lot of interesting features. For example, they allow movement of packets between the ports of one NIC card without using system RAM at all. Such solutions may be used to move packets other than TCP, as they are not interesting for Lynx in any case. Also, Direct Cache Access (DCA) can reduce number of read operations and as a result, decrease RAM usage. The next stop for Lynx is 100 Gbps. Stay tuned! •

PAWEL JAKUB DAWIDEK is cofounder and CTO at Wheel Systems and the main architect of the company's products. He is also a longtime FreeBSD committer and the author of GELI disk encryption, HAST highly available storage, auditdistd, the ZFS filesystem port and many contributions to Capsicum and GEOM.

MILOSZ KANIEWSKI is a programmer and FreeBSD user. He works for Wheel Systems and is responsible for development of the Lynx SSL/TLS Decryptor. He is mostly interested in network programming, especially in software utilizing the TLS protocol.