# 15:08 Zero Overhead Networking

*by Robert Graham*

The kernel is a religion. We programmers are taught to let the kernel do the heavy lifting for us. We the lay folks are taught how to propitiate the kernel spirits in order to make our code go faster. The priesthood is taught to move their code into the kernel, as that is where speed happens.

This is all a lie. The true path to writing high-speed network applications, like firewalls, intrusion detection, and port scanners, is to completely bypass the kernel. Disconnect the network card from the kernel, memory map the I/O registers into user space, and DMA packets directly to and from user-mode memory. At this point, the overhead drops to near zero, and the only thing that affects your speed is you.

## Masscan

Masscan is an Internet-scale port scanner, meaning that it can scan the range `/0`. By default, with no special options, it uses the standard API for raw network access known as `libpcap`. `Libpcap` itself is just a thin API on top of whatever underlying API is needed to get raw packets from Linux, macOS, BSD, Windows, or a wide range of other platforms.

But Masscan also supports another way of getting raw packets known as `PF_RING`. This runs the driver code in user-mode. This allows Masscan to transmit packets by sending them directly to the network hardware, bypassing the kernel completely (no memory copies, no kernel calls). Just put "zc:" (meaning `PF_RING` ZeroCopy) in front of an adapter name, and Masscan will load `PF_RING` if it exists and use that instead of `libpcap`.

In the section below, we are going to analyze the difference in performance between these two methods. On the test platform, Masscan transmits at 1.5 million packets-per-second going through the kernel, and trasnmits at 8 million packets-per-second when going though `PF_RING`.

We are going to run the Linux profiling tool called `perf` to find out where the CPU is spending all its time in both scenarios.

Raw output from `perf` is difficult to read, so the results have been processed through Brendan Gregg's FlameGraph tool. This shows the call stack of every sample it takes, showing the total time in the caller as well as the smaller times in each function called, in the next layer. This produces SVG files, which allow you to drill down to see the full function names, which get clipped in the images.

I first run Masscan using the standard `libpcap` API, which sends packets via the kernel, the normal way. Doing it this way gets a packet rate of about 1.5 million packets-per-second, as shown in Figure 5.

To the left, you can see how `perf` is confused by the call stack, with [unknown] functions. Analyzing this part of the data shows the same call stacks that appear in the central section. Therefore, assume all that time is simply added onto similar functions in that area, on top of `__libc_send()`.

The large stack of functions to the right is `perf` profiling itself.

In the section to the right where Masscan is running, you'll notice little towers on top of each function call. Those are the interrupt handlers in the kernel. They technically aren't part of Masscan, but whenever an interrupt happens, registers are pushed onto the stack of whichever thread is currently running. Thus, with high enough resolution (faster samples, longer profile duration), `perf` will count every function as having spent time in an interrupt handler.

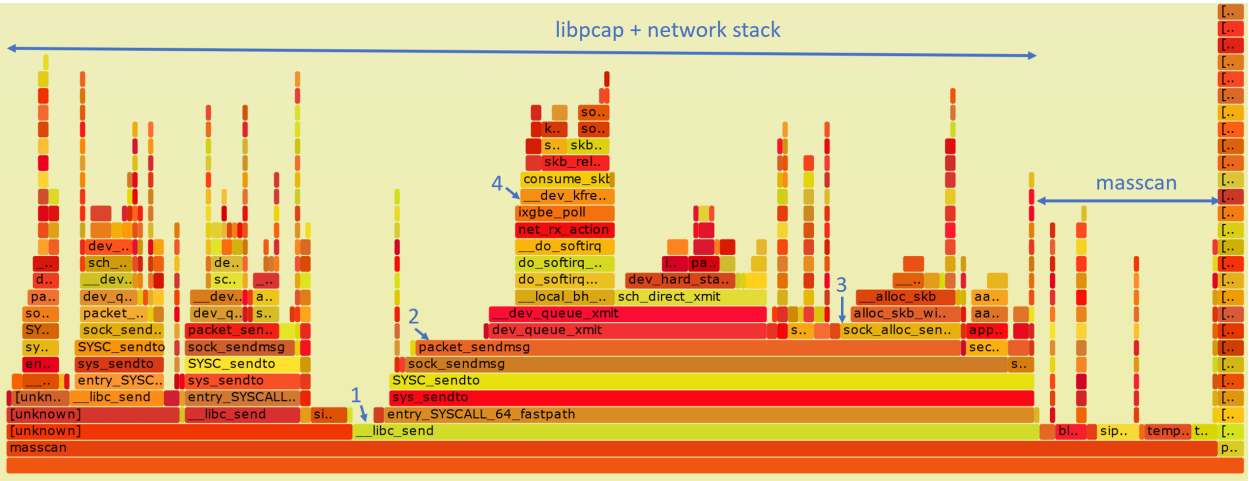
The next run of Masscan bypasses the kernel completely, replacing the kernel's Ethernet driver with the user-mode driver `PF_RING`. It uses the same options, but adds "zc:" in front of the adapter name. It transmits at 8 million packets-per-second, using an Ivy Bridge processor running at 3.2 GHz (turboed up from 2.5 GHz). Shown in Figure 6, this results in just 400 cycles per packet!

The first thing to notice here is that 3.2 GHz divided by 8 mpps equals 400 clock cycles per packet. If we looked at the raw data, we could tell how many clock cycles each function is taking.

Masscan sits in a tight scanner loop called `transmit_thread()`. This should really be below all the rest of the functions in this flame graph, but apparently `perf` has trouble seeing the full call stack.

The scanner loop does the following calculations:

- It randomizes the address in `blackrock_shuffle()`
- It calculates a SYN cookie using the `siphash24()` hashing function




1 marks the start of `entry_SYSCALL_64_fastpath()`, where the machine transitions from user to kernel mode. Everything above this is kernel space. That's why we use `perf` rather than user-mode profilers like `gprof`, so that we can see the time taken in the kernel.

2 marks the function `packet_sendmsg()`, which does all the work of sending the packet.

3 marks `sock_alloc_send_pskb()`, which allocates a buffer for holding the packet that's being sent. (`skb` refers to `sk_buff`, the socket buffer that Linux uses everywhere in the network stack.)

4 marks the matching function `consume_skb()`, which releases and frees the `sk_buff`. I point this out to show how much of the time spent transmitting packets is actually spent just allocating and freeing buffers. This will be important later on.

Figure 5. Performance profile of Masscan with `libpcap`.

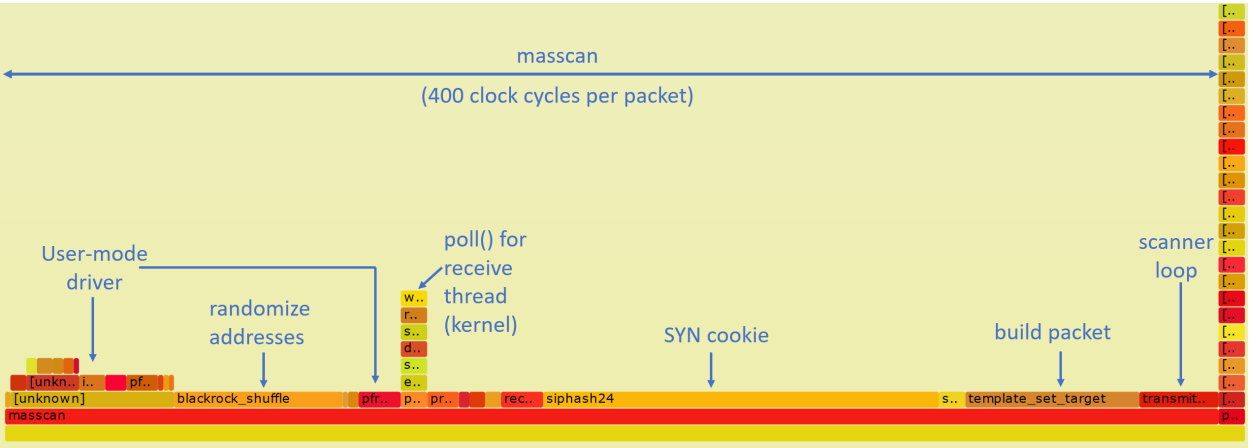



Figure 6. Performance profile of Masscan with `PF_RING`.

- It builds the packet, filling in the destination IP/port, and calculating the checksum
- It then transmits it via the `PF_RING` user-mode driver

At the same time, the `receive_thread()` is receiving packets. While the transmit thread doesn't enter the kernel, the receive thread will, spending most of its time waiting for incoming packets via the poll() system call. Masscan transmits at high rates, but receives responses at fairly low rates.

To the left, in two separate chunks, we see the time spent in the `PF_RING` user-mode driver. Here `perf` is confused: about 1/3 of this time is spent in the receive thread, and the other 2/3 in the transmit thread.

About ten to fifteen percent of the time is taken up inside `PF_RING` user-mode driver or an overhead 40 clock cycles per packet.

Nearly half of the time is taken up by `siphash24()`, for calculating the SYN cookie. Masscan doesn't remember which packets it's sent, but instead uses the SYN cookie technique to verify whether a response is valid. This is done by setting the Initial Sequence Number of the SYN packet to a hash of the IP addresses, port numbers, and a secret. By using a cryptographically strong hash, like `siphash`, it assures that somebody receiving packets cannot figure out that secret and spoof responses back to Masscan. Siphash is normally considered a fast hash, and the fact that it's taking so much time demonstrates how little the rest of the code is doing.

The *build packet* takes ten percent of the time. Most of the this is spent needlessly calculating the checksum. This can be offloaded onto the hardware, saving a bit of time.

The most important point here is demonstrating that the transmit thread doesn't hit the kernel. The receive thread does, because it needs to stop and wait, but the transmit thread doesn't. `PF_RING`'s custom user-mode driver simply reads and writes directly into the network hardware registers, and manages the transmit and receive ring buffers, all memory-mapped from kernel into user mode.

The benefits of this approach are that there is no system call overhead, and there is no needless copying of packets. But the biggest performance gain comes from not allocating and then freeing packets. As we see from the previous profile, that's where the kernel spends much of its time.

The reason for this is that the network card is normally a shared resource. While Masscan is transmitting, the system may also be running a webserver on that card, and supporting SSH login sessions. Sharing these resources ultimately means allocating and freeing `sk_buff`s whenever packets are sent or received.

`PF_RING`, however, wrests control of the network card away from the kernel, and gives it wholly to Masscan. No other application can use the network card while Masscan is running. If you want to SSH into the box in order to run m̂asscan, you'll need a second network card.

If Masscan takes 400 clock cycles per packet, how many CPU instructions is that? Perf can answer that question, with a call like `perf -a sleep 100`. It gives us an IPC (instructions per clock cycle) ration of 2.43, which means around 1000 instructions per packet for Masscan.

To reiterate, the point of all this profiling is this: when running with `libpcap`, most of the time is spent in the kernel. With `PF_RING`, we can see from the profile graphs that the kernel is completely bypassed on the transmit thread. The overhead goes from most of the CPU to very little of the CPU. Any performance issues are in the Masscan, such as choosing a slow cryptographic hash algorithm instead of a faster, non-cryptographic algorithm, rather than in the kernel!

## How to Replicate This Profiling

Here is brief guide to reproducing this article's profile flamegraphs. This would be useful to compare against other network projects, other drivers, or for playing with Masscan to tune its speed. You may skip to the next section on a first reading, but if, like me, you never trusted a graph you could not reproduce yourself, read on!

Get two computers. You want one to transmit, and another to receive. Almost any Intel desktop will do.

Buy two Intel 10gig Ethernet adapters: one to transmit, and the other to receive and verify the packets have been received. The adapters cost $200 to $300 each. They have to be the Intel chipset, other chipsets won't work.

Install Ubuntu 16.04, as it's the easiest system to get `perf` running on. I had trouble with other systems.

The `perf` program gets confused by idle threads. Therefore, for profiling, I rebooted the Linux computer with `maxcpus=1` on the boot command

line. I did this by editing `/etc/default/grub`, adding `maxcpus=1` to the line `GRUB_CMDLINE_LINUX_DEFAULT`, then running `update-grub` to save the configuration.

To install `perf`, Masscan, and FlameGraph.

```
apt-get install linux-tools-common \
    linux-tools-`uname -r` git \
    build-essential libpcap-dev

git clone https://github.com/brendangregg/
    FlameGraph
# Get masscan from source and build it:
git clone https://github.com/
    robertdavidgraham/masscan
cd masscan
make
make test
ln bin/masscan /usr/local/sbin/masscan
cd ..
# Get PF_RING from source and build it:
git clone https://github.com/ntop/PF_RING
cd PF_RING
make
cd kernel
make install
insmod pf_ring.ko
cd ../userland/tools
make install
cd ../drivers/intel/ixgbe/ixgbe-5.0/src
make
sh load_drivers.sh
cd ../../../../../..
```

The `pf_ring.ko` module should load automatically on reboot, but you'll need to rerun `load_drivers.sh` every time. If I ran this in production, rather than just for testing, I'd probably figure out the best way to auto-load it.

You can set all the parameters for Masscan on the command line, but it's easier to create a default configuration file in `/etc/masscan/masscan.conf`:

```
source-ip = 00:11:22:33:44:55
adapter-mac = 00:22:22:22:22:22
router-mac = 00:11:22:33:44:55
include = 0.0.0.0-255.255.255.255
exclude = 255.255.255.255
port = 0-65535
```

Since there is no network stack attached to the network adapter, we have to fake one of our own. Therefore, we have to configure that source IP and MAC address, as well as the destination router MAC address. It's really important that you have a fake router MAC address, in case you accidentally cross-connect your 10gig hub with your home network and end up blasting your Internet connection. (This has happened to me, and it's no fun.)

Now we run Masscan. For the first run, we'll do the normal adapter without `PF_RING`. Pick the correct network adapter for your machine (on my machine, it's `enp2s03`.)

```
masscan -e enp2s0f1 --rate 100000000
```

In another window, run the following. This will grab 99 samples per second for 60 seconds while Masscan is running.

```
cd FlameGraph
perf record -F 99 -a -g -- sleep 60
perf script | ./stackcollapse-perf.pl > out.
    perf-folded
./flamegraph.pl out.perf-folded > masscan-
    pcap.svg
```

You'll have to wait 60 seconds, then it'll produce the file `masscan-pcap.svg` with the FlameGraph pictures.

Now, repeat the process to produce `masscan-pfring.svg` with the following command. It's the same as the original Masscan run, except that we've prefixed the adapter name with `zc:`. This disconnects any kernel network stack you might have on the adapter and instead uses the user-mode driver in the `libpfring.so` library that Masscan will load:

```
masscan -e zc:enp2s0f1 --rate 100000000
```

At this point, you should have two FlameGraphs. Load these in any web browser, and you can drill down into the specific functions.

Playing with `perf` options, or using something else like `dtrace`, might produce better results. The results I get match my expectations, so I haven't played with them enough to test their accuracy. I challenge you to do this, though—for reproducibility is the heart and soul of science. Trust no one; reproduce everything you can.

Now back to our regular programming.

## How Ethernet Drivers Work

If you run `lspci -v` for the Ethernet cards, you'll see something like the following.

```
02:00.1 Ethernet controller: Intel Corporation 82599 10
      Gigabit TN Network Connection (rev 01)
        Subsystem: Intel Corporation 82599 10 Gigabit
      TN Network connection
        Flags: bus master, fast devsel, latency 0, IRQ
      17
        Memory at df200000 (64-bit, non-prefetchable) [
      size=2M]
        I/O ports at e000 [size=32]
        Memory at df600000 (64-bit, non-prefetchable) [
      size=16K]
        Capabilities: <access denied>
        Kernel driver in use: ixgbe
        Kernel modules: ixgbe
```

There are five parts to notice.

- There is a small 16k memory region. This is where the driver controls the card, using memory-mapped I/O, by reading and writing these memory addresses. There's no actual memory here—these are registers on the card. Writes to these registers cause the card to do something, reads from this memory check status information.

- There is a small amount of I/O ports address space reserved. It points to the same registers mapped in memory. Only Intel x86 processors support a second I/O space along with memory space, using the `inb`/`outb` instructions to read and write in this space. Other CPUs (like ARM) don't, so most devices also support memory-mapped I/O to these same registers. For user-mode drivers, we use memory-mapped I/O instead of x86's "native" `inb`/`outb` I/O instructions.

- There is a large 2-megabyte memory region. This memory is used to store descriptors (pointers) to packet buffers in main memory. The driver allocates memory, then writes (via memory-mapped I/O) the descriptors to this region.

- The network chip uses Bus Master DMA. When packets arrive, the network chip chooses the next free descriptor and DMAs the packet across the PCIe bus into that memory, then marks the status of the descriptor as used.

- The network chip can (optionally) use interrupts (IRQs) to inform the driver that packets have arrived, or that transmits are complete. Interrupt handlers must be in kernel space, but the Linux user-mode I/O (UIO) framework allows you to connect interrupts to file handles, so that the user-mode code can call the normal poll() or `select()` to wait on them. In Masscan, the receive thread uses this, but the interrupts aren't used on the transmit thread.

There is also some confusion about IOMMU. It doesn't control the memory mapped I/O—that goes through the normal MMU, because it's still the CPU that's reading and writing memory. Instead, the IOMMU controls the DMA transfers, when a PCIe device is reading or writing memory.

Packet buffers/descriptors are arranged in a ring buffer. When a packet arrives, the hardware picks the next free descriptor at the head of the ring, then moves the head forward. If the head goes past the end of the array of descriptors, it wraps around at the beginning. The software processes packets at the tail of the ring, likewise moving the tail forward for each packet it frees. If the head catches up with the tail, and there are no free descriptors left, then the network card must drop the packet. If the tail catches up with the head, then the software is done processing all the packets, and must either wait for the next interrupt, or if interrupts are disabled, must keep polling to see if any new packets have arrived.

Transmits work the same way. The software writes descriptors at the head, pointing to packets it wants to send, moving the head forward. The hardware grabs the packets at the tail, transmits them, then moves the tail forward. It then generates an interrupt to notify the software that it can free the packet, or, if interrupts are disabled, the software will have to poll for this information.

In Linux, when a packet arrives, it's removed from the ring buffer. Some drivers allocate an `sk_buff`, then copy the packet from the ring buffer into the `sk_buff`. Other drivers allocate an `sk_buff`, and swap it with the previous `sk_buff` that holds the packet.

Either way, the `sk_buff` holding the packet is now forwarded up through the network stack, until the user-mode app does a `recv()`/`read()` of the data from the socket. At this point, the `sk_buff` is freed.

A user-mode driver, however, just leaves the packet in place, and handles it right there. An IDS, for example, will run all of its deep-packet-inspection right on the packet in the ring buffer.

Logically, a user-mode driver consists of two steps. The first is to grab the pointer to the next available packet in the ring buffer. Then it processes the packet, in place. The next step is to release the

packet. (Memory-mapped I/O to the network card to move the tail pointer forward.)

In practice, when you look at APIs like `PF_RING`, it's done in a single step. The code grabs a pointer to the next available packet while simultaneously releasing the previous packet. Thus, the code sits in a tight loop calling `pfring_recv()` without worrying about the details. The `pfring_recv()` function returns the pointer to the packet in the ring buffer, the length, and the timestamp.

In theory, there's not a lot of instructions involved in `pfring_recv()`. Ring buffers are very efficient, not even requiring locks, which would be expensive across the PCIe bus. However, I/O has weak memory consistency. This means that although the code writes first A then B, sometimes the CPU may reorder the writes across the PCI bus to write first B then A. This can confuse the network hardware, which expects first A then B. To fix this, the driver needs memory fences to enforce the order. Such a fence can cost 30 clock cycles.

Let's talk `sk_buffs` for the moment. Historically, as a packet passed from layer to layer through the TCP/IP stack, a copy would be made of the packet. The newer designs have focused on "zero-copy," where instead a pointer to the `sk_buff` is forwarded to each layer. For drivers that allocate an `sk_buff` to begin with, the kernel will never make a copy of the packet. It'll allocate a new `sk_buff` and swap pointers, rewriting the descriptor to point to the newly allocated buffer. It'll then pass the received packet's `sk_buff` pointer up through the network stack.

As we saw in the FlameGraphs, allocating `sk_buff`s is expensive!

Allocating `sk_buff`s (or copying packets) is necessary in the Linux stack because the network card is a shared resource. If you left the packets in the ring buffer, then one slow app that leaves the packet there would eventually cause the ring buffer to fill up and halt, affecting all the other applications on the system. Thus, when the network card is shared, packets need to be removed from the ring. When the network card is a dedicated resource, packets can just stay in the ring buffer, and be processed in place.

Let's talk zero-copy for a moment. The Linux kernel went through a period where it obsessively removed all copying of packets, but there's still one copy left: the point where the user-mode application calls `recv()` or `read()` to read the packet's contents. At that point, a copy is made from kernel-mode memory into user-mode memory. So the term zero-copy is, in fact, a lie whenever the kernel is involved!

With user-mode drivers, however, zero-copy is the truth. The code processes the packet right in the ring buffer. In an application like a firewall, the adapter would DMA the packet in on receive, then out on transmit. The CPU would read from memory the packet headers to analyze them, but never read the payload. The payload will pass through the system completely untouched by the CPU.

Let's talk about interrupts for a moment. Back in the day, an interrupt was generated per packet. Indeed, at one time, two interrupts could be generated, one after the TCP/IP headers were received, so processing could start immediately, and another after the rest of the packet had been received.

The value of interrupts is that they provide low latency, important for devices that forward packets (firewalls, IPS, routers), or for fast responses to packets. The cost of interrupts, though, is that they cause large CPU overhead. When an interrupts happens, it forces execution of an interrupt handler. Even medium rates of packets can overwhelm the system with interrupts, so that as soon as the system leaves an interrupt handler, it immediately enters another one. In such cases, the system has essentially locked up. The mouse won't even move on the screen until the packet rate decreases, after which point the system will behave normally.[32]

The obvious solution to this is to turn off interrupts from the network card. Instead, the software can sit in a tight loop and `poll()` to see if new packets arrive. Another strategy is to program the timer chip for frequent interrupts. The card can bounce back and forth among these strategies, depending on the current network speed. Polling consumes a lot of CPU time. Using delayed timer interrupts increases latency.

Those writing custom drivers have used these strategies since the 1980s. Around 2006, Linux drivers started doing the same, using the NAPI API to enable polling when packets arrived at high speed. Around that time, network hardware also improved, adding support for coalescing interrupts, so that it generated fewer at high speed, generating only one interrupt after many packets have arrived.

In the graphs, you saw that the `libpcap` had

[32]If caught during the late stages of booting, the system might not even boot up until the packet flow eases up.

some small overhead with interrupts, but it's not overwhelming, because NAPI interrupt moderation kicks in. Using `pfring` gets rid of this overhead.

Let's talk system call overhead. A recent paper by Livio Soares and Michael Stumm does a good job measuring it.[33] The basic cost of entering or leaving kernel space is around 150 clock cycles. This alone takes more time than all the user-mode driver processing done by `PF_RING`, according to our measurements.

There are further expenses to the system call. It has to walk through a bunch of kernel data structures. This then pollutes the caches on the chip. According to the Soares paper, it evicts about half the data in the L1 cache. This will cause data access to go from 4 clock cycles (often masked by the out-of-order processing of the CPU) to 12 clocks in L2 cache, or 30 clocks in L3 cache. The effective cost can thus equal hundreds of extra clock cycles.

On the other hand, the cost can easily be amortized by doing multiple packet reads or writes per system call. Linux has a `recvmsg()` system call that does this, to good effect.

Combining all this together, we see why a user-mode driver has such big gains (or conversely, why the kernel has such big losses): (a) it avoids the allocation/deallocation of memory; (b) it avoids any memory copies; (c) it avoids system call overhead, and (d) it avoids interrupts.

## Some History of Ethernet Drivers

Since the dawn of networking there have been people dissatisfied with the standard Ethernet drivers who have written their own.

An example were packet sniffers, like the Network General "Sniffer" product. Back in the day, they wrote custom drivers so they could capture at "wire speed" on an 80286 microprocessor. The major feature was simply disabling interrupts. Portable MS-DOS computers were used as packet sniffers because "real" computers like SPARCstations running Solaris couldn't handle high traffic rates.

Early drivers were hard, because hardware sucked. There was no bus master DMA in the early ISA bus days, so for DMA, you had to use the motherboard's DMA controller. Only, it wasn't really that fast. So instead, drivers used the Programmed I/O (PIO) mode to read packets from the adapter.

There was also the problem of bus bandwidth. Early PCI supported 1 Gbps in theory (32 bits times 33 MHz), but various overheads made that impractical. It wasn't until wider PCI (64-bit) or/and faster PCI (66 MHz) that true wirespeed gigabit Ethernet was possible.

Also, with PCI, all the slots were shared on the same bus, so other devices impacted yours. This was especially difficult when building firewalls, routers, or IPS applications that needed to both transmit and receive. Luckily, motherboards started supporting multiple independent PCI buses. Still, PCI was still single-plexed, meaning it couldn't transfer in both directions at the same time.

Virtually all these concerns have gone away now. Even a single lane of PCIe 1.0 is 2 Gbps, bidirectional, with more than enough bandwidth to handle sending and receiving at full 1 Gbps.

The early Intel 1 Gbps card had only 256 descriptors. Timing was tight enough that at full bandwidth; there wasn't enough time to process packets before the ring buffer would fill up. With BlackICE, we solved this by allocating an effective ring buffer of several thousand descriptors. Then, when packets arrived, we replaced the existing descriptors with new descriptors from the preallocated set. We used two CPUs, one dedicated to running the user-mode driver doing this, and another reading and processing packets from the large virtual ring buffer. I mention this trick because, at the time, Intel engineers told us it wasn't possible to capture packets at wirespeed, and we were able to prove them wrong.

Historically, and often today, the reality is that few hardware vendors test their hardware at maximum speed. Since operating systems can't handle it, they don't test for it. That makes writing drivers for practical hardware much harder than it would seem in theory, as driver writers have to overcome bugs in the hardware.

Today, custom drivers are common. Back in the day, they were black magic.

## Core Concept

In 1998, I created BlackICE, an IDS/IPS using a custom driver. A frequent question at the time was why we didn't write it on Linux, or even BSD, which everyone knew was faster. In particular, some papers at the time "proved" that the BSD networking was the fastest.

[33] `unzip pocorgtfo15.pdf flexsc-osdi10.pdf`


This bothered me because I was unable to explain the core concept. If we are completely bypassing the operating system, then the operating system doesn't matter. As the graphs show, Masscan spends no time in the operating system. Given the same version of GCC, and the same hardware, it'll run at nearly identical speed, regardless if the operating system is Windows, Linux, or BSD. It's like any other CPU-bound (rather than OS-bound) task.

Yet, people couldn't appreciate this. They knew in their hearts that some operating system was better, and couldn't see the concept of bypassing it.

BlackICE used poll mode, instead of interrupts, so it didn't lock up under high packet rates. Now, with NAPI, and poll-mode drivers like `PF_RING`, it's something everyone can play with and understand. Back then, it was some weird black magic that people refused to believe actually worked. My 11-inch laptop computer happened to use 3Com's 3c905 chip, the only 100 Mbps card we wrote a driver for. Even after demonstrating it handling the maximum rate of 148,800 packets-per-second, people refused to believe it worked. There's a Defcon video where the presenter claims that this is impossible, that the notebook would literally melt under such a load. Nowadays, cheap notebooks easily handle max 1 Gbps speeds (1,488,000 packets-per-second) using things like `PF_RING`.

In 2003, Gartner came out with a report that software IDS was dead, because it couldn't handle line-rate gigabit Ethernet, and that "hardware" was needed. That was based on experience with Snort, which had no custom drivers available at the time. Even when customers explained to Gartner they were successfully using our product at line rate, they refused to believe.

More interesting was the customers who tested our software product side-by-side with "hardware" competitors in the lab, and found our product faster. They still bought the competitors', because of FUD. Nobody got fired for buying a hardware product that turned out to be slow.

Even today, discussions of these drivers still get questions like "What about Endace?" Endace builds custom cards with FPGAs to accelerate processing. This doesn't apply. The overhead for Masscan using `PF_RING` is nearly zero, and would have the identical overhead working with an Endace card, also near zero. The FPGA doesn't reach outside the card and somehow make Masscan's code faster.

Yes, Endace does have some advantages. You can push filters to card, so that fewer packets arrive in a system. This is needed in some networks. However, most people use Endace for things that `PF_RING` would solve just fine, because they believe in the power of hardware.

Finally, the same sorts of prejudices exist with kernel code. Programmers are indoctrinated to believe code runs faster in the kernel, which is not true. The reason you push stuff into the kernel is to avoid the kernel/user transition. There's otherwise no inherent advantage. Pushing things like the driver to user mode is just doing the same thing, avoiding the kernel/user transition. Indeed, that's all micorokernels are, operating systems that aggressively push subsystems outside the kernel.

## Several Drivers to Choose From

Masscan uses `PF_RING` because of compile dependencies—there is no actual dependency. You compile Masscan without any dependency on `PF_RING`, yet that compiled code will go hunt for the `pfring.so` library and dynamically load it. Thus, in the replication instructions, I have you compile Masscan first, and `PF_RING` second.

But there are two other options of note.

Intel has a system called DPDK, the Data-Plane Development kit. It contains not only a user-mode driver similar to `PF_RING`, but a whole toolkit to solve other problems, like multi-CPU synchronization and multi-socket NUMA memory handling. It's a real awesome toolkit. However, it's also an enormous dependency for code. That's why Masscan uses `PF_RING`—it's an optional feature that most users will never see. Had I used DPDK, I would've forced users into dependency hell trying to build a massive toolkit for my little application.

Another option is `netmap`. This is a kernel-mode driver that is otherwise identical to the user-mode stuff. It memory maps the packet buffers in user space, so it's truly zero copy. It also disconnects the driver from the network stack, and gives exclusive access to the application, so there's no allocation and freeing of `sk_buff`s. It batches multiple reads and writes with a single system call, amortizing the cost of system calls across many packets.

The great thing about `netmap` is that it's built into the latest Linux kernels. Assuming you have Intel Ethernet, or even a Realtek Gigabit card, it should work immediately with no special software. I haven't gotten around to adding this to Masscan, but the overhead should be comparable to `PF_RING`—despite being tainted with evil kernel-mode code.

## Some notes on IDS design

One place to use these "user-mode no-interrupt zero-copy ring-buffer" drivers is with a network intrusion detection system, or even an inline version called and intrusion prevention system.

None of the existing open-source IDS projects (Snort, Bro, Suricata) are really designed for speed. They were written using `libpcap` where, at high speed, the kernel consumed most of the CPU power. As a consequence, there were only so much performance improvements that could be made before it wasn't worth it. Optimizations that made the software infinitely fast would still not even double the practical performance of the IDS, because the kernel would be eating up all the time.

But, with near zero overhead in the drivers, some interesting optimizations become worthwhile.

One problem with the Snort IDS is how it does TCP reassembly. It must copy packets into the same buffer in order to perform regex searches. This adds two things which we know to be bad: memory allocations and memory copies.

An alternative is to not do this, to neither do regex as the basis of signatures, nor do reassembly.

This approach is demonstrated in Masscan in several places. Masscan can establish a TCP connection and interact with the service. When it needs to search for patterns, instead of a regex it uses an Aho-Corasick (AC) pattern matcher. Whereas a normal regex needs to have a complete buffer, so that it can do back tracking, an AC pattern matcher does not. It accepts input a sequence of fragments, saving the state of the search at the end of one fragment and continuing at the start of the next fragment.

This has the same practical ability to search a TCP stream, but without the need to "reassemble" fragments, allocate memory, or do memory copies.

In abstract computer science terms, this is the tradeoff between NFAs (non-deterministic finite automata) which can consume a lot of CPU power, and DFAs (deterministic finite automata), which consume a fixed amount of CPU power, but at the expense of using a lot of memory for the tables it builds.

Another thing you'll see in Masscan is protocol decoders based on state machines. Again, instead of reassembling packets, the protocol decoder saves state at the end of one fragment and continues with that state at the start of the next. An example of this is the X.509 parser, `proto-x509.c`. The unit test calls this two ways, one with an entire certificate to be parsed, and one where the bytes are processed one at a time, as if they had arrived in fragments over TCP.

Such state-machine parsers are really weird, but by avoiding memory allocations and copies, they become really fast at high network speeds. It's a difficult optimization to make the code that would add little value when using kernel mode drivers, but becomes an important way of building an IDS if using these zero-overhead drivers.

— — — — — — — — — —

The kernel is a lie.