Network Function Virtualization using Native Client

Shivam Handa
(Dated: May 15, 2017)

I. ABSTRACT

Network Functions like firewalls, filters are built upon specialized hardware built specially to run each function. This is done due to performance reasons. These devices lack customization options and implementing new functions require to build new hardware devices.
In recent years new NICs have appeared which provide high speed network access to commodity machines. These NICs have made implementing these functions on commodity hardware possible. But currently how commodity OS are built, exploiting this speed require you to build these functions within the OS. Running multiple untrusted functions require you to run them on VMs which have performance penalties.
An alternate choice is to run these functions as processes and map device driver memory to userspace so that we dont have to go to kernel for every packet. But we lose all security guarantees in this case.
We provide a solution to the above problem by protecting device memory using native client. We also extend native client such that zero coping takes place while sending and receiving packets.

II. NON ZERO COPY IDEA

We use e1000 device driver for this case. E1000 compatible devices are DMA devices and require no privileged instructions to send and receive packets. The device can be controlled by just reading and writing on device memory.
These devices have a send ring and a receive ring (Queues). For the send queue the driver controls the tail of the queue and the device controls the head of the queue. To send a packet we point the tail slot to our send buffer and increase the tail by 1. Device will see that there is a packet to be sent and will send the packet and update the head by 1.
For receive queue the driver controls the head and the device controls the tail. The driver first maps every slot with a buffer address. When a packet comes in the device writes the packet to he buffer pointed by the header slot and increases the head by 1. The important thing here is that buffer in which it copies has to be decided before the packet is received by the device. This become important as we need a trusted code which can give the packet correctly to the process which is allowed to receive it.
Native client provides us a sandbox to run similar to OS-process boundary. The device memory and code is part of the trusted code and the untrusted code can make syscall-like calls to use the device. But since the sandbox's memory is fixed, the trusted code has to copy receive buffer data to its untrusted memory and for safety reasons have to copy send buffer out of its memory.

III. ZERO COPY IDEA

This solution is only for the x86-64 version of native client. It works differently from the 32 bit version we read in our class. This is due to the fact that segment registers are now not supported by all processors which was the core of the sandbox for 32 bit version.
x86-64 introduced new 64 bit registers (r8 - r15) and 64 bit extensions to previous registers (rax for eax). Whenever you do a 32-bit operation (like addl %eax %ebx) all the higher bits of that register is cleared out.
It also introduces relative addressing i.e. address can be written in the form $base + index * scale$.
64-bit native client only accepts 32-bit code (i.e. all operations are 32-bit non privileged code). It makes the register r15 and replaces ever memory operation with relative addressing for example $movl * \%eax, \%eax$ is replaced by $movl(r15, rax, 1), \%eax$ (This reads $r15 + rax * 1$ location). Since we are only doing 32-bit operations on eax registers its top bits will be zero. Thus by changing eax , untrusted code can index $r15$ to $r15$ + 4GB space. If we put $r15$ as the base of the sandbox the code will run correctly. (I am jumping over details of esp, ebp, eip which are also required to be correct.)
The Zero copy idea is simple (I am brushing details here). What we do is allow our code to do relative addressing by not only using $r15$ register but also other $r8 - r14$ registers. Let me take $r14$, based on the value of $r14$, the code can index $r14$ to $r14$ + 4GB space. If we keep only one buffer in each 4 GB space, we can at runtime change the value of $r14$(only trusted code can change this) and change which buffer our code can index.
We use the above concept to make zero copy happen. (Again I am brushing over details here).
As a part of the implementation I had to change the native client verifier to allow operations like above.

IV. LIES

There is currently a problem with current code (not the idea). I explained above that I have to allocate one buffer in each 4GB space. What we also need is that that to be pinned (because we dont want OS to swap them out and we need physical address for our driver rings.) This can be done easily (allow kernel module for our device driver

to allocate them in which case they are not swapped out and we can map them to userspace). The problem is right now mmap implementation of the OS doesnot allow me to map one page in 4GB space. This can be circumvented by bootstrapping the loading of native client (Currently this is how native client does it for its own code right now for the 4GB sandbox space). And I will also have to change the malloc implementation for native client such that it never allocates in these spaces. Currently, all the buffers are in one continuous space. (I didnt have the courage in me to get into it.)
This was meant to be a proof of concept something like this can be done, which I feel this succeeds in doing that.

V. CODE LOCATION

The code runs on 64-bit VM and you still need to make changes to VM spec so that you have correct device working and other details. Code is at `https://drive.google.com/file/d/0B1qVwLBrF61ITzRjdGZfTE5Eczg/view?usp=sharing`