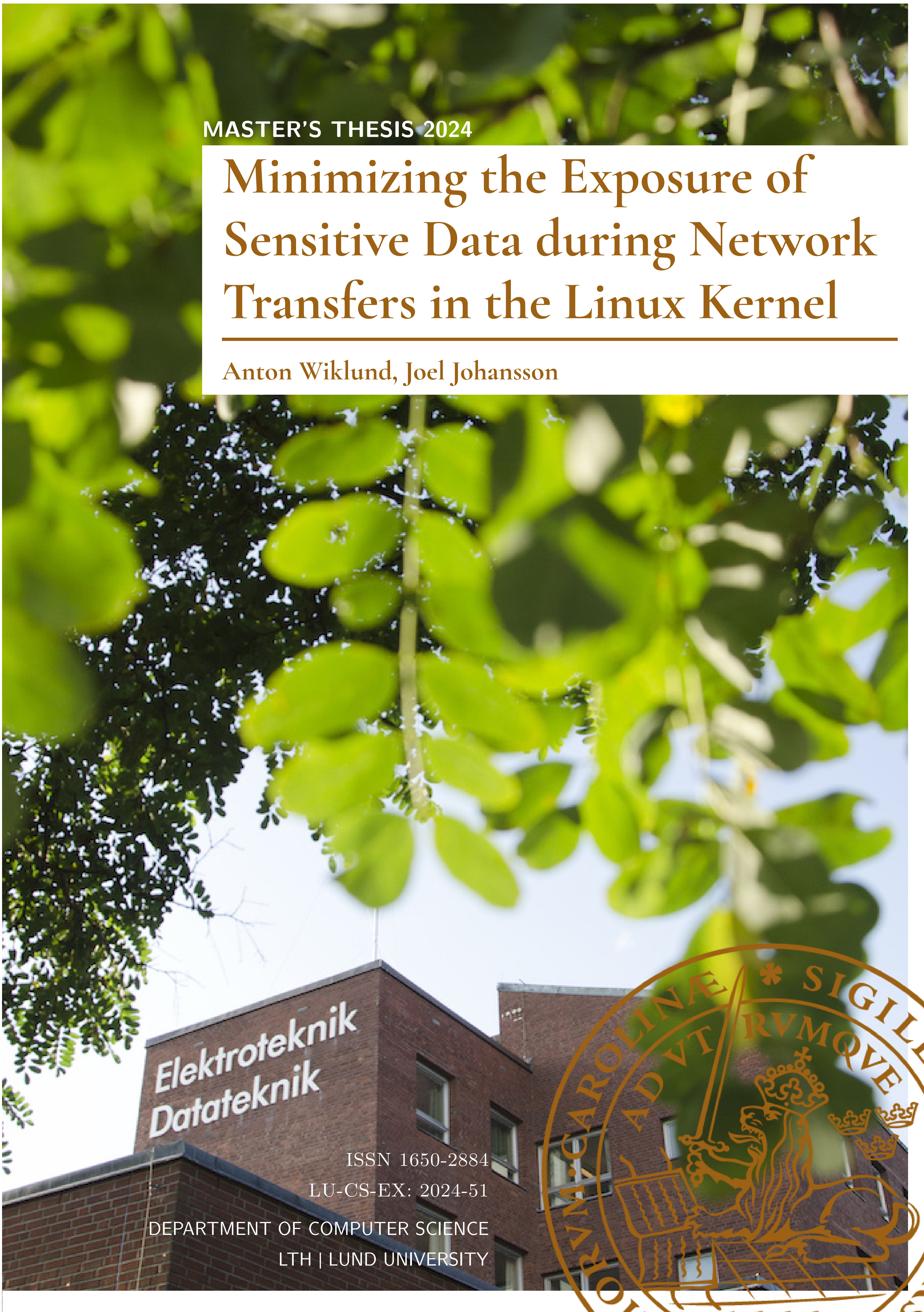

MASTER'S THESIS 2024

Minimizing the Exposure of Sensitive Data during Network Transfers in the Linux Kernel

Anton Wiklund, Joel Johansson

ISSN 1650-2884
LU-CS-EX: 2024-51

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-51

Minimizing the Exposure of Sensitive Data during Network Transfers in the Linux Kernel

Minimera exponeringen av känslig data för nätverksöverföringar i Linux kärnan

Anton Wiklund, Joel Johansson

Minimizing the Exposure of Sensitive Data during Network Transfers in the Linux Kernel

(Implementations of zeroing and their performance penalties)

Anton Wiklund
antonwiklund199@gmail.com

Joel Johansson
joel.k.johanson@gmail.com

August 16, 2024

Master's thesis work carried out at Asperiq AB.

Supervisors: Reine Johansson, reine.johansson@asperiq.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This master thesis focuses on minimizing the exposure of sensitive data in the Linux kernel during network transfers and how much such a solution would affect performance. After studying the network stack and network drivers, the kernel was modified to overwrite packet data with zeroes before freeing it. Consequently, we could halve the amount of exposed data during transmissions, and after the transmission, no data would be left in memory. This solution came with some performance costs as using the regular memset function to zero increased the cache miss rate by 4% and CPU utilization by 1.5%. When the CPU and memory were under high load, the CPU utilization increased by around 4%. Using non-temporal stores instead, the cache miss rate was unaffected and used less of the CPU than memset when the system was under pressure, roughly 2.5% more than no zeroing. However, the CPU usage was slightly higher than memset, around 0.5%, when the system was under no pressure.

Keywords: sk_buff, NIC, Linux kernel, network stack, NAPI, memory management, zeroing, cache pollution

Acknowledgements

We would like to thank Jonas Skeppstedt for his help and feedback on our work.

We would also like to thank Reine Johansson for his help and guidance throughout the project.

Thanks also to Martin Ågren, Marcel Tovar, and the rest of the Asperiq team for their help and support.

Contents

Chapter 1

Introduction

The Linux kernel, like any other large software project, is prone to bugs that malicious actors can exploit. The question is often not if but when bugs and exploits will be discovered; therefore, a good approach is to try to limit the damage they can do [6]. There have been numerous security vulnerabilities during the years that have allowed an attacker to read memory they are not supposed to. In recent years, attacks such as Meltdown and Spectre have exposed vulnerabilities in modern hardware using out-of-order execution and branch prediction and allowed user processes to read arbitrary kernel memory on most modern operating systems, including Linux. While mitigations exist, these are still a threat, and new forms have been discovered. An example is RETBLEED, a speculative execution attack exploiting the ret instruction rather than jmp, as in the original Spectre attack, on Intel and AMD processors that can leak arbitrary kernel memory [14].

Threats such as these, which allow the reading of protected memory, can be a genuine concern for a system that handles sensitive information, as the information at some point is stored in the memory in cleartext. Therefore, it would be preferable to be able to guarantee that sensitive data is in memory as short time as possible. This is already done to some extent in the Linux kernel. There are multiple places where kfree_sensitive(), which clears the memory before freeing it, is used rather than kfree(), for example, when handling various encryption keys. However, this is not done for network transfers and packet data. This is because clearing memory no longer in use comes with performance penalties relating to CPU workload and memory/caching. The performance aspect of clearing memory is actively worked on in the kernel [8][7]. It will be especially noticeable for frequently allocated and deallocated memory, such as packet buffers. However, for a system where the data in the network packets is sensitive, this might be an acceptable trade-off.

1.1 Problem definition

Our work aims to minimize the time that sensitive data is exposed in memory during network transfers in the Linux kernel while minimizing the impact on performance. We focus especially on the case where a system acts as a router and forwards packets between network interfaces.

1.1.1 Research questions

- How can the network stack in the Linux kernel be modified to clear packet data after it is no longer in use?
- What are the implications on performance (throughput, CPU load, cache performance) of clearing packet data from memory after it is no longer in use?
- How much less data is exposed in memory by clearing the packet data from memory after it is no longer in use?

The overall question is, then, given the performance and effectiveness, is it worth clearing packet data in memory after it is no longer in use?

1.1.2 Related work

Chow et al. [4] explored the topic of secure deallocation, meaning that the data in memory was cleared directly at deallocation or within a short time. A part of their work focused on the Linux kernel, and they modified the slab and page allocators and deallocators in the kernel to clear the memory at deallocation. The way this was implemented was pages that contained sensitive data, e.g., network data, were marked as "dirty" when freed and put into a special pool. These pages would then be zeroed either when reallocated or by a kernel thread within some time. They tested this modified kernel with a networking workload and saw no performance differences neither in CPU usage, latency, or bandwidth. Another interesting takeaway from their work is the fact that they found that even if memory with sensitive data was reallocated, it was common for only part of that data to be overwritten, so there were "holes" left of sensitive data that could reside in memory from minutes up to days.

The idea of clearing memory after it is deallocated, or as it is also called, memory sanitization, is not new in the kernel and has been up on the table before. The previous approach mentioned used by Chow et al. was based on a series of patches by Christopher Lameter relating to pre-zeroing allocated pages similarly with a kernel daemon called scrubd [5]. There are also specialized hardened kernels, such as the Pax/GRSecurity one that has an added kernel option *CONFIG_PAX_MEMORY_SANITIZE*, that will add zeroing with every deallocation. Patches adding the same kernel option and features have been submitted and discussed multiple times [1][3][9]. But nothing has been merged to the mainline kernel, the reason for this is the concern for performance penalties and more complicated code for a feature that is not needed and appropriate for the majority of use cases. Also, the feature is not something that has been prioritized, and the efforts mentioned earlier have all petered out and been abandoned. This means that there is no easy way to enable memory sanitization, and this is what has motivated our work, adding an easier, less intrusive way to clear memory for network transfers.

The main difference between our work and the previous work mentioned is that we focus only on the networking part and clearing packet data from memory rather than clearing all memory. This means that instead of modifying the memory management systems in the kernel, the allocators, and deallocators, we modify the particular functions used by the networking subsystem for deallocations. On one hand, this leads to less added code since the memory management system does not have to be rewritten and changed. Our implementation will be less susceptible to changes in this system and have an easier time being applied to newer kernel versions. But on the other hand, we will be more susceptible to changes in the networking part of the kernel.

The previous approaches will still work for clearing packet data since they clear a page whenever it is freed or a short time later; the difference here is that our implementation will zero the data directly when it is deallocated. This is relevant since packet data is usually stored in page fragments (part of a page). Multiple page fragments belonging to different packets might point to the same page, meaning that even if a packet is done and freed, its data will stay in memory and not be cleared until all of the other packets with their corresponding page fragments that point to the same page are freed. We found that this could be an issue when a page fragment is pre-allocated for data to be written to it at a later time by a network card; then, it could block another page from being freed that other packets point to.

1.1.3 Contribution

Our work further explores the possibility of decreasing the lifespan of sensitive data in network packets, i.e., data stored as cleartext in memory. We examine the possibility of mitigating sensitive data leaks in the kernel and to what scale such mitigations would affect performance. Developers will gain a deeper understanding of how methods to decrease the lifespan of sensitive data during network transfers work and possibly apply them to other parts of the kernel. Linux-based system developers will also gain knowledge of ways to enhance security in their systems and whether or not it is sensible to implement. Our aim is also to provide a detailed description of memory management in the Linux networking stack.

1.1.4 Distribution of work

Most of the work has been done by both students together with pair programming. Ideas and implementation choices have been discussed among both students. However, Anton has focused more on the non-temporal memset, while Joel has focused more on gathering and visualizing the performance metrics. Regarding the report, Anton has focused more on the method, while Joel has focused more on the results. The remaining work has been divided evenly; one student wrote a section and the other reviewed it.

Chapter 2
Background

This chapter provides an overview of key concepts such as Network Interface Cards (NICs), network drivers, socket buffers, and memory management mechanisms within the Linux kernel. Understanding the fundamental components and processes of the Linux networking stack is essential to the modifications and improvements proposed in our research.

2.1 NICs and network drivers

A network interface card (NIC) is a piece of hardware that connects a computer to a network. It allows the computer to receive and transmit network packets. They are usually connected to the rest of the computer (CPU and memory) through a PCIE (Peripheral Component Interconnect Express) bus. The CPU communicates with the NIC via register writes and interrupts, and the packets are usually put in a ring buffer that the NIC can read from and write to.

A ring buffer is a circular buffer that contains descriptors. Each descriptor contains some metadata and a pointer to a memory buffer. To increase performance, the ring buffer and associated data buffers are DMA (Direct Memory Access) mapped on modern NICs. This means the NIC can read and write directly to the memory via a DMA controller without going through the CPU.

A network driver is the software that handles communicating with the NIC. In Linux, each network driver defines a net_device_ops struct. This struct defines various operations that each network driver should be able to perform, for example, ndo_start_xmit(), which is the function used for sending a packet with the NIC. Drivers that use the NAPI (New API) scheduler also define a napi_struct, which contains device parameters and data used for scheduling and defines a poll() method. The poll() method tries to read any new packets received by the NIC.

The NIC, along with the drivers, serves as both the entry and exit point for processing a network packet. For our work, it is important to understand how the packet data gets written to memory and the conditions under which it is freed. The following paragraphs will give an overview of the key steps involved in the transmission and reception processes.

A typical transmission process for a NIC can be seen in figure 2.1. This varies between NICs, but in general, the steps are:

1. An upper-layer protocol handler creates a packet and adds headers, etc. The routing rules then decide what interface this packet is supposed to go to. The packet is then scheduled for transmission on the network device corresponding to that interface and put in a queue. After this, the packet is dequeued, and **ndo_start_xmit()**, defined in the network driver, is called on the packet.

2. The driver DMA maps the data corresponding to the packet and takes enough TX descriptors to fit the full packet. The TX descriptors are filled with pointers to the DMA-mapped data.

3. The driver signals that new TX descriptors are ready for transmission.

4. The NIC reads the new TX descriptors, and DMA reads the data pointed at by the descriptors into an internal memory buffer. The read packet is then sent out.

5. The NIC raises an interrupt that signals that descriptors have been read and are available for the CPU to write to. For our purpose, this means that the packets are ready to be zeroed.

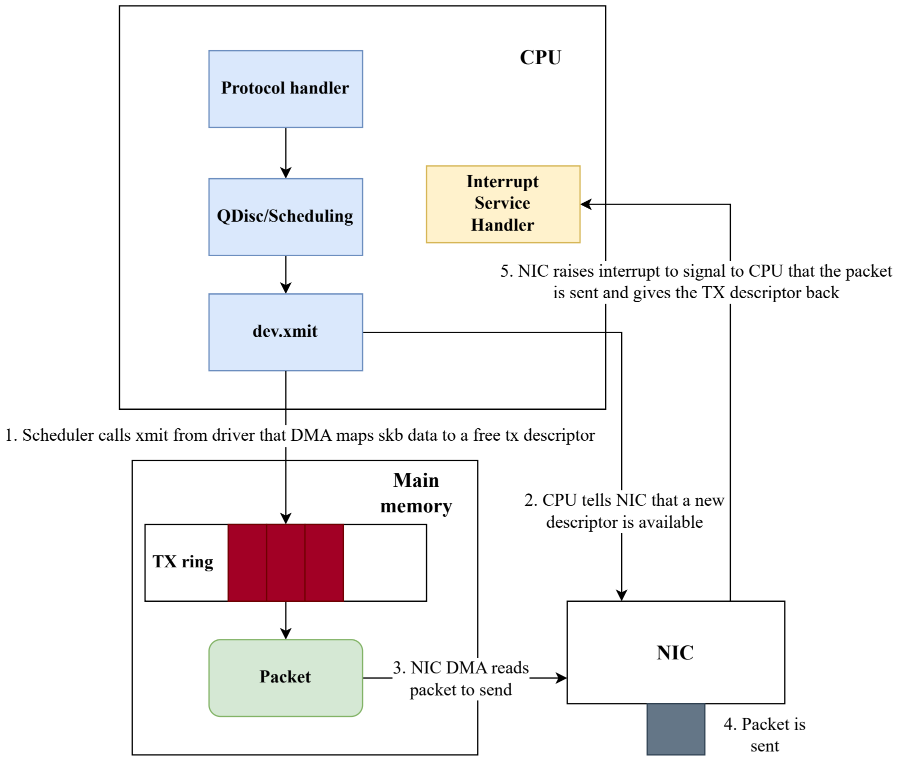
CPU
Protocol handler
QDisc/Scheduling
dev.xmit
Interrupt Service Handler
5. NIC raises interrupt to signal to CPU that the packet is sent and gives the TX descriptor back
1. Scheduler calls xmit from driver that DMA maps skb data to a free tx descriptor
Main memory
TX ring
Packet
2. CPU tells NIC that a new descriptor is available
3. NIC DMA reads packet to send
NIC
4. Packet is sent

Figure 2.1: NIC TX Control Flow

A typical reception process for a NIC can be seen in figure 2.2. In general, the steps are:

1. The NIC receives a frame, which it then writes through DMA to the DMA-mapped memory area specified in an RX descriptor.

2. The NIC raises an interrupt, signaling that a new packet has been received. On the first of these interrupts, the driver masks the interrupt, which disables it and calls napi_schedule(), which will start scheduling the NET_RX_SOFTIRQ (more on softirqs and NAPI later).

3. NAPI starts polling the NIC with the poll method set up in the driver. The poll method reads available RX descriptors of packets that have been received and creates sk_buff structs representing the packets that are then sent up through the network stack via napi_gro_receive() (or similar methods). Some drivers will copy the data from the DMA mapping in the RX descriptor into a sk_buff, others create the sk_buff around the data pointer using napi_build_skb() (or a similar method) and then allocate and DMA maps a new memory area to the RX descriptor. Either way, the RX descriptors are then handed back to the NIC.

4. If all available packets in the NIC have been read, the interrupt is enabled again. Otherwise, NAPI will continue polling.

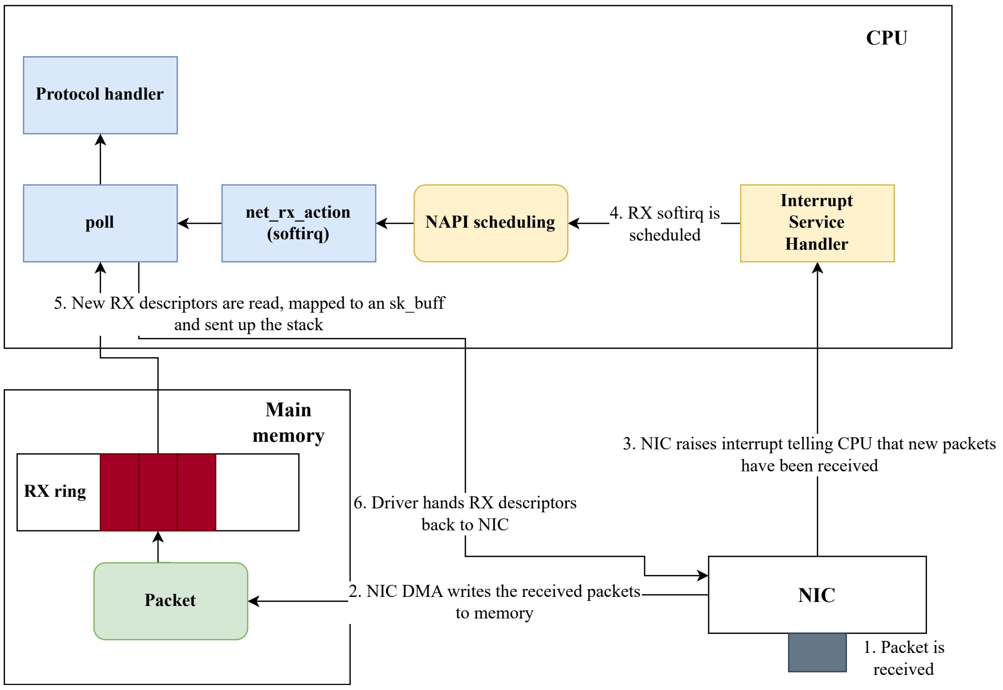
CPU
Protocol handler
poll
net_rx_action (softirq)
NAPI scheduling
4. RX softirq is scheduled
Interrupt Service Handler
5. New RX descriptors are read, mapped to an sk_buff and sent up the stack
Main memory
RX ring
Packet
3. NIC raises interrupt telling CPU that new packets have been received
6. Driver hands RX descriptors back to NIC
2. NIC DMA writes the received packets to memory
NIC
1. Packet is received

Figure 2.2: NIC RX Control Flow

2.2 sk_buff (socket buffer)

A sk_buff, or skb, is the main data structure representing a network packet. Every packet that is transmitted or received is handled by an skb. The data structure holds no packet data, only metadata, and some header fields. Instead, the data is held in associated buffers that the skb points to. The whole sk_buff structure can be seen in listing A.1. When clearing packet data it is important to know what data there is and how that data is stored in order to be able to clear it. For that, we need to go a bit into detail about the sk_buff struct and how it stores the data.

The skb data is divided into two parts:

- Head buffer - also called the linear data or non-paged part.
- Paged data - also called the non-linear data part.

The head buffer itself is divided into two parts. The first part is the data buffer, where the packet data is stored (protocol headers and payload). The second part, stored at the end, is the shared info represented by the struct skb_shared_info. This structure contains data that needs to be shared between different skbs pointing to the same head buffer [2]. The structure can be found in listing A.2.

A sk_buff struct has four pointers pointing to the different locations in the head buffer. These are:

- **sk_buff->head** - points to the start of the head buffer
- **sk_buff->data** - points to the start of the data segment in the head buffer, i.e., the data after the headers.
- **sk_buff->tail** - points to the end of the data segment.
- **sk_buff->end** - points to the end of the head buffer, where the skb_shared_info structure is stored.

The skb_shared_info struct contains fields used to manage the non-linear part of an skb and IP fragments. Some of these fields are:

- **skb_shared_info->frag_list**: Pointer to the first fragment of a chain of IP fragments that this skb is a part of. NULL if the skb is not an IP fragment.
- **skb_shared_info->frags**: Array of pointers to skb_frag_t structures, which represents a page buffer. These page buffers are the non-linear part of the skb data and store packet data that is not in the head buffer.
- **skb_shared_info->nr_frags**: Number of page buffers in **skb_shared_info->frags**.
- **skb_shared_info->dataref**: Number of users of the head buffer that the skb is pointing to.

Each page buffer can be multiple pages or a segment of a single page. These are represented by the skb_frag_t struct, which has three fields:

- **skb_frag_t->bv_page**: First page associated with the page buffer.
- **skb_frag_t->bv_len**: Size in bytes of the page buffer.
- **skb_frag_t->bv_offset**: Start of the page buffer relative to the start of the first page.

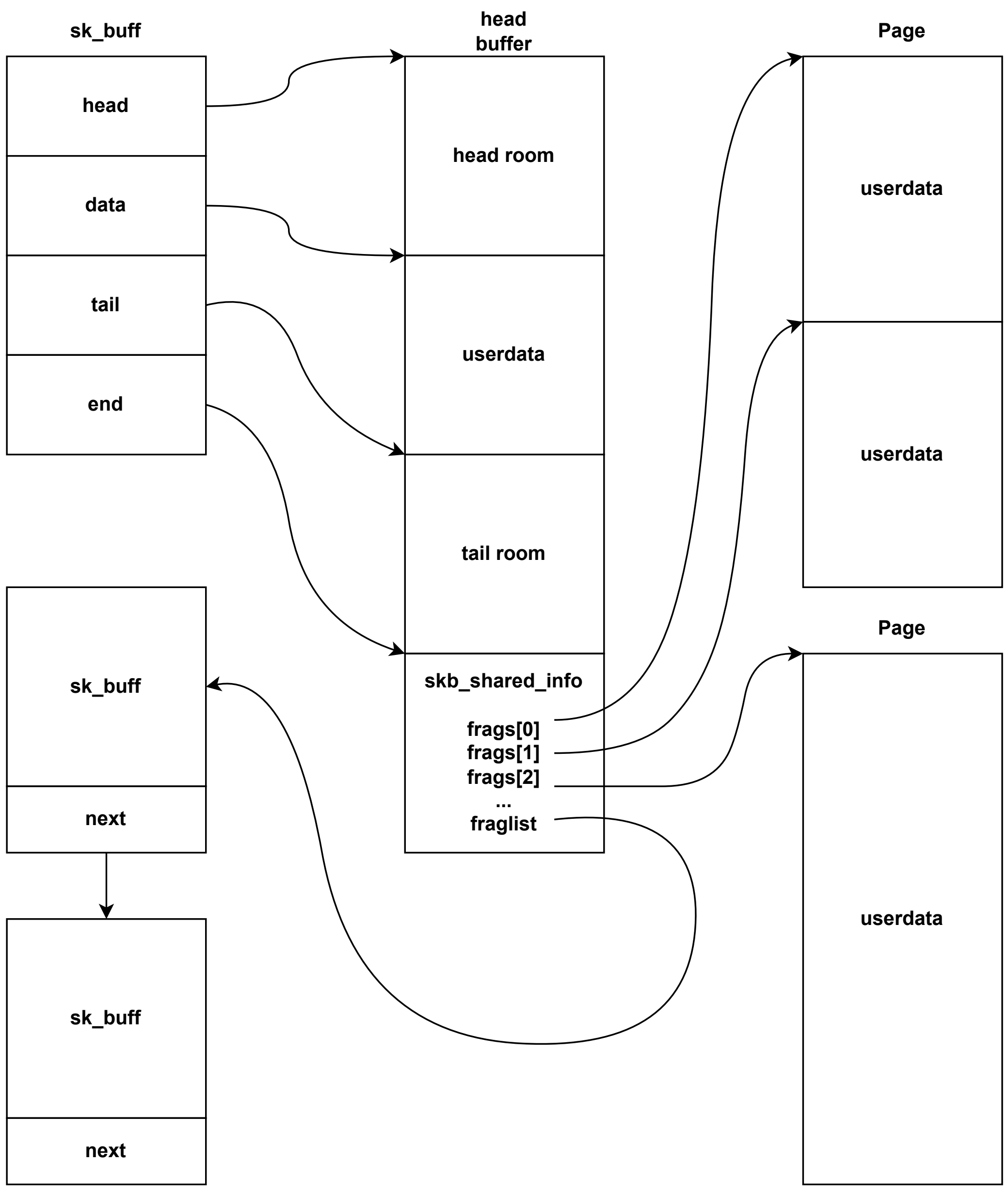

Figure 2.3: Socket buffer

A visual representation of a socket buffer can be seen in figure 2.3.

The length of the actual data in the package is stored in sk_buff->len. This length accounts for both the head buffer and the page buffers. The length of only the data in the page buffers is stored in sk_buff->data_len. To get the length of the data stored in the head buffer, sk_buff->len is subtracted by sk_buff->data_len. There is a function for this called skb_headlen(). So when zeroing the head buffer, we need to zero skb_headlen() bytes from sk_buff->data.

To handle multiple skbs as a list, the sk_buff structure has two fields, next and prev. These fields link together skbs in a doubly linked list. For example, the skbs in skb_shared_info->frag_list are linked together this way. If the skb has a frag_list when freed, those skbs are also freed. This means that we do not have to zero the fragment list when zeroing an skb since the same function that frees the skb frees each fragment as well.

An skb can also be cloned by the function skb_clone(). This function creates a new sk_buff struct but copies the pointers from the original. The clone will then point to the same head buffer and the same skb_shared_info struct, which means that the clone and the original will also share the same paged fragments. In both the clone and the original, the sk_buff->cloned will be set to one, and the dataref field in skb_shared_info will be incremented. If the dataref field is more than 1 when freeing the skb, the data buffers will not be freed and are, in our case, not safe to zero.

In certain cases, you would want an entirely new copy of another skb. The function skb_copy() does this. It copies the entire sk_buff structure of the original and the memory blocks it is pointing at to a new skb. Unlike clones, which share memory, the copy and the original will not share memory blocks, and thus, the data buffers are safe to zero.

The skb_shared_info structure also contains a field called flags. Descriptions of the flags can be found in listing A.3. The SKBFL_SHARED_FRAG is the most important to us and tells us if at least one of the frags is shared, i.e., used somewhere else (vmsplice, send file, etc.), and thus, we should not zero it.

A few functions are defined to move around the pointers of the skb. The head and end pointer should always point to the head and end of the allocated head buffer. However, the data and tail pointers can be moved around to increase or decrease each area of the head buffer.

You can also decrease the size of a page fragment. If you want to discard some data at the beginning of a page fragment, you would add some bytes to the offset and decrease the size by the same amount. To discard something at the end of a page fragment, you simply decrease the size with some bytes.

2.3 Zero-copy

When an application sends data to be transmitted, the kernel usually copies it to a kernel buffer to pass it down the network stack. However, the Linux kernel supports something called zero-copy. Zero-copy means the userspace buffer is not copied to a kernel buffer but passed directly down the network stack. If the kernel later wants to modify that buffer, it would instead need to copy the userspace buffer to a kernel buffer first. If an skb contains zero-copied data, the data buffers are not safe to zero.

2.4 Linux kernel memory management

The fundamental element of Linux memory management is the page [11]. The whole physical memory is split into chunks called pages, usually 4KB or 8KB in size. Each page is represented by the page struct, which is a very complex struct with many different fields and unions depending on how it is used. It contains a reference counter to keep track of the number of users of the page so it can be freed and reused when it is no longer in use.

Pages can be allocated using __alloc_pages() or through wrapper functions, with allocations specified in orders (e.g., zero-order allocation for one page). Memory allocated in bytes rather than pages is managed with kmalloc() and freed using kfree(), similar to the userspace malloc(). To take a new reference to a page, you call get_page(), which increments the reference counter. To dereference a page, you call put_page(), which decrements the reference counter and frees the page if the reference counter has reached zero.

2.5 sk_buff allocation

For allocating the actual sk_buff struct, a slab allocator is used. But the important part for us is how the data buffers are allocated to know how data is written and where it is stored. There are different ways to allocate the data.

The linear part of the skb (skb->head) is either allocated using a page frag cache or by kmalloc(). The field sk_buff->head_frag indicates if the head buffer is allocated as a page fragment or through kmalloc() (1 if the head buffer is a page fragment).

The non-linear part of the sk_buff is allocated differently. The alloc_skb_with_frags() function allocates data in both skb->head and skb_shinfo->frags. Typically, the headers are in the linear part (skb->head), while additional data is in the non-linear part (skb_shinfo->frags). This avoids unnecessary copying, as seen in functions like skb_segment(), which splits large sk_buffs without copying data. Multiple sk_buff instances will then point to the same memory, so when the original sk_buff is freed, its data cannot be zeroed out.

When data buffers are first allocated, and data is later copied to these buffers, both the original data buffer and the sk_buff data buffers need to be zeroed. However, this scenario is rare. More commonly, drivers allocate a page fragment, DMA map it, and place it in an RX descriptor. Upon receiving a packet, the driver unmaps the DMA area and calls build_skb() to construct a new sk_buff around the allocated data. Therefore, once the sk_buff data buffers are zeroed, the NIC's page fragment will also be zeroed.

2.6 sk_buff deallocation

The freeing of sk_buffs is more straightforward than the allocation. There are multiple functions for freeing skbs, but they are mainly used to create different tracepoints and have no functional difference. The two variants are the NAPI variant called from NAPI context and the normal kfree_skb() called from everywhere else. Both of these functions work the same way. They both call skb_release_all() on the skb. This function calls skb_release_head_state(), which frees some structs in the skb. Next, it calls skb_release_data(), which decrements the dataref counter in skb_shared_info, and if it has reached zero or skb->cloned is not set, it frees all of the data in the following steps:

1. Go through each frag in skb_shared_info->frags and call napi_frag_unref() on them. This function will dereference the page (decrement the page counter) associated with the frag. If there are no more references to the page, it is freed.

2. If there is a frag_list call kfree_skb_list_reason() that goes through it and calls kfree_skb() on each of the skbs in it. This means that this skb is part of a fragmented message, and we want to free all of them.

3. Call skb_free_head(), which checks if skb->head_frag is true, meaning that the skb->head has been allocated with page fragments. If it was allocated with page fragments, it dereferences the page, which decrements the page reference counter, and if there are no more references to the page, it is freed. Otherwise, if it was allocated with kmalloc(), it simply calls kfree() on skb->head.

After freeing the data, the sk_buff struct is put back into the appropriate cache.

The most important part for us here is that, ultimately, the freeing of the data in an skb boils down to calling skb_release_data(). There are exceptions to this when manipulating the non-linear part of an skb. In some places, a fragment is simply dropped by calling napi_frag_unref() or skb_frag_unref(). There are also places where the size of a fragment is decreased, which means that a part of the fragment is discarded.

2.7 Non-temporal stores on x86

On x86 processors, which we are testing on, you can do what is called non-temporal stores that write directly to memory without first writing to the cache [10]. Non-temporal instructions are made for writing data that is written once and then never again, i.e., there is no temporal locality. To gain performance, they use write combining, meaning that you write to a buffer, typically consisting of 4-6 cache lines, that is evicted and written to memory at a later time (buffer is full or some time has passed since last write). This means you do not have to wait for the full memory write to complete and that multiple smaller writes can be combined into a larger one. Since they use write combining, the stores are weakly ordered, so you have to add memory fences yourself (SFENCE or MFENCE). The width of the memory bus is a cache line, usually 64 bytes, so writing less causes a partial write that decreases the memory bandwidth. Therefore, partial writes should be avoided.

These instructions are available as part of the Intels SIMD supplementary instruction set, SSE, and SSE2 (Streaming SIMD Extension). This means that they can only be used on 64-bit x86 processors supporting SSE/SSE2. However, most modern x86 processors from both Intel and AMD support these.

Chapter 3

Method

3.1 Implementation

This section gives an overview of our implementation; for more details and a description of the process that led us to our final implementation, see chapter 4.

All methods that free an skb eventually call skb_release_data(). In this method, all of the fragments are dereferenced, and then the skb->head is freed by calling skb_free_head(), so it is in this function we want to zero the data before it is freed. Before each fragment is freed, we zero the memory area that it is pointing to by mapping the page and doing a memset. Then we do the same before the skb->head is freed. If it is a page fragment, we map the page and then do a memset; otherwise, if it was allocated with kmalloc, we simply do a memset from the data pointer to the tail pointer.

To avoid copying data between skbs when you want to transfer data from one to the other, data is usually added to the frags as a page fragment pointing to that data. This means that multiple skbs can point to the same data, causing corruption concerns when freed. For example, if one skb is freed and its data zeroed, another skb pointing to the same data will also have its data zeroed, even if it is still in use. To define whether data should be zeroed or not when a skb is freed, we added new fields to skb_shared_info. Using these fields we can avoid the corruption issue.

We decided to implement zeroing as a sysctl parameter to enable easy switching between zeroing and no zeroing in the kernel. The zeroing can be turned on or off by setting the net.core.skb_zeroing to 0 or 1, respectively.

3.1.1 Different ways of zeroing

One can imagine that writing zeroes to large memory areas that are then freed and not used again would be sub-optimal for cache performance and pollute the cache with useless data. This will be the case when zeroing packet data since we write to the packet data memory and then free it, meaning that it will not be used again until reallocated. As mentioned in 2.7, non-temporal stores are a way to combat this and write directly to the main memory, bypassing the cache and, therefore, avoiding cache pollution. We wanted to test if using these instructions would improve the performance of our zeroing, so we decided to implement zeroing (memset) in three different ways:

- Simple - just a simple for loop going through each byte and setting it to zero. This was used as a baseline to compare to the other more complicated memset implementations.

- Store string - this is the normal memset implementation for x86_64 in the Linux kernel. This implementation was used as a way to compare an optimized memset to our implementation with non-temporal stores to see if using non-temporal stores was worthwhile. More recent x86 CPUs support fast-string operations, referred to as fast-short REP in the kernel. This means that string operations are more optimized and can operate on the string in groups that may include multiple data elements effectively [10]. If the processor supports fast-string operations, i.e., if it is a modern Intel x86 processor, the memset uses the store string instruction (STOSB) with REP, which will repeat the string instruction until done. If the processor does not support fast-string operations, another function that uses regular MOV instructions and loops is used. The processor we tested supports fast-string operations, so the fast-short REP STOSB memset is used. The asm code for the fast-short REP STOSB memset and original memset with a more detailed explanation can be found in B.1 and B.2, respectively. If the elements to zero are known at compile time, the compiler will optimize the call away for memsets smaller than 8192 bytes and inline stores instead. However, the elements are not known at compile time for our use case, so the memset function will always be called.

- NT - uses non-temporal instructions, specifically MOVNTI, which does a non-temporal store from a general-purpose register. This implementation was used to see the impact of cache pollution and to see if using non-temporal instructions would improve performance compared to the regular memset (store string). It is a copy of the original memset implementation (without store string), where all of the MOVQ instructions have been replaced by MOVNTI, and the code for handling the tail (if the end of our storing is not aligned with 8 bytes) has been rewritten to use only use non-temporal instructions. The original implementation used single-byte stores (MOVB) to handle the tail, but there is no non-temporal equivalent, so we have to do an unaligned store instead. This also means the non-temporal implementation only uses NT instructions for memsets larger than 8 bytes. Otherwise, we simply use byte stores. See B.3 for the full assembly code and a more in-depth explanation of the non-temporal memset.

3.2 Setup

Most of our development and testing was done on virtual machines, primarily QEMU. We used this rather than actual hardware because of the faster turnaround time for testing new changes and the added debugging facilities. For instance, QEMU has support for GDB, meaning that you can run the kernel in GDB, add breakpoints, and step through code. It was also possible to adjust machine parameters such as memory size and use different network cards to

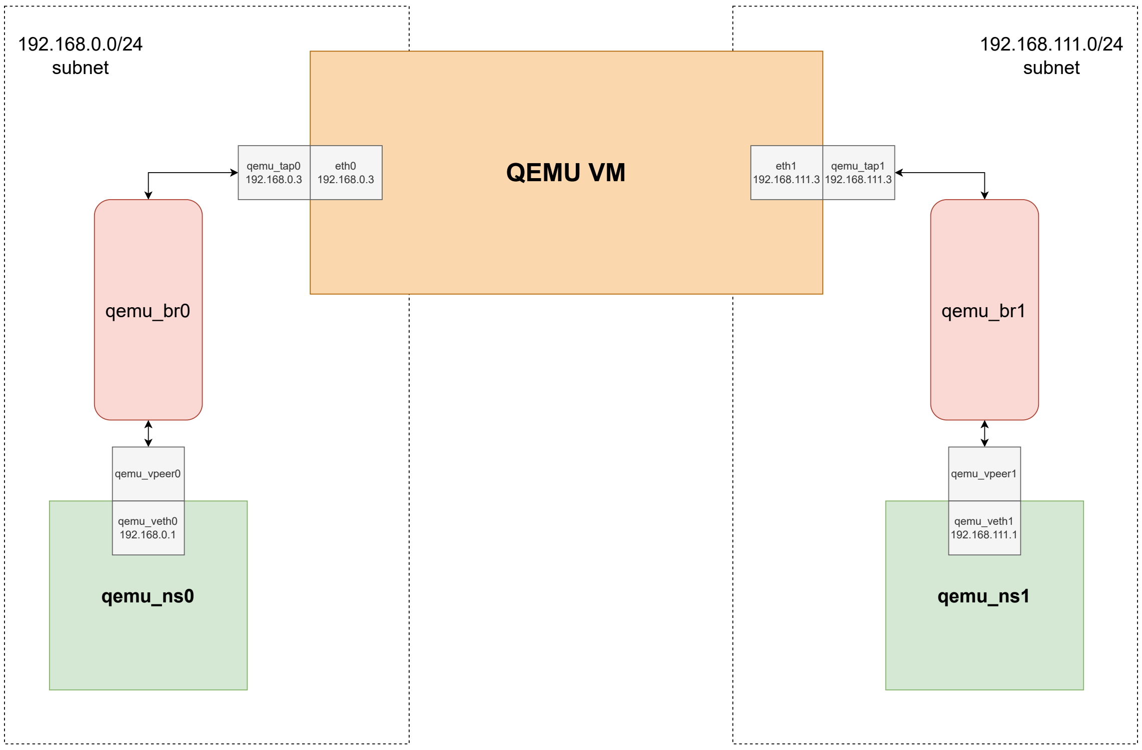

Figure 3.1: QEMU test setup

test more possibilities. We ran QEMU with our kernel, and a minimal filesystem was created with buildroot.

The networking setup we used for QEMU can be seen in figure 3.1. On the virtual machine, we created two virtual network cards. These can emulate real network cards, such as e1000, or be virtual, such as virtio. We created two taps on the host attached to the virtual machine ports (qemu_tap). We also created two separate namespaces (qemu_ns) with a veth (qemu_veth) and corresponding peer (qemu_vpeer). The namespaces were then connected to one of the QEMU taps through a bridge (qemu_br). The interfaces connected to port 0 were given IP addresses in the 192.168.0.0/24 subnet and port 1 in the 192.168.111.0/24 subnet. We set up routing inside the namespaces so that packets sent to the other subnet were routed through the virtual machine. This setup allowed us to both test forwarding /routing through the virtual machine and direct communication from the host to the virtual machine by running commands inside the namespaces. All of the scripts for setting up QEMU, networking, and other programs used during our thesis can be found at https://github.com/antonwiklund99/exjobb.

For testing on actual hardware, we used a minimal live boot Debian image created using debootstrap from Debian stable. We used squashfs for the filesystem to get a small image and added our custom kernel to it. The hardware we were given had multiple RJ45 ports, but we mainly used two ports, each connected to an external computer via a USB-C to Ethernet adapter. The USB-C to Ethernet adapter was a Ugreen USB-C to Ethernet Adapter. The setup can be seen in figure 3.2

The performance testing was carried out on a machine with the following specifications:

- **CPU**: Intel(R) Atom(TM) CPU C3558 @ 2.20GHz (x86_64)

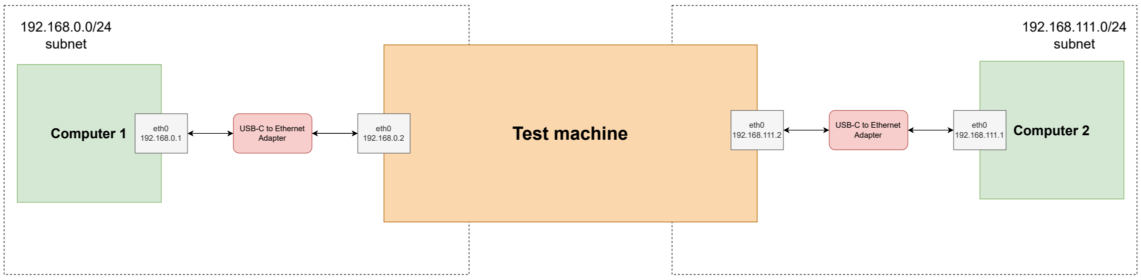

Figure 3.2: Hardware test setup

- **Cores**: 4
- **NICs**: 4x Ethernet Connection X553 1GbE
- **Network driver**: IGB
- **RAM**: 4 GB DDR4 @ 2133 MT/s
- **L1d**: 96 KiB
- **L1i**: 128 KiB
- **L2**: 8 MiB

3.3 Testing

The Linux kernel is a large and complex system, so testing changes can be quite tricky because the kernel handles so many cases. Testing our implementation, therefore, required extensive testing to ensure that the kernel still worked as intended and that memory was zeroed correctly.

3.3.1 Functionality

The easiest way to test if the kernel still works is to simply send packets to it or through it and see that the data you sent is the same as the one you received. We wrote both clients and servers to test this for TCP, UDP, and TLS. All of these worked similarly: the server listens for incoming connections, and the client opens a socket to the server. After the connection is established, the client sends a lot of predefined data (the same string repeated multiple times). The server reads all of this and checks that the data received matches what it expects. We also check the total number of bytes sent versus the number of bytes received to make sure nothing got lost. We ran these tests with both the server and client on the machine with the custom kernel and host. We also ran the tests with the machine acting as a router, forwarding packets from one interface to another by running the server in one namespace/computer and the client on the other.

The Linux kernel self-tests, kselftests, are a suite of tests for different subsystems in the Linux kernel. These tests cover many code paths and different kernel functionality. We used the net self-tests to test that our implementation worked, as it did not break anything for most of the various use cases of the networking code. These tests create internal interfaces and send traffic to the host only, so the tests mentioned earlier were also needed to test the network drivers and interaction with actual network cards.

3.3.2 Zeroing

To determine if our zeroing works, we needed some way to see what is in memory. When using QEMU, this was quite easy since QEMU allows you to dump the entire memory of the virtual machine using the QEMU monitor and the command dump-guest-memory. We used the same server-client setup as mentioned in 3.3.1 with a predefined string as data. After the packets have been sent, we dump the memory and search for the string. We decided to use 4 bytes repeated as the predefined data to avoid false positives but still not miss any remaining data. We chose 4 bytes after testing different lengths and determining the number of false positives, with 3 bytes or less there were always multiple false positives, while with 4 it was much more rare.

For testing on real hardware, we instead used LiME. This is a loadable kernel module that can dump the entire memory over a TCP connection when you load it. We tested with the same client and server on QEMU but had them running on the external computers connected to the machine.

When testing on both QEMU and hardware, we ran both the server and client on the machine and also tested just having the machine forwarding. We ran these with different protocols, such as TCP, UDP, and TLS. On QEMU, we also tried varying the RAM size to verify that it did not influence the behavior of the kernel and that data was zeroed correctly. These tests cover most common cases and machines, however it does guarantee absolute correctness and there might be issues in the zeroing with more uncommon socket options or protocols.

3.4 Performance measurement

For the performance measurements, we ran the server and client on different computers, with the machine with the modified kernel between them acting as a router and forwarding packets from one computer to the other. To measure throughput/bandwidth, we used iperf3, a tool that sends data over TCP or UDP and measures the bandwidth. You can run this tool in both directions or only one. We ran all of our tests in both directions. The reason for measuring the bandwidth is to see if our added zeroing has an impact on the throughput, i.e., the added code slows down the critical path enough that the kernel is not able to put out enough data to transmit at the NIC's maximum bandwidth or if it's unaffected.

When running iperf3, we used various tools to measure the impact on system performance. For measuring cache misses, we used perf, a profiler tool for Linux that uses performance counters to count hardware events, such as cache references and misses. We wanted to measure the cache miss rate in order to see the effects of cache pollution that our zeroing had. A higher cache miss rate indicates that more cache pollution might have occurred.

We used a tool called Netto to measure the CPU utilization of the networking stack. Netto is an eBPF-based monitoring tool that measures how much time is spent in different parts of the networking stack e.g. RX and TX softirq, protocol handlers, etc. [13]. This is an important measurement to see the impact of zeroing, even if the bandwidth is unaffected, we can use this to see how much more time is spent if we use zeroing. Correlating CPU utilization with cache miss rate will also give us more insight into why the different memset implementations behave differently and how the implementation might scale with more ports.

We also wanted to measure performance when the system was under load, which we did using the Linux utility stress. With stress, it is possible to put a load on both the CPUs and memory, so we ran all of the tests, once without stress, once with full CPU load, and once with both full CPU and memory load.

To measure how much data is exposed, i.e., in memory, during transmission, we used the QEMU setup described in 3.2 and the server-client setup as in 3.3.1. The client sent a predefined string from one namespace to a server in another namespace routed through the virtual machine with our modified kernel. Every second, we dumped the entire memory of the virtual machine. We could then estimate how much data is exposed during transmission from these memory dumps by searching through each one of them for the predefined string and counting the number of matches. We used the same predefined string of 4 bytes as in 3.3.2.

To get an idea of what happens with data in the memory after transmission when there is more traffic routed through the QEMU, we did the same memory exposure test. But this time, we set up a bidirectional iperf3 connection right after the sensitive data transmission was completed. We did the test with 10, 100, and 1000 Mbits/sec connections to see the effect of different amounts of data being sent. The reason for doing this is to model a more realistic system where traffic simply does not just stop after one transmission but is sent continuously. This way, we also get an idea of how much of the data is left in memory after transmission and how much of the same memory is reallocated and overwritten as new packet data is received and sent. The different bandwidths were chosen to see the effect of different bitrates on how much data is overwritten, and one can imagine that a higher bitrate would lead to more packet data having to be stored in memory at a time since network cards send packets in bulk and buffer them a bit, thus leading to more memory being reallocated and overwritten.

We also did a similar test on actual hardware, but since dumping memory each second would mean transferring 4GB of data over the network, we could not capture it each second as we did on QEMU. Instead, we first sent data routed through the machine, then ran iperf3 at a specific bitrate for some time, and after this, performed a memory dump using LiME.

Chapter 4
Implementation

In this section, we will describe our implementation and reasoning around the choices we made. The major difference between our implementation and earlier work on memory sanitization is that we zero right before freeing rather than after freeing and zero the specific memory used instead of just full pages when they are released. Instead of modifying the kernel memory management system, we modify the networking code. This makes our solution more precise but more involved/specific. As mentioned in 1.1.2, it also means that we can zero page fragments directly when they are dereferenced rather than having to wait for all other page fragments that point to the same page to be freed before zeroing the data, which in some cases can be an issue.

As mentioned, all methods that free an skb eventually call skb_release_data(). Here, all of the skb_frags in skb->frags are dereferenced, and the data in skb->head is freed, so we want to add our zeroing to this function.

The naive implementation is simply calling memzero_explicit() on the entire skb->head. This will zero the head and add a memory barrier to prevent the compiler from optimizing it away. Then, for the frags, call memzero_page() with the fragment page, offset, and size. However, this creates some problems, especially for the non-linear part, since different frags can be referenced by multiple skbs, and the only reference counter we have is to the actual page and not each fragment. This means that if we zero a fragment that another skb is still using, we will corrupt its data. Also, by calling skb_head_frag_to_page_desc(), an skb->head can be converted into a fragment in another skb, so by zeroing either the created frag or the original head, data can be corrupted.

To keep track of this and avoid zeroing the data that is still in use, we added two new fields to the skb_shared_info struct, one for keeping track of whether we should zero the skb->head or not (skb_shinfo->dont_zero_head) and one for where in the frags we should start zeroing (skb_shinfo->frags_zero_index). Both of these default to zero, so everything is still zeroed if they are not modified. This helped us solve most of the problems. When a method such as skb_segment() transfers a fragment to another skb, we simply set the frags_zero_index such that it would only be zeroed by the new skb and not the original.

But some methods still needed fixing, for example, the method pskb_carve_inside_nonlinear(), which carves off some bytes at the start of the skb. In this method, a new skb is created from the data in the original one from a certain offset, and then the original one is freed. In this case, we do not want to zero everything after a certain index. Instead, we want to zero everything before the newly split data. So, another field was needed that tells us if we should zero everything below or above frags_zero_idx. That field was skb_shinfo->frags_zero_below. The fields we have added to skb_shared_info are then:

- **dont_zero_head** - Indicates whether the head buffer is shared with another sk_buff.
- **frags_zero_index** - Used to only zero up to the n:th, or from the n:th frag in the sk_buff.
- **frags_zero_below** - If true, then zero the frags up to **frags_zero_index**, else zero from it.

The skb_shared_info struct is optimized to fit perfectly into five cache lines if you use the default number of skb->frags (17) and the cache line size is 64 bytes. This optimization minimizes cache misses and enhances performance on systems with 64-byte cache lines, which is the case for most 64-bit systems, and most modern CPUs are 64-bit. So it is a good idea to keep this cache alignment. The cache alignment is destroyed by adding three new fields, and thus, performance is degraded. See listing 4.1 for the structure and alignment. It also caused the IGB driver to break since it depended on the size of skb_shared_info.

Listing 4.1: pahole output for struct skb_shared_info without bit fields

```
struct skb_shared_info {
        __u8                        flags;                 /*    0     1 */
        __u8                        meta_len;              /*    1     1 */
        __u8                        nr_frags;              /*    2     1 */
        __u8                        tx_flags;              /*    3     1 */
        short unsigned int          gso_size;              /*    4     2 */
        short unsigned int          gso_segs;              /*    6     2 */
        struct sk_buff *            frag_list;             /*    8     8 */
        union {
                struct skb_shared_hwtstamps hwtstamps;     /*   16     8 */
                struct xsk_tx_metadata_compl xsk_meta;     /*   16     8 */
        };                                                 /*   16     8 */
        unsigned int                gso_type;              /*   24     4 */
        u32                         tskey;                 /*   28     4 */
        atomic_t                    dataref;               /*   32     4 */
        unsigned int                xdp_frags_size;        /*   36     4 */
        void *                      destructor_arg;        /*   40     8 */
        __u8                        frags_zero_idx;        /*   48     1 */
        bool                        frags_zero_below;      /*   49     1 */
        bool                        dont_zero_head;        /*   50     1 */

        /* XXX 5 bytes hole, try to pack */

        skb_frag_t                  frags[17];             /*   56   272 */

        /* size: 328, cachelines: 6, members: 17 */
        /* sum members: 323, holes: 1, sum holes: 5 */
        /* last cacheline: 8 bytes */
};
```

To fix this, we instead used bitfields and borrowed some bits from the gso_type field. This field is 32 bits and contains flags, but only 18 bits are used. Therefore, we used one bit for the dont_zero_head and one for frags_zero_below. The maximum number of frags that can be set in the kernel config is 45 (CONFIG_MAX_SKB_FRAGS), which fits into 6 bits, and therefore 6 bits were used for the frags_zero_idx. Bitfields come with some performance penalties since the bits need to be masked out when used. But it should be negligible since these fields are rarely used, and gso_type already needs to mask when checking for different flags (bits set). The final struct and its alignment can be seen in listing 4.2.

Listing 4.2: pahole output for final struct skb_shared_info

```
struct skb_shared_info {
        __u8                          flags;                     /*     0      1 */
        __u8                          meta_len;                  /*     1      1 */
        __u8                          nr_frags;                  /*     2      1 */
        __u8                          tx_flags;                  /*     3      1 */
        short unsigned int            gso_size;                  /*     4      2 */
        short unsigned int            gso_segs;                  /*     6      2 */
        struct sk_buff *              frag_list;                 /*     8      8 */
        union {
                struct skb_shared_hwtstamps hwtstamps;           /*    16      8 */
                struct xsk_tx_metadata_compl xsk_meta;           /*    16      8 */
        };                                                       /*    16      8 */
        u32                           gso_type:24;               /*    24: 0   4 */
        u32                           frags_zero_idx:6;          /*    24:24   4 */
        u32                           frags_zero_below:1;        /*    24:30   4 */
        u32                           dont_zero_head:1;          /*    24:31   4 */
        u32                           tskey;                     /*    28      4 */
        atomic_t                      dataref;                   /*    32      4 */
        unsigned int                  xdp_frags_size;            /*    36      4 */
        void *                        destructor_arg;            /*    40      8 */
        skb_frag_t                    frags[17];                 /*    48    272 */

        /* size: 320, cachelines: 5, members: 17 */
};
```

To perform the zeroing of the fragments, we defined four different functions that were used in different cases. Those functions were:

- **skb_zero_frags(struct sk_buff *skb, int from, int to)** - Zero the paged fragments in **skb** from index **from** until index **to**.
- **skb_zero_frag(struct sk_buff *skb, int i)** - Zero the paged fragment in **skb** at index **i**.
- **__skb_frag_zero(skb_frag_t* frag)** - Zero the paged fragment **frag**.
- **skb_zero_frag_off(struct sk_buff *skb, int i, int off, int size)** - Zero **size** at **off** from the paged fragment in **skb** at index **i**. Here, **off** is relative to the starting offset of the fragment.

Both skb_zero_frags() and skb_zero_frag() call __skb_frag_zero() to do the actual zeroing. skb_zero_frag_off() does the zeroing itself since it has to consider offset and size. When zeroing one or more fragments, we first check whether the SKBFL_SHARED_FRAG is set for the sk_buff. The skb_has_shared_frag() function does this for us. If it is set, it means that one or more fragments are shared, and in that case, it is not safe to zero. The function implementations can be found in Appendix C.3.

In skb_release_data, where the fragments are released, we call skb_zero_frags(). If the field frags_zero_below is true, then we call skb_zero_frags() to zero from the first fragment in skb_shared_info->frags up to frags_zero_idx. Otherwise, we call skb_zero_frags() to zero from the fragment at frags_zero_idx and each fragment after it. The modified skb_release_data() function can also be found in Appendix C.1.

The skb_frag_zero() function is used when a single fragment is released. There are two places where this is the case. The first one is in __pskb_pull_tail(), which expands the header of a sk_buff and copies necessary data from the fragments into the skb->head. The fragments pulled into the head are then released, which, in our case, means we have to zero each one. The second place is __pskb_trim_head(), which does the same thing, except that the pulled fragments are not copied to the header but instead discarded.

When only a part of the data in a fragment is pulled, we do not want to zero the whole fragment, just the pulled part. This is where the skb_zero_frag_off() function is used. A few different functions do this, for example, pskb_carve_inside_nonlinear() and skb_segment().

Chapter 5

Results

5.1 Exposed data during transmission

The results of the memory exposure measurements described in 3.4 with QEMU can be seen in figure 5.1. Data was sent with TCP and forwarded through the virtual machine, and the entire memory was dumped and searched through every second. This means that the exposed data in memory is the data in the entire memory, however, since packets are forwarded all of this data will be in network buffers and not copied elsewhere. In the case of no zeroing, there is about twice as much data in the memory as compared to the case where you zero the data. It is also interesting to see that when you do not zero the data, it is left in the memory indefinitely, i.e., until it is overwritten.

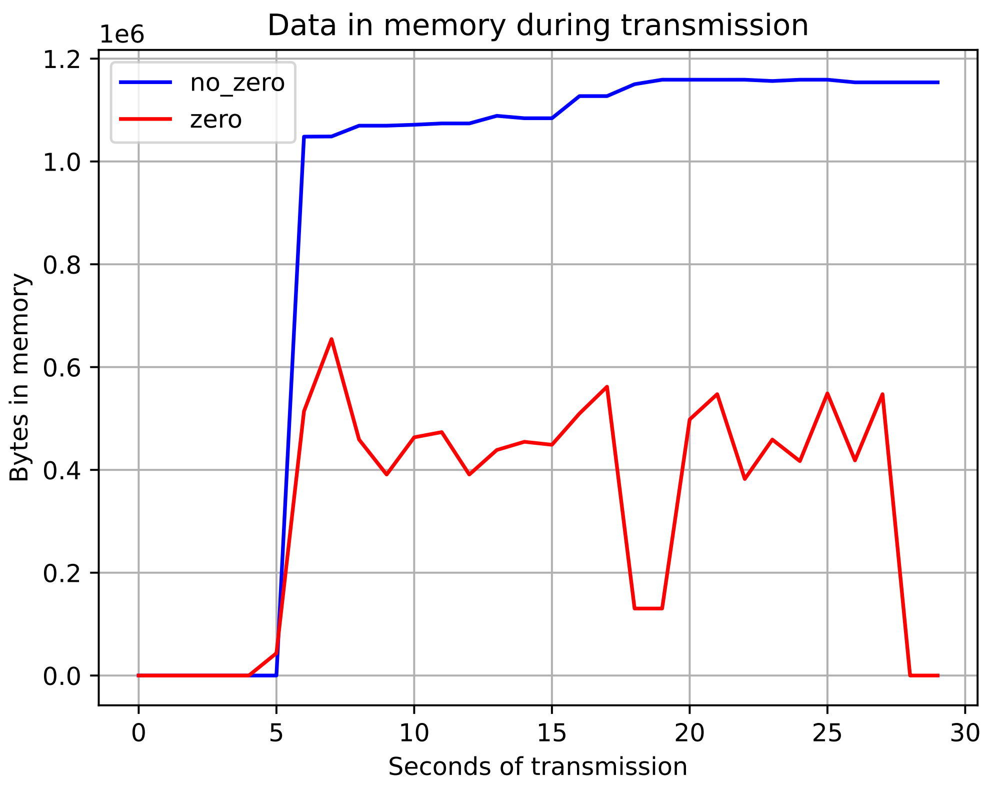

Figure 5.1: Data in memory during transmission

5.2 Exposed data after transmission

The results of the memory exposure test with traffic routed after transmission, described in 3.4, can be found in figure 5.2. Data was sent with TCP forwarded through the virtual machine, and the entire memory was dumped and searched through every second. After the TCP transmission was done, data was sent at a specific bitrate afterward. At around 26 seconds, a significant dip in exposed memory can be seen. That is when the sensitive data transmission stopped, and iperf3 started transmitting. As suspected, the larger bitrate overwrote more data in the memory since it requires more memory for buffering packets, and thus, more of the old packet data will be reallocated and overwritten by the new traffic coming through. This can be seen by the lower amount of remaining data for the higher bitrates. It also seems like some memory with sensitive data is not overwritten by the data iperf3 sent and is left in memory, which can be seen by none of the different bitrates reaching zero in the graph. Interestingly, even after running 1000 Mbits/sec for around 20 seconds through the virtual machine, sensitive data is still exposed in the memory.

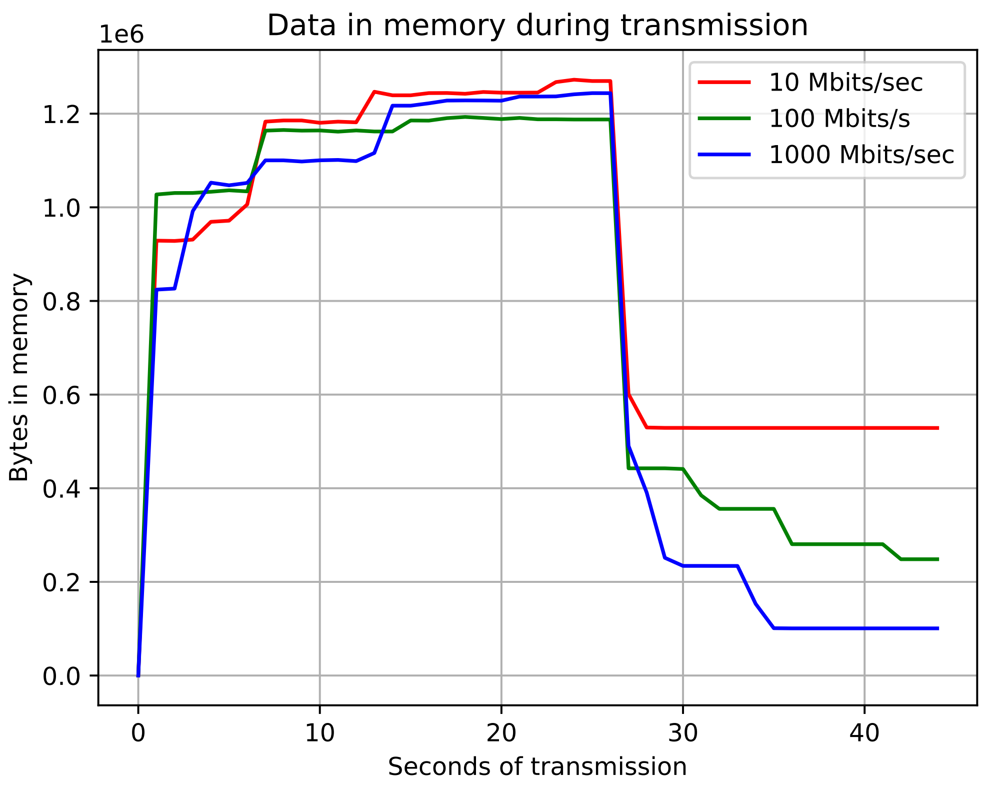

Figure 5.2: Data in memory during and after transmission

We did a similar test on the hardware to see whether it would produce the same results as on the virtual machine. The results are in table 5.1 and are from single runs. This test gave quite inconsistent results. It is hard to say what may happen with the sensitive data after transmission. It might be that the amount of sensitive data still exposed depended on which memory the hardware decided to use for routing the iperf3 connection. None of the sensitive data remained in memory for 10 Mbits/sec for 3 seconds. But for 100 Mbits/sec for 6 seconds, there were still 786820 bytes of sensitive data in the memory.

IPerf3 Bitrate	Iperf3 Time	Remaining data
-	0	845200 B
1 Mbits/sec	1s	1103888 B
1 Mbits/sec	3s	1208148 B
1 Mbits/sec	6s	784752 B
1 Mbits/sec	10s	10636 B
10 Mbits/sec	1s	708 B
10 Mbits/sec	3s	0 B
10 Mbits/sec	6s	824764 B
100 Mbits/sec	1s	888440 B
100 Mbits/sec	3s	777128 B
100 Mbits/sec	6s	786820 B

Table 5.1: Data in memory after transmission on hardware

5.3 Bandwidth

The results from the bandwidth tests using iperf3 results can be found in table 5.2. These measurements are from a single run, running iperf3 for three minutes. Regardless of the implementation and zeroing or no zeroing, the average bandwidth was unaffected.

	TX	RX
No zero	926 Mbits/sec	928 Mbits/sec
Zero simple	926 Mbits/sec	931 Mbits/sec
Zero store string	928 Mbits/sec	930 Mbits/sec
Zero non-temporal	926 Mbits/sec	927 Mbits/sec

Table 5.2: iperf3

We did the same test while maximizing the CPU and memory load. The results are in table 5.3. Again, there were no effects on the average bandwidth.

	TX	RX
No zero	928 Mbits/sec	930 Mbits/sec
Zero simple	928 Mbits/sec	930 Mbits/sec
Zero store string	926 Mbits/sec	931 Mbits/sec
Zero non-temporal	925 Mbits/sec	928 Mbits/sec

Table 5.3: iperf3 (Stress CPU and memory)

5.4 Cache miss rate

The results from measuring cache misses with perf can be found in table 5.4; those were the results of single runs, running for one minute. We ran these tests multiple times but with little variation in the final results. The table shows that looping through each byte in the memory and writing zero to it will have the most significant impact on the miss rate. Using the standard memset function, even though not as great, also affected the miss rate. Possibly because it is more optimized than just looping; however, by a large margin, it has the most cache references. When we used non-temporal stores, the cache references and miss rate barely increased compared to not zeroing at all (within one percent).

	Cache References	Cache Misses	Miss Rate
No zero	1556063965	206439289	13.267%
Zero simple	1993791304	512640484	25.612%
Zero store string	2600751429	456596840	17.556%
Zero non-temporal	1350383971	190084247	14.076%

Table 5.4: Perf

We also recorded cache references and misses when stress-testing the CPU. The results of those tests can be found in table 5.5. Each implementation of zeroing, along with no zeroing, had a slightly higher miss rate when maximizing the CPU load.

	Cache References	Cache Misses	Miss Rate
No zero	1125436296	176940872	15.722%
Zero simple	1724690690	465089172	26.967%
Zero store string	2597597475	504757291	19.432%
Zero non-temporal	1211218000	202488996	16.718%

Table 5.5: Perf (Stress CPU)

In addition, we recorded the cache references and miss rate when both stressing the CPU and memory consumption. The results are in table 5.6. When stressing the memory, we also increased the cache references by more than ten times, so the misses caused by the networking code are drowned out. However, the difference in miss rate is still noticeable, with non-temporal and not zeroing being about the same and store string slightly higher.

	Cache References	Cache Misses	Miss Rate
No zero	26581337550	4843918736	18.223%
Zero simple	25865549215	4884662107	18.885%
Zero store string	26838374613	4953481209	18.457%
Zero non-temporal	26306867500	4801968198	18.254%

Table 5.6: Perf (Stress CPU and memory)

5.5 Networking CPU utilization

The results from the networking CPU utilization measurements using Netto can be found in figure 5.3. These are single runs, running for three minutes. The zero simple implementation had the highest CPU consumption of them all. Even though using non-temporal writes when writing zeroes to the memory had fewer cache references and a lower miss rate, the regular implementation of memset had lower CPU utilization, probably because the non-temporal writes are more time-consuming.

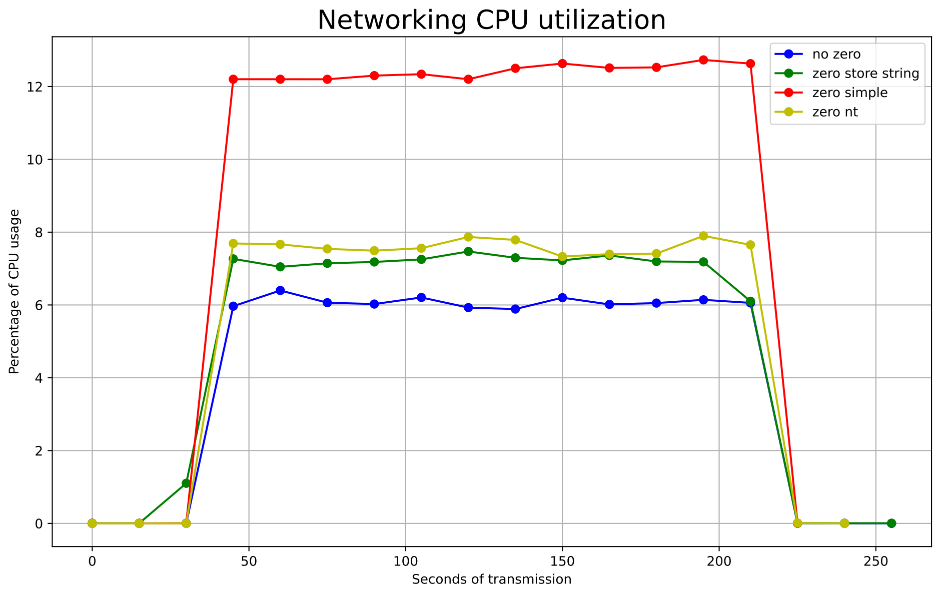

Figure 5.3: Networking CPU utilization

In addition, we measured the CPU utilization of the network stack when maximizing the CPU load. The results can be seen in figure 5.4. Each implementation had slightly less CPU utilization, but their relationship stayed the same.

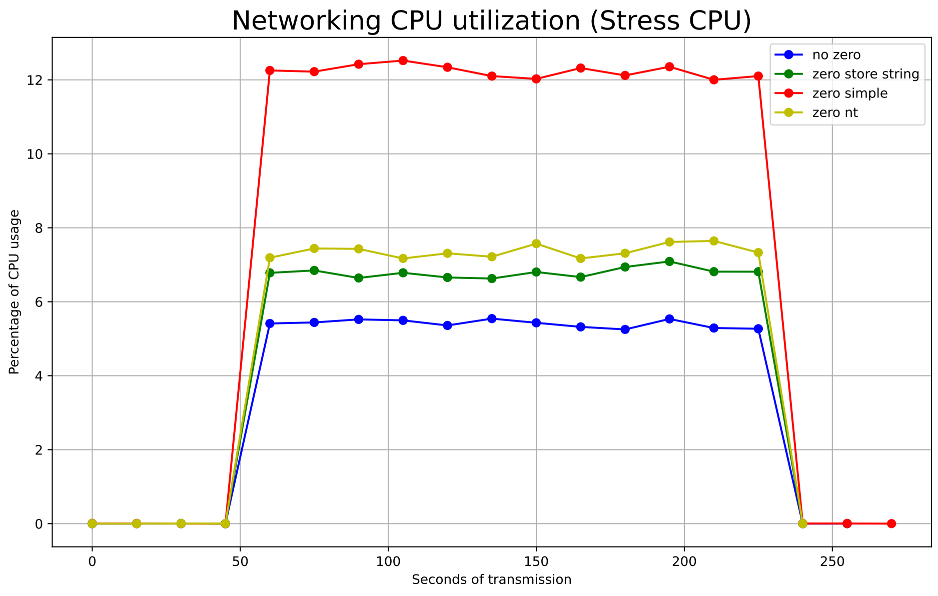

Figure 5.4: Networking CPU utilization (Stress CPU)

We also measured the CPU utilization of the network stack when maximizing the CPU load and the memory consumption. The results can be seen in figure 5.4. When stressing memory consumption as well, the non-temporal memset implementation consumes less of the CPU than the regular memset implementation, possibly because it uses less of the memory, as can be seen by the cache references in table 5.4.

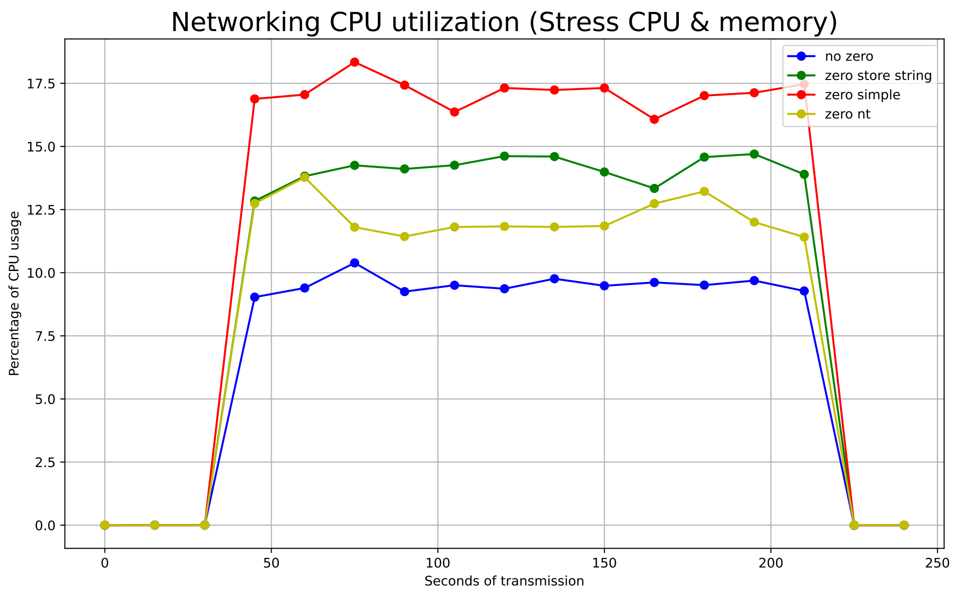

Figure 5.5: Networking CPU utilization (Stress CPU and memory)

Chapter 6
Discussion

6.1 What are the implications of zeroing on performance?

The bandwidth seems unaffected by whether or not we are zeroing or any of the different zeroing implementations. This indicates that the processing power of the CPU is not the bottleneck in the system, even at max throughput. Instead, it is the NICs that can only transmit up to 1000 Mbits/sec. This can also be seen in the CPU utilization, which never reaches above 13% for any of the implementations. This was not unexpected since we tested it on a relatively powerful CPU with only two 1 Gbit network ports. For other systems with less processing power and more or faster network ports, this might be different, and the CPU could limit the throughput. On our hardware, we saw an increase of about 10%, when the total CPU utilization of the network stack is only around 10%, which only equates to 1% total CPU utilization. On a more limited system, where the network stack uses more processing power, the impact of the zeroing on the total CPU usage would then also be larger and have more of an impact, and then it could affect the throughput. However, most of the freeing and then also zeroing is not done in the critical path of the network stack but instead deferred to later, when the CPU is free, so it is hard to predict if this would be true.

The performance metrics that do differ are CPU utilization and cache misses. The store string and non-temporal memset are within 1% of each other, with the store string using slightly less and about 1.5 % more than no zeroing when running with no memory load. The simple memset is around double the CPU utilization, which can be expected since it will simply do a bunch of bytes stores in a loop. However, when running stress with memory load enabled, the NT version performs better than the store string, having up to a 2.5 % difference in utilization. It may seem surprising that the store string slightly outperforms the NT version when there is no memory load. Something similar can be seen in measurements performed by Li et al. [12] concerning whether logging should be cached or not. They discovered that the execution time was worse for non-cached writes compared to normal cached writes, even if the cache miss rate was lower. The reasons they gave for this are that the MOVNTI has more overhead and takes more cycles to execute because of additional data movements and longer write latency to memory than caches. They also mention the issue of partial writes, which is when the size of the data in the write combining buffer (WCB) is less than the size of a cache line when it decides to write to memory. The memory bus and WCB will then be underutilized. This is likely what we are seeing as well. The main reason for using NT instructions is to avoid cache pollution, but if we are not using a lot of memory, this will have less of an impact since cache lines still in use will not have to be kicked out prematurely. The store string will then edge out slightly since the stores have less overhead. This is also likely why the NT version outperforms the store string when there is more memory pressure. There is a higher risk of kicking out something still in use since more things will have to be kicked out. Also, the NT instructions are not affected by whether the memory they are writing to is in the cache or not. In other words, the cache miss rate has a higher impact when memory consumption is high.

6.2 How much less data is exposed?

From figure 5.1, we can see that at an arbitrary time during transmission, the amount of sensitive data in memory is around half of what it would be if we do not zero. Then, after the transmission is completed, no data will be left in the memory. If we do not zero the data, it is hard to know exactly how long it will remain in the memory. It is only when the memory block in which the data resides gets used again that it is overwritten and thus cleared. However, as shown in figure 5.2, even when running a 1000 Mbits/sec connection for 20 seconds after transmitting sensitive data, some still reside in the memory. So, the memory blocks used for some data are not guaranteed to be used again directly. Still, the "exposed" data here is stored in network buffers, i.e., protected kernel memory, and would require some exploit or kernel bug to be read by an attacker.

Chapter 7

Conclusion

As observed, zeroing packet data once it is no longer in use lessened the amount of packet data left in memory by a large margin. At an arbitrary time during transmission, we could cut the amount of data in memory to around half what it would be without zeroing. After transmission, no data was left in memory, compared to when not zeroing, when data would reside in memory for an indefinite period, i.e., until overwritten.

Zeroing did come with some performance costs, as, when using the regular memset function to zero, it had around 1.5% higher CPU usage than no zeroing and about 4% higher cache miss rate. Because of worsened cache performance, the zeroing with memset used around 4% more of the CPU when stressing the CPU and memory. However, the cache miss rate was unaffected when using a memset that uses non-temporal stores to zero. Due to not polluting the cache, it only used around 2.5% more of the CPU than no zeroing when stressing the CPU and memory. However, due to using more time-consuming operations, it used slightly more of the CPU when the CPU and memory were under no pressure.

Our zeroing implementation should be applicable across different platforms other than x86, such as ARM. However, the non-temporal memset is x86 specific and will only work on x86 processors that support SSE/SSE2, since it uses special instructions from these extensions and is written in x86 assembly.

In conclusion, this work has demonstrated that zeroing networking data effectively reduces the lifespan of sensitive information in memory. If data security is a high concern, or if you want to ensure that sensitive data is not retained in memory, zeroing can be a valuable strategy, provided the performance penalties are acceptable. The non-temporal version is preferable for a system with high memory consumption. Otherwise, the normal memset has a slightly better performance.

7.1 Future work

7.1.1 Testing for different drivers

When testing our implementation on the e1000 driver, we noticed some data was still exposed after transmission. After investigating, we saw that when receiving a small packet, the e1000 driver allocates a new skb and copies the data into the skb without freeing the original data received. The data would be left in the DMA-mapped page until it is overwritten. We only tested on the e1000, rtl8139, and IGB drivers, so we suspect there may be cases similar to the e1000 driver. This work could benefit from more testing on different drivers.

7.1.2 Testing while maximizing Networking CPU Utilization

In our tests, the maximum Networking CPU Utilization never went above 14%. It would be interesting to see the impact zeroing would have on the throughput when Networking CPU Utilization is close to 100%. This could be done using more networking devices/ports, a more underpowered machine, or a faster NIC.

References

[1] kernel-hardening - Merge in PAX_MEMORY_SANITIZE work from grsec to linux-next. https://www.openwall.com/lists/kernel-hardening/2017/01/18/1. [Accessed 18-01-2024].

[2] struct sk_buff. https://docs.kernel.org/networking/skbuff.html. [Accessed 08-05-2024].

[3] Laura Abbott. [RFC][PATCH 0/7] Sanitization of slabs based on grsecurity/PaX. https://lore.kernel.org/lkml/1450755641-7856-1-git-send-email-laura@labbott.name. [Accessed 18-01-2024].

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security Symposium*, pages 22–22, 2005.

[5] Jonathan Corbet. Faster page faulting through prezeroing. https://lwn.net/Articles/117881/. [Accessed 18-01-2024].

[6] Jonathan Corbet. Kernel security: beyond bug fixing. https://lwn.net/Articles/662219/. [Accessed 18-01-2024].

[7] Jonathan Corbet. Revisiting the kernel's preemption models (part 1). https://lwn.net/Articles/944686/. [Accessed 18-01-2024].

[8] Jonathan Corbet. Some upcoming memory-management patches. https://lwn.net/Articles/875587/. [Accessed 18-01-2024].

[9] Jake Edge. Sanitizing kernel memory. https://lwn.net/Articles/334747/. [Accessed 18-01-2024].

[10] Intel. Intel 64 and ia-32 architectures software developer's manual volume 1. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2024. [Accessed 03-05-2024].

[11] Robert Love. *Linux Kernel Development (3rd edition)*. Addison-Wesley Professional, 2010.

[12] Jishen Zhao Mengjie Li, Matheus Ogleari. Logging in persistent memory: to cache, or not to cache? In *MEMSYS '17: Proceedings of the International Symposium on Memory Systems*, pages 177–179, 2017.

[13] Davide Miola. *Assessing the impact of Linux networking on CPU consumption*. PhD thesis, Politecnico di Torino, 2023.

[14] Johannes Wikner and Kaveh Razavi. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, 2022.

Appendices

Appendix A
struct sk_buff

Listing A.1: sk_buff structure

```
struct sk_buff {
    union {
        struct {
            /* These two members must be first to match sk_buff_head. */
            struct sk_buff      *next;
            struct sk_buff      *prev;

            union {
                struct net_device *dev;
                /* Some protocols might use this space to store information,
                 * while device pointer would be NULL.
                 * UDP receive path is one user.
                 */
                unsigned long       dev_scratch;
            };
        };
        struct rb_node      rbnode; /* used in netem, ip4 defrag, and tcp stack */
        struct list_head    list;
        struct llist_node   ll_node;
    };

    union {
        struct sock     *sk;
        int             ip_defrag_offset;
    };

    union {
        ktime_t         tstamp;
        u64         skb_mstamp_ns; /* earliest departure time */
    };
    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a skb_clone()
     * first. This is owned by whoever has the skb queued ATM.
     */
    char            cb[48] __aligned(8);

    union {
        struct {
            unsigned long   _skb_refdst;
            void        (*destructor)(struct sk_buff *skb);
        };
        struct list_head    tcp_tsorted_anchor;
#ifdef CONFIG_NET_SOCK_MSG
        unsigned long       _sk_redir;
#endif
    };

#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
    unsigned long        _nfct;
#endif
    unsigned int        len,
                data_len;
    __u16           mac_len,
                hdr_len;

    /* Following fields are _not_ copied in __copy_skb_header()
     * Note that queue_mapping is here mostly to fill a hole.
     */
    __u16           queue_mapping;

/* if you move cloned around you also must adapt those constants */
#ifdef __BIG_ENDIAN_BITFIELD
#define CLONED_MASK    (1 << 7)
#else
#define CLONED_MASK    1
#endif
#define CLONED_OFFSET         offsetof(struct sk_buff, __cloned_offset)

    /* private: */
    __u8            __cloned_offset[0];
    /* public: */
    __u8            cloned:1,
                nohdr:1,
                fclone:2,
                peeked:1,
                head_frag:1,
                pfmemalloc:1,
                pp_recycle:1; /* page_pool recycle indicator */
#ifdef CONFIG_SKB_EXTENSIONS
    __u8            active_extensions;
#endif

    /* Fields enclosed in headers group are copied
     * using a single memcpy() in __copy_skb_header()
     */
    struct_group(headers,
    /* private: */
    __u8            __pkt_type_offset[0];
    /* public: */
    __u8            pkt_type:3; /* see PKT_TYPE_MAX */
    __u8            ignore_df:1;
    __u8            dst_pending_confirm:1;
    __u8            ip_summed:2;
    __u8            ooo_okay:1;

    /* private: */
    __u8            __mono_tc_offset[0];
    /* public: */
    __u8            mono_delivery_time:1;   /* See SKB_MONO_DELIVERY_TIME_MASK */
#ifdef CONFIG_NET_XGRESS
    __u8            tc_at_ingress:1;  /* See TC_AT_INGRESS_MASK */
    __u8            tc_skip_classify:1;
#endif
    __u8            remcsum_offload:1;
    __u8            csum_complete_sw:1;
    __u8            csum_level:2;
    __u8            inner_protocol_type:1;

    __u8            l4_hash:1;
    __u8            sw_hash:1;
#ifdef CONFIG_WIRELESS
    __u8            wifi_acked_valid:1;
    __u8            wifi_acked:1;
#endif
    __u8            no_fcs:1;
    /* Indicates the inner headers are valid in the skbuff. */
    __u8            encapsulation:1;
    __u8            encap_hdr_csum:1;
    __u8            csum_valid:1;
#ifdef CONFIG_IPV6_NDISC_NODETYPE
    __u8            ndisc_nodetype:2;
#endif

#if IS_ENABLED(CONFIG_IP_VS)
    __u8            ipvs_property:1;
#endif
#if IS_ENABLED(CONFIG_NETFILTER_XT_TARGET_TRACE) || IS_ENABLED(CONFIG_NF_TABLES)
    __u8            nf_trace:1;
#endif
#ifdef CONFIG_NET_SWITCHDEV
    __u8            offload_fwd_mark:1;
    __u8            offload_l3_fwd_mark:1;
#endif
    __u8            redirected:1;
#ifdef CONFIG_NET_REDIRECT
    __u8            from_ingress:1;
#endif
#ifdef CONFIG_NETFILTER_SKIP_EGRESS
    __u8            nf_skip_egress:1;
#endif
#ifdef CONFIG_TLS_DEVICE
```

```
    __u8            decrypted:1;
#endif
    __u8            slow_gro:1;
#if IS_ENABLED(CONFIG_IP_SCTP)
    __u8            csum_not_inet:1;
#endif

#if defined(CONFIG_NET_SCHED) || defined(CONFIG_NET_XGRESS)
    __u16           tc_index;   /* traffic control index */
#endif

    u16             alloc_cpu;

    union {
        __wsum          csum;
        struct {
            __u16   csum_start;
            __u16   csum_offset;
        };
    };
    __u32           priority;
    int             skb_iif;
    __u32           hash;
    union {
        u32         vlan_all;
        struct {
            __be16      vlan_proto;
            __u16   vlan_tci;
        };
    };
#if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
    union {
        unsigned int    napi_id;
        unsigned int    sender_cpu;
    };
#endif
#ifdef CONFIG_NETWORK_SECMARK
    __u32       secmark;
#endif

    union {
        __u32       mark;
        __u32       reserved_tailroom;
    };

    union {
        __be16          inner_protocol;
        __u8        inner_ipproto;
    };

    __u16           inner_transport_header;
    __u16           inner_network_header;
    __u16           inner_mac_header;

    __be16              protocol;
    __u16           transport_header;
    __u16           network_header;
    __u16           mac_header;

#ifdef CONFIG_KCOV
    u64             kcov_handle;
#endif

    ); /* end headers group */

    /* These elements must be at the end, see alloc_skb() for details.  */
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head,
                *data;
    unsigned int        truesize;
    refcount_t      users;

#ifdef CONFIG_SKB_EXTENSIONS
    /* only usable after checking ->active_extensions != 0 */
    struct skb_ext      *extensions;
#endif
};
```

Listing A.2: skb_shared_info structure

```
struct skb_shared_info {
    __u8        flags;
    __u8        meta_len;
    __u8        nr_frags;
    __u8        tx_flags;
    unsigned short gso_size;
    /* Warning: this field is not always filled in (UFO)! */
    unsigned short gso_segs;
    struct sk_buff *frag_list;
    union {
        struct skb_shared_hwtstamps hwtstamps;
        struct xsk_tx_metadata_compl xsk_meta;
    };
    unsigned int    gso_type:24;
    u32         tskey;

    /*
        * Warning : all fields before dataref are cleared in __alloc_skb()
        */
    atomic_t dataref;
    unsigned int    xdp_frags_size;

    /* Intermediate layers must ensure that destructor_arg
      * remains valid until skb destructor */
    void *          destructor_arg;

    /* must be last field, see pskb_expand_head() */
    skb_frag_t    frags[MAX_SKB_FRAGS];
};
```

Listing A.3: sk_buff flags

```
/* Definitions for flags in struct skb_shared_info */
enum {
    /* use zcopy routines */
    SKBFL_ZEROCOPY_ENABLE = BIT(0),

    /* This indicates at least one fragment might be overwritten
     * (as in vmsplice(), sendfile() ...)
     * If we need to compute a TX checksum, we'll need to copy
     * all frags to avoid possible bad checksum
     */
    SKBFL_SHARED_FRAG = BIT(1),

    /* segment contains only zerocopy data and should not be
     * charged to the kernel memory.
     */
    SKBFL_PURE_ZEROCOPY = BIT(2),

    SKBFL_DONT_ORPHAN = BIT(3),

    /* page references are managed by the ubuf_info, so it's safe to
     * use frags only up until ubuf_info is released
     */
    SKBFL_MANAGED_FRAG_REFS = BIT(4),
};
```

Appendix B
memset implementations

All of the information about x86 processors in this chapter is from the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1. For non-temporal instructions, see 10.4.6.2 Caching of Temporal vs. Non-Temporal Data, and for string operations, see 7.3.9 String Operations [10].

The signature of the memset function is: **void memset(void *s, int c, size_t n)**, where **s** is the starting address of the memory to set, **c** is the value to set each byte to, and **n** is the count/number of bytes to set.

B.1 Store string

This is the memset implementation in the Linux kernel used for x86 processors with support for fast-string operations. This is an optimization that exists on Intel processors based on Ice Lake Client microarchitecture or later, meaning that this implementation will be used for modern Intel x86 processors and will be faster on these than the original implementation that uses normal stores.

String instructions work on a string with the index/address stored in the ESI or RDI register. String instructions can be combined to operate on larger strings than a double word by using one of the repeat prefixes (REP, REPE/REPX, or REPNE/REPNZ). By storing the count (number of iterations/bytes) in the ECX register, the string instructions will be repeated, and the ESI/RDI and ECX automatically incremented/decremented until it is done.

The memset implementation using string operations starts by storing the start address in the RDI register, the count in the RCX register, and the set value in the RAX register. It then uses REP STOSB, which repeats STOSB (store string byte in RAX to the memory address in RDI) until the RCX register reaches zero, automatically incrementing RDI and decrementing RCX for each iteration.

Listing B.1: memset using fast-short REP STOSB

```
movq %rdi,%r9
movb %sil,%al
movq %rdx,%rcx
rep stosb
movq %r9,%rax
RET
```

B.2 Original

This is the memset implementation in the Linux kernel used on processors that do not support fast-store string, i.e., older Intel and non-Intel processors. This implementation does not use any special features and will work on any 64-bit x86 processor. The implementation uses quad-word stores (MOVQ) as much as possible and single bytes stores (MOVB) to handle remaining bytes at the end not aligned with a quad-word. It also tries to write as many full cache lines with 8 MOVQs between jumps at a time to improve performance (.Lloop_64).

It works like this:

- We start by moving the start address to the RDI register and the number of bytes to set to RDX.
- We expand the byte value we want to set each byte to from the argument into the 64-bit register RAX.
- We check if the starting address is aligned with 8 bytes (one quad-word). If it is not, we jump to the bad alignment label (.Lbad_alignment); here, we check if the count is less than 8. If it is, we jump to the handle_7 label (.Lhandle_7). Otherwise, if the count is 8 or larger, we do an unaligned store (MOVQ). An unaligned store is a store that is not aligned on the store instruction's data width, i.e., a quad-word store that is not aligned on 8 bytes. These still work, but they will be split into two separate stores by the processor and take longer to execute. The address in RDI is then incremented so it is aligned on 8 bytes, and we jump back to where we would have been if we did not jump (.Lafter_bad_alignment).
- We copy the count from RDX to RCX and right shift RCX by 6. This is the same as dividing by 64, i.e., the RCX register is how many full cache lines we can write (.Lafter_bad_alignment).
- Loop through each cache line, storing RAX into each memory address in the cache line with multiple MOVQs, incrementing RDI, and decrementing RCX each iteration until RCX reaches zero. (.Lloop_64)
- We copy RDX to RCX again so it has the original count and mask out the bits for the number of bytes after the last full cache line. If it zero we jump to .Lhandle_7. Otherwise, we shift RCX by 3 so it contains the number of quad-words left to set (.Lhandle_tail)
- We loop through each remaining quadword, storing RAX into each memory address with MOVQ, incrementing RDI, and decrementing RCX each iteration until RCX reaches zero. (.Lloop_8).

- We then reach the .Lhandle_7 label, mask out the 3 lowest bits from RDX to get the number of bytes left to set. If it is zero, jump to the end label (.Lende).
- We loop through each remaining byte doing a single byte store (MOVB) on RDI, decrementing RDX and incrementing RDI each iteration until RDX reaches zero. (.Lloop_1)
- We are done, return. (.Lende)

Listing B.2: memset with only normal stores

```
    movq %rdi,%r10

    /* expand byte value */
    movzbl %sil,%ecx
    movabs $0x0101010101010101,%rax
    imulq  %rcx,%rax

    /* align dst */
    movl  %edi,%r9d
    andl  $7,%r9d
    jnz   .Lbad_alignment
.Lafter_bad_alignment:

    movq  %rdx,%rcx
    shrq  $6,%rcx
    jz    .Lhandle_tail

    .p2align 4
.Lloop_64:
    decq  %rcx
    movq  %rax,(%rdi)
    movq  %rax,8(%rdi)
    movq  %rax,16(%rdi)
    movq  %rax,24(%rdi)
    movq  %rax,32(%rdi)
    movq  %rax,40(%rdi)
    movq  %rax,48(%rdi)
    movq  %rax,56(%rdi)
    leaq  64(%rdi),%rdi
    jnz     .Lloop_64

    /* Handle tail in loops. The loops should be faster than hard
       to predict jump tables. */
    .p2align 4
.Lhandle_tail:
    movl        %edx,%ecx
    andl     $63&(~7),%ecx
    jz          .Lhandle_7
    shrl        $3,%ecx
    .p2align 4
.Lloop_8:
    decl   %ecx
    movq  %rax,(%rdi)
    leaq  8(%rdi),%rdi
    jnz     .Lloop_8
```

```
.Lhandle_7:
    andl        $7,%edx
    jz       .Lende
    .p2align 4
.Lloop_1:
    decl     %edx
    movb        %al,(%rdi)
    leaq        1(%rdi),%rdi
    jnz      .Lloop_1

.Lende:
    movq        %r10,%rax
    RET

.Lbad_alignment:
    cmpq  $7,%rdx
    jbe   .Lhandle_7
    movq  %rax,(%rdi)      /* unaligned store */
    movq  $8,%r8
    subq  %r9,%r8
    addq  %r8,%rdi
    subq  %r8,%rdx
    jmp   .Lafter_bad_alignment
.Lfinal:
```

B.3 Non-temporal

This is the implementation of memset with non-temporal stores. As explained earlier, non-temporal stores are a way to write to memory without polluting the cache; the writes go directly to memory using write combining to increase efficiency. Non-temporal instructions, e.g. MOVNTI, are available as part of Intels SIMD supplementary instruction instruction set, SSE and SSE2. So this implementation only works on 64-bit x86 processors supporting SSE or SSE2, however on most modern x86 processors from both Intel and AMD these are supported.

It is similar to the original memset (see B.2) but with some differences:

- The MOVQ instructions have been substituted with the non-temporal store instruction MOVNTI. MOVNTI does a non-temporal store from a general-purpose register. This can either be a quad-word or a double-word store depending on the width of the register; in our case, we use RAX, which is a 64-bit register; hence MOVNTI will do a quad-word store.

- Instead of doing byte stores when reaching the Lhandle_7 label, we move the RDI back so that it is 8 bytes between RDI and the last address we want to store to, e.g., if there are 5 more bytes to store, we move RDI back 3 bytes so we can do an unaligned quad-word store.

- In .Lbad_alignment, if the count is less than 8 bytes, we jump to Lhandle_small instead of Lhandle_7. This is because we cannot move back RDI to do a non-temporal store since we would then be setting bytes before the starting offset of the target range. Instead, we do single-byte stores on the bytes.

- Since the non-temporal stores use write combining semantics, the stores will be weakly ordered. This means that when we are done, we need to add an SFENCE instruction to ensure coherence (.Lende).

Listing B.3: memset implementation using MOVNTI

```
        movq %rdi,%r10

        /* expand byte value  */
        movzbl %sil,%ecx
        movabs $0x0101010101010101,%rax
        imulq  %rcx,%rax

        /* align dst */
        movl  %edi,%r9d
        andl  $7,%r9d
        jnz  .Lbad_alignment
        cmpq $7,%rdx
        jbe      .Lhandle_small

.Lafter_bad_alignment:
        movq  %rdx,%rcx
        shrq  $6,%rcx
        jz       .Lhandle_tail

        .p2align 4
.Lloop_64:
        decq  %rcx
        movnti  %rax,(%rdi)
        movnti  %rax,8(%rdi)
        movnti  %rax,16(%rdi)
        movnti  %rax,24(%rdi)
        movnti  %rax,32(%rdi)
        movnti  %rax,40(%rdi)
        movnti  %rax,48(%rdi)
        movnti  %rax,56(%rdi)
        leaq  64(%rdi),%rdi
        jnz     .Lloop_64

        /* Handle tail in loops. The loops should be faster than hard
           to predict jump tables. */
        .p2align 4
.Lhandle_tail:
        movl     %edx,%ecx
        andl     $63&(~7),%ecx
        jz                .Lhandle_7
        shrl     $3,%ecx
        .p2align 4
.Lloop_8:
        decl    %ecx
        movnti  %rax,(%rdi)
        leaq  8(%rdi),%rdi
        jnz      .Lloop_8

.Lhandle_7:
        andl     $7,%edx
        jz       .Lende
        /* Move rdi back to do an unaligned store. */
        movq $8,%r8
        subq %rdx,%r8
        subq %r8,%rdi
        movnti %rax,(%rdi)        /* unaligned store */

.Lende:
        sfence
        movq     %r10,%rax
        RET

.Lbad_alignment:
        cmpq $7,%rdx
        jbe      .Lhandle_small
        movnti %rax,(%rdi)        /* unaligned store */
        movq $8,%r8
        subq %r9,%r8
        addq %r8,%rdi
        subq %r8,%rdx
        jmp .Lafter_bad_alignment

.Lhandle_small:
        andl     $7,%edx
        jz       .Lende
        .p2align 4
.Lloop_1:
        decl     %edx
        movb     %al,(%rdi) /* not a non-temporal store */
        leaq     1(%rdi),%rdi
        jnz      .Lloop_1
        jmp      .Lende
.Lfinal:
```

Appendix C
Zeroing functions

Listing C.1: modified skb_release_data function

```
static void skb_release_data(struct sk_buff *skb, enum skb_drop_reason reason,
              bool napi_safe)
{
    struct skb_shared_info *shinfo = skb_shinfo(skb);
    int i;

    if (skb->cloned &&
         atomic_sub_return(skb->nohdr ? (1 << SKB_DATAREF_SHIFT) + 1 : 1,
                  &shinfo->dataref))
        goto exit;

    if (skb_zcopy(skb)) {
        bool skip_unref = shinfo->flags & SKBFL_MANAGED_FRAG_REFS;

        skb_zcopy_clear(skb, true);
        if (skip_unref)
            goto free_head;
    }

    if (shinfo->frags_zero_below)
        skb_zero_frags(skb, 0, shinfo->frags_zero_idx);
    else
        skb_zero_frags(skb, shinfo->frags_zero_idx, shinfo->nr_frags);
    for (i = 0; i < shinfo->nr_frags; i++)
        napi_frag_unref(&shinfo->frags[i], skb->pp_recycle, napi_safe);

free_head:
    if (shinfo->frag_list)
        kfree_skb_list_reason(shinfo->frag_list, reason);

    skb_free_head(skb, napi_safe);
exit:
    /* When we clone an SKB we copy the reycling bit. The pp_recycle
     * bit is only set on the head though, so in order to avoid races
     * while trying to recycle fragments on __skb_frag_unref() we need
     * to make one SKB responsible for triggering the recycle path.
     * So disable the recycling bit if an SKB is cloned and we have
     * additional references to the fragmented part of the SKB.
     * Eventually the last SKB will have the recycling bit set and it's
     * dataref set to 0, which will trigger the recycling
     */
    skb->pp_recycle = 0;
}
```

Listing C.2: modified skb_free_head function

```
static void skb_free_head(struct sk_buff *skb, bool napi_safe)
{
    unsigned char *head = skb->head;

    if (READ_ONCE(sysctl_skb_zeroing) && skb_headlen(skb)
            && !skb_shinfo(skb)->dont_zero_head) {
        memzero_explicit(skb->data, skb_headlen(skb));
    }

    if (skb->head_frag) {
        if (skb_pp_recycle(skb, head, napi_safe))
            return;
        skb_free_frag(head);
    } else {
        skb_kfree_head(head, skb_end_offset(skb));
    }
}
```

Listing C.3: modified skb_zero_frag* functions

```
void static __always_inline __skb_frag_zero(skb_frag_t* frag) {
    struct page *p;
    u32 p_off, p_len, copied;
    skb_frag_foreach_page(frag, skb_frag_off(frag),
            skb_frag_size(frag), p, p_off, p_len,
            copied) {
        memzero_page_explicit(p, p_off, p_len);
    }
}

void skb_zero_frag_off(struct sk_buff *skb, int i, int off, int size) {
    if (READ_ONCE(sysctl_skb_zeroing) && !skb_has_shared_frag(skb)) {
        skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
        memzero_page_explicit(skb_frag_page(frag), skb_frag_off(frag) + off, size);
    }
}

void skb_zero_frag(struct sk_buff *skb, int i) {
    if (READ_ONCE(sysctl_skb_zeroing) && !skb_has_shared_frag(skb)) {
        __skb_frag_zero(&skb_shinfo(skb)->frags[i]);
    }
}

void skb_zero_frags(struct sk_buff *skb, int from, int to) {
    if (READ_ONCE(sysctl_skb_zeroing) && !skb_has_shared_frag(skb)) {
        for (int i = from; i < to; i++) {
            skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
            __skb_frag_zero(frag);
        }
    }
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-13

EXAMENSARBETE Minimizing the exposure of sensitive data during network transfers in the Linux kernel
STUDENTER Joel Johansson, Anton Wiklund
HANDLEDARE Jonas Skeppstedt (LTH), Reine Johansson (Asperiq AB)
EXAMINATOR Flavius Gruian (LTH)

Minimera exponeringen av känslig data för nätverksöverföringar i Linux kärnan

POPULÄRVETENSKAPLIG SAMMANFATTNING **Joel Johansson, Anton Wiklund**

System som skickar känslig data över nätverk behöver vid något tillfälle ha den datan i minnet. Detta arbete undersöker hur man mest effektivt kan minimera tiden den datan ligger i minnet för operativsystemet Linux, för att mildra potentiella dataläckor.

Tänk dig att du ska skicka ett krypterat brev till din kompis. Du skriver först ner det du vill skicka på ett anteckningsblock. Sedan krypterar du meddelandet och skriver ner den krypterade texten på ett brev som du skickar iväg. Om någon läser ditt brev när det skickas kommer den inte kunna förstå meddelandet. Men om någon bryter sig in i ditt hus och läser ditt anteckningsblock, kommer den kunna läsa meddelandet eftersom det står i klartext där. Det här kan liknas vid en nätverksöverföring, där man krypterar meddelandet som ska skickas ut, men originalet finns kvar i datorns minne. Om någon hackar sig in i datorn och lyckas läsa minnet, kommer den personen kunna läsa den icke-krypterade datan. Det vi undersöker i detta arbete är hur man kan nollställa minnet där datan lagrats, dvs, sudda ut anteckningblocket när man har krypterat och skickat iväg brevet.

För att skicka ut eller ta emot paket på nätverket behöver Linux kärnan vid något tillfälle spara paketdata någonstans i minnet. Det gör den för att kunna behandla paketet i alla de olika nätverksdelarna. När ett paket inte längre behövs, dvs, det har skickats iväg eller levererats till en applikation, så frigörs minnet som användes för att lagra paketdatan. I vårt examensarbete har vi modifierat Linux kärnan så att när minnet för paketdata frigörs, skrivs det också över med nollor, vi suddar ut minnet. På det sättet kommer paketdatan endast ligga i minnet när kärnan behöver det. I den normala implementationen kommer datan ligga kvar i minnet till det minnet allokeras av någon annan och den skriver över det. Våra tester visar att den tiden datan ligger i minnet kan variera mycket och att ibland kan data ligga kvar i en obestämd tid.

Resultaten av vår implementation visade att nollställningen inte påverkade hur mycket nätverkstraffik vi kunde skicka, hastigheten var den samma oavsett hur vi nollställde eller om vi inte nollställde alls. Dock såg vi att den använde något mer av datorns resurser. Vi testade även att skriva direkt till minnet utan att skriva till cachen för att undvika "cache pollution", dvs, vi förhindrar att hämta minne som endast används en gång till cachen. Det visade sig vara bättre när systemet var mer belastat - lite tillgängligt minne och hög processor användning - men sämre när det inte var belastat. Vi såg även att paketdatan som låg i minnet under nätverksöverföringar halverades och att inget var kvar efter.