MIKELANGELO

D3.3

Super KVM - Fast virtual I/O hypervisor

Workpackage	3	Hypervisor Implementation	
Author(s)	Joel Nider, Kalman Meth		IBM
	Shiqing Fan, Fang Chen		Huawei
	Gabriel Scalosub, Niv Gilboa		BGU
	Justin Činkelj, Gregor Berginc		XLAB
Reviewer	Gregor Berginc		XLAB
Reviewer	Nadav Harel		SCYLLA
Dissemination Level	PU		

Date	Author	Comments	Version	Status
8.11.2017	Kalman Meth	Initial draft	V0.1	Draft
11.12.2017	Kalman Meth, all authors	Updates by partners on ZeCoRx, vRDMA, SCAM, UNCLOT	v0.2	Draft
16.12.2017	Kalman Meth	Document ready for review	V0.2	Review
19.12.2017	Kalman Meth, all authors	Document ready for submission	V1.0	Final

Executive Summary

sKVM, the super KVM, is an enhanced version of the popular Kernel-based Virtual Machine (KVM) hypervisor. The enhancements are both in virtual I/O performance as well as VM security. sKVM improves the performance of virtual I/O devices, such as disks and network interfaces, by changing the underlying threading model in the virtual device backend (vhost) and by avoiding unnecessary copies of the data. sKVM furthermore provides a new type of virtual I/O device: virtualized RDMA (Remote Direct Memory Access) device. This device abstracts the behaviour of the physical RDMA device and offers the flexibility of the virtualized infrastructure by reducing the performance overhead of existing solutions. In addition, the SCAM security module enables sKVM to protect its virtual machines from attacks perpetrated by co-located virtual machines that may be trying to steal SSH cryptographic keys. Finally, sKVM offers the underlying infrastructure to allow efficient intra-host communication via shared memory. This allows cross-layer optimisation targeting OSv specifically increasing the overall bandwidth and reducing the latencies.

During the past year (third year of the MIKELANGELO project) the RDMA and SCAM features that enhance KVM have been further developed. Additional performance testing and evaluation have been performed on these. In addition, several new features have been introduced based on these evaluations and new research findings: ZeCoRx and UNCLOT. This report explains the modifications made to bring the features to this point. Instructions are provided to download and build the code required for reproducing the experiments.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

Table of contents

Table of Figures

MIKELANGELO

Table of Tables

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
HPC	High Performance Counter or High Performance Computing (depending on the context)
I/O	Input / Output
IOcm	I/O Core Manager
KVM	Kernel-based Virtual Machine
LLC	Last Level Cache
NIC	Network Interface Controller
OFED	Open Fabrics Enterprise Distribution
PAPI	Performance Application Programming Interface
PID	Process ID
PMU	Performance Measuring Unit
QEMU	Quick Emulator
RAM	Random Access Memory
RAM-disk	In Memory filesystem
RDMA	Remote Direct Memory Access
RoCe	RDMA over Converged Ethernet
RSA	Rivest-Shamir-Adelman (public key cryptographic protocol)
SCAM	Side Channel Attack Monitoring and Mitigation Module
SKB	SocKet Buffer
sKVM	super KVM
TCP	Transmission Control Protocol
TLS	Transport Layer Security (communication cryptographic protocol)
VM	Virtual Machine
vRDMA	virtual RDMA

1 Introduction

The sKVM (super KVM) architecture has first been described in detail in report D2.13[[1]], and later updated in D2.20[2] and D2.21[3]. The following figure shows all the components that constitute the highly optimised new kernel-based hypervisor: IOcm, ZeCoRx, vRDMA, SCAM, and UNCLOT. These features together are known as sKVM, and each enhances a different aspect of the hypervisor to strengthen various use cases.

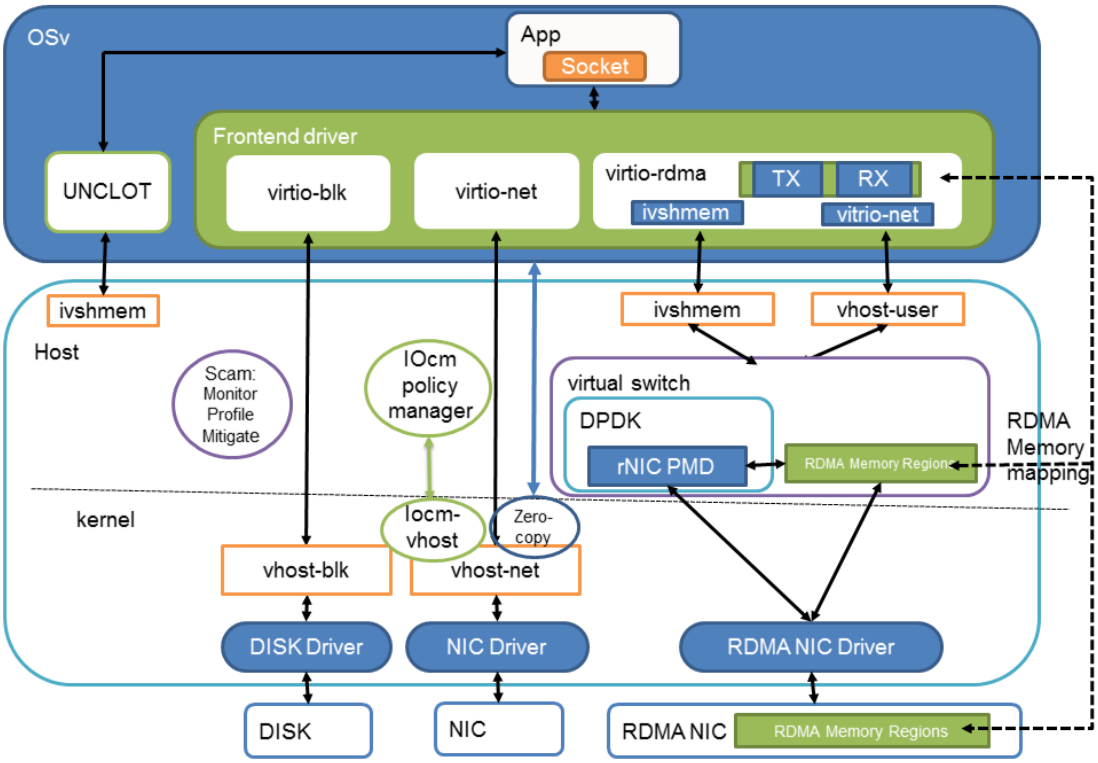

Figure 1: The high-level architecture diagram of sKVM

This document describes our developments during the past year of the following features of sKVM: zero-copy receive (ZeCoRx), lightweight RDMA para-virtualization (vRDMA), side channel attack monitor and mitigation (SCAM), and UNikernel Cross Level cOmmunication opTimisation (UNCLOT). The IOcm component was completed in year 2 of the project and is described in detail in D3.2[4].

The KVM hypervisor depends on the virtio[5] protocol to implement virtual I/O devices for guests. The backend of the virtio protocol is implemented in a Linux kernel module called vhost. IOcm extends the vhost module to provide more control over the allocation of resources for handling virtual I/O. The resources are controlled from a user-space application, namely the IO Policy Manager, which monitors system performance and tunes vhost parameters accordingly. Details about the design, implementation and evaluation of IOcm

appeared in D3.2. In the performance evaluation of IOcm we discovered that although IOcm improved performance, there was still a very large overhead incurred for copying data when network packets were large. A different solution needed to be developed to address large sized packets. This led us to develop the zero-copy receive (ZeCoRx) feature, described in section 2. IOcm was not further developed in year 3 and our main effort in year 3 went into ZeCoRx.

A lightweight RDMA para-virtualization solution (a.k.a. vRDMA) has been designed and the third phase has been implemented this past year. Unlike IOcm which relies on virtio, vRDMA implements its own protocol and a backend driver that manipulates the real RDMA operation on the physical RDMA-capable device. The details of the implementation are found in section 3.

Side channel attack monitor and mitigation (SCAM) is designed to detect and defeat a covert attack perpetrated by a co-located VM intended to discover secret cryptographic keys from the victim VM. The attack is carried out by gleaning knowledge through the difference in behaviour of shared resources (in this case, the cache) depending on the value of the keys. Updated details of the attack, defence of the defense are found in section 4.

Finally, unikernel cross-level optimisation (UNCLOT) component has been introduced in D2.21. UNCLOT employs shared memory offered by the hypervisor to address efficient communication between unikernels running on the same host. Shared memory is used to bypass nearly complete networking stack of OSv increasing the overall throughput and decreasing the latency. Details of the final architecture, proof of concept implementation and a short tutorial on how to use it can be found in section 5.

2 ZeCoRx (Zero-Copy Receive)

2.1 Overview

Some Linux network drivers support zero-copy on transmit (Tx) messages. Zero-copy Tx avoids the copy of data between VM guest kernel buffers and host kernel buffers, thus improving Tx latency and reducing CPU overhead. Buffers in a VM guest kernel for a virtualized NIC are passed through the host device drivers and DMA-d directly to the network adapter, without an additional copy into host memory buffers. Since the Tx data from the VM guest is always in-hand, it is quite straight-forward to map the buffer for DMA and to pass the data down the stack to the network adapter driver.

Zero-copy for receive (Rx) messages is not yet supported in Linux. A number of significant obstacles must be overcome in order to support zero-copy for Rx. Buffers must be prepared to receive data arriving from the network. Currently, DMA buffers are allocated by the low-level network adapter driver. The data is then passed up the stack to be consumed by the intended destination of the data. When Rx data arrives, it is not necessarily clear a-priori for whom the data is designated. The data may eventually be copied to VM guest kernel buffers. The challenge is to allow the use of VM guest kernel buffers as DMA buffers, at least when we know that the VM guest is the sole consumer of a particular stream of data.

One solution that has been tried is page-flipping, in which the page with the received data is mapped into the memory of the target machine (VM guest) after the data has already been placed in the buffer. The overhead to perform the page mapping is significant, and essentially negates the benefit we wanted to achieve by avoiding the copy[6].

Our proposed solution requires the introduction of several interfaces that enable us to communicate between the high and low level drivers to pass buffers down and up the stack, when needed. We also need to deal with the case when insufficient VM guest buffers have been made available to receive data from the network. A high-level architecture is detailed in D2.21. In this document we summarize implementation details and current status.

Our initial design and implementation concentrate on an end-to-end solution for a single example, but the method is applicable to the general case where a distinct MAC address is identified with a guest VM network interface. We define a VM guest with a macvtap[7] device. (See the figure below.) Macvtap is a device driver meant to simplify virtualized bridged networking; it replaces the combination of the tun/tap and bridge drivers with a single module based on the macvlan device driver. On the guest side, the data goes through virtio. On the host, the data goes through vhost_net, macvtap, macvlan, and the Ethernet adapter device drivers (e.g. ixgbe). We describe the changes to these drivers in order to support Zero-Copy Rx.

2.2 Architecture

2.2.1 Main idea

To perform Zero-Copy Rx we need to take guest-provided buffers and make them available to the Ethernet adapter driver. These buffers must be placed in a ring buffer that receives data specifically for this guest. This means, first of all, that the Ethernet adapter must support the ability to assign a particular receive queue to a particular guest. After setting up the mapping between the guest and the Ethernet adapter receive queue, guest-provided buffers must be posted to the queue. As data arrives and is placed in the guest-provided buffers, the buffers are passed up the stack to be returned to the guest. Code needs to be written in each of the intermediate layers (virtio, vhost-net, macvtap, macvlan, ethernet device driver (e.g. ixgbe)) to support the new functionality.

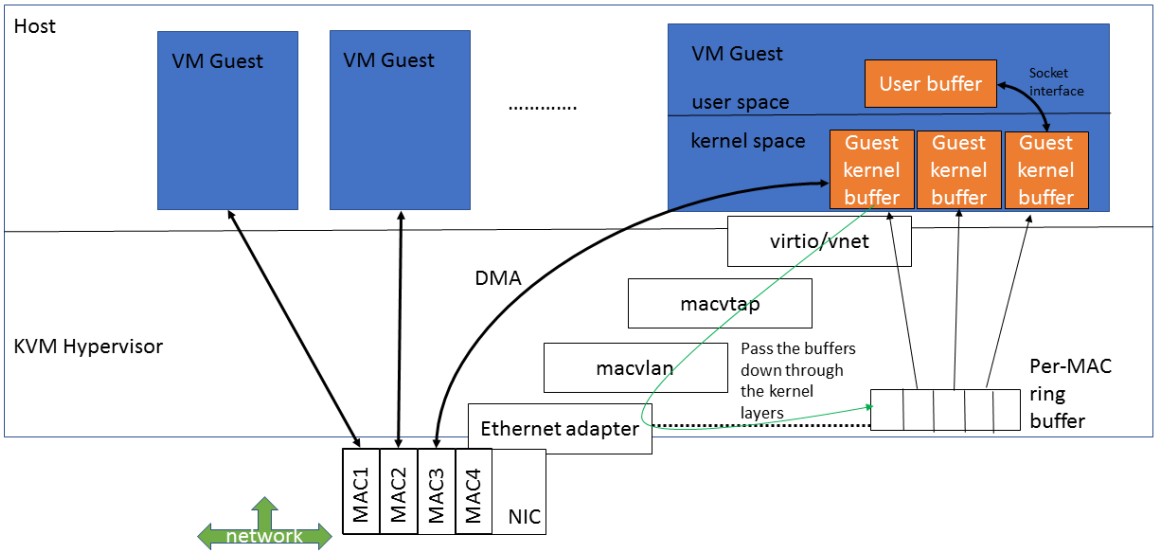

Figure 2: Zero-Copy Receive Architecture

2.2.2 Challenges

There are, however, numerous difficulties that have to be overcome in order to implement Zero-Copy Receive. When a buffer is provided by a guest, it must be mapped to its proper Ethernet ring buffer. After data is placed in the buffer, the buffer must be directed back to the proper socket interface (through which macvtap is connected to the ethernet driver) so that it arrives at the guest that originally allocated the page. Sometimes a message for the guest arrives that is not in its dedicated ring and is not placed in its pre-allocated buffers (as in a broadcast request). A mechanism is needed to recycle pages when a packet is corrupted and the page must be returned to its original owner for re-use. If we enable TCP segmentation and reassembly offloading, it turns out that Ethernet headers of intervening packets are

discarded, but they take up space at the beginning of the receive buffers; offsets within the buffers must now be taken into consideration when returning those buffers to the guest.

Other challenges include:

- Build a solution to work within the existing network stack framework without changing too many things.
- Provide sufficient buffers from user space (guest) to the Ethernet driver (in hypervisor) so that we don't need to drop too many packets.
- Proper interaction (timing) between the various threads. We need to minimize the times we run out of buffers at Ethernet level. We need to minimize the times we over-stuff the socket buffer resulting in dropped packets at the vhost level.

2.2.3 Relevant threads to perform virtio_net/vhost IO

There are 3 relevant threads that interact to perform IO on a vitio device:

- the guest virtio thread, which provides buffers to vhost and cleans up buffers that have been used by vhost;
- the interrupt thread taking the next buffer received by the Ethernet adapter and figuring out to which upper level driver the buffer should be delivered;
- the vhost worker thread, which is awakened every so often to process buffers that have been placed on its socket fifo.

The Ethernet driver interrupt thread typically takes the next used buffer from the Ethernet ring and posts it to the appropriate upper level socket. The vhost Rx worker thread typically takes the next buffer that was posted on its socket fifo and copies the data into the next available buffer(s) that are provided in the virtio ring. The guest virtio thread places available buffers in the virtio ring and cleans up buffers that have been consumed. Events and timers are used to wake up the threads when they have work to perform.

2.3 Implementation

As we encountered the challenges described above, we made adjustments to our original design in order to address those challenges. This section describes the new interfaces we introduced, some directions in the implementation we had to abandon (false starts), and details of the current implementation.

We started with the linux source tree, version 4.8.0, which was the latest stable version available when we started work on ZeCoRx. Subdirectories referenced below are relative to this source tree.

2.3.1 New interfaces defined at the netdev level

In order to support Zero-Copy Rx, we added 2 functions to the list of generic functions (net_device_ops in include/linux/netdevice.h) defined for a network device. These functions must be implemented for each zero-copy enabled device in the network stack that supports the net_device_ops interface. In our implementation this includes the macvlan driver and the ixgbe Ethernet driver.

2.3.1.1 ndo_set_zero_copy_rx(struct net_device *dev, struct net_device *base_dev)

The ndo_set_zero_copy_rx() function is called when a zero-copy enabled device is opened. In our implementation, this occurs in macvtap_open(), which calls the ndo_set_zero_copy_rx() of the underlying macvlan device, which in turn calls the ndo_set_zero_copy_rx() function of the underlying Ethernet driver (ixgbe in our first implementation). The first parameter is the device object to which the function belongs, and the second parameter points to the device object of the high level device (macvtap in our case) for which we are enabling zero-copy receive. The macvlan ndo_set_zero_copy_rx() function simply calls the low-level Ethernet device ndo_set_zero_copy_rx() function, passing it the original base_dev pointer obtained from the macvtap level. The base_dev structure contains the MAC address associated with the macvtap device. If that address matches the MAC address associated with a particular Ethernet ring buffer, then we can proceed to run with zero-copy receive on that ring buffer. Upon success return 0; upon failure return an appropriate code (ENOTSUPP, ENOMEM, ENXIO, etc).

2.3.1.2 ndo_post_rx_buffer(struct net_device *dev, struct sk_buff *skb)

The ndo_post_rx_buffer() function is called to pass a buffer from a guest down to the driver that will fill the buffer (in our case, the Ethernet adapter driver). The first parameter is the device object to which the function belongs, and the second parameter points to an skb (socket buffer) object that points to the provided buffer. Skb-s are used throughout the linux network stack to pass buffers between components. In addition to pointing to the provided buffer, the skb has a field pointing to a device structure with which the buffer is associated. This field is filled in with the high-level device structure (macvtap in our implementation). Among other things, this gives access to the MAC address for which the buffer is designated. This information is used by the low level Ethernet driver to determine the correct queue into which to place the buffer. The macvlan ndo_post_rx_buffer() function simply passes the buffer down the stack by calling the ixgbe Ethernet driver ndo_post_rx_buffer() function.

2.3.2 False Starts

This section describes some directions in the implementation we had to eventually abandon.

We aimed to implement Zero-Copy Rx with minimal changes to the surrounding eco-system in the hope that this would ease upstreaming of our changes. In particular, we hoped to avoid changes to the guest VM, so that existing VMs could run without having to recompile. Typically, Ethernet adapter drivers provide full (4K aligned) pages to the adapter into which to place data arriving over the network via DMA (Direct Memory Access). We therefore want to take 4K aligned full pages provided by the guest and pass them down to the Ethernet driver. It turns out, however, that there was no provision for virtio drivers to provide 4K aligned full pages. We therefore had to make changes to the virtio_net driver (in the VM guest) to provide 4K aligned full pages and to process those pages properly after data had been placed in them. We later realized that we also had to make changes to Qemu in order to negotiate between the guest and the host as to which method to memory allocation to use.

At first we attempted to mimic code used for zero-copy transmit (Tx) as much as possible. In zero-copy Tx, one skb is allocated in macvtap per message. Each message is associated with a single vhost descriptor. A special data structure (ubuf) attached to the skb is used to store critical information about the corresponding descriptor. The ubuf includes the vhost descriptor number and a pointer to a callback function to be called upon completion. After the data is transmitted by the Ethernet driver, when the skb is freed, the callback function is called to inform the guest that the descriptor has been consumed. At first, we did the same for Zero-Copy Rx. We allocate one skb per buffer in macvtap and post the buffer to the Ethernet driver. The Ethernet driver places the buffer in the appropriate rx ring and saves the skb pointer in the rx_buffer data structure associated with that ring buffer entry. The callback information (including the vhost descriptor number) is thus available when we need it. When the buffer is consumed, we used the same skb to pass the buffer back up the stack, and we were thus able to recover the vhost descriptor number for the buffer. This worked as long as we had a separate skb for each buffer. However, when doing TCP reassembly, multiple buffers are returned with a single skb, and we lost the vhost descriptor information for the attached buffers. For a while, we continued development with TCP reassembly (GRO - Generic Receive Offload) turned off so that we continued to have a single skb per buffer. This required the guest to perform the TCP reassembly and could not take advantage of the GRO hardware support in the adapter. In order to enable TCP reassembly by the adapter, we had to use a different scheme to recover the vhost descriptor numbers of the buffers, and we could not rely on callback information we had saved in the skb to post the buffer. Instead we use a hashtable that maps each buffer to its vhost descriptor number. When a single skb points to multiple buffers, we look up each returned buffer in the hashtable, and identify the full list of vhost descriptors that are being consumed by the buffers of the skb.

Our design builds upon the assumption that the Ethernet driver has multiple queues and that we can map a particular MAC address to a specific receive queue. Most modern Ethernet adapters have multiple queues and enable dedication of individual queues to support virtual

functions for VMs and SRIOV (Single Root I/O Virtualization). We want to exploit this feature of new adapters to map a guest VM's macvtap MAC address to a particular queue, and post the buffers from that VM to the particular queue. We obtained adapters (ixgbe) that support multiple queues and SRIOV and we tried to figure out how to map a MAC address to a particular queue. If this were supported directly by the card, then it should have been possible to perform a command like:

```
ethtool -U eth12 flow-type ether dst 50:e5:49:3d:90:5b m
ff:ff:ff:ff:ff:ff action 3
```

This command specifies that Ethernet packets with the specified destination address should be directed into the queue specified by 'action'. It turned out that the adapters we had (ixgbe) did not support this level of granularity of filtering with the ethtool command. We dug into the code to see how to patch the code to provide the functionality we needed. In parallel we inquired with contacts at Intel to find out what newer adapters provided the desired functionality. Eventually we discovered that if we enable the l2-fwd-offload feature and apply a patch that enabled this feature for macvtap/macvlan devices, then it essentially allocated a dedicated queue for our macvtap device (based on MAC address).

2.3.3 VHOST Descriptors

Each virtio/vhost provided buffer is identified by a descriptor. A single descriptor may be associated with multiple linked buffers. In the existing vhost implementation (mergeable buffers, big buffers), data is copied from host buffers to guest buffers, filling in one descriptor at time. The next descriptor to be used is specified by the vhost layer (in handle_rx()), and its data buffers are passed to the macvtap_recvmsg() function in a scatter-gather list. The macvtap code then copies as much data as it can that arrives on its socket to the vhost-provided buffers, returning the number of bytes written. The vhost layer then declares the descriptor as consumed and reports to the guest how many bytes were used in that descriptor. In Zero-Copy Rx, each descriptor is associated with a single buffer. The vhost layer no longer knows in advance which descriptor will be filled next, since the buffers can arrive from the Ethernet layer in any order. When we fill a buffer, we need to specify which descriptor has been consumed, and we need to specify how many bytes have been written to that buffer/descriptor. In some cases we also have to report an offset within the buffer where the data starts (as when doing TCP reassembly).

Buffers consumed by the Ethernet adapter are not necessarily consumed in the same order in which they were provided. Some special handling in the network stack may cause buffers to be delivered out of order. Also, sometimes we need to grab a buffer to copy an skb header that has no buffers attached to it (as when receiving an ARP or other broadcast packet). Vhost maintains tables of descriptors that are available and descriptors that have been used,

etc. Typically vhost reports the descriptor number and the number of bytes consumed from the specified descriptor. We needed to add a new field to inform the guest the offset in the buffer from which valid data begins. This required changes to the data structures used in the virtio/vhost protocol and in code that processes those data structures.

The default virtio/vhost implementation provided 256 descriptors per queue, while the Ethernet adapter we used managed queues of size 512 buffers. With buffers now being provided only by the VM guest (and not by the Ethernet adapter) we quickly ran out of available buffers for DMA and lots of packets were dropped. We were able to raise the number of virtio/vhost descriptors to 1024 by using a more recent version of Qemu (we used version 2.8.1).

2.3.4 Changes in virtio (Guest)

The existing virtio driver (drivers/net/virtio_net.c) allocates buffers of fixed size depending on parameters negotiated with the host. There exist 3 different schemes for the size of receive buffers: small buffers, big buffers and mergeable buffers. The scheme used is determined by negotiating capabilities provided by the host. A single scheme is used for all virtio devices of a particular VM. (Suggestion for improvement: allow different schemes for different virtio devices, depending on capabilities of underlying transport device.) The order of preference of the implementation is to use mergeable buffers, followed by big buffers, followed by small buffers. The small buffer scheme provides buffers of size about 1500, not necessarily page aligned. The big buffers scheme provides 16 4K buffers at a time for data via a scatter-gather type mechanism. The mergeable buffer scheme uses large data areas (e.g. 32K) and carves out of it multiple buffers of 1500 bytes into which multiple Ethernet packets can be copied. In all these schemes, data that arrives from the Ethernet adapter is passed up the hypervisor network stack to the macvtap/tun level, and is then copied from the hypervisor buffers into the guest-provided buffers. Because the data is copied, there are no gaps in the guest-provided buffers. The code in the guest takes advantage of the fact that the data is always at the beginning of the buffer (offset = 0) and the data is consecutive.

In order to implement Zero-Copy Rx, the underlying Ethernet adapter performs DMA directly into the guest buffers. The adapter expects its individual data pages to be page-aligned, so that is what we need to provide. We implemented a 4th buffer scheme in virtio to provide individual 4K pages (page-aligned) to be passed from the guest to the Ethernet adapter.

In the virtio protocol, in addition to the buffer allocated to hold data, there is also a (12 byte) buffer allocated to hold the vnet header. For big buffers, an additional 4K buffer is allocated from which the first 12 bytes are used for the vnet header and the remaining space is available into which to copy data. An skb points to the the list of buffers. For small buffers, the entire data buffer (including vnet header) is allocated in the body of an skb. Neither of

these options work when we need to provide 4K aligned buffers. Instead, we allocate a small skb header to point to a 4K aligned page, and use space in the cb (control block) field of the skb to hold the vnet header. We need to perform some extra bookkeeping to be able to recover the page from the skb when it returns with data. Additionally, since we do not perform the extra copy, we sometimes have to deal with the possibility of the data beginning at a non-zero offset in the data buffer. This occurs when multiple buffers are returned after using TCP reassembly offloading, in which the Ethernet headers of subsequent packets are stripped by the adapter code and combined into a large TCP packet. We therefore had to add an offset parameter to the virtio protocol to specify the offset within the page where the data starts.

2.3.5 Changes in Qemu

Qemu performs emulation of the underlying hardware to support running of different types of VMs on the same hardware. The emulation results in a large overhead for IO. Virtio was developed to bypass Qemu on part of the IO path (a paravirtualized driver instead of emulated) in order to improve performance. Even when using virtio, Qemu is still used to set up the association between the frontend guest device and the backend host device. The Virtio implementation supports several types of buffers (small buffers, big buffers, mergeable buffers) depending on the capabilities of the underlying hardware. Qemu facilitates the negotiation between VM and host during VM setup to decide what type of buffers will be used. We added an option to virtio devices to use 4K-page aligned buffers in order to support ZeCoRx. Therefore, support to negotiate this option had to also be added to Qemu.

2.3.6 Changes in ixgbe driver

We added several functions to the ixgbe Ethernet driver. First of all, we added the new functions required by Zero-Copy Rx in the net_device_ops structure.

```
.ndo_post_rx_buffer     = ixgbe_post_rx_buffer
.ndo_set_zero_copy_rx   = ixgbe_set_zero_copy_rx
```

The ixgbe_set_zero_copy_rx() function receives a base_dev parameter pointing to the (macvtap) device structure containing the MAC address to which buffers will be designated. We check whether this MAC address matches the MAC address specified for one of the queues (ring buffers) of the ixgbe adapter. If it matches, we mark that ring buffer as zero-copy enabled and handle it accordingly. We free all the buffers that were previously allocated to that ring buffer by the ixgbe driver. Only buffers provided by the VM guest are subsequently posted to the ring buffer. If the provided base_dev MAC address does not

match the MAC address of any ring buffer, it means that no ring buffer (queue) has been designated for that MAC address, and we return an error indication.

The ixgbe_post_rx_buffer() function is called to add a VM guest provided buffer to a ring buffer associated with a particular (macvtap) device. The skb parameter points to the buffer as well as to the device structure of the designated (macvtap) device. If a match is found, the buffer is mapped for DMA and is placed in the corresponding ring buffer; otherwise an error indication is returned.

After data is placed in a buffer, a software interrupt is generated to process the buffer and pass it up the network stack. This eventually causes the ixgbe_clean_rx_irq() function to be called, which then calls ixgbe_fetch_rx_buffer(). We implemented a zero-copy version of ixgbe_fetch_rx_buffer() to perform the different processing needed for a zero-copy ring buffer. After processing a buffer, it is sent up the stack via the existing napi_gro_receive() function.

2.3.7 Changes in macvlan

The macvtap driver sits on top of the macvlan driver, which in turn sits on top of the Ethernet driver. Interaction between layers is through function calls defined in the net_device_ops data structure (defined in include/linux/netdevice.h). When macvtap is called to post a buffer, it calls the lower layer ndo_post_rx_buffer() function. This actually calls the corresponding macvlan function, which, in turn, must call its lower layer ndo_post_rx_buffer() function (i.e. the corresponding Ethernet adapter function). Similarly, for the ndo_set_zero_copy_rx() interface, macvlan simply needs to forward the function parameters (base_dev) to the lower level Ethernet adapter function. No additional processing is required at the macvlan level. We did need to apply a patch to macvlan that allowed L2 forwarding offloads with macvtap. (See https://patchwork.ozlabs.org/patch/826704/).

2.3.8 Changes in macvtap

In macvtap, we added two main functions: one function (macvtap_post_rx_buffer()) to send buffers down the stack to the Ethernet driver and one function (macvtap_do_read_zero_copy()) to process filled buffers. In addition, we needed to make some changes to the macvtap_handle_frame() function to properly handle zero-copy buffers during failure.

2.3.8.1 macvtap_post_rx_buffer

The macvtap_post_rx_buffer() receives a single buffer from the vhost layer and performs the following:

- Verify that the socket is zero-copy enabled (i.e. that ndo_set_zero_copy_rx() had previously been run successfully).
- Allocate an skb to which to attach the buffer. Set the dev field in the skb to point to the macvtap device structure.
- Pin the buffer in memory and attach it to the allocated skb.
- Send the buffer down the stack by calling the lower level device ndo_post_rx_buffer() function.
- Add the buffer to the hash table of live buffers that holds the mapping between the vhost descriptor number and the buffer.

2.3.8.2 macvtap_do_read_zero_copy

The macvtap_do_read_zero_copy() consumes a single skb of data. The skb may point to multiple buffers, as when the system performed TCP reassembly. The main operations are the following:

- Get the next skb from the socket.
- For each buffer attached to the skb:
  - Obtain its vhost descriptor, data length, and offset;
  - Unpin the page;
  - Organize the information so that it can be consumed by vhost handle_rx_zcopy().
- Return the number of buffers being consumed.

2.3.8.3 macvtap_handle_frame

Packets are sent up the network stack by the Ethernet interrupt handler, eventually calling macvtap_handle_frame() to determine which macvtap queue is supposed to receive the packet. The packet is placed on the appropriate queue and is later taken off the queue (in another thread) in the function macvtap_do_read_zero_copy(). In macvtap_do_read_zero_copy(), the buffer is unmapped (unpinned), We return to the vhost driver the descriptor of the buffer and the amount of data in the buffer, and we then release the skb. Sometimes a packet is dropped in macvtap_handle_frame() because it is badly formed or because there is no room in the socket structure for more skb-s. These (now unused) buffers must also be returned to virtio with an indication that none of their space has been used.

2.3.9 Changes in vhost-net

Previously, the main function in vhost-net on receive was handle_rx(). The main loop in this function does the following:

- Get the number of bytes available on the socket from the lower-level drivers (Ethernet driver).
- Obtain sufficient virtio/vhost buffers and their descriptors from the vhost descriptor ring in order to copy the data from system buffers to guest buffers.
- Pass the virtio buffers and descriptors to the virtual device handler (macvtap) to read the socket and copy the data into the virtio buffers.
- Signal the user space thread to consume the descriptors that were used, indicating how many bytes from each descriptor contain valid data.

In Zero-Copy Rx, we need to separate the functionality into separate operations. We now want to post as many buffers as possible (up front) from the vhost descriptor ring to the Ethernet device driver. As these buffers get used, we want to inform the user space thread of which descriptors (corresponding to the buffers) were consumed. We therefore have two main functions to perform these operations: post_buffers() and handle_rx_zcopy(). In addition, we have a callback function, vhost_rx_zc_callback(), to free a buffer in case of error.

2.3.9.1 post_buffers

The main loop of post_buffers() does the following:

- Obtain a vhost descriptor and corresponding buffer.
- Pass the buffer and descriptor to the virtual device handler (macvtap) to send the buffer down the network stack.

Buffers are posted to the lower level until there are either no more buffers available or until there is no more room in the lower level driver (indicated by EAGAIN return value).

2.3.9.2 handle_rx_zcopy

The main loop of handle_rx_zcopy() does the following:

- Receive the next batch of buffers that were consumed by the virtual device handler (macvtap) together with relevant information (descriptors, lengths consumed from each buffer, offset within each buffer).
- Communicate to the virtio/vhost ring the descriptors that were consumed, their lengths and offsets.
- Signal the user space thread that buffers have been consumed.

2.3.9.3 vhost_rx_zc_callback

Sometimes a buffer gets rejected before it completes its full path through the network stack. The vhost_rx_zc_callback() function is provided to clean up the buffer so that its resources can be cleaned up and reused.

2.3.9.4 v_page_info structure

In Zero-copy Tx, a special data structure (ubuf) is provided by vhost to the virtual device driver (macvtap) to map a buffer to its vhost descriptor. This structure is passed to the virtual device driver by using the control field of the msghdr structure that is passed between them. We use a similar mechanism to pass information between vhost and macvtap for Zero-Copy Rx. We define a data structure that contains fields to hold various information that we want to preserve about the buffer and to pass information between the layers. We call this structure vhost_page_info and it looks like this:

```
struct vhost_page_info {
        struct page *page;
        int desc;
        int offset;
        int len;
        void (*callback)(struct vhost_page_info *);
        void *vnet_hdr;
        void *virt_page_addr;
        struct vhost_virtqueue *vq;
};
```

The desc field contains the vhost descriptor number and the page field points essentially to the buffer. The offset and len fields are filled in by macvtap to specify how many bytes of valid data are contained in the buffer and at what offset the data begins. The callback function pointer is used by macvtap_handle_frame() in case the handling of a buffer cannot be properly completed.

2.3.10 Interaction between vhost-net and macvtap

The interaction between vhost-net and macvtap is via the sendmsg() and recvmsg() functions defined in the proto_ops structure (include/linux/net.h). The recvmsg() function has a flags parameter. In order to implement Zero-Copy Rx, we needed 2 functions on the receive path: one to post buffers and one to receive buffers. We overloaded the recvmsg() function to support both of these functions by adding a flag to indicate which operation is being performed.

Sometimes (when we receive an ARP or other broadcast message) an skb arrives at macvtap that does not point to a buffer. The entire message was generated internally and fits in the skb. We need to grab an available VM guest buffer, copy the skb into it, and return the buffer to the VM guest. This required us to always provide macvtap with a spare virtio buffer and to implement a mechanism to indicate that this buffer was used.

2.4 Evaluation

The development of ZeCoRx is still work-in-progress. While we have implemented most of the pieces and we have end-to-end data flow, the code is not yet stable, and the TCP HW-assisted reassembly does not yet work properly. Without the TCP reassembly provided by the hardware, we cannot hope to achieve the performance gains that the existing implementations (mergeable, big buffers) achieve.

We performed an initial experiment to estimate how much we can gain by avoiding the data copy between host and guest buffers. When commenting out the lines that perform the copy when performing large data transfers we see a CPU usage savings of over 20%. This is our target. Once our code is stabilized, we intend to measure the performance improvements and hopefully write a paper summarizing our results.

A test plan was prepared and is summarized in a section of deliverable D6.2 (Architecture and Implementation Evaluation).

2.5 Configuration

In order to use Zero-copy Receive, the following operations must be performed:

- Download a linux source tree (version 4.8.0). Define one branch for the changes to the host and one branch for changes to the guest.
- Apply patch to host branch to enable macvtap/macvlan to run with l2-fwd-offload (https://patchwork.ozlabs.org/patch/826704/).
- Download recent version of Qemu that supports rx_queue_size=1024.
- Apply patch to virtio_net.c in guest branch for VM Guest to supply 4K page-aligned buffers.
- Apply patch to Qemu to negotiate 4K page-aligned buffers.
- Apply patches to host branch (macvtap, macvlan, ixgbe, vhost-net, etc) to support Zero-Copy Rx.
- Put Ethernet adapter into mode to allow l2 forward offload: (sudo ethtool -K eth_12 l2-fwd-offload on).
- Bring up VM via Qemu specifying rx_queue_size=1024 for virtio on the macvtap device.

2.6 Future

We intend to write a paper summarizing our results and (assuming we have significant performance gain) we intend to push our code to upstream into the Linux code base. We also

continue to investigate other potential performance improvements when running VMs in the Cloud.

3 vRDMA

3.1 Overview

In this section, we focus on the updates of the design and implementation of virtual RDMA (vRDMA) in the last year of the MIKELANGELO project. Based on the updated final architecture, more details about the updated components will be discussed. Then we present how network issues and configuration challenges can be solved for vRDMA driver and Linux host. More details about our contributions in each prototype and component are presented in D4.6 (OSv - Guest Operating System – Final Version) with a complete example.

In the very beginning of the project, the vRDMA has been designed with three prototypes, each of which has different architectures, scenarios and challenges. The goal of the final output of vRDMA is to provide a RDMA virtual driver with better communication performance between VMs and high flexibility for the virtualization environment. After vRDMA prototype II has been implemented, we have done a significant update for the final version of overall architecture design in D2.21. The new architecture, as shown in Figure 3 successfully merged prototype II and III into a bigger view, in order to summarize all the components and scenarios. Prototype I, which was taken as a starting point for the other two prototypes. has been excluded due to its only slightly improved performance. The overall design aims to provide solutions to support different types of guest applications with RDMA virtualization, allowing these applications to exploit benefits of virtio-rdma without any changes in the application code itself. Within the scope of MIKELANGELO project, we support guest applications including linux and OSv. Both socket and RDMA verbs are targeted in the design to support cloud and HPC applications.

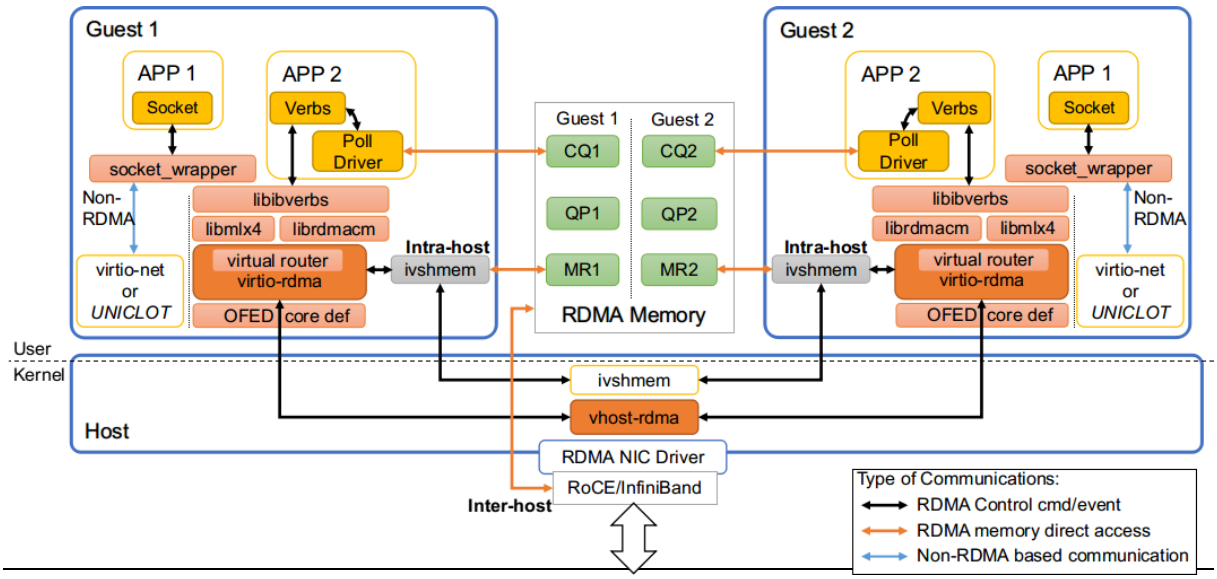

Figure 3: Overall Architecture of the final design of vRDMA based on Prototype II and III

3.2 Architecture

3.2.1 Main Updates

The architecture shown in Figure 3 was based on Linux guest, which has been implemented as planned in prototype III. However, when we continued working with the same architecture in OSv, the components were changed based on the supporting libraries and kernel implementation of OSv. Because OSv doesn't distinguish user space and kernel space, there will be a confusion of the InfiniBand data structure in user space and kernel space. For example, the rdma_cm_id is defined in user space (rdma_cm.h in librdmacm) as following:

```
struct rdma_cm_id {
      struct ibv_context      *verbs;
      struct rdma_event_channel *channel;
      void                    *context;
      struct ibv_qp           *qp;
      struct rdma_route        route;
      enum rdma_port_space     ps;
      uint8_t                  port_num;
      struct rdma_cm_event    *event;
      struct ibv_comp_channel *send_cq_channel;
      struct ibv_cq           *send_cq;
      struct ibv_comp_channel *recv_cq_channel;
      struct ibv_cq           *recv_cq;
      struct ibv_srq          *srq;
      struct ibv_pd           *pd;
```

```
      enum ibv_qp_type        qp_type;
};
```

And the InfiniBand kernel driver uses a different definition in the kernel rdma_cm.h header file:

```
struct rdma_cm_id {
      struct ib_device        *device;
      void                    *context;
      struct ib_qp            *qp;
      rdma_cm_event_handler    event_handler;
      struct rdma_route       route;
      enum rdma_port_space     ps;
      enum ib_qp_type          qp_type;
      u8                      port_num;
};
```

The InfiniBand kernel driver will be in charge of translating the user space data structure into kernel space. But we do not have such kernel driver support in OSv, because the additional InfiniBand kernel drivers cannot be easily ported to OSv, which might involve porting the whole Linux or FreeBSD kernel drivers into OSv. As a result, we fixed this issue by keeping only the user space data structure in OSv, and handed over the translation work in the backend driver on the host by implementing additional hypercalls. The implemented new hypercalls will be discussed in section 3.3.1 in more detail.

Another change of the architecture is that the virtual router functionality in the guest has been moved to host, and it has been simplified by utilizing a special address mask. This saved quite a lot of coding effort for implementing new functionality, and also made the deployment easier by configuring the network addresses and masks. More details about the related implementation will be presented in section 3.3.2.

Finally, due to underestimating the implementation workload, the shared memory support has been replaced with writing a technical white paper, which will be published on the project website.

3.2.2 Challenges

The most difficult challenge for the vRDMA design is security, e.g. how to avoid intruders from accessing the publicly shared RDMA memory regions, how to separate different RDMA regions among application and users. To achieve this goal, special effort needs to be contributed to supporting ivshmem, in order to have higher security mechanism of the public shared memory.

Live migration needs also additional update for the current design, i.e. snapshotting the current RDMA communication state, moving the RDMA memory region to a new VM, updating the RDMA context on the new VM, and finally restoring the communication from the snapshot.

Flow control is also an important future topic for vRDMA, as the RDMA device has only the knowledge of the RDMA context, e.g. Queue Pairs (QP), Completion Queues (CQ), and it doesn't know which VM the QP and CQ belong to. A mapping mechanism has to be designed and implemented to provide the possibility of flow control.

3.3 Implementation

3.3.1 RDMA Communication Manager

The RDMA Communication Manager (CM) provides neutral API, which follows the concept of socket API but adapted for Queue Pair based semantics, in order to setup reliable, connected and unreliable datagram data transfers. The setup communication of RDMA CM must be over a specific RDMA device, and data transfers are message based.

The RDMA CM manages the communication of the QP and controls the connection setup and teardown. It works in conjunction with the verbs API defined by the libibverbs library, which provides the underlying interfaces needed to send and receive data.

The RDMA CM is implemented with asynchronously or synchronously communication modes. The mode of operation is controlled by the user through the use of the rdma_cm event channel parameter in specific calls. If an event channel is provided, an rdma_cm identifier will report its event data, on that channel. If a channel is not provided, then all rdma_cm operations for the selected rdma_cm identifier will block until they complete.

The RDMA CM support is a major part for vRDMA prototype III, because rsocket and several communication modules in Open MPI depend on it. In order to support RDMA CM in vRDMA, several new hypercalls were implemented:

- vrdmacm_post_event: Function to forward RDMA event to the guest.
- vrdmacm_create_id: Corresponding backend function of the rdma_create_id hypercall from guest. It calls directly the host kernel to generate the RDMA CM id. For OSv, it also translates the guest user space id into host kernel space id.
- vrdmacm_destroy_id: Corresponding backend function of the rdma_destroy_id hypercall from guest. It calls directly the host kernel to destroy the RDMA CM id. For OSv, it also translates the guest user space id into host kernel space id.

- vrdmacm_bind_addr: Corresponding backend function of the rdma_bind_addr hypercall from guest. It translates the guest IP address to the host IP address, and binds the host IP address and port number to the RDMA device.
- vrdmacm_listen: Corresponding backend function of the rdma_listen hypercall from guest. It translates the guest IP address to the host IP address, and creates a listen RDMA CM id on the IP address and port number.
- vrdmacm_init_qp_attr: Corresponding backend function of the rdma_init_qp_attr hypercall from guest. It is used to change the QP state, for example, from initiated to RTS (Ready to Send) state. It calls ibv_modify_qp function internally.
- vrdmacm_resolve_addr: Corresponding backend function of the rdma_resolve_addr hypercall from guest. It translates the guest IP address to host IP, then resolves destination and optional source addresses.
- vrdmacm_resolve_route: Corresponding backend function of the rdma_resolve_route hypercall from guest. It resolves the route information needed to establish a connection.
- vrdmacm_connect: Corresponding backend function of the rdma_connect hypercall from guest. It initiates an active connection request.
- vrdmacm_disconnect: Corresponding backend function of the rdma_disconnect hypercall from guest. It disconnects a connection.
- vrdmacm_accept: Corresponding backend function of the rdma_accept hypercall from guest. It accepts a connection request.
- vrdmacm_reject: Corresponding backend function of the rdma_reject hypercall from guest. It rejects a connection request.
- ucma_copy_ib_route and ucma_copy_iboe_route: Ported functions originally from rdma_ucm.ko, which is used when calling vrdmacm_query_route to determine the route information based on the link types. This is only for OSv, as the InfiniBand route information can be only accessed by additional kernel driver, which doesn't exist on OSv. This can only be done by a hypercall that can perform the same call on the host side and return the results back to guest.
- vrdmacm_query_route; Corresponding backend function of the rdma_query_route hypercall from guest. This is only for OSv. It translates the guest source and destination IP addresses to the host ones, and then queries the route information for the guest.

3.3.2 Network Address Translation and Routing

As shown in Figure 3, the overall architecture is a hybrid of two different communication modes, including intra-host inter-VM and inter-host inter-VM communications. It was initially designed to use the Global Unique ID (GUID) of the RDMA device as the key, which can be

shared among the communication peers to identify whether the peers are on the same host or not. The virtio-rdma frontend driver maintains a list of the GUIDs of the local host, and checks if the communication peers' GUIDs, appended with the remote and source QPs, are matched in the list.

However, using the GUID as the key will involve at least one hypercall between guest and host, and additionally at least one broadcast communication from each host and guest for exchanging this key all over the network. The work for exchanging this information costs a lot additional coding effort to modify the existing data structure and related functions, and introduces huge virtio and network overhead.

We updated this design with a simpler solution, which requires only some network and IP address configurations for the guest and host. By using a special address mask, we will be able to identify if the communication peers are on the same host or not. We define the host IP address according to the following IPv4 format, where host_id is a unique id for each host:

```
subnet_id.host_id.port_num.1
```

The guest IP address is defined as following:

```
subnet_id.host_id.port_num.guest_id
```

The network mask for both guest and host IP address would be 255.0.0.0 in order to allow the routing through all the guests and hosts. The subnet_id defines the subnet, port_num is the RDMA port number specified from guest application, and the guest_id cannot be 1, which is reserved only for the host IP address.

We also defined a few masks for easy translation of the IP addresses:

```
// masks for network order address
#define ADDR_MASK_PORT 0x00ff0000
#define ADDR_MASK_HOST 0x0000ff00
#define ADDR_MASK_NET  0x000000ff
#define ADDR_MASK_GATE 0x01000000
#define TRANSLATE_MASK (ADDR_MASK_PORT | ADDR_MASK_HOST | ADDR_MASK_NET)
```

For example, we have two hosts and two guests on each host, then we will have following configuration of the virtual network as shown in Table 1.

Table 1: Example Configuration of a vRDMA virtual network.

Host 1	10.1.0.1	Host 2	10.2.0.1

Guest 1 on Host 1	10.1.0.101	Guest 1 on Host 2	10.2.0.101
Guest 2 on Host 1	10.1.0.102	Guest 2 on Host 2	10.2.0.102

With such configuration, the backend driver on the host can easily verify if a guest is on the same host or not by applying the mask to the sockaddr_in address of the guest:

host_address = (guest_address & TRANSLATE_MASK) | ADDR_MASK_GATE

Please note that, the port number in the address definition is the RDMA port, i.e. 1 or 2, and the IP address port number is stored in the socket_addr structure. The bit definition can be changed and extended, for example, the port_num can take higher 2 binary bits, and guest_id can be extended upto 14 binary bits in total, which is 16383 (2^14-1) guests per host.

The mask configuration is at moment hard coded in the backend driver, but it can be extended with startup parameters in order to be able to dynamically configure the masks.

3.4 Future

As we have the performance numbers available for prototype II and III, a paper is under preparation based on the achievements so far and will be published. The source code of vRDMA driver for Linux guest, OSv guest and Linux Host has been contributed into a private repository of the project, and they will be turned into public after the final cleaning up.

4 SCAM

This section provide the details of the updated SCAM (Side-Channel Attacks Monitoring and Mitigation) module developed within the MIKELANGELO project. SCAM serves as the security module being developed and implemented as part of sKVM, which is targeted at increased security against last-level cache (LLC) side-channel attacks. We recall that SCAM is designed as a software module, is essentially agnostic to any specific security-enabled hardware support, nor any specific patches made on the application level. SCAM is designed to provide security on the hypervisor level, specifically KVM, and as such it is an integral part of the sKVM implementation being developed within the project.

A detailed account of the components being developed within the SCAM module, as well as the attack surface it is targeting to secure, was provided in some of the previous deliverables as follows: (1) D2.13 provided a high-level description of the architecture of both sKVM, and in particular of SCAM. In section 4.1.2 we provide an update on this architecture. In a nutshell, the SCAM module does not make use of a kernel module, and is implemented completely in user-space. (2) D3.4[8]. described the details of our implementation of a prime and probe attack targeting the extraction of a private RSA key used by a co-located machine. Our attack is able to obtain this key within minutes. This attack, and the ubiquitousness of RSA as a cryptographic primitive, serves as the main motivation for our security solution being developed within the MIKELANGELO project, which targets both HPC infrastructure, as well as public (and private) clouds. (3) D3.2 provided the first report of the modules being developed as part of SCAM. These include three major components, namely, monitoring, profiling, and mitigation. Our initial results there show that monitoring High Performance Counters (HPC) is an effective way to identify prime and probe attacks of LLCs, and we present there a simple threshold-based profiling of malicious activity. We further first introduce our mitigation approach, noisification, which meticulously introduces noise into the samples obtained by an attacker, so as to significantly diminish its ability to extract information from the LLC side-channel being targeted.

The following sections are structured so as to remind the reader of the major concepts required to evaluate the progress of our work on SCAM. Specifically, in section 4.1.2 we describe the updated architecture of SCAM, as well as some of the major concepts underlying the attack, which are required to evaluate our improved monitoring and mitigation methods described thereafter. In section 4.2.1 we describe our improved monitoring component, which now focuses on the target alone. This allows us to provide monitoring "as-a-service", and does not require we have any information of the process acting as the attacker. In section 4.3 we provide a detailed account of our mitigation component, performing careful noisifcation of specific sets. We discuss its performance, as well as a tradeoff between security and required resources.

We note that all the results presented in this section are based on our implementation of a prime-and-probe attack against a TLS-server performing TLS authentication using a private 4096-bit RSA key.

4.1 Preliminaries

4.1.1 The Attack

We first review some of the concepts related to performing a prime-and-probe last-level-cache side-channel attack. The reader can refer to deliverable D3.4 for a detailed description of the attack and its performance.

We recall that the objective of the attack is to extract a private RSA key used by some server co-located on the same host as the attacker. Since the attacker and the server each run on their own VM, no target data is directly observable by the attacker. A common mechanism for extracting information by an attacker is to make use of a side-channel attack. We focus specifically on such side-channels arising from the fact that both the attacker and the target make use of shared hardware resources, and specifically in our case, they both make (inadvertent) use of the last-level cache (LLC).

Every time the target performs a decryption of a message using its private key, a sequence of actions is performed depending on whether the bits in the key are zero bits, or one bits. Conceptually, the standard implementation of the target decrypting a message can be described as follows:

```
for every private key bit b, in sequence, do
      multiply operation
      if b == 1
            another multiply operation
      endif
endfor
```

The attacker essentially tried to identify whether the if-statement is performed, for any given key bit. Being able to discern, for each bit, whether the if-statement is performed or not, would allow the attacker to identify which of the key bits are zero, and which of them are one, and in turn allow the attacker to extract the private key. As we have shown in deliverable D3.4, we were able to actually implement an attack based on these concepts, and extract the private key of a co-located machine within minutes.

At the core of the attack lies the prime-and-probe procedure. Under this regime, the attacker invalidates all the lines storing the target's data in a cache set. This set stores (some of) the instructions required for performing the multiply operation before each such operation

commences. By invalidating all these lines, the cache sets contains only lines corresponding to the attacker. In this case, having the target perform its multiply operation causes a cache-miss, which in turn causes the server to store its own cache lines instead of some of the attacker's lines.

By measuring its own hit-miss rate for each such invalidation of a cache set, the attacker can discern between the case where a multiply operation took place or not, and in particular whether the if-statement is performed or not. Clearly such measurements are not noise-free, but by repeatedly performing such prime-and-probes, the attacker can accumulate sufficiently many key-length samples of candidate keys, and using some further algorithmic approaches (e.g., solving problems of synchronization, pattern recognition, alignment, voting etc.), the attacker can recover the key.

As described in deliverable D3.2, SCAM components target two of the main characteristics of such attacks. These characteristics, which we refer to as *activity* and *sensitivity*, are briefly described here for completeness:

- ***Activity***: When the attacker *primes* a cache set, i.e., when it accesses all lines of a cache set so as to invalidate all lines that were previously potentially used by the target process/VM, the attacker causes various cache misses. This is an intrinsic component of the attack.
- ***Sensitivity***: When the attacker *probes* a cache set, i.e., access the lines of a cache set and evaluate whether there was a hit-miss on any of them, the attacker views any cache miss as an indication that the target has performed an operation which caused one of the attacker lines to be evicted from the cache. Such misses (or lack thereof) are the basic units of information gathered by the attacker throughout the attack. In particular, the attacker wishes to identify whether a cache miss occurred after the target has finalized its first multiply operation. In any such case, the attacker deduces that another multiply operation took place, which implies that the current key bit is a 1-bit.

We again refer the reader to deliverable D3.4 for a more complete description of the attack.

4.1.2 SCAM Architecture

For completeness, we provide here the high-level architecture of SCAM, as described and discussed in deliverable D2.13:

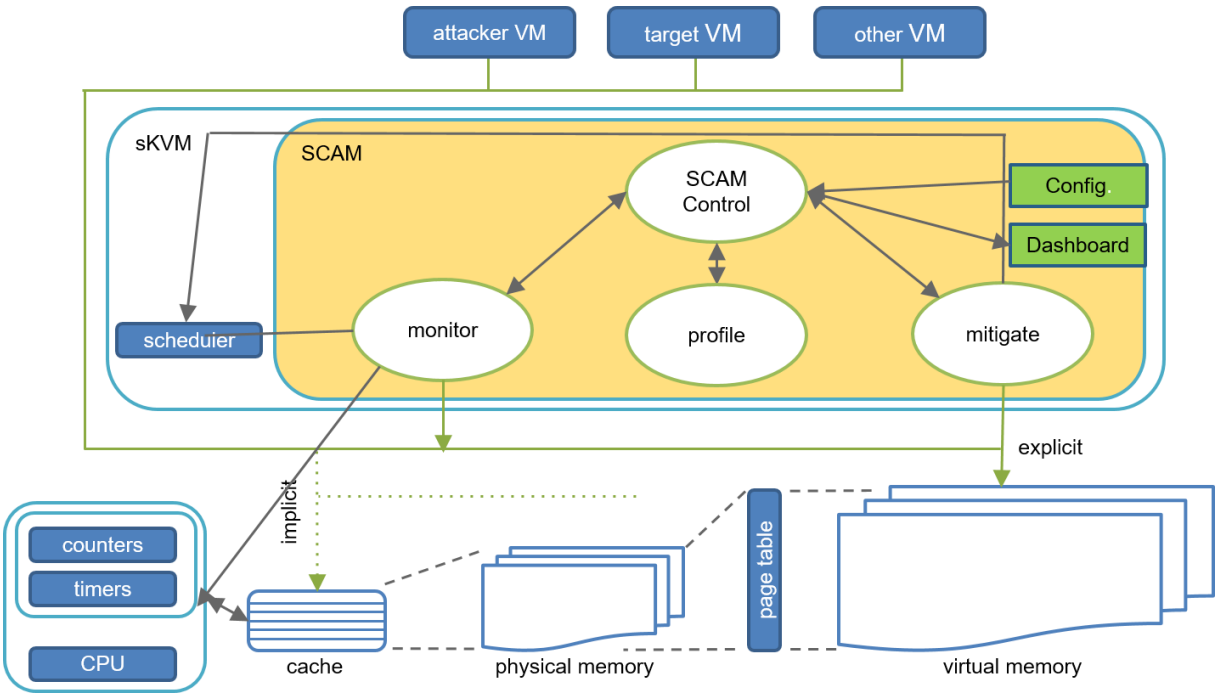

Figure 4: SCAM Architecture

We note that all of SCAM components are implemented entirely in user-space. This would facilitate seamless porting of SCAM to other kernels, without requiring any kernel recompilation. We note that the module interacts with various system counters (for monitoring) via the PAPI[9] package. Our module also requires access to CPU tick counters, which serve as system timers, for the offline setup procedure of the noisification component. However, this access is not required once the service has been set up, and timers are not accessed once the component is active and responds to malicious activity as identified by the monitoring service. After setup, the noisification process accesses cache lines implicitly by explicitly accessing specific virtual memory locations, according to the results of the reconnaissance phase performed during setup.

4.2 Monitoring and Profiling

In this section we present the progress of our monitoring and profiling services, as part of the SCAM module. We describe both details of its implementation, as well as the results of our performance study. We note that we only highlight the aspects required for evaluating our improved solution, and refer the reader to deliverable D3.2 where our high-level design and details of the initial implementation were described in depth.

4.2.1 Monitoring

Our proposed service for monitoring prime-and-probe LLC side channel attacks focuses on measuring the LLC activity, namely, the rate of accesses and misses. This exploits one of the

characteristics of the attack, namely, the *activity* characteristic. We adopt this solution due to two major properties: (1) low performance overhead, and (2) high detection accuracy. Our monitoring solution is based on data obtained from the PMU (Performance Measuring Unit), which is part of the vast majority of Intel's modern processors families. These units control a collection of HPC (High Performance Counter) registers, which collect statistics on various processor activities, including memory events and CPU events. Of particular interest to us are the counters that follow LLC events, namely LLC cache misses and LLC cache accesses. Due to the improved granularity of sampling, we access these counters via the PAPI (Performance Application Programming Interface) package, which may provide repeated access to these counters in intervals of a few microseconds (pending OS scheduling decisions).

The initial monitoring service provided by SCAM considered both the target and the attacker activities, as the subject being monitored. As can be seen in Figure 5, when considered side-by-side, these activities are highly correlated during an attack.

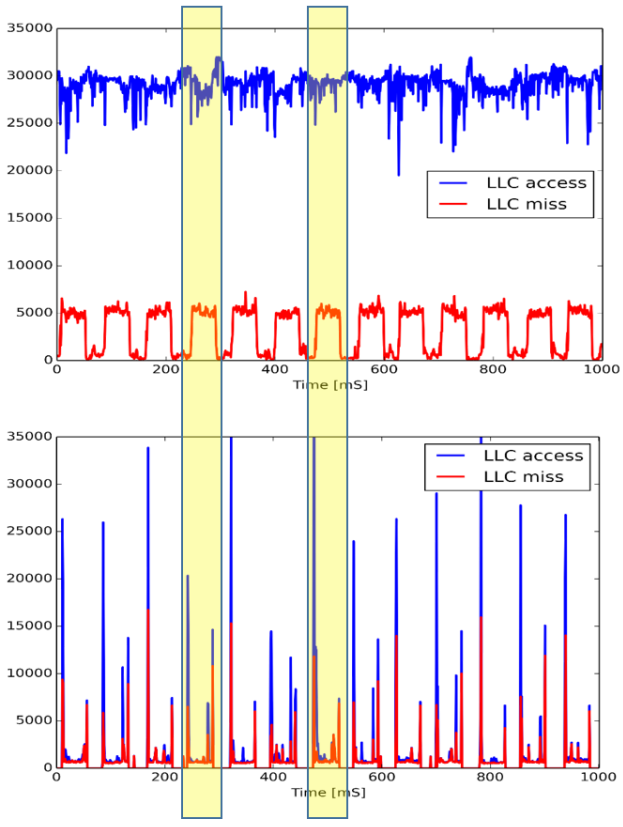

Figure 5: Correlation of LLC misses between attacker (top) and target (bottom) during an attack. The y-axis represents the number of LLC accesses and misses in a sample time window of 1ms, where the blue line represents the total number of LLC accesses (TCA), and the red line represents the total number of LLC misses (TCM) during the time window.

The distinct access pattern exemplified by the attacker, aligned with the effect it inherently has on the target, allow for clear identification of an ongoing attack, which can be detected in

a few milliseconds (where we recall that the overall act of decryption of a single message using RSA in our system takes on the order of 70 milliseconds). This effectively allows the attacker to obtain but a fraction of a key sample before the attack is detected.

While the above approach has proven itself to be extremely effective, it does have its downsides. In particular, the above approach requires monitoring all the VMs operating on the host, since any of them may be an attacker, and any of them may be a target. In our improved implementation of the monitoring service we focus on the target alone. This approach allows us to provide the SCAM monitoring capabilities as a service to any specific VM which requires increased security. This also reduces the overhead required for monitoring multiple VMs and processes in order to provide the service to (potentially) only one VM that may be a target.

When designing the monitoring and profiling of an attack based solely on the observable characteristics of the target (i.e., a VM which has registered for the SCAM monitoring service), we consider the characteristics of the target without any attacker present, as opposed to its characteristics while an attack is taking place. Figure 6 provides a side-by-side view of the LLC activity of the target in two settings; when the target is the only VM active on the host, and during an attack. As can be seen by these figures, the pattern exhibited by the target during an attack is significantly different than the its "peacetime" pattern. This is the key to our improved monitoring solution provided within SCAM.

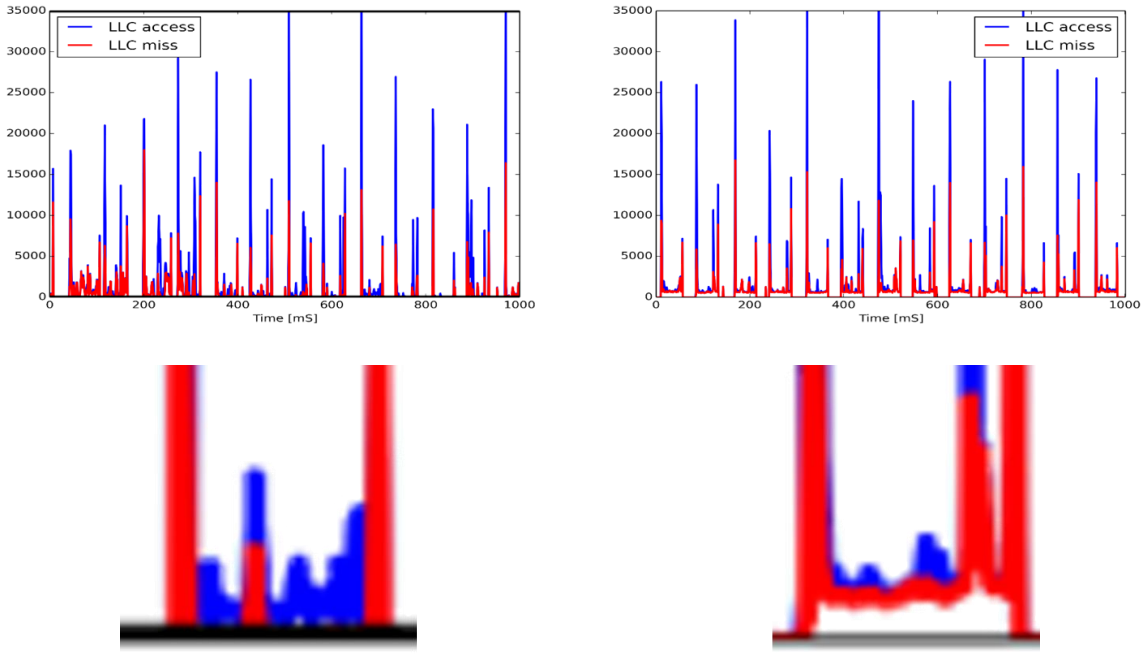

Figure 6: Comparison of the target VM cache activity with exclusive activity on host (left) and during an attack by a co-located VM (right). The top figures show the overall pattern, whereas the bottom figures provide a magnified view of a typical pattern observable during a single decryption performed by the target.

4.2.2 Profiling

Whereas the previous version of our monitoring service, as described in detail in deliverable D3.2, focused on identifying the attacker (which indeed exhibits a very distinct pattern), the focus of our improved monitoring and profiling service is to identify a target that is currently under attack. The underlying idea for this is that such monitoring would enable SCAM monitoring to be offered as a service to security-aware VMs. Furthermore, as mentioned earlier, such a target-targeted approach would also incur a smaller toll in terms of performance, since only the VM requiring the service is being monitored, and only its respective HPCs are being sampled and analyzed.

Our monitoring service considers two HPCs corresponding to the process ID (PID) of the target: the LLC access count, and the LLC miss count. Each of these counters is sampled every *s* microseconds, where we make use of the quickhpc package of PAPI to obtain such high-resolution of sampling. Each such interval of *s* microseconds comprises a *time slot*. The monitoring service calculates the cumulative miss-ratio in each time slot, which is the ratio between the number of misses and the number of accesses during this time slot. The service calculates a moving average over a sliding window of width *W*, and maintains a local counter *C* that is incremented by 1 in every time slot where the value of the moving average is above some predefined threshold $T \in [0,1]$. If at any time slot the moving average value drops below *T*, then the counter *C* is reset to 0. The monitoring service identifies an abnormal activity associated with a potential attack whenever the counter reaches some minimum threshold value R>1.

We recall that the performance overhead of sampling the HPCs was provided in D3.2. The results of the experiments performed for monitoring within the SCAM module are provided in deliverable D6.2. Our experiments show that the above method can effectively identify prime-and-probe LLC side channel attacks.

4.3 Mitigation by Noisification

In the current section we describe our method for mitigating cache side-channel attacks in virtualized environments, which is based on introducing deliberate noise into the cache thereby reducing the effectiveness of an attacker who extracts information from cache activity. We refer to our proposed mitigation approach as "noisification", and refer to the module implementing this approach as the noisification module (NM). It is implemented as a user-space process running on the host.

The goal for noise mitigation is to prevent the attacker from obtaining any data from a protected cache set. Recall that the attacker reads a bit by priming a cache set, that is by

storing a line from its memory space in every line of the cache set and then probing the set by reading all the lines and analyzing the number of cache misses during the probe.

The module has two phases, an offline phase and an online phase. In the offline phase a VM that requests protection for some part of its code runs the code in a loop with all other VMs inactive. The NM uses this phase to learn which cache sets are active during the sensitive code's execution. In the online phase the noisification module iteratively reads memory addresses that are mapped to the same cache sets as the sensitive code and data of the target. An attacker using prime and probe to obtain data on the cache-usage pattern of the target is expected to be thrown off by the noise that the module adds to its measurements.

The following subsections include a detailed description of the operation of the NM. Deliverable D6.2 presents an evaluation and measurements of the NM.

4.3.1 Offline Phase (Setup and Reconnaissance)

This phase includes two parts: constructing a mapping of virtual memory addresses to cache sets and discovering which cache sets are active when the protected target runs its sensitive code.

Construction of memory-cache mapping. The NM begins its execution on a bare system by building a mapping of its memory space to cache sets so as to be able to fill all lines in each cache set. The procedure for performing this task is the same as the one employed by the attacker when building its mapping of memory to cache sets, described in detail in deliverable D3.4. This procedure results in a mapping **φ: NM-memory -> LLC-sets**, such that each LLC-set **S** has at least **w** memory locations mapped to S, where w is the set-associativity of the LLC in the system. In formal terms, this can be described by having

$|\varphi^{-1}(S)| \geq w$.

Discovery of target cache activity. The target is activated without any additional active VMs and repeatedly executes the sensitive code portion. In the setting of a server performing RSA private key operations that would mean running that operation in a loop, possibly with appropriate messages from an external client. The NM discovers which cache sets are active by performing the same prime-and-probe procedure that the attacker uses, see deliverable D3.4.

Cache sets which show a distinct and repeated activity profile during this process are assumed to include sensitive code and data and are therefore potential targets for the attacker. We denote the collection of sets thus identified by M.

4.3.2 Online Phase (Noisification)

As stated previously the goal of the online phase is to prevent the attacker from obtaining information from cache sets in M. An ideal scenario for NM is to access every line of the cache set that the attacker reads between any two probes by the attacker. That way the attacker gets only cache misses regardless of target activity in the interval between the probes. The obvious problem with this scenario is that the NM does not know which cache set the attacker probes and in fact it is quite possible for the attacker to switch cache sets dynamically making any attempt to locate this set difficult. The NM is therefore reduced to adding noise to *each* of the sensitive cache sets. As a consequence there is a race between the attacker reading data from a single set and the NM accessing every sensitive set.

However, the critical component in the ideal scenario is that given the attacker's view, i.e. cache hits and misses, target activity or inactivity in the cache sets seems identical. If the NM can induce the view of the attacker to be identical, or very nearly so, whether the target is active or inactive in each sensitive cache set then the attacker does not learn information from the cache set.

The noisification algorithm is very simple. It takes as input M, the list of protected cache sets, **k,** the number of cores that can be used to add noise and ℓ_i, the number of lines in each set that the i-th core should read. The i-th core loops through reading $|M|\ell_i$ memory locations and all the cores together read $\ell_1+,...,+\ell_k$ locations that are mapped to each set in M and in particular to the set that the attacker probes. On the i-th core the following algorithm runs.

ActivateNoisification(M,ℓ_i)

1. For every set m in M let s(m)=$s_1,...,s_{\ell i}$ be a set of ℓ_i lines in S such that $\varphi(s_j)$=m for each s_j.
2. Let S(M) be a circular list of one word from each line in s(m) for all m.
3. Let head point to an arbitrary element of S(M)
4. While (true)
    a. Read the address that head points to.
    b. Advance head to next element of S(M).

The distribution of the hits and misses that the attacker's probe obtains depends on the number of lines that the noisification process inserts between two successive probes into the set that the attacker reads. The number of inserted lines between probes is correlated to the time it takes a core to read each location in S(M), and that time depends on the location of code of the algorithm and the addresses of S(M) in the memory hierarchy of the physical machine.

Addresses can be located in one of three cache levels or in main memory. Typical access time to each level is given as: 3 clock cycles for L1, 12 clock cycles for L2, 38 clock cycles for L3 and 195 if the address only resides in memory[10]. All the code can be safely assumed to reside in the L1 instruction cache since it is so small. Each line of cache in an Intel architecture is 64 bytes long. Therefore, S(M) requires $64|M|\ell_i$ bytes in cache storage. If $|M|\ell_i$ is very small then S(M) may fit in L1, but in that case there is a risk that ℓ_i is too small to effectively obscure the difference between attacker readings of target activity and target inactivity. In most typical cases S(M) fits into L2.

The number of cache lines per LLC set that are already in L1 or L2 are limited not only by the total size of a lower level cache but also by the set size in this cache which is typically smaller than the number of cache lines in the LLC. Covering more lines in an LLC set than exist in an L2 cache may require more than one core participating in the noise process, each contributing its own L2.

M is made up of cache sets in the LLC that can be roughly divided into the following categories:

1. Cache sets in which the only activity other than noisification is by the target. Recall that all sets in M are assumed to include target activity, so these sets do not have any other activity.
2. Cache sets in which the target, noisification and other random processes are active.
3. The cache set in which the attacker is active.

As long as $\ell_1+,...,+\ell_k$ is smaller than w (the associativity of the LLC) minus the number of lines other VMs access cache set c then all the accesses of the noisification in c are in L2 and are therefore fast. As a consequence setting $\ell_1+,...,+\ell_k \geq w$ will result in many LLC misses when the target is active since it inserts at least one line into each of the M sets invalidating a noisification line.

In some cases a memory access in the noisification process results in an LLC miss. That is often the case in the set that the attacker primes, regardless of the choice of $\ell_1,...,\ell_k$, since the attacker inserts w lines from its own memory into the cache set, invalidating any other line that was previously stored in the set, including all the noisification lines. However, even in that case the performance hit that the noisification takes is often much less than the expected time for main memory access. The reason is that modern CPUs provide Instruction Level Parallelism (ILP) and out-of-order execution exactly for such scenarios. While an instruction that caused a cache miss waits for data from main memory the next instruction can be executed as long as it doesn't depend on the blocked instruction. Since in the noisification process none of the instructions depend on the data read from memory, ILP

ensures that the subsequent instructions execute while an LLC miss is waiting for data from main memory.

To summarize, as long as the values of $\ell_1,...,\ell_k$ are chosen with care the noisification loop accesses data that is almost exclusively in the L2 cache except for the data in the cache set that the attacker probes. In that cache set on average the noisification process and attacker will have the same number of cache misses. However, while the noisification process can take advantage of ILP to continue its loop, the attacker has to intentionally block its execution until a cache miss is serviced from memory. The reason for this difference is that the attacker needs to measure the time an access takes to determine whether it is a cache hit or miss since that is exactly the mechanism it uses to obtain information on the target.

The difference in processing between the noisification and the attacker lead to roughly equivalent execution time for the attacker to prime and probe a single set and for the NM to add noise to many sets. Deliverable D6.2 includes an evaluation and typical measurements of the noisification process from the point of view of an attacker, showing the difficulty for an attacker in distinguishing between target activity and inactivity when the noisification is active.

5 UNikernel Cross Level cOmmunication opTimisation - UNCLOT

Report D2.21[11] (The final MIKELANGELO architecture) was the first report to include the UNCLOT component to address some of the burning issues of the OSv guest operating system. As the name suggests, UNikernel Cross Layer opTimisation explores ways to improve the performance of the unikernel with close collaboration between the hypervisor and the guest operating system. UNCLOT introduces a novel communication path for collocated virtual machines, i.e. VMs running on the same physical host. With the support of the hypervisor (KVM) the guest operating system (OSv) is able to choose the optimal communication path between a pair of VMs. The default mode of operation is to use the standard and unmodified TCP/IP network stack. However, when VMs reside on the same host, the standard TCP/IP stack is replaced with direct access to the shared memory pool initialised by the hypervisor. This allows the cloud and HPC management layer to distribute VMs optimally between NUMA nodes.

The aforementioned report also discussed serverless architectures as one of the pillars supporting the decision to research and implement a proof of concept component. Serverless architectures, adopted by all major cloud providers (Amazon, Google and Microsoft) have been using containers traditionally to successfully handle provisioning and management at scale. We argued[12] that OSv is ideal to implement such an architecture, because it is extremely lightweight, boots extremely fast and supports running most of the existing workloads, i.e. it mimics the advantages of containers. All the benefits of the hypervisor are furthermore preserved (for example, support for live migration, high security, etc.).

In light of the aforementioned facts, optimising the networking stack for collocated unikernels allows for an even better resource utilisation because the interrelated services will be able to bypass the complete network stack when sharing data between each other.

As shown in the figure below, the change of the architecture of OSv networking stack is rather simple. The host (hypervisor) needs to establish a shared memory pool and pass it to the virtual machines running on this specific host. Because we are looking for a solution that will allow running unmodified applications on top of UNCLOT, the standard networking API should not be changed, i.e. it should remain POSIX-compliant. On the other hand, we are interested in reducing as much of the networking stack as possible. To this end, functions implementing this API need to be aware of the parallel communication path through shared memory. UNCLOT modifies the implementation of these functions to identify communication between collocated VMs and redirects all packet transfers through the shared memory.

For all other communication, either between host and the VM or between VMs running on different hosts, OSv's default network stack is employed without any interaction with UNCLOT.

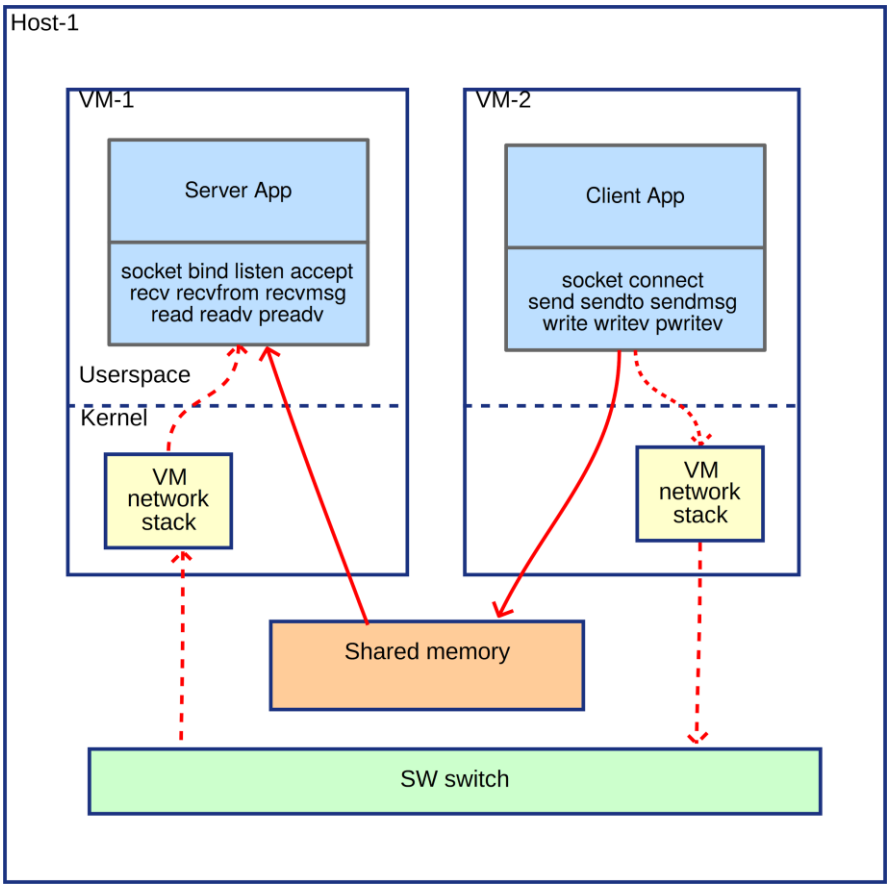

Figure 7: High-level architecture of UNCLOT.

Further details on the design of UNCLOT are presented in report D2.21. The following subsections provide information about the specific implementation details that were addressed during the implementation of the proof of concept. Ideas for future work are briefly outlined as well as instructions on how to use the current version of this component.

5.1 Implementation

Because UNCLOT is a completely new component in its initial phase, this section will focus primarily on the technical details of the implementation. This will explain some of the rationale and allow for comprehensive understanding of the design decisions made. It will further help interested readers to dive into the component and experiment with the improvements and extensions.

UNCLOT implementation is based on replacement of the standard network socket API implementation with a shared memory implementation. For inter-VM shared memory the

IVSHMEM was used. IVSHMEM shared memory is supported in mainline QEMU, and is exposed to VM as a PCI device. Hence we first needed to develop a device driver for IVSHMEM PCI device for OSv.

Using shared memory as a single large block of memory is rather difficult and error prone. It is more user friendly if arbitrary sized chunks of data can be allocated on demand. When shared memory is used in a standard Linux application, the System V shared memory interface (http://www.tldp.org/LDP/lpg/node21.html) is one of most commonly used interfaces. A similar interface for allocation of memory chunks from IVSHMEM device was therefore implemented in OSv.

When multiple VMs need to allocate memory from the same IVSHMEM device at the same time, synchronization is required to prevent data corruption. Some care is required as usual OSv synchronization primitives (i.e. mutex class) are only applicable within a single address space of one virtual machine instance and cannot be used across many VMs. Thus the first step was to implement a dedicated lock class, suitable for use from multiple VMs. This lock ensures stable and synchronized allocation of new memory chunks from an arbitrary VM running on the same host.

Allocated memory chunks are used to hold in-flight TCP data - e.g. as a send or receive buffer. Each TCP connection requires two memory chunks for proper operation. One is used as the send and the other as the receive buffer. The allocated memory chunk is managed as a circular ring buffer. This way the same memory chunk can be reused continuously. When the end of the buffer is reached, it continues at the beginning. Because that data has already been read and processed, this approach does not corrupt the transfer of the data.

TCP connections are uniquely identified by source IP, source port, destination IP and destination port. In case of collocated VMs where shared memory can be used, special logic needs to be built into the networking logic. When data is sent, the network socket file descriptor needs to be converted into the corresponding circular ring buffer. After this, the data is written into the ring buffer instead of the corresponding network socket. In a similar way data needs to be read from a corresponding ring buffer instead of from the network socket. A utility class sock_info has been implemented to maintain relationships between the TCP connection source IP, source port, destination IP, destination port, file descriptor and the send/receive ring buffer. Lookup functions are provided to find correct the ring buffer based on source/destination IP/port or based on file descriptor.

Before data can be sent/received via network socket, a TCP connection needs to be established. This requires one VM to bind(), listen() and accept(), and the second VM to connect() to the first VM. In order to fill sock_info class instances with connection information during connection establishment, a set of hooks has been inserted into the network packet

send and receive path (this is discussed in more detail in subsection Intercepting TCP Connection Establishment below). This ensures that once OSv finalizes a TCP connection by assigning it a file descriptor, all other details about the connection (IPs and ports) are also known.

This simple mechanism already provides sufficient capabilities to support the simplest scenario where one VM only sends some data and the other, collocated, only receives the data.

However, as soon as the application wishes to read data before they have been actually sent, it has to be blocked until data is available. Traditionally this is done with the help of select(), poll() and/or epoll() system functions. In order to be able to support such (common) scenarios these functions required specific changes for when a shared memory channel is used. Changes are comprised of two parts. The first part implements two functions that tell whether data can be read or written from/to the socket without blocking: soreadable() and sowriteable(). In case a thread needs to be blocked, the second part ensures that it can be woken up once the data that the socket expects are available, either for reading or writing.

5.1.1 IVSHMEM device driver

The IVSHMEM shared memory region is presented to the guest OS as a PCI device. The IVSHMEM device implements 3 PCI BARs (base address registers):

- BAR0 holding the device register,
- BAR1 holding the MSI-X table and PBA (pending bit array), and
- BAR2 mapping the shared memory object.

The BAR2 register, i.e. the actual shared memory object, is sufficient for the basic operations.

The driver has been implemented based on existing virtio drivers. Similar to other PCI drivers, it is registered with OSv in the arch_init_drivers() function (file arch/x64/arch-setup.cc).

```
#include "drivers/virtio-ivshmem.hh"
...
drvman->register_driver(virtio::ivshmem::probe);
```

The driver is implemented in `drivers/virtio-ivshmem.hh` in the OSv drivers/ directory. This driver is invoked when OSv finds an IVSHMEM device on its PCI bus. The main task of the driver is to map BAR2, so that a PCI (physical) memory address can be accessed from a virtual memory address. This is done in the ivshmem class constructor:

```
namespace virtio {
ivshmem::ivshmem(pci::device& pci_dev)
        : virtio_driver(pci_dev)
        , _irq(pci_dev, [&] { return ack_irq(); }, [&] { handle_irq(); })
{
      pci::bar *bar;
      bar = pci_dev.get_bar(IVSHMEM_DATA_BAR_ID);
      bar->map();
      _data = (void*)bar->get_mmio();
      _size = bar->read_bar_size();
...
}
```

A VM with IVSHMEM device allocating 128 MB of shared memory can be configured through QEMU command line configuration: `--device ivshmem,shm=ivshmem,size=128M.` This shared memory is accessible to all VMs. It is up to individual VMs to ensure that they don't write to parts intended to be used by other VMs (e.g. to not corrupt memory of other VMs).

Running OSv VMs can further be simplified using the scripts/run.py script found in the OSv source tree: `./scripts/run.py --novnc -nv -c4 --pass-args '--device ivshmem,shm=ivshmem,size=128M'`. The IVSHMEM is configured by simply passing the argument to the QEMU command.

More advanced usage of interrupt notification is possible using the other two registers, i.e. BAR0 and BAR1. To properly support interrupts QEMU version 2.6 or above must be used. Furthermore, IVSHMEM device must be configured to use device type ivshmem-plain or ivshmem-doorbell, i.e.

- `--device ivshmem-plain,shm=ivshmem,size=128M` or
- `--device ivshmem-doorbell,shm=ivshmem,size=128M`.

5.1.2 IVSHMEM System V - like interface

The IVSHMEM data memory is difficult to use directly. To simplify usage, a System V - like interface is implemented. System V shared memory interface (http://www.tldp.org/LDP/lpg/node21.html) is implemented in the following functions:

- shmget to allocate a memory segment (returns a new segment ID)
- shmat to attach to memory segment (returns a void* pointer to a memory chunk) and
- shmdt to detach a memory segment.

Our interface mimics System V interface, and is implemented in functions:

- ivshmem_get to allocate a new chunk of memory. An integer ID is returned.
- ivshmem_at to obtain a void* pointer to chunk of memory.
- ivshmem_dt to free pointer returned by ivshmem_at.

To implement the interface the size and address of each memory segment (class ivshmem_segment) are stored internally. A list of all ivshmem_segment instances is stored in a list at the beginning of IVSHMEM mapped memory. This way these segments are shared between all VMs using the particular IVSHMEM memory. The list is implemented as a fixed size array.

Memory segments are allocated in a sequential manner. The first free memory chunk that is large enough to allocate the segment is used. This could lead to memory fragmentation after repeated allocation/deallocation if memory segments were of random size. To this end, the current implementation of UNCLOT relies on fixed sized buffers for all TCP connections. In case future implementations would require variable sized buffers, more advanced memory allocation strategies would be employed to prevent fragmentation. Such techniques are well researched and known from basic memory allocators (e.g. malloc implementations).

To test for correct operation and to demonstrate usage, a test case for the interface (functions Ivshmem_get/at/dt) has been implemented in tests/misc-ivshmem-get.cc. The test runs in a single VM, where 2 threads allocate and deallocate memory.

5.1.3 IVSHMEM Synchronisation

Allocation and deallocation of memory segments from multiple VMs have to be synchronized to prevent memory corruption. When two VMs simultaneously search for a new free memory segment, they should get two distinct pieces of memory. In order to achieve this, the system must provide a mechanism that will allow at most one thread to modify the list of allocated memory segments, which is, in turn, used to find available memory. However, the available synchronisation primitives such as mutex class from include/osv/mutex.h and include/lockfree/mutex.hh work only for threads from the same VM instance. These primitives maintain a list of threads waiting for the specific lock as shown in the following code segment (variable waitqueue):

```
class mutex {
protected:
    std::atomic<int> count;
    // "owner" and "depth" are only used for implementing a recursive
mutex.
    // "depth" is not an atomic variable - only the lock-owning thread
sets
    // and reads its own depth. "owner" is atomic - one thread doing
```

```
lock()
    // needs to read the current owner possibly set by another thread -
but
    // it can be accessed with relaxed memory ordering.
    unsigned int depth;
    std::atomic<sched::thread *> owner;
    queue_mpsc<wait_record> waitqueue;

    //...
}
```

This mutex, when initialised in one instance (VM1) can therefore not be used from another instance (VM2), even when running in the same host. This is because the pointers to threads from VM1 is invalid in VM2. To overcome this, UNCLOT currently uses the simplest synchronisation primitive, spinlock. Spinlock acquires a lock by setting a specific flag. If lock is not free, the thread/waiter just cycles in a loop until a flag is clear. This is acceptable for our use case, as only a small amount of work is done while the lock is being held: we only need to traverse a list of memory segments to find a free memory chunk. The spinlock is stored in the IVSHMEM area, so that all VMs can access it.

5.1.4 Circular ring buffer

Allocated memory segments are used to hold sent and received data. A wrapper class around plain memory block was developed to ease usage of allocated memory blocks. The RingBuffer class allows push and pop or variable sized data. This class automatically handles wrap around when end of memory block is reached, starting again from the beginning. It also provides information about the amount of free memory (when writing data to the block) and the amount of available data (when reading data).

The implementation is based on OSv lockfree ring from include/lockfree/ring.hh. The main difference is that our implementation does not store the list of fixed sized items, but rather a variable sized items, i.e. the data which will be sent to/received from a TCP stream.

5.1.5 TCP Connection Descriptor

Each ring buffer belongs to a single TCP connection (one TCP connection contains two ring buffers). A central list of opened TCP connections is maintained in the IVSHMEM area, shared between all VMs running on the same host (more specifically, all that share the same IVSHMEM device, as multiple devices can be created on the same host). Each connection (socket) is described with a sock_info class. The main data stored for each connection are source and destination IP and port, and the ring buffer containing the actual data. The corresponding file descriptor ID (integer) is furthermore stored to simplify mapping between

TCP connections, which are most commonly accessed via file descriptor, and ring buffers. This file descriptor is therefore used as the key to lookup the sock_info instance belonging to a given file descriptor.

5.1.6 Intercepting TCP Connection Establishment

TCP connection is established by calling connect() from VM1 to a listening socket in VM2. The socket in VM2 enters a listening state by calling bind() (to select IP addresses/interfaces it is willing to use) and listen() to allow connection attempts. Afterwards, it is ready to wait on connection attempt from peer with listen(). After peer VM1 initiates the connection, listen() in VM2 returns the file descriptor of the newly created TCP connection. After connect() in VM1 returns successfully, VM1 file descriptor can be used to send and receive the data.

In order to establish corresponding buffers in shared memory, the code of bind(), listen() and connect() needs to be changed. Changes identify the relation between the file descriptor in VM1 and the same TCP connection in VM2.

Because the listening socket must be able to accept connection attempts from collocated VMs at the same time as those VMs running on a different host, UNCLOT can not just replace the existing real TCP accept() with the version that would allow only UNCLOT connections. As soon as connect() is called, the destination IP address is known. Consequently, it is already possible to choose between the standard (real TCP) path and the path using shared memory, skipping real TCP path completely. However, the listening socket already waits for a real TCP connection attempt, so skipping the real TCP connect would require additional modifications in accept() to stop the waiting. To circumvent this UNCLOT connect() always establishes the real TCP connect().

In case of blocking sockets connect() waits, until the peer VM accepts the connection (more accurately, until 3-way handshake is finished). After connect(), the returned file descriptor can be used immediately. The same is true for the file descriptor returned by accept().

In case of non blocking sockets connect() returns immediately with errno set to EINPROGRESS. At that point we cannot yet write to or read from the socket. We also do not know when (or if) the other peer will actually accept the connection attempt or not. The connection establishment continues in background in OSv network stack - see function tcp_do_segment() in tcp_input.cc, executed by thread virtio-net-rx. The tcp_do_segment drives TCP state machine for 3-way handshake (SYN - SYN/ACK - ACK), and by inserting calls to our hook function at these places, UNCLOT can be notified when client VM gets SYN/ACK reply from the server and when the server VM gets ACK reply from the client. At that point OSv finalizes TCP connection setup on client and server side, respectively. This is also the time

when UNCLOT can properly finalise the setup of sock_info class instance used to represent the given TCP connection.

5.1.7 Intercepting Basic TCP Data Functions

Data can be sent to TCP connection via: write(), writev(), sendmsg(), send(), sendto(), and pwritev().

All of them call either sys_write() or sosend() under the hood. The call chain is:

- write -> pwrite -> sys_write
- writev -> pwritev -> sys_write
- sendmsg -> linux_sendmsg -> linux_sendit -> kern_sendit
- send -> linux_send -> sys_sendto -> sendit -> kern_sendit
- sendto -> linux_sendto -> linux_sendit -> kern_sendit

The sys_write(struct file *fp, ...) calls fp->write, which is a write implementation specific to file class. For TCP sockets this is implemented in socket_file::write(), which calls sosend_generic(). Similarly kern_sendit() calls sosend_generic() responsible for actual sending of the data.

In a similar way the calls to recv(), read(), recvfrom(), recvmsg(), readv() and preadv() use sys_read() and/or soreceive_generic(). This makes sosend_generic() the most generic place to send data via shared memory instead of via network.

Despite this, our initial implementation of UNCLOT adds support for bypassing the networking stack by writing to a ring buffer from within write() function. This way we were able to focus on the simplest mechanism to send the data allowing us to identify potential pitfalls of the implementation. This also reduces some cycles that are otherwise lost while traversing the above call path. Later the bypass was extended with a similar code into send/sendto/sendmsg implementations.

5.1.8 Implementing epoll support

The epoll support is required to allow a reader to release the CPU (e.g., to “sleep”) when there is no data to be read. In a similar way the writer releases the CPU if it cannot write to a socket at the moment. Similar functionality is also provided by the select() and poll() system calls. In OSv, select(), poll() and epoll() call common functions under the hood; poll_scan() and soreadabledata/sowritabledata() are main functions required to implement the functionality.

The thread wishing to do network IO is put to sleep if none of the file descriptors registered with epoll() is ready for IO. When that happens, the thread is added to the epoll_file _waiters list. When the file descriptor becomes ready, the thread has to be woken up, so that it can process received data (or send data to socket). For UNCLOT sockets we trigger wake up by

reusing code from standard sockets. When new data is written, we call poll_wake() in the receiver VM, with file pointer of the receive socket as a parameter. This ensures to wake up the reader thread. But before passing processing from epoll_wait() or poll() to application code, the OSv also checks if there is actual data to be read (or space to write data when sending). This is done by soreadabledata(). We modified soreadabledata() and sowritabledata() to also check the UNCLOT ring buffer in case of UNCLOT sockets.

The sender thread is in a different VM as the receiver thread. This means that the code writing data to the UNCLOT ring buffer cannot directly wake the receiver thread. The sender VM should send an interrupt to receiver VM to signal new data. As an initial solution we implemented a simpler approach, where a helper thread periodically scans UNCLOT sockets for unread data and wakes up threads waiting on them.

5.2 How to use UNCLOT

UNCLOT is available as a separate branch of the OSv source code. Using it requires building an OSv image with UNCLOT support. The TCP applications are unmodified. The example below shows how to build and run an UNCLOT-enabled image with

- Nginx HTTP server and
- wrk HTTPtest client.

The OSv branch is available in the project's public Github repository[13] (branch unclot-v1). The Nginx and wrk are available in upstream osv-apps repository[14].

To build the image following commands are used:

```
git clone https://github.com/mikelangelo-project/osv/
cd osv
scripts/setup.py
git fetch origin
git checkout unclot-v1
git submodule update --init --recursive
scripts/build image=cli,nginx,wrk -j8
```

This results in release image at build/release/usr.img. Next, we run two VMs from the image. We assume that VM0 with ethernet MAC address of 52:54:00:12:34:56 gets IP address 192.168.122.90 and that VM1 with MAC address 52:54:00:12:34:57 gets IP address 192.168.122.91.

```
sudo /bin/cp build/release/usr.img /tmp/vm0.img;
sudo ./scripts/run.py --novnc --gdb 1234 -nv -c2 --mac 52:54:00:12:34:56
```

```
-V -i /tmp/vm0.img --pass-args '--device ivshmem,shm=ivshm,size=1024M' -e
"/cli/cli.so"

sudo /bin/cp build/release/usr.img /tmp/vm1.img;
sudo ./scripts/run.py --novnc --gdb 1235 -nv -c2 --mac 52:54:00:12:34:57
-V -i /tmp/vm1.img --pass-args '--device ivshmem,shm=ivshm,size=1024M' -e
"/cli/cli.so"
```

This starts two VMs, both paused at their command line interface. Now, we can start Nginx in VM0 CLI:

```
/# run '/nginx.so -c /nginx/conf/nginx.conf'
```

After Nginx is initialised, we can also start wrk in VM1 CLI.

```
/# run --newprogram '/wrk -c1 -d1 -t1
http://192.168.122.90:80/basic_status'
```

The --newprogram option is required for wrk, as it uses luajit library, which is incompatible with liblua used by OSv CLI. Both LUA implementations use identical function names for some of the functions. This results in luajit calling the implementation from liblua and prevents wrk from starting. Running wrk in a separate ELF namespace (the --newprogram option) resolves the issue.

After wrk finishes, it prints result of the performance test. For example:

```
  Thread Stats   Avg    Stdev       Max   +/- Stdev
     Latency     3.68us      1.12us   61.00us    96.04%
     Req/Sec   225.94k       5.50k   230.33k    93.33%
  188633 requests in 989.35ms, 45.05MB read
Requests/sec: 190663.37
Transfer/sec:     45.53MB
```

6 Concluding Remarks

This report summarizes the final status of the components developed in sKVM. Some components have reached an advanced level of maturity: IOcm, vRDMA and SCAM. These components have been tested individually, and shown to work on the various use cases according to the design of the components. Synthetic benchmarks, focusing on specific aspects that these components address show that theoretical improvements are beneficial. Real-world use cases, on the other hand, demonstrate less significant improvements which can be attributed to the fact that the chosen workloads are not network bound, i.e. limited by the capacity of the networking. These components are available to the community as Open Source.

sKVM furthermore introduces two new components, ZeCoRx and UNCLOT, that were designed based on the needs identified during the project. These two components are still work-in-progress and their implementations are considered proof-of-concept. This report described them in more detail, providing technical details and caveats. Both are to be released as open source to make them available to the community. Future work is also presented because we believe both of these components address modern architectures and technologies such as software defined networking and microservice (serverless) architectures.

7 Appendix: Using SCAM

The following serves as a guide for using SCAM. The code is available on git[15], and the following information is available in README.md in the said repository.

7.1 Overview

SCAM is a user space module that identifies cache side-channel attacks using the prime-and-probe technique and mitigates the effects of these attacks. The first function of SCAM is monitoring, which identifies attacks by analyzing the data of CPU counters collected through the PAPI library. The second function of SCAM is mitigation of an attack by adding noise to the cache in a way that makes it hard for an attacker to obtain information from priming and probing the cache.

Testing SCAM is possible with three virtual machines that can be downloaded separately. The VMs include a target, which is a TLS server based on a slightly modified GNU-TLS server, a TLS client and an attacker that obtains the server's private RSA key.

7.2 Installation

7.2.1 SCAM

- Download all files from the GIT.
- Go to: /quickhpc/papi-5.5.1/src and run ./configure followed by make.
- Go to: /quickhpc and run make.
- Go to: /scam_noisification and run make.
- **If the number of cores is not a power of two**: -Go to /scam_noisification/src/l3.c and make sure that the FULLMAPPING flag is defined.

7.2.2 VM Setup

Load the VMs from:

http://opendata.mikelangelo-project.eu/index.php/s/sz6mZtBIQzjdsgk

Configure the memory pages to "huge pages" and setup networking by running the following instructions:

```
sudo mount -t hugetlbfs hugetlbfs /hugepages/
sudo chmod 777 /hugepages/
sudo brctl addbr br1
sudo ip addr add 192.168.179.1/24 broadcast 192.168.179.255 dev br1
```

```
sudo ip link set br1 up
sudo ip tuntap add dev tap1 mode tap
sudo ip link set tap1 up promisc on
sudo brctl addif br1 tap1
sudo dnsmasq --interface=br1 --bind-interfaces --dhcp-
range=192.168.179.10,192.168.179.254 #add dhcp for all interfaces bound
to the bridge.
```

7.2.3 VM Launch

Launch the three virtual machines by:

Attacker

```
sudo kvm -m 2G -hda $PATH$/VM1.qcow2 -mem-path /hugepages -boot c -smp 2
-name attacker -device e1000,netdev=tap1,mac=52:54:00:00:02:04 -netdev
tap,id=tap1,script=/etc/qemu-ifup &
```

Username: attacker & Password: Qweasdd.

Go to /FillingTheCache/ and type make

Attacker program- (./FillingTheCache/FillingTheCache)

Server (Target)

```
sudo kvm -m 1G -hda $PATH$/VM2.qcow2 -boot c -smp 1 -name SERVER -device
e1000,netdev=tap1,mac=52:54:00:00:02:05 -netdev
tap,id=tap1,script=/etc/qemu-ifup &
```

Username: attacker & Password: 1234567

Go to /Desktop/Project/CseProject/GnuTLS_Server/ and type make

Server program- (./Desktop/Project/CseProject/GnuTLS_Server/GnuTLS_Server)

Client

```
sudo kvm -m 512M -hda $PATH$/VM3.qcow2 -boot c -smp 1 -name CLIENT -
device e1000,netdev=tap1,mac=52:54:00:00:02:03 -netdev
tap,id=tap1,script=/etc/qemu-ifup &
```

Username: attacker & Password: 1234567

Go to /Desktop/Project/CseProject/GnuTLS_Client/ and type make

Client program- (./Desktop/Project/CseProject/GnuTLS_Client/GnuTLS_Client)

7.3 Test

Testing SCAM requires several steps which are described below. SCAM must be launched ahead of all the virtual machines to map the Last Level Cache by associating memory addresses in SCAM's virtual memory to cache sets. Then the target server and the client are initialized (see above) and run the handshake. In this phase SCAM learns the cache sets that the server uses. The last step is to launch the attacker.

Perform the following steps to run the test:

Start SCAM by

```
sudo python scam.py
```

An xterm window will open with the scam-noisification program output which is only used for debug.

Wait until the notification 'Turn on the target, start the decryption process, and press any key...' is shown and launch the server and client VMs. Verify that TLS handshakes are executed in a loop by messages printed on the screen.

Press any key and wait for SCAM to identify all the cache sets active during a TLS handshake. The completion of this phase is indicated by the 'to start monitoring, please enter target PID:' message. Enter the Target process PID from the hypervisor point-of-view by the following steps. Run htop, enter F5 for the tree view, locate the process tree of the VM running the TLS server by its name (in the attached VM this name is "qemu SERVER") and select the PID of the lowest sub-process of this VM's hierarchy.

Launch the attacker.

7.4 Configuration parameters

/Scam/scam.ini configurable parameters:

- **scam_cores** - the ID of the CPU core that SCAM will use.
- **plot_enable** - for debug, shows the latest data on cache accesses and cache misses for the target PID.
- **window_avg_thresh** - monitoring assumes that an attack is taking place when the ratio of cache misses to cache accesses is high for a long period of time. The parameter window_avg_thresh defines a threshold on this ratio, that takes values between 0 and 1 and is 0.8 by default. Reducing the threshold may result in more false positives (classifying legitimate traffic as attacks) and may also identify attacks that otherwise go undetected. Increasing the threshold has the opposite effect.

- **detect_thresh** - the length of time that monitoring data must stay above window_avg_thresh to constitute an attack. The threshold is 20 milli-seconds out of a totla of 65 ms for a 4096-bit private key operation. This threshold should be modified to a fraction (no more than a third) of the time necessary to complete a private key operation by the server VM.
- **min_rumble_duration** - minimum time of a "noisification" phase after detecting an attack.

8 References

[1] MIKELANGELO Report D2.13 "The first sKVM hypervisor architecture", https://www.mikelangelo-project.eu/deliverables/deliverable-d2-13
[2] MIKELANGELO Report D3.1 - "The First Super KVM – Fast virtual I/O hypervisor", https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP3.1-IBM-v1.0.pdf
[3] MIKELANGELO Report D2.21 - "The final MIKELANGELO architecture", https://www.mikelangelo-project.eu/wp-content/uploads/2017/09/MIKELANGELO-WP2.21-USTUTT-v2.0.pdf
[4] MIKELANGELO deliverable D3.2, "The intermediate Super KVM - Fast virtual IO hypervisor", https://www.mikelangelo-project.eu/wp-content/uploads/2017/01/MIKELANGELO-WP3.2-IBM-v2.0.pdf
[5] Russell, R. "virtio: Towards a De-Facto Standard For Virtual I/O Devices." 2012. ftp://ftp.os3.nl/people/nsijm/INR/Week%201/papers/32_virtio_Russel.pdf
[6] J. A. Ronciak, J. Brandeburg, and G. Venkatesan, "Page-Flip Technology for use within the Linux Networking Stack," in Proceedings of the Linux Symposium, Vol 2, 2004, pp. 175-180. Available: https://www.landley.net/kdocs/ols/2004/ols2004v2-pages-175-180.pdf
[7] Macvtap - http://virt.kernelnewbies.org/MacVTap
[8] MIKELANGELO Report D3.4 "sKVM Security Concepts – First Version" https://www.mikelangelo-project.eu/wp-content/uploads/2016/06/MIKELANGELO-WP3.4-BGU-v1.0.pdf
[9] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho, "Papi: A portable interface to hardware performance counters", In Proceedings of the department of defense HPCMP users group conference, pages 7–10, 1999.
[10] https://stackoverflow.com/questions/1126529/what-is-the-cost-of-an-l1-cache-miss. The figure of 195 cycles per memory access is an average value with very high variability depending on previous accesses to the same memory row and on load within the system.
[11] Report D2.21, The final MIKELANGELO architecture, https://www.mikelangelo-project.eu/wp-content/uploads/2017/09/MIKELANGELO-WP2.21-USTUTT-v2.0.pdf
[12] Serverless Computing With OSv, http://blog.osv.io/blog/2017/06/12/serverless-computing-with-OSv/
[13] UNCLOT Github branch, https://github.com/mikelangelo-project/osv/tree/unclot-v1
[14] OSv applications repository, https://github.com/cloudius-systems/osv-apps/
[15] https://github.com/mikelangelo-project/SCAM