MIKELANGELO

D3.2
Intermediate Super KVM - Fast virtual I/O hypervisor

Workpackage	3	Hypervisor Implementation	
Author(s)	Joel Nider		IBM
	Ofer Biran		IBM
	Razya Ladelsky		IBM
	Shiqing Fan		Huawei
	Fang Chen		Huawei
	Gabriel Scalosub		BGU
	Niv Gilboa		BGU
Reviewer	Gregor Berginc		XLAB
Reviewer	Nadav Harel		SCYLLA
Dissemination Level	PU		

Date	Author	Comments	Version	Status
2016-12-01	Joel Nider	Initial draft	V0.0	Draft
2016-12-11	Niv / Gabriel	SCAM section ready	V0.1	Draft
2016-12-14	Fang/Shiqing	vRDMA section ready	v0.2	Draft
2016-12-15	Joel Nider	Ready for review	v1.0	Review
2016-12-22	Joel Nider	Ready for submission	v2.0	Final

© Copyright Beneficiaries of the MIKELANGELO Project

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). 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.

During the past year (second year of the MIKELANGELO project) the 3 features that enhance KVM have developed and matured to a point that allowed for performance testing and evaluation under lab conditions. This report explains the modifications made to bring the features to this point, and also shows the evaluation of features under various use cases. Instructions are provided to download and build the code required for reproducing the experiments.

In the coming year, the goal is to complete the implementations to maximize performance and functionality, and integrate these features to operate together to truly become super KVM.

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)

© Copyright Beneficiaries of the MIKELANGELO Project

Table of contents

1 Introduction 7
2 IOcm and Future Features 9
2.1 IOcm Architecture 10
2.2 Building And Configuration IOcm 10
2.2.1 Getting the IOcm source 10
2.2.2 Building and Installing IOcm kernel 11
2.2.3 Configuring and running the IOcm manager 11
2.3 IOcm Evaluation 12
2.4 IOcm - Conclusions 20
2.5 Future – Zero Copy RX (ZeCoRX) 21
3 vRDMA 22
3.1 Overview 22
3.2 Architecture 22
3.3 Usage Requirements 24
3.4 Building and Configuring vRDMA Backend Driver 24
3.5 Evaluation 26
3.6 Future 28
4 SCAM 29
4.1 Review of attack 30
4.2 SCAM Architecture 31
4.3 Monitoring and Mitigation Modules 32
4.3.1 Monitoring 32
4.3.1.1 Performance Monitoring Unit 33
4.3.1.2 Standalone Measurements 33
4.3.1.3 Correlated Measurements 35
4.3.1.4 Identification 36
4.3.1.5 Performance 37
4.3.2 Mitigation by Noisification 38
4.3.2.1 Overview 38

© Copyright Beneficiaries of the MIKELANGELO Project

4.3.2.2 Detailed Description 39
4.3.2.3 Evaluation 40
4.3.3 Future 43
5 Concluding Remarks 45
6 References and Applicable Documents 46
7 Appendix A: Kernel config file changes 48

© Copyright Beneficiaries of the MIKELANGELO Project

Table of Figures

Figure 1: The high-level architecture diagram of sKVM 7
Figure 2: IOcm architecture. IOcm changes the number of I/O cores according to the workload. 10
Figure 3: Test system setup 13
Figure 4: Performance comparison of netperf TCP stream between the baseline (unmodified), the best I/O core configuration, and IOcm 14
Figure 5: Performance of TCP stream with a varying delay in the hypervisor when transmitting packets 15
Figure 6: Running Apache Bench with static HTML page sizes of 64B-1MB 16
Figure 7: Dynamic benchmark running of a Netperf TCP stream w. 2KB messages and seven phases of various busy-loop configurations 17
Figure 8: MessageSight test setup 18
Figure 9: MessageSight experiment set against 1 big VM 19
Figure 10: MessageSight experiment set against 6 small VMs 20
Figure 11: Architecture of vRDMA prototype II. Left: updated architecture design with vhost moved to kernel space. Right: original architecture proposed last year. Control path (black arrows) is managed by the hypercalls. Data path (green arrows) is directly mapped and shared between guest and host. 23
Figure 12: Bandwidth tests of vRDMA prototype II on Linux hosts and guests. 27
Figure 13: Latency tests of vRDMA prototype II on Linux hosts and guests. 28
Figure 14: SCAM Architecture 31
Figure 15: Standalone web server cache accesses and misses. 34
Figure 16: Measurements of standalone attacker's cache accesses and misses. 35
Figure 17: Attacker cache activity while working concurrently with server. 36
Figure 18: Server connection time for different probe rates of the PMU 38

© Copyright Beneficiaries of the MIKELANGELO Project

Table of Tables

Table 1: Noisification evaluation results 42

© Copyright Beneficiaries of the MIKELANGELO Project

1 Introduction

This document describes our developments during the past year of the 3 features of super KVM (sKVM): I/O core manager (IOcm), lightweight RDMA para-virtualization (vRDMA), and side channel attack monitor and mitigation (SCAM).

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

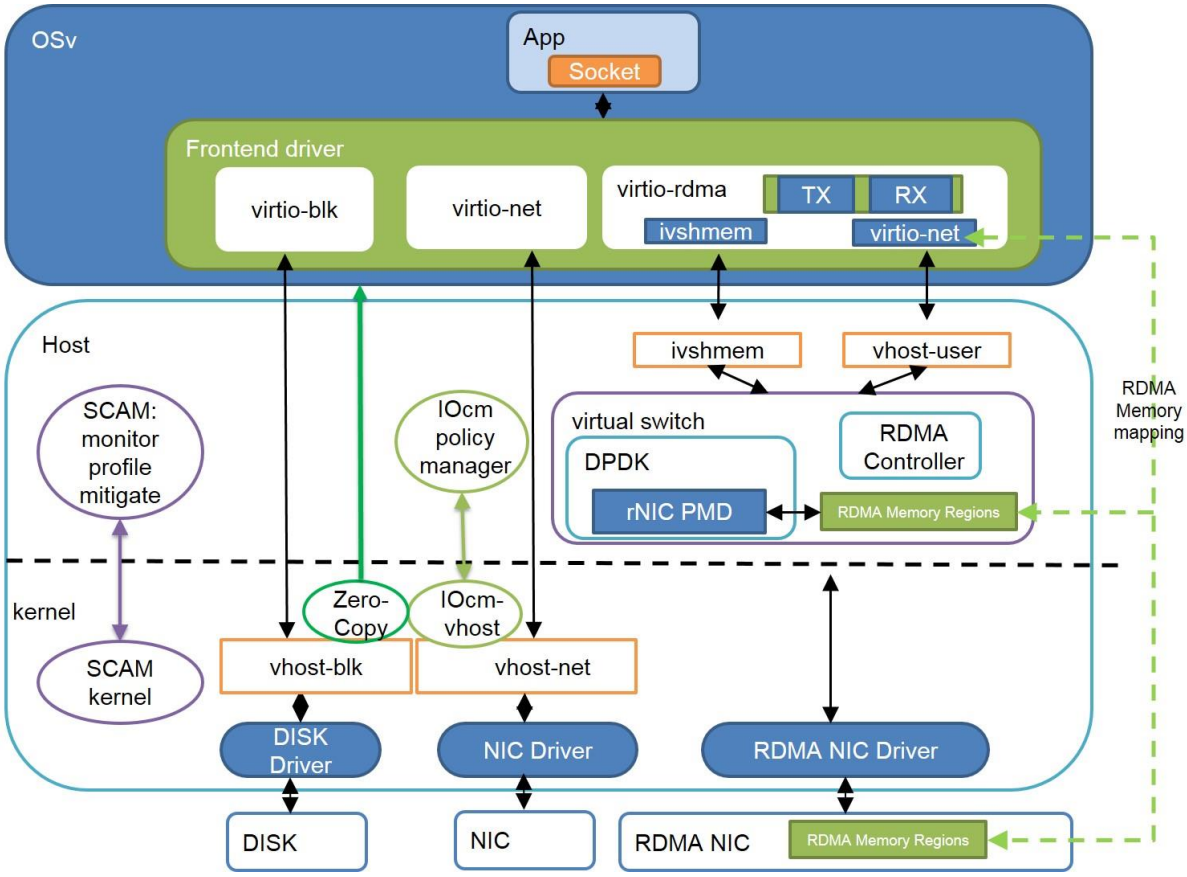

Figure 1: The high-level architecture diagram of sKVM

The KVM hypervisor depends on the virtio[4] 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 IOmanager, which monitors system performance and tunes vhost parameters

© Copyright Beneficiaries of the MIKELANGELO Project

accordingly. Section 2 provides details about the design, implementation and evaluation of IOcm.

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

The third feature of sKVM is not a performance feature, but a security feature. 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. The details of the attack, defence and evaluation of the defense are found in section 4.

© Copyright Beneficiaries of the MIKELANGELO Project

2 IOcm and Future Features

IOcm is a mechanism to control shared virtual I/O threads. It makes decisions about resource allocation to ensure maximum I/O performance for virtual machines. The architecture, design and implementation of IOcm is described in its entirety in deliverables D3.1[2] and D2.13[1], and updated by D2.20 - The intermediate MIKELANGELO architecture[3]. In a nutshell, the principle is to segregate a set of CPU cores whose purpose is dedicated to exclusively handling virtual I/O. Each such CPU core runs only a single thread - a shared virtual I/O thread - to avoid context switching on this core. We call these dedicated cores "I/O cores". The remaining CPU cores run remaining processes including VMs and other hypervisor threads (not related to virtual I/O).

This year we implemented the user-space IOcm policy manager, which is an application that samples the I/O activity statistics, calculates how pressured the VMs and I/O cores are, and selects the appropriate number of I/O cores for maximum performance.

In some cases, the baseline model works better than the side-core model, where cores are dedicated for I/O. The policy manager can decide to release all I/O cores and fall back to the baseline model, where each virtual device has a corresponding I/O thread. In the case where the side-core model is beneficial, the manager also needs to decide which I/O devices are handled by which I/O core, in order to balance the load across the I/O cores.

We have constructed an adaptive algorithm that the policy manager uses to make its decisions. The basic policy is based on measuring the CPU load on the two types of cores:

- if the I/O cores are loaded, and the VM vCPU cores are not, then add an I/O Core
- if the I/O cores are not loaded, and the VM vCPU cores are loaded, remove an I/O Core

The manager uses the IOcm API to constantly determine the best the configuration of vcpu cores vs. I/O cores, at any point of the workload run, in order to maximize throughput. It uses the statistics coming from the kernel to make a decision about the number of I/O cores that would be optimal at that moment, and uses the control mechanism to take the proper action. The statistics are checked by the IO manager every 0.01 seconds (100 times/second), but requires several iterations to build a history on which to base a decision.

© Copyright Beneficiaries of the MIKELANGELO Project

2.1 IOcm Architecture

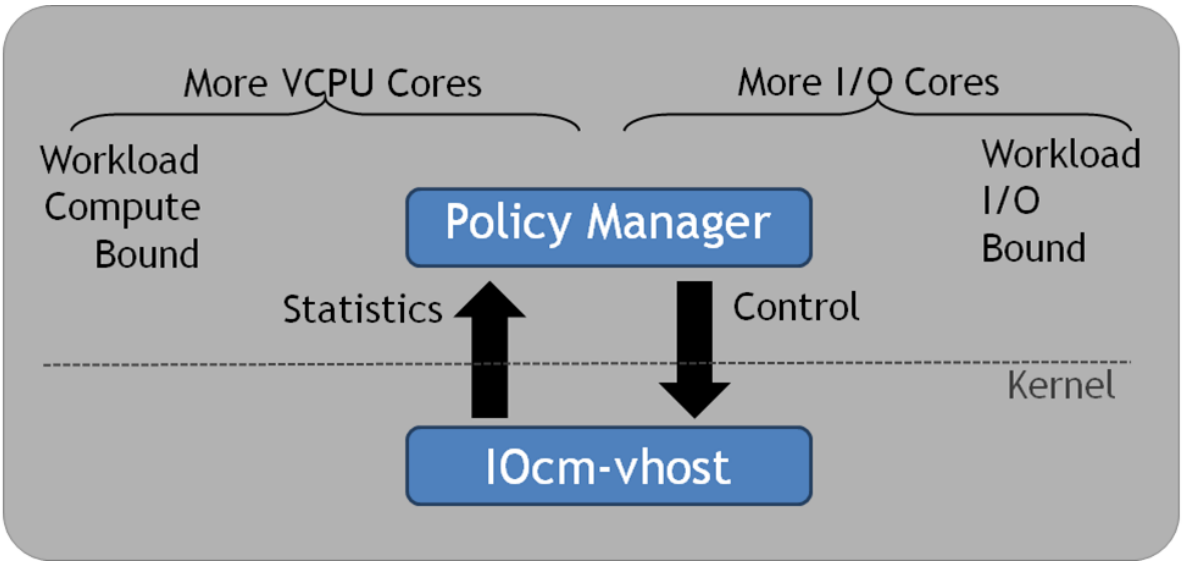

Figure 2: IOcm architecture. IOcm changes the number of I/O cores according to the workload.

Figure 2 depicts the two components of IOcm: IOcm Policy manager, and IOcm-vhost. The policy manager collects statistics regarding the I/O activity (collected and exposed by IOcm-vhost) and uses them to control the number of IO cores. The number of I/O cores needed to handle the I/O load depends on the (possibly dynamic) workload type (compute vs. I/O bound).IOcm API implementation was introduced in D3.1. Functionality was implemented in the kernel, as part of the vhost module, and exposed to user-space via the sysfs mechanism[6]. The API includes functions to control the I/O cores and devices, and retrieve statistics about the resource utilization.

2.2 Building And Configuration IOcm

2.2.1 Getting the IOcm Source

We have completed the development of the IOcm feature in sKVM, and the code is made available as part of the deliverable through a publicly available git repository.

To get a copy of the IOcm kernel code, clone this git repository:

https://github.com/mikelangelo-project/linux-3.18.y

To get the IOcm policy manager code, clone this git repository:

https://github.com/mikelangelo-project/dynamic-io-manager

© Copyright Beneficiaries of the MIKELANGELO Project

2.2.2 Building and Installing IOcm Kernel

IOcm kernel code is implemented as a set of patches on top of the official upstream Linux kernel version 3.18. The repository above reflects that, and contains the IOcm patches as well as the base Linux kernel in a single location. To compile a kernel that supports IOcm, please follow the normal method for building a Linux kernel, which is summarized here for convenience.

```
make defconfig
<add modifications to the .config according to appendix A (as needed)>
make
make modules_install
make install
```

Once the kernel containing IOcm support is built and installed, the host machine must be rebooted in order to use this new kernel. The new kernel only adds support required for IOcm functionality, and will not affect the operation of any existing interfaces or applications.

2.2.3 Configuring and running the IOcm Manager

The IOcm manager is a python application. It requires a configuration file which specifies parameters such as the number of CPU cores that are to be managed, and the network interfaces that can route traffic to virtual machines. When setting up the environment, each virtual machine should be connected to the physical network through macvtap[24], a Linux (virtual device) driver which connects the the VMs to a physical NIC without the need for a virtual bridge, thereby improving performance.

To create a configuration file, run:

```
src/create_configuration_file.py <num cores> <if_name[,2nd_if_name]> [OPTIONS]


Where:
num_cores:     The number of cores to be managed by IOcm (for both virtual machines and
I/O threads). This must be less-than or equal to the number of active cores in your
system.
if_name:       The physical network interface name used by the virtual machines.
2nd_if_name:   An additional physical network interface name (for balancing the load
between two physical interfaces)
By default, the configuration file (io_manager_configuration.json) is stored in /tmp.
OPTIONS
--config       override the default configuration file path
--min          Minimum number of IOcores allowed (default: 0)
--max          Maximum number of IOcores allowed (default: 4)
```


© Copyright Beneficiaries of the MIKELANGELO Project

To start the iomanager, use:

```
src/start_io_manager.py [OPTIONS]

OPTIONS
-c, --config  Override the default configuration file path
-p, --process Run as a regular process (not a daemon) and direct all output to
stdout/sterr
```

To stop the iomanager if the iomanager is running as a daemon (default), simply use:

```
src/stop_io_manager.py
```

Otherwise (if the -p option is used when starting), Ctrl+C is sufficient to stop the iomanager. If you wish to change the configuration of IOcm for some reason (i.e. the maximum number of IOcores), the service would need to be stopped, and then restarted with the new configuration file.

2.3 IOcm Evaluation

Test-bed

Our testbed system consists of two physical machines: a Host and a Load Generator, as shown in Figure 3. The two servers are IBM System x3550 M4 machines, each equipped with: two 8-core Intel 2.2GHz Xeon E5-2660 CPUs; 56GB memory; and two Intel x520 dual port 10Gbps NICs (network interface cards), allowing a total throughput of 40Gbps. The hypervisor is KVM with QEMU 2.4, hosting virtual machines configured with one virtual CPU and 1GB of memory each, backed by huge pages of 2MB to maximize performance (by minimizing paging and page management overhead).

All host machines run Linux 3.18. Hyperthreading and all power optimizations are turned off, namely sleep states (C-states) and DVFS (Dynamic Voltage and Frequency Scaling). This is done in order to obtain consistent results and to avoid reporting artifacts caused by non-deterministic events.

In all our experiments, unless stated otherwise, we use 12 virtual machines, each with a single paravirtual NIC. We divided the virtual machines into 4 groups, each of which shares one 10Gbps NIC port. Each virtual machine is connected to the physical network through the hypervisor, but without any software bridge to connect the virtual machines to each other. Instead, the virtual machines connect to the network through a macvtap virtual interface.

© Copyright Beneficiaries of the MIKELANGELO Project

Macvtap is a device driver in the Linux kernel that connects the VMs to a physical NIC without the need for a virtual bridge, improving performance.

The machines are connected in a back-to-back fashion as depicted in Figure 3.

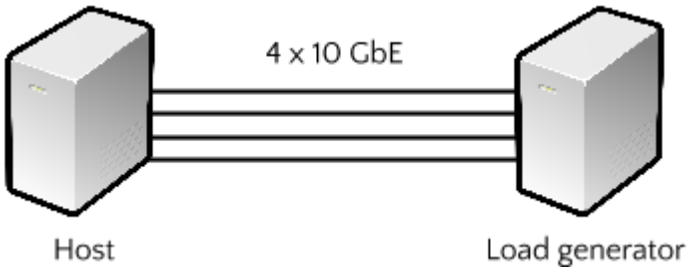

Figure 3: Test system setup

Methodology

We consider ELVIS as the state-of-the-art side-core approach. We evaluate IOcm against the I/O models we have considered thus far: the KVM/virtio baseline as the state-of-practice, and a group of ELVIS configurations we denote as *Best*. In each of our experiments we tested a wide range of ELVIS configurations as shown in this section. The *best* line in each graph represents the best performing ELVIS configuration for each test case (i.e. optimum number of IO cores). We perform each experiment 5 times and present averages.

Benchmarks

Previous work[5] concentrated on presenting and optimizing the side-core approach for under-committed systems with a single side-core per socket. We concentrate on over-committed systems with multiple side-cores per socket. The side-core approach divides the cores in the system into I/O cores and VM cores, so increasing the number of I/O cores causes the number of VM cores to decrease. As such, adding more I/O cores does not necessarily increase the overall throughput of the system, as the resources available to the virtual machines decrease. To improve system performance, the I/O manager must find the balance.

We use the following benchmarks:

1. Netperf TCP stream [7]
2. Netperf TCP stream with varying message cost
3. Apache HTTP web server [8]
4. Dynamic workload

© Copyright Beneficiaries of the MIKELANGELO Project

For a micro benchmark, we choose to test our system with a well-known and versatile network based benchmark, Netperf, which is used to test both throughput and latency of the network as a system. We use Netperf with different configurations and perform various tests to simulate a real-life workload.

Netperf TCP Stream is a unidirectional transfer of data over a TCP/IP connection using Berkeley sockets. This test is a common measurement for the maximum throughput achievable in the system. The "Best" line is created by running all possible ELVIS configurations (with 1 I/O core, 2 I/O cores, 3, etc) and selecting the configuration with the best performance for each message size. The number of I/O cores selected is written above each data point.

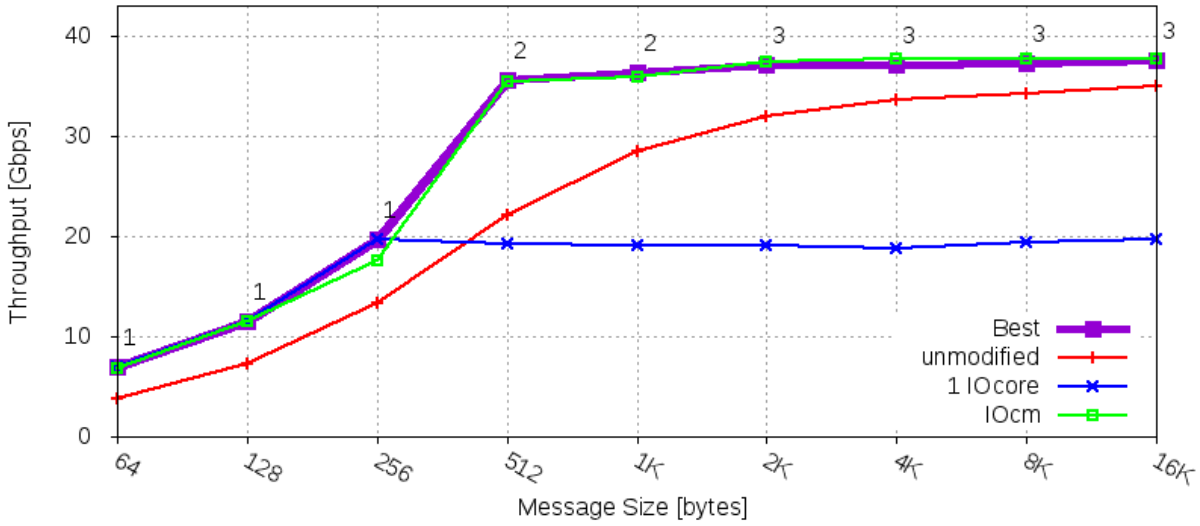

Figure 4: Performance comparison of netperf TCP stream between the baseline (unmodified), the best I/O core configuration, and IOcm

As we see in Figure 4, IOcm performed on par with the best static side-core configuration, trailing by at most 10% (2.1Gbps), with a message size of 256B. This gap is due to the fact that the I/O manager thresholds make it fluctuate between three and four side-cores. When four IO-cores are in use, the VM-cores are fully saturated and the IO-cores' utilization hovers on the threshold that triggers the I/O manager to reduce the number of IO-cores, which results in a loss of throughput. After the I/O manager removes a core, the regret policy identifies the degradation in throughput and reverts to the previous configuration. IOcm found the optimum configuration within about 0.47 seconds, or 47 iterations. About half of the iterations were spent on making sure that the change in configuration is beneficial - the first part of the regret policy.

TCP Stream with varying message cost uses the Netperf TCP/IP stream test, but with a modification inside vhost which performs a variable time, busy-loop delay before sending

© Copyright Beneficiaries of the MIKELANGELO Project

data. The busy-loop delay simulates a CPU-intensive workload that could potentially be done by the hypervisor during interposition, such as compression, encryption, or network traffic analysis (such as virus or intrusion detection). We send 2KB messages with busy-loop delays of 0-35000 cycles before each transmission.

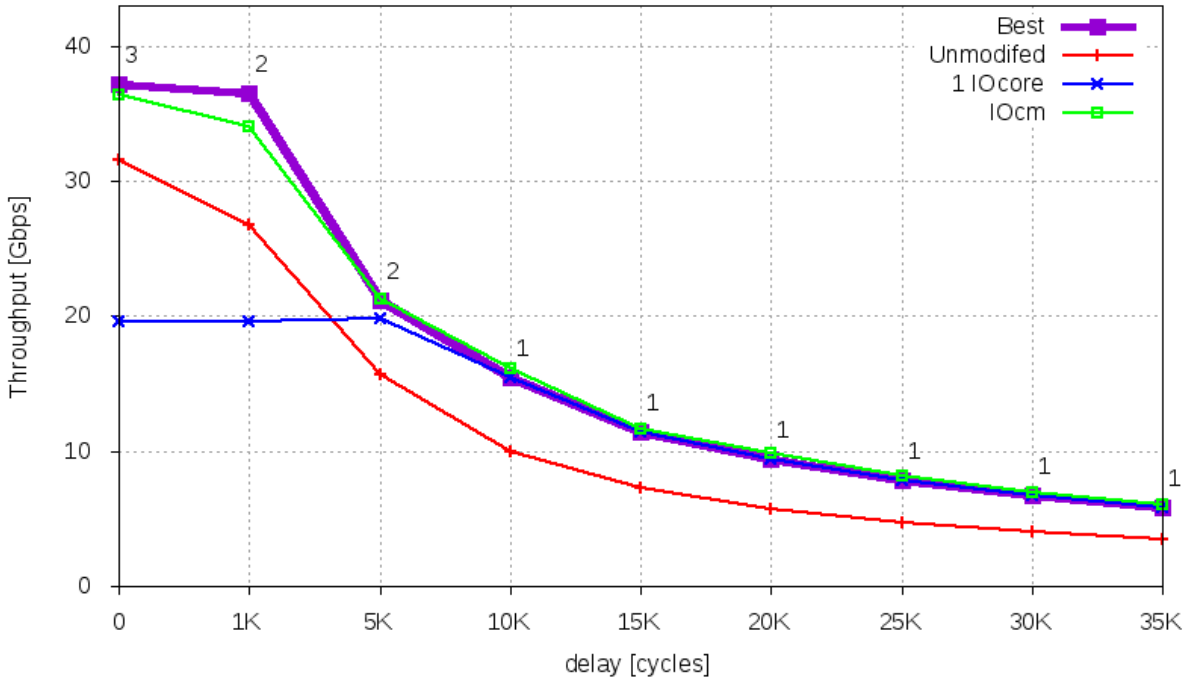

Figure 5: Performance of TCP stream with a varying delay in the hypervisor when transmitting packets

By the results shown in Figure 5 we can see that IOcm performs as well as the best static configuration, with a maximum of 1% deviation. Moreover, we see that IOcm (green) outperforms the baseline (red) in every case by between 15% to 72% (average of 54%).

Apache HTTP web server. We configure the Apache server to send static HTML pages with various sizes, from 64 bytes to 1MB. As shown in Figure 6, IOcm approaches the optimum defined by the best static side-core configuration for all HTML page sizes, with a deviation of at most 1%. When compared to the baseline paravirtual configuration, IOcm surpasses the performance for all HTML page sizes.

© Copyright Beneficiaries of the MIKELANGELO Project

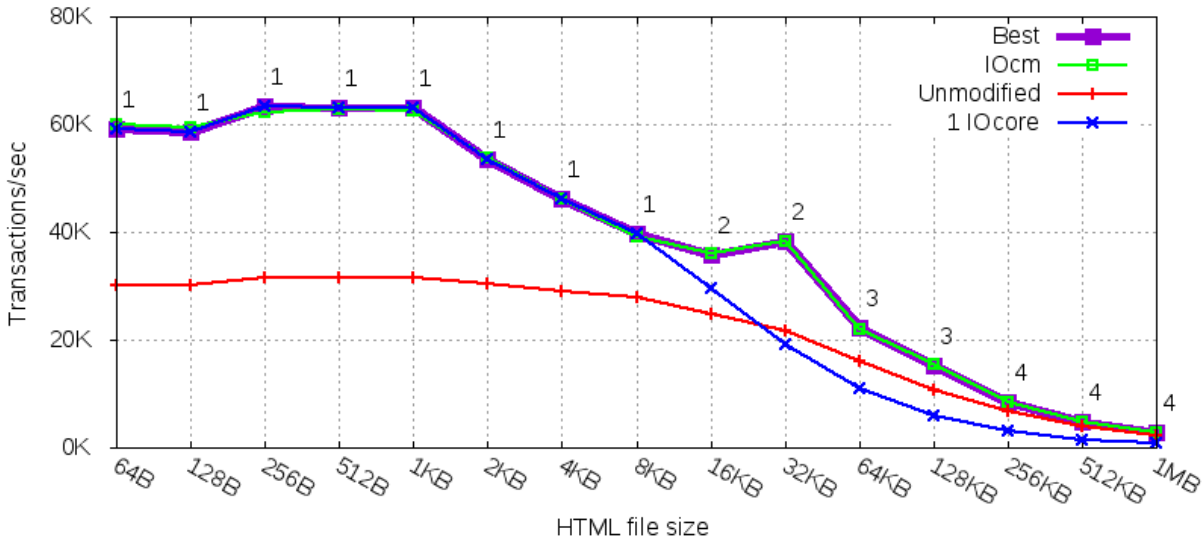

Figure 6: Running Apache Bench with static HTML page sizes of 64B-1MB

Figure 6 shows the benefit of side-cores in Apache bench (macro benchmark). The benefit of using the side-core approach declines for HTML page sizes between 2KB-16KB, then peaks for 32KB page sizes. For larger page sizes, the benefit continues to decay, although remains better than the baseline.

Dynamic workload. We demonstrate the real strength of having a dynamic environment that is able to adapt to change. We test the system with a workload that varies over time. This is achieved by running a Netperf TCP stream with a busy-loop test that sends 2KB messages, while using changing amounts of busy-loop cycles. The number of busy-loop cycles ranges from 0 to 35000, and the number of cycles used for each phase of the test is marked above the phase in Figure 7 as the first number in the pair. The test comprises seven phases, each running for 30 seconds. The throughput is sampled every two seconds.

© Copyright Beneficiaries of the MIKELANGELO Project

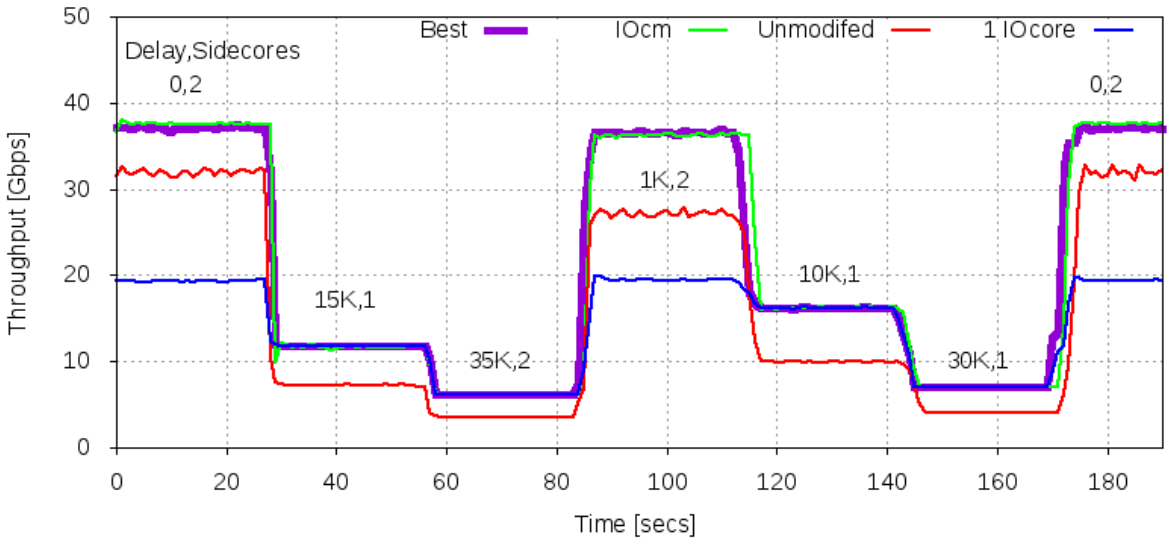

Figure 7: Dynamic benchmark running of a Netperf TCP stream w. 2KB messages and seven phases of various busy-loop configurations

The results for the dynamic workload experiment are shown in Figure 7, comparing the optimum performance achievable in each stage (the static best) vs IOcm.

IBM MessageSight Application

IBM MessageSight [20] is the bridge between incoming data from IoT devices and the consumers within the enterprise up to the HPC/analytics applications that consume the data. The MessageSight bridging is planned to be provided in several deployment models, including as-a-Service, where a tenant receives a dedicated, hosted MessageSight cluster running on VMs over the KVM hypervisor. Due to the expected high throughput of IoT messages, the incoming I/O performance is the major constraint on the number of MessageSight VMs per physical server. Thus, the performance improvement of the incoming I/O in sKVM will enable more MessageSight VMs per physical server, and as a result, improve the efficiency of the MessageSight service per a given number of physical servers.

Test Setup The experiments were done with one (or in some cases two) servers running benchmarking tools, which were connected back-to-back (i.e. no network switch) over a 10GbE link to a third server which hosted the MessageSight VMs (Figure 8). The experiments were conducted with a MqttBench, a proprietary IBM tool used for benchmarking MessageSight by simulating a large number of IoT devices sending messages over the mqtt protocol.

© Copyright Beneficiaries of the MIKELANGELO Project

Figure 8: MessageSight test setup

The MessageSight host was an IBM System x3650 M4, with 2 sockets, 8 cores Intel(R) Xeon(R) CPU E5-2680 2.70GHz, HyperThreading enabled, 128G memory, Intel 82599ES 10-Gigabit Ethernet network card. The host ran Linux with a 3.18 kernel, sKVM (with IOcm support and a vhost backend) and QEMU version 2.0.0 (Debian 2.0.0+dfsg-2ubuntu1.21).. We ran a series of experiments on MessageSight running in VMs over sKVM to simulate the as-a-Service deployment model with the IOcm technology, and compared it to the standard KVM (*vanilla*). These experiments show a significant improvement in the maximum total message throughput obtained by a cluster of MessageSight VMs running on a single physical server.

Evaluation The experiments were conducted to determine maximum message rates for different numbers of connections in various cluster sizes (numbers of MessageSight VMs). The results showed an increased message rate enabled by sKVM with the IOcm technology in the range of 20-50%. The high throughput is gained through both the number of connections (up to 100K) and increased message rate per VM.

Following are the results and diagrams of experiment set against one MessageSight VM consisting of 8 vCPUs, 32GB of memory, and a multiqueue virtio NIC with 8 queues. These experiments simulate 100K (separate) TCP connections, sending at a requested message rate overall (across these 100K connections).

© Copyright Beneficiaries of the MIKELANGELO Project

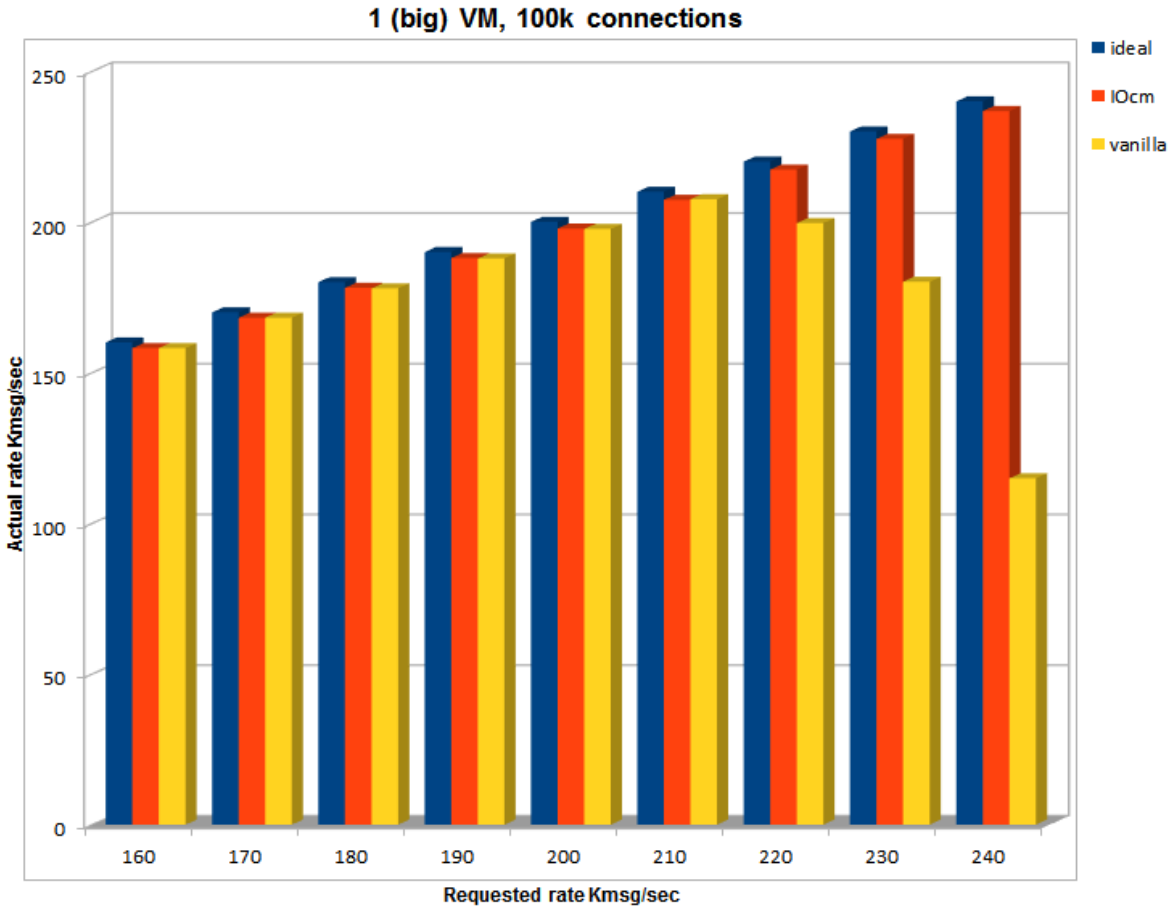

Figure 9: MessageSight experiment set against 1 big VM

In Figure 9 and Figure 10, the X axis shows the requested message rate in each experiment, and the Y axis represents the actual message rate attained by MessageSight processing. The blue bars represents the ideal state where the full requested rate is achieved. It can be seen that starting at 220K msg/sec the MessageSight running on the vanilla KVM is starting to get into trouble, while the IOcm enables a reasonable operation.

Following is the result diagram of experiment set against 6 MessageSight VMs, each consisting of 2 vcpus, 32GB of memory, and a virtio NIC (with 1 queue). This experiment simulated 100K IoT devices, each with its own persistent TCP connection (100K TCP connections).

© Copyright Beneficiaries of the MIKELANGELO Project

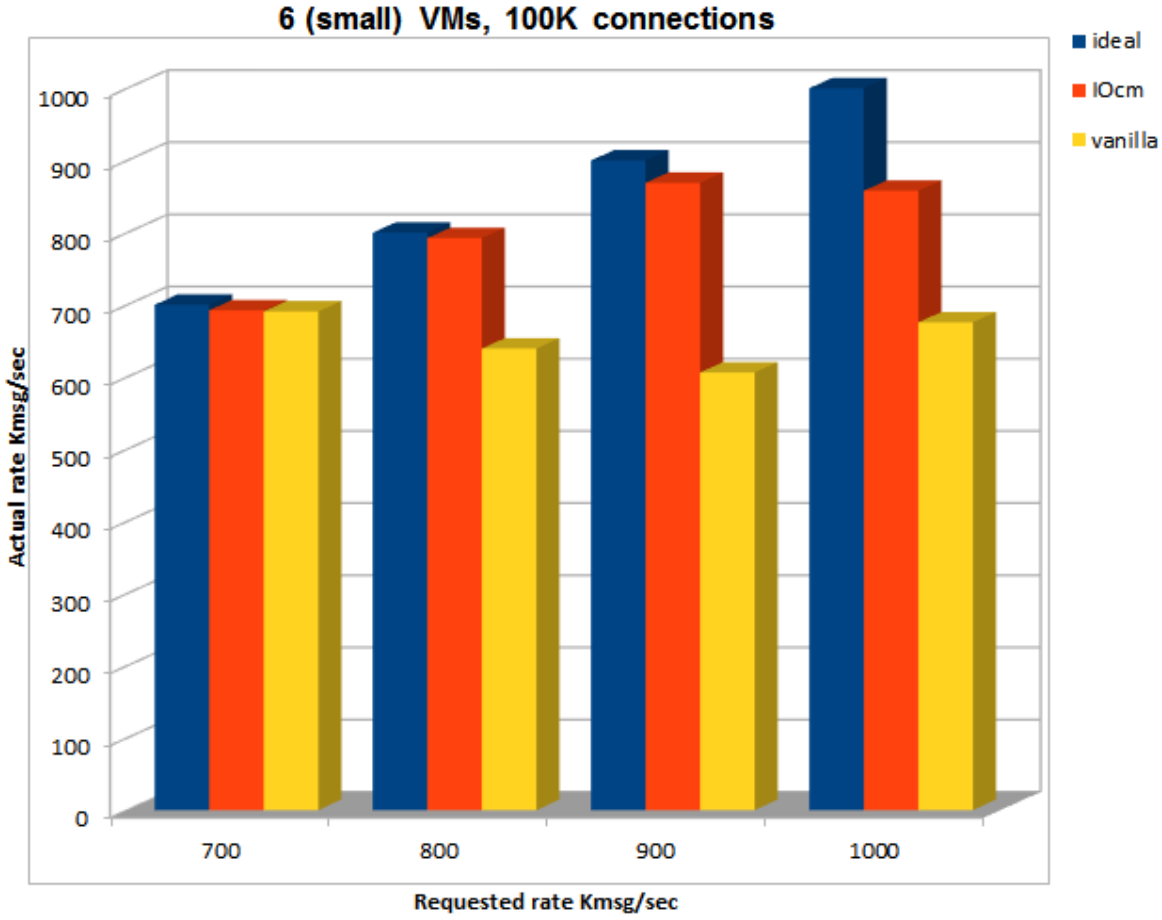

Figure 10: MessageSight experiment set against 6 small VMs

Here the message load is spread across 6 VMs and it is possible to get higher total message rates (the reason is due to an application scaling limitation of a single MessageSight instance), and it can be seen that in the case of a message rate of total 900K msg/sec, IOcm causes an increased actual throughput of up to 40%.

2.4 IOcm - Conclusions

In year 2 we analyzed the results of the IOcm development and evaluation, focusing on the IO cores policy management. We demonstrated the IOcm advantage both for standard benchmarks as Netperf and Apache HTTP server, and for the real life application IBM MessageSight. For IBM MessageSight, the results show, as expected, the importance of IOcm for heavy I/O applications with throughput intensive streams of small packets. We conclude that despite some impressive results with small packet sizes, we must continue to search other directions to improve I/O intensive applications that use larger packet sizes.

© Copyright Beneficiaries of the MIKELANGELO Project

Therefore during the latter part of year 2 we started to work on the second vHost technology, ZeCoRX, that also aims to further improve the sKVM I/O, as we describe on the next section.

2.5 Future – Zero Copy RX (ZeCoRX)

We are now in the final design phase of the ZeCoRX technology, which aims to eliminate copy operations that are performed in the processing of incoming packets that involves the NIC, the hypervisor and the guest VM. The hypervisor vhost_net module copies the incoming packet content into a shared ring buffer, from where it is copied by the guest to its own buffer. With ZeCoRX, the physical NIC will be configured to place incoming packets directly on the buffers allocated by the guest, thus eliminating the copying operation normally needed from the buffer allocated by the kernel driver to user space. This raises challenges of synchronization between the guest buffer allocation system and the vhost module. The intention is to support the macvtap interface only (a network interface that can be attached to a software device) which will make it easier for upstream contribution, since this approach requires less intervention with the existing host networking stack, provides flexibility for enabling/disabling ZeCoRX per virtual NIC, and minimizes overhead incurred by the control plane.

Our design is being made with the cooperation of key figures in the Linux kernel community, to help smooth the path to contributing the changes back upstream. Based on lessons learned in our past research, we have taken a policy to work with the kernel community from day-1, in order to gain agreement about our direction, and make any necessary modifications to the design before getting too deep into the project.

© Copyright Beneficiaries of the MIKELANGELO Project

3 vRDMA

3.1 Overview

In deliverable 2.13 [1], we proposed three designs for enabling virtualized RDMA (vRDMA). Virtualized RDMA is allowing guests to use RDMA hardware in the host to perform more efficient communication between different guests - whether the two communicating guests are on the same host or on different hosts. Prototype I focused on developing vRDMA solutions that support socket based API. This year, we concentrate on the development of prototype II, which aims to support guest applications that directly use RDMA verbs. Future work of prototype III will focus on combining the two prototypes to support both socket and RDMA API with automatic selection of communication protocols, i.e. RDMA for inter-host communication or shared memory for intra-host communication, while the guest application is unaware of which protocol is being used.

In vRDMA prototype II, we reuse an existing backend driver on the host, and focus on the development of frontend drivers for OSv that support RDMA over virtualized devices. Therefore, the major contribution and efforts were made on the guest side, which is presented in deliverable D4.5 [19].

In the following sections, we present the architectural design of prototype II backend driver, the hardware and software requirements that need to be met, how to build and use it with detailed example instructions, and finally a brief evaluation of the updated backend driver.

3.2 Architecture

The backend driver of prototype II manages the I/O requests on the host side which directly accesses hardware devices. The proposed backend driver in prototype II modifies and extends the existing HyV [22] vhost implementation to the target Linux version 3.18. Hyv implements a frontend driver and a kernel vhost backend driver for Linux 3.13. Using the backend driver of Hyv directly for vRDMA saves the effort of re-implementing the backend driver from scratch, and only the frontend for OSv needs to be developed.

The difference between prototype II and the HyV implementation on the guest side is that we eliminate unnecessary context switches between user and kernel space, porting only the required OFED kernel definitions to support the kernel verb structure in the user space library. The OFED resources, like context handles and user command data, will never be necessarily copied between guest and kernel spaces. Many *memcpy* operations are avoided and making the frontend driver more efficient.

© Copyright Beneficiaries of the MIKELANGELO Project

On the host side, the major part of prototype II design stays unchanged comparing to the initial plan proposed in D2.13. Figure 11 shows the current architecture design (left) compared to the one proposed at the beginning of the project (right). The current design architecture moves the vhost driver from user space to kernel space, in order to reuse the existing backend driver of HyV.

The original HyV driver uses a communication scenario, called hypercall, between guest and host. The control path for the virtio-rdma driver are managed by the hypercall implementation, and the verb commands are prepared in a specific and formatted message buffer which can be parsed by the frontend and backend driver. Meanwhile, the data path is using the directly shared RDMA memory regions, which is the same as the prototype II architecture. We follow the same semantic that HyV uses, i.e. hypercall communication between guest and host for the control path communication, in order to reuse HyV's backend driver on the host. For each RDMA verb call, e.g. ibv_query_device [23], the virtio-rdma driver will prepare the parameters into a communication buffer and send it to the host driver to execute the requested verb call. The host driver then knows the correct kernel verb call to run with sufficient parameters. However, as OSv doesn't have any RDMA support, we still need to implement the frontend driver, which has been done and explained in D4.5 [19].

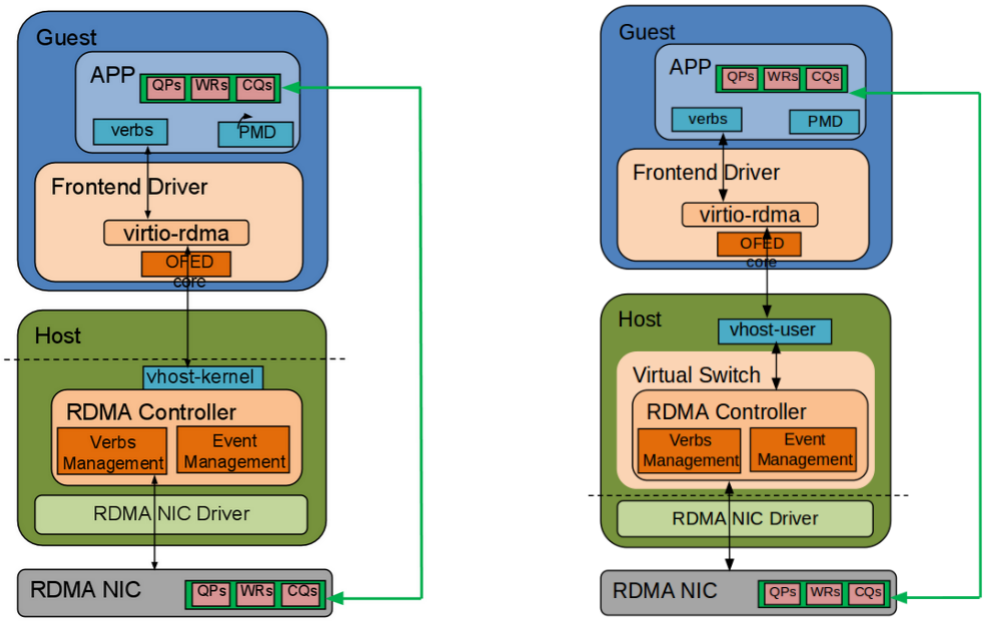

Figure 11: Architecture of vRDMA prototype II. Left: updated architecture design with vhost moved to kernel space. Right: original architecture proposed last year. Control path (black arrows) is managed by the hypercalls. Data path (green arrows) is directly mapped and shared between guest and host.

A QEMU patch is also needed for providing the virtual device support for the guest, which has been provided within the Git repository of HyV [22]. The original patch from HyV was made for an ancient git version of QEMU. We have updated this patch to support QEMU 2.4.0

© Copyright Beneficiaries of the MIKELANGELO Project

accordingly. As HyV supports only Linux 3.13, and the target kernel for MIKELANGELO is 3.18, it then needs to be updated. Changes have to be made to the QEMU patch for handling the memory mapping between the guest and the newer host kernel and managing the virtual device.

3.3 Usage Requirements

To use vRDMA prototype II, the following system configuration and hardware requirements must be met:

- Virtualization is enabled in BIOS. For Intel CPUs, VT-x must be enabled, and for AMD CPUs, AMD-v must be enabled,
- ConnectX-3 adapter cards with Virtual Protocol Interconnect (VPI) is used.
- QEMU 2.4.0 and the patch made for Hyv
- Host side needs HyV backend driver
- On the host side, install OFED user libraries: libibverbs, libmlx4, librdmacm, opensm. libibverbs provides the user linkage to the verbs API. libmlx4 is the user space service provider. Librdmacm is the RDMA connection manager. opensm is the system services for subnet management.

Once the above requirements are fulfilled, you are ready to set up the host side for using vRDMA following instructions given in Section 3.4.

3.4 Building and Configuring vRDMA Backend Driver

On the host side, we utilize the host drivers from HyV design. To install and setup the host side for vRDMA, follow the steps below.

Make sure you are using linux version 3.18. Install kernel source (the following commands are based on Ubuntu distribution. For other distributions, please replace the *sudo apt-get* command with the correct system commands):

```
$ sudo apt-get install linux-headers-3.18.0
$ sudo apt-get update
$ sudo apt-get install linux-source
$ sudo apt-get source linux-image-3.18.0
$ cd /usr/src/linux-source-3.18.0
$ sudo tar -xf linux-source-3.18.0.tar.bz2
```

Obtain and build HyV source code into your working directory:

```
$ git clone https://github.com/zrlio/hyv
$ export KSRC=/usr/src/linux-source-3.18.0/linux-source-3.18.0
```


© Copyright Beneficiaries of the MIKELANGELO Project

Next, install the following packages:

```
$ sudo apt-get install libibverbs1 libibverbs-dev librdmacm1 librdmacm-dev infiniband-
diags ibverbs-utils libmls4-1 libmthca1 libipathverbs1
```

Now download, patch and compile qemu

```
$ wget http://wiki.qemu-project.org/download/qemu-2.4.0.tar.bz2
$ tar xvf qemu-2.4.0.tar.bz2
$ cd qemu-2.4.0
$ path patch -p1 < ../hyv/host/qemu_diff.patch
$ cp ../hyv/include/virtio_hyv_config.h ../hyv/include/virtio_rdmacm_config.h
./hw/virtio/
$ sudo apt-get install zlib1g zlib1g-dev libglib2.0-dev zlib1g-dev gcc automake
$ git submodule update -- init pixman dtc
$ ./configure
$ make
$ sudo make install
```

To prepare the host, replace the following kernel modules by the ones from HyV

```
$ sudo rmmod rdma_ucm
$ sudo rmmod ib_ucm
$ sudo rmmod ib_ipoib
$ sudo rmmod vhost_rdmacm
$ sudo rmmod rdma_cm
$ sudo rmmod ib_cm
$ sudo rmmod mlx4_ib
$ sudo rmmod ib_sa
$ sudo rmmod iw_cm
$ sudo rmmod ib_umad
$ sudo rmmod ib_mad
$ sudo rmmod ib_uverbs
$ sudo rmmod ib_ipoib
$ sudo rmmod vhost_rdmacm
$ sudo rmmod rdma_cm
$ sudo rmmod mlx4_ib
$ sudo rmmod mlx5_ib
$ sudo rmmod vhost_hyv
$ sudo rmmod ib_core
$ sudo modprobe ib_core
$ sudo modprobe vhost
$ sudo insmod /path/to/hyv/host/vhost_hyv/vhost_hyv.ko
$ sudo modprobe mlx4_ib
$ sudo modprobe rdma_cm
$ sudo insmod /path/to/hyv/host/vhost_rdmacm/vhost_rdmacm.ko
$ sudo modprobe ib_uverbs
$ sudo modprobe ib_umad
```


© Copyright Beneficiaries of the MIKELANGELO Project

opensm provides an implementation of an InfiniBand Subnet Manager and Administration [21]. To start opensm on one of the hosts, use command:

```
$ /etc/init.d/opensm start
```

Before starting a virtual machine, obtain the node GUID of the InfiniBand card with command *ibv_devinfo*, which comes with *libibverbs*. Next, copy this GUID into the config file */path/to/ibnode_guid*. Add "allow br0" in the */usr/local/etc/qemu/bridge.conf* file. Please note that *em1* is the ethernet port name. This name can differ based on your system and network card.

```
$ ibv_devinfo
$ sudo brctl addbr br0
$ sudo brctl addif br0 em1
$ sudo ifconfig em1 0
$ sudo dhclient br0
```

Use the following commands to start virtual machines. The user can choose any unused mac addresses and give them to your virtual machines.

```
$ sudo qemu-system-x86_64 -enable-kvm -boot d -drive  file=~/workspace/vm-images/Linux-
3.18-Ubuntu-14.04.qcow -m 2048M -cpu Nehalem -smp 2 -name "VM2" -device virtio-hyv-
pci,config_path="/path/to/ibnode_guid" -device virtio-rdmacm-pci -netdev bridge,id=hn0 -
device virtio-net-pci,netdev=hn0,id=br0,mac=52:54:00:12:34:56 &

$ sudo qemu-system-x86_64 -enable-kvm -boot d -drive file=~/workspace/vm-images/Linux-
3.18-Ubuntu-14.04.qcow -m 2048M -cpu Nehalem -smp 2 -name "VM2" -device virtio-hyv-
pci,config_path="/path/to/ibnode_guid" -device virtio-rdmacm-pci -netdev bridge,id=hn0 -
device virtio-net-pci,netdev=hn0,id=br0,mac=52:54:00:12:34:57 &
```

Finally we have set up the host side. For guest drivers installation, please refer to deliverable D4.5.

3.5 Evaluation

We present the I/O performance evaluation of vRDMA prototype II. We evaluate bandwidth and latency tests on a local testbed. The testbed is using two HP ProDesk 600 with a 4-core Intel i7-4790 processor and 16GB of RAM. HyV uses Linux kernel version 3.13. Our project aims to support Linux version 3.18. Hence, we test vRDMA prototype II on a host operating system with Linux Kernel 3.18. The Linux guest OS is configured with two vCPUs and 2GB of memory.

© Copyright Beneficiaries of the MIKELANGELO Project

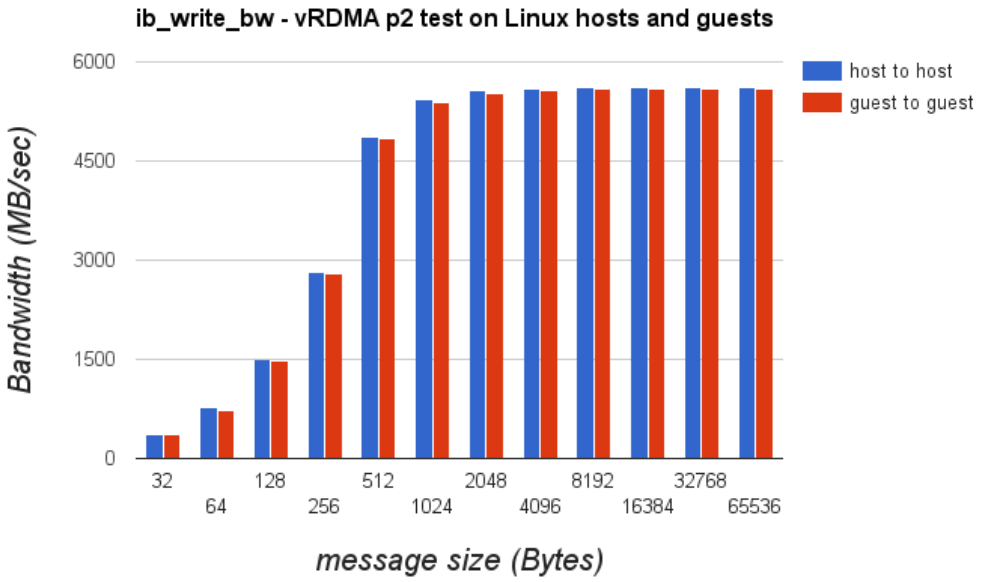

Figure 12: Bandwidth tests of vRDMA prototype II on Linux hosts and guests.

Figure 12 illustrates the measured bandwidth when using *ib_write_bw* command. Blue bars indicate ib_write_bw bandwidth from host to host, while red bars indicate the bandwidth from guest to guest. Bandwidth is measured in MB/s for a message size varying from 32 to 65536 bytes. We consider a system without VM installed a bare metal. Therefore, the bandwidth of host to host is considered to be bare metal performance, which are demonstrated by the blue bars. Overall, it can be seen from the bandwidth tests that prototype II has a performance close to bare metal, which has only 1~3% difference. The throughput increases as the message size increases. For a message size larger than 2048 bytes, the throughput reaches its limit of approximately 5.6 GB/s, which is the physical upper limit of the ConnectX-3 card.

Figure 13 illustrates the measured latency when using *ib_write_lat* command. Blue bars indicate ib_write_lat latency from host to host, while red bars indicate the latency from guest to guest. Latencies are measured in microseconds (μs) for a message size varying from 32 to 65536 bytes.

Both ib_write_bw and ib_write_lat are using the user space verbs API provided by libibverbs. When running them in OSv guest, the verb calls are processed by the virtio-rdma driver, and only the control path is managed by the hypercall implementation, i.e. control commands like create CQs and QPs and releasing the unused resources. The data path is not transferred through the hypercall or any other virtio drivers, as the RDMA memory regions are directly shared between the InfiniBand hardware to the host and also to the guest by virtual to

© Copyright Beneficiaries of the MIKELANGELO Project

physical address translation in the virtio-rdma driver. As a result, prototype II has bandwidth and latency performance close to bare metal.

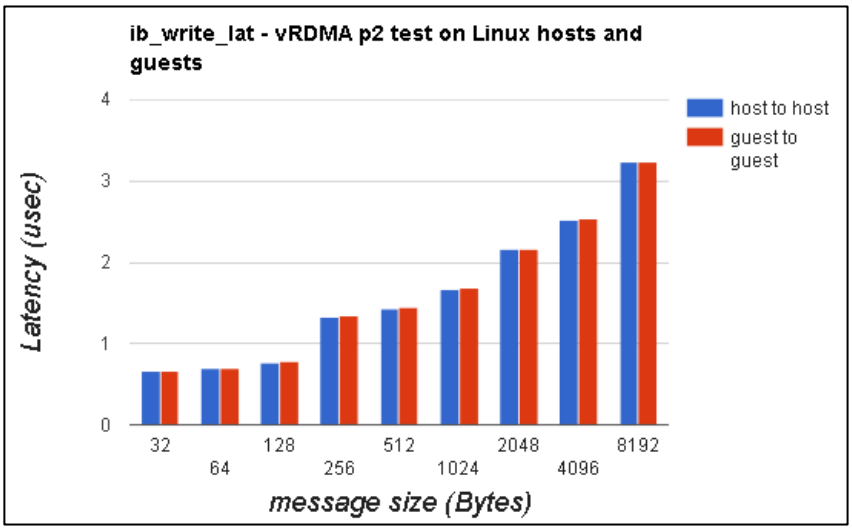

Figure 13: Latency tests of vRDMA prototype II on Linux hosts and guests.

3.6 Future

Next year we will move onto the development of vRDMA prototype III. Prototype III will combine the previous two prototypes together, enabling vRDMA through both socket-API and infiniband verbs with the mode automatically selected.

In prototype III, we will provide a conversion of the socket calls from guest application into RDMA verbs, and then pass these verbs to the backend driver. Prototype III will allow frontend driver to actively poll the completion event, avoiding the event sending between guest and host. Consequently, TX/RX ring buffers are directly translated to Queue Pairs (QPs) in the frontend driver, and Work Requests (WRs) and Completion Queues (CQs) are shared between the frontend driver and the RDMA device.

© Copyright Beneficiaries of the MIKELANGELO Project

4 SCAM

In this section we provide a detailed update on the status and plans of SCAM (Side-Channel Attacks Monitoring and Mitigation); the security module being developed and implemented as part of the MIKELANGELO stack, targeted at thwarting cache side-channel attacks (which will be briefly explained below, and described in detail in D3.4 [16]). SCAM is designed as a software module, and does not require any specific hardware, nor does it assume any intrinsic hardware capabilities. SCAM is designed to be part of the hypervisor level, specifically KVM, and is an inherent part of the sKVM implementation being developed within the MIKELANGELO project.

We note that a detailed account of SCAM, as well as the attack surface it is targeting to secure, was provided in deliverable D2.13[1] and deliverable D3.4[16]. The former described the high-level architecture of sKVM in general, and of SCAM in particular. The latter gives a detailed account of the implementation of an attacker running in a VM, which is able to extracts a private RSA key within a matter of minutes from a co-located VM running on the same host. We recall that since RSA keys are typically valid for months or years and are used to identify important networking components the implemented attack is attractive to a wide range of malicious actors, such an attack poses a significant security threat in virtualized environments, most predominantly in public clouds.

In the following sections we provide a detailed update on the work done in developing the various SCAM components. In section 4.1 we provide a quick review of the attack method, which will be useful both as a refresher on the attack itself, as well as highlights several of the mechanisms employed by the attacker, which are also used (with some variation) by our monitoring and mitigation modules. In section 4.2 we provide an update on the overall SCAM architecture, detailing the parts and components implemented thus far, as well as a high-level view of the future plans that remain to be developed. In section 4.3, we provide the bulk details of the modules developed within SCAM. These modules include a monitoring (and initial profiling) module, as well as a mitigation module that performs very fine-grained and carefully-timed cache accesses so as to introduce noise and thwart the attacker's attempts at recovering the target's private RSA key. For both of these modules we provide a detailed performance evaluation study highlighting their various properties, and also discuss several tradeoffs that must be taken into account when deploying and using our proposed solutions. Finally, in section 4.4 we describe our future plans for the remainder of this project, which focus on extending our monitoring and mitigation techniques, integration of our solutions into the MIKELANGELO stack, and further performance evaluation.

© Copyright Beneficiaries of the MIKELANGELO Project

4.1 Review of Attack

In this section we review a cache side-channel attack, and focus our attention on a prime-and-probe based attack. For a complete description of the attack we refer the reader to deliverable D3.4 [16].

The goal of the attack is to extract a private RSA key used by a server co-located on the same host as the attacker. Since the attacker and the server each run on their own VM, any leakage of information from the target useful to the attacker is done via shared hardware resources, predominantly, the last-level cache (LLC).

The goal of the attacker is to follow closely the decryption process performed by the target using its private key, and differentiate between its behavior when a specific bit in the key is 0, and when this bit is 1. The standard implementation of the decryption process contains a conditional, which in pseudo-code looks as follows:

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

In such an implementation the multiply operation uses the same instructions, but is executed an additional time for any 1-bit in the private key. Assume an attacker invalidates all the lines storing the target's data in a cache set storing (some of) the instructions required for performing the multiply operation before each such operation commences. In this case, each such operation causes a cache-miss causing the server to store its own cache lines instead of some of the attacker's lines.

By measuring its own hit-miss ratio for each such invalidation of a cache set, the attacker can discern between the case where a multiply operation took place or not. It can therefore identify whether or not the condition evaluated to a true value (i.e., the current bit of the private key is 1), or not. Using sufficiently many samples, and further optimization (addressing issues of synchronization, pattern recognition, alignment, voting etc.), the attacker can hence recover the key.

To better understand our monitoring and mitigation approaches, it is important to consider with some care the exact operations performed by an attacker in such scenarios. The above attack technique, commonly referred to as prime-and-probe, consists of the following two

© Copyright Beneficiaries of the MIKELANGELO Project

basic operations performed consecutively by the attacker for each cache set potentially used by the target:

- Prime:
  accessing **all** lines of a cache set so as to invalidate all lines that were previously potentially used by the target process/VM.

- Probe:
  after allowing sufficient time for the target process/VM to access any of the lines in the cache set, repeat the access to some of the lines and test for cache misses.

If the probe operation indicates cache misses, the attacker infers that the cache set in question has been accessed by another process, presumably the target. If, on the other hand, the probe operation indicates no cache misses, the attacker infers that the cache set in question has not been used by any process.

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

4.2 SCAM Architecture

For completeness, we provide here the high-level architecture of SCAM, as described and discussed in deliverable D2.13. We note that in the current interim implementation not all functionalities are implemented as described in the figure (e.g. we currently have no kernel module, and implementation is all in user-space).

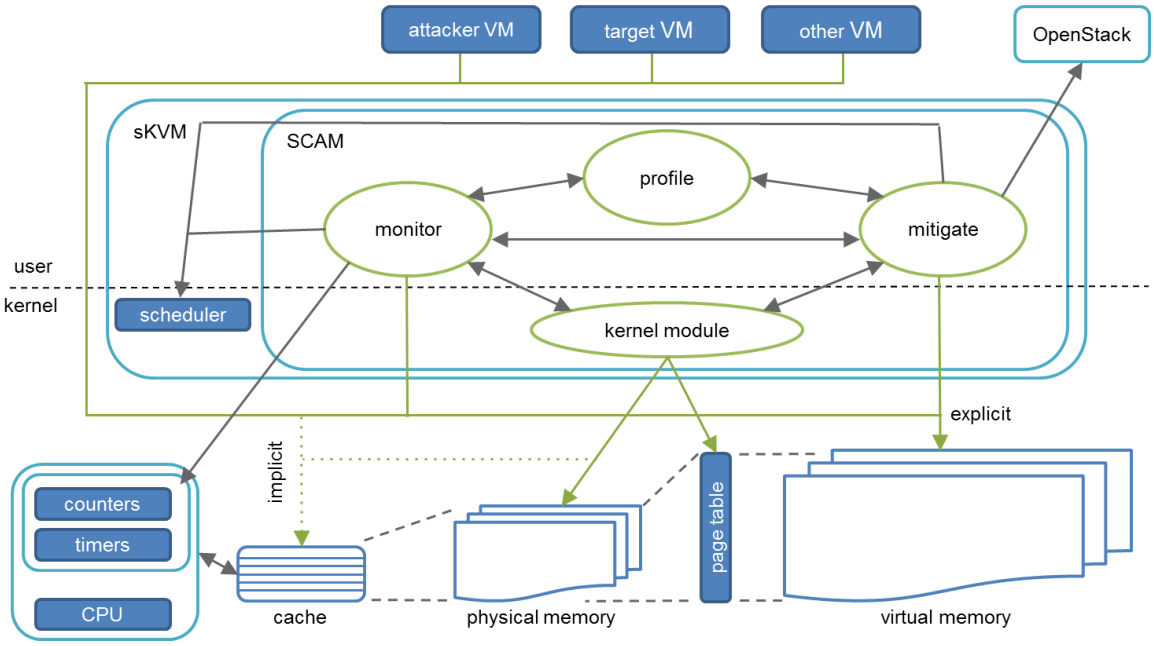

Figure 14: SCAM Architecture

© Copyright Beneficiaries of the MIKELANGELO Project

The current modules implemented within the architecture include the monitoring module, along with initial profiling, as well as the initial implementation of the mitigation tools which focuses on a specific noisification module. The entire implementation is currently in user-space. The modules implemented so far interact with the various system counters (for monitoring), and timers (for noisification). Furthermore, the noisification module accesses cache lines implicitly by explicitly accessing specific virtual memory locations.

As we elaborate in section 4.4, our future plans for extending SCAM include implementing some of the functionality in kernel-space (for both improved performance, as well as better control of resources), and more specifically we plan to further develop and validate our proposed monitoring and mitigation approaches (and potentially explore the effectiveness of additional techniques).

4.3 Monitoring and Mitigation Modules

In this section we describe the design and implementation of the two security modules contributing to SCAM. In section 4.3.1 we describe the Monitoring module, and provide an evaluation study of its performance. In section 4.3.2 we provide a complete rundown of our proposed mitigation technique, the noisification module, along with a performance evaluation of its effectiveness and tradeoffs involved in its operation.

4.3.1 Monitoring

A method for monitoring VM cache activity to identify cache attacks should ideally be lightweight and as accurate as possible. We propose and implement two methods for such monitoring. The first relies on reading the PMU (Performance Measuring Unit) which includes CPU counters that collect statistics on cache activity. It is lightweight and as our experiments prove, can identify most attack strategies with very high accuracy. The second method relies on running the prime-and-probe algorithm to detect attacks given the cache sets which the attacker monitors. This second method, while extremely accurate, should only be used as an addition to the first, since its performance toll is unacceptable in most scenarios.

Other monitoring strategies are possible but have significant drawbacks. They include static analysis of VM code, searching for code patterns that are likely for prime-and-probe attacks and full emulation of the VM, dynamically inspecting each instruction it executes, looking for the same patterns. The first option may be defeated by code obfuscation and may not be applicable in some settings due to regulation and privacy concerns. The second option implies a severe downgrade in performance, but its potential accuracy may make it attractive as a complement to other methods.

© Copyright Beneficiaries of the MIKELANGELO Project

Any strategy for identifying prime-and-probe cache attacks must rely on the low-level characteristics of the attack. The attacker repeatedly fills a cache set with its data, which requires cache accesses and incurs cache misses if the target works in parallel to it. That is, *the cache activity of the attacker and the target are correlated*, with cache misses for both processes increasing when they repeatedly use the same cache sets.

The current version of SCAM uses the PMU extensively to identify attacks and has the option to use prime-and-probe to complement PMU measurements. SCAM monitoring integrates the PAPI interface to run the PMU.

4.3.1.1 Performance Monitoring Unit

Modern processors by Intel, ARM, IBM and others are equipped with a Performance Monitoring Unit (PMU), which provides the user with data and statistics on the performance of various CPU components. The user (in our case the hypervisor) selects a list of hardware events to monitor such as memory events (LLC cache misses, LLC cache accesses, etc.) and CPU events (branch mispredictions, branch predictions, etc.). The PMU controls a set of special purpose registers called hardware performance counters (HPCs) which count the events. The PMU is typically used for performance profiling, but in our case can be repurposed for monitoring. There are two common ways to read PMU data, either through perf or by using the PAPI interface.

Perf - The user can configure the PMU and read the HPCs state either by using the perf_event_open system call or a command line application called perf and perf-stat. The HPCs may be configured to count the events generated by a specific thread. The maximum effective sampling frequency of the perf_event_open is a sample every several microseconds, but the perf application is limited to sampling at most once every 100 milliseconds.

Performance Application Programming Interface (PAPI) is an open source library [17] which supports a wide range of processors and offers two levels of platform-independent interface for monitoring events and a high level management interface. PAPI's maximum sampling frequency is in order of a few microseconds (depending on system workload and OS scheduling).

4.3.1.2 Standalone Measurements

We first show measurements of the target server running without interference from the attacker and the attacker running independently with the server inactive. In both cases SCAM code was executed on a different core.

The target is a web server configured with a 4096-bit RSA key. A client repeatedly connects to it using the TLS handshake protocol, which is the way to create secure HTTP (i.e. https)

© Copyright Beneficiaries of the MIKELANGELO Project

connections. In Figure 15 the horizontal axis counts the number of probes that SCAM monitoring makes of the PMU through PAPI. A typical window of 8000 probes is presented out of a million probes that were performed as part of the experiment. The vertical axis counts the number of cache accesses and cache misses between two successive probes of the PMU. The sampling frequency of the PMU is one sample per 100 µseconds, which is sufficient for our purposes. The blue lines represent the number of Total Cache Accesses (TCA) while the red lines represent Total Cache Misses (TCM).

Note the periodic behavior of the server, which is caused by repeated connections from the client with no other activity. A single connection requires about 69 milliseconds, out of which computing the signature, using the RSA private key requires 65 milliseconds.

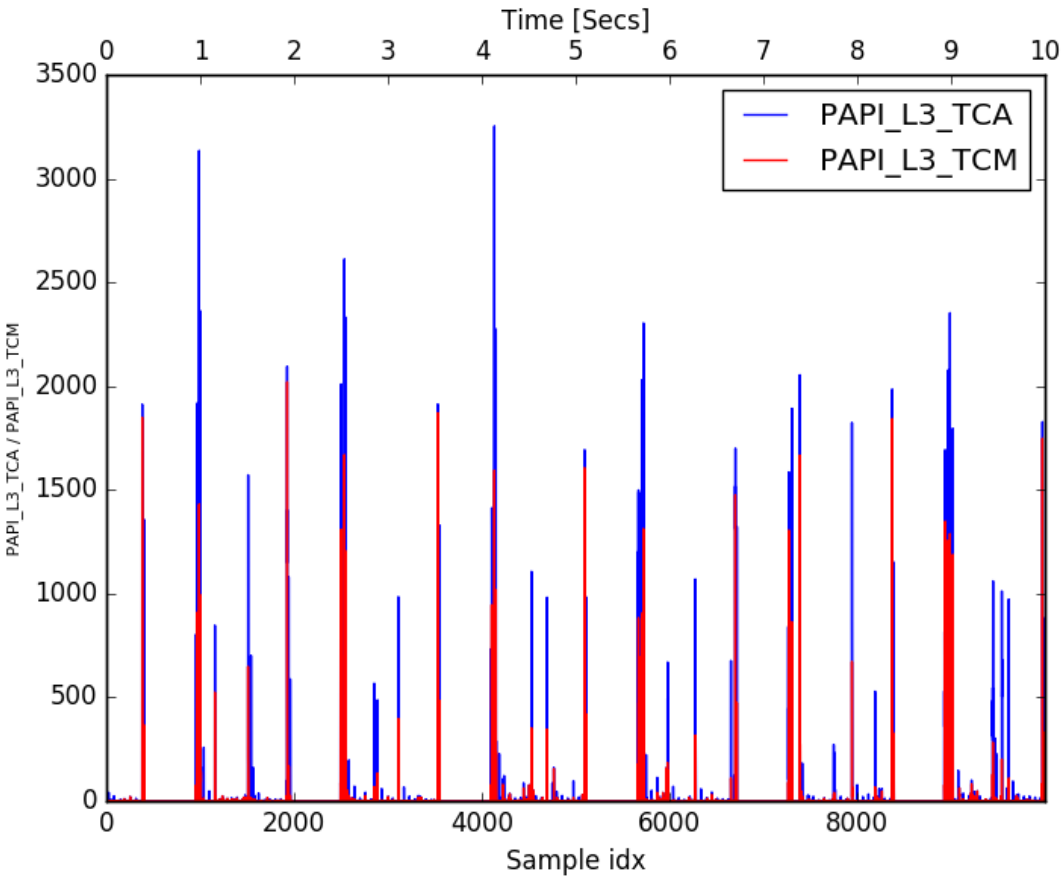

Figure 15: Standalone web server cache accesses and misses.

The next figure presents similar cache activity for the attack we developed as a way to test SCAM. The attacker's rate of accesses is relatively high (this is a side-effect of our attempt to make the attack fast rather than stealthy) while the rate of misses is relatively low. The low miss rate is expected when the server is not active since the attacker fills cache sets with its data, but that data is typically still in the cache when the attacker reads it back. Note that the

© Copyright Beneficiaries of the MIKELANGELO Project

accesses and misses of the attacker do not resemble the pattern that was measured for the web server.

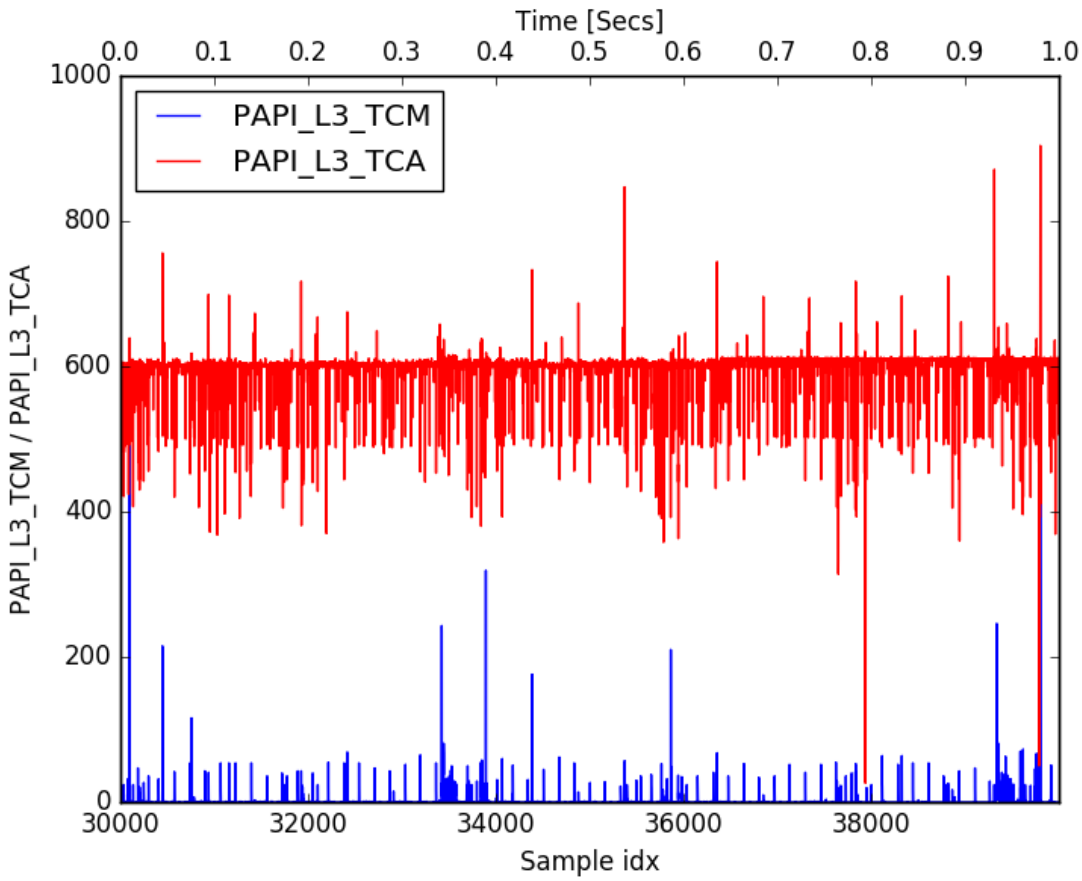

Figure 16: Measurements of standalone attacker's cache accesses and misses.

4.3.1.3 Correlated Measurements

When both the server and an attacker work concurrently the ratio of misses to accesses increases as both processes access the same cache sets. Furthermore, the attacker fills the cache sets with its own data, causing cache misses for the server, while each access by the server causes a cache miss for the attacker. The following diagram shows attacker and server cache activity when working concurrently with each other.

© Copyright Beneficiaries of the MIKELANGELO Project

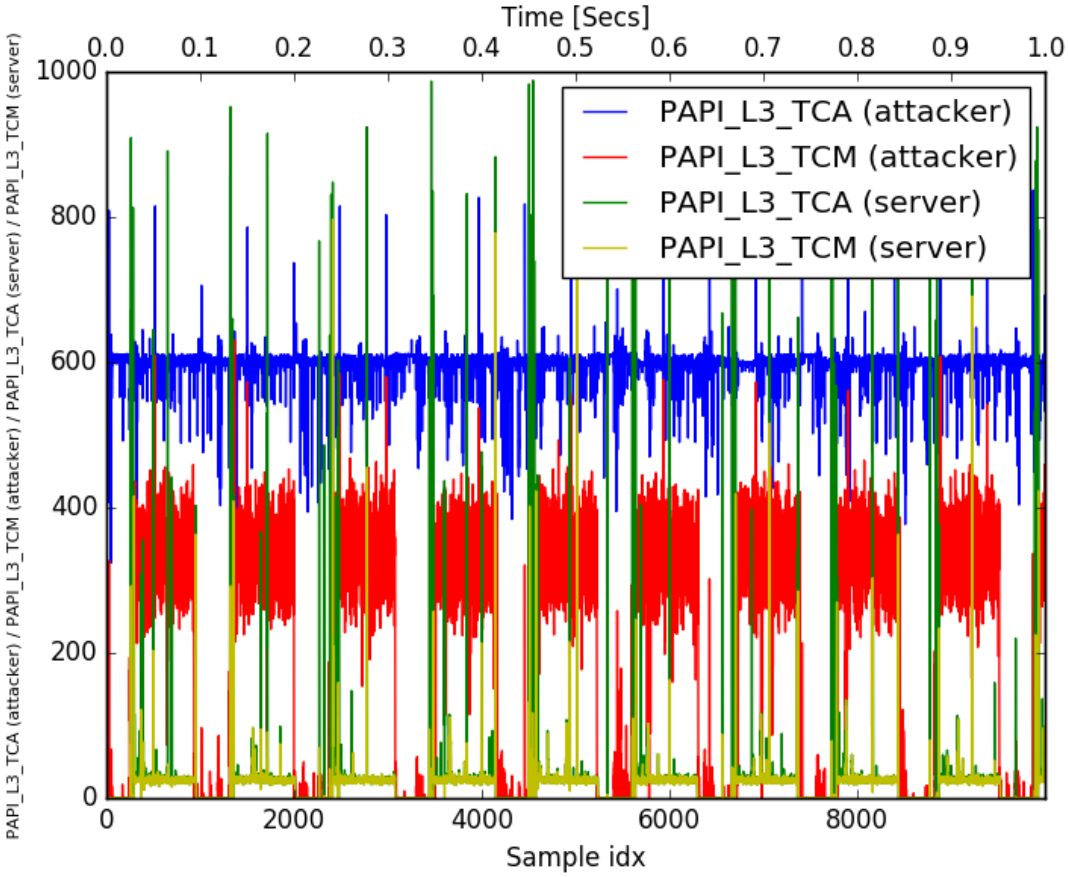

Figure 17: Attacker cache activity while working concurrently with server.

Figure 17 clearly shows that the number of accesses by an attacker is largely unchanged when working in standalone mode or concurrently with the server. However, the miss ratio is much higher and is exactly correlated with server activity[1]. Note that the number of misses in Figure 17 is increased compared with the measurement of standalone server activity in Figure 15.

4.3.1.4 Identification

The monitoring module assigns a score to each VM that determines how closely its activity resembles a cache attack, or in other words how suspicious it is. The critical data for each VM is its miss ratio, i.e. the ratio between the number of cache misses and cache accesses in a time interval. The monitoring module stores a low miss ratio and a high miss ratio for each VM. The two ratios are computed by computing the miss ratio for each PMU probe, taking the average of the higher half of the measurements as the high miss ratio and the lower half

[1] The number of misses is synchronized with the RSA private operations that the server performs.

© Copyright Beneficiaries of the MIKELANGELO Project

as the low miss ratio. Outliers are removed before computing both ratios. The two ratios are updated dynamically throughout the operation of a VM.

For each time interval of 100 milliseconds the monitoring tests if the miss ratio of a VM was close to the low miss ratio for at least 4 milliseconds and then close to the high miss ratio for at least 15 milliseconds. The values represent measurements for a TLS handshake in which the high values occur when the server computes RSA signatures (which takes ~65 milliseconds in the setup we use and could be four times faster using shorter keys) and the low values occur during the rest of the handshake. If both conditions are met then the VM's score is increased by 1. After a threshold is reached (currently set to 10) in a predetermined time interval T, the VM is considered to be a likely cache attacker. If not, then the counter is reset. Determining T While the current monitoring algorithm is successful in identifying the current attack, we intend to look into likely extensions and improvements in year 3. Some of the natural extensions include testing it on different hardware, using different workloads with possibly other RSA implementations and craftier attackers.

4.3.1.5 Performance

To check the impact of monitoring on system performance we measured the effect of different rates of monitoring on the server. The average setup time for a HTTPS connection between a web client and the server is summarized in Figure 18.

Probing the PMU at 100 µseconds incurs less than 1% overhead and is sufficient for successfully identifying the attacker, which is why we intend to use that rate in the future. Probing at higher rates induces reduced performance as shown in Figure 18 and does not significantly add to detection rates. Additional tests on the average number of cycles and instructions in the server present a similar picture of little difference in server performance between no monitoring at all and probing the PMU every 100 µseconds.

© Copyright Beneficiaries of the MIKELANGELO Project

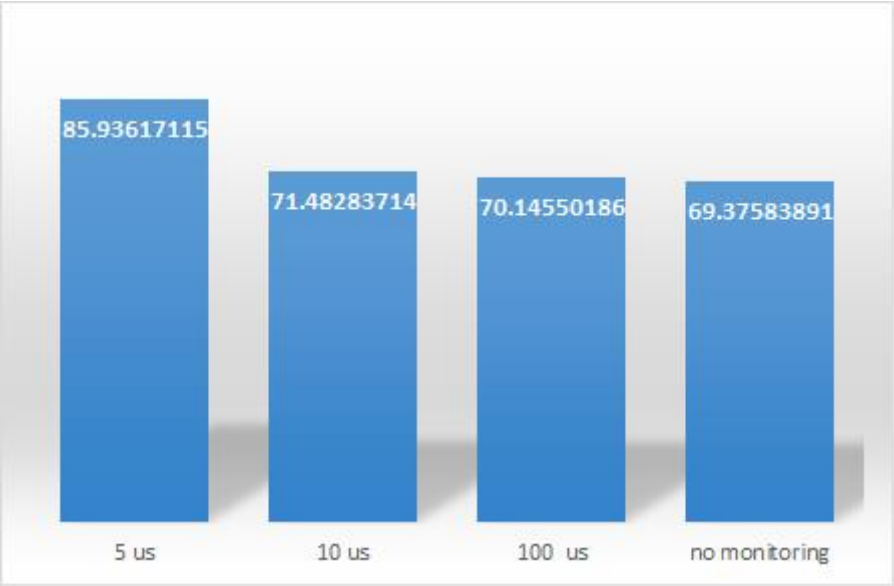

Figure 18: Server connection time for different probe rates of the PMU

4.3.2 Mitigation by Noisification

In the current section we describe a method for mitigating cache side-channel attacks in virtualized environment. Our method is based on a process which introduces noise into a potential attacker's attempt to discern cache activity of a co-located target. We refer to our proposed mitigation approach as "noisification", and refer to the module implementing this approach as the noisification module (NM).

We first provide a complete description of our method, then turn to present our performance analysis, and finally conclude with a discussion of the limitations and further extensions of our approach.

4.3.2.1 Overview

Our implementation of the noisification module is implemented as a user-space process running on the host.

The goal of the NM is to cause cache misses in the attacker's cache accesses, so as to significantly reduce, and ideally obliterate, its ability to extract bits of the private key used by the target server.

In a nutshell, the ideal mitigation method would cause *all* access made by the attacker to induce a cache miss. In such a case the attacker cannot infer any information from following hit-miss patterns in its cache access activity. The toll of implementing such an approach on the hypervisor-level would be a severe degradation in performance, and might require unreasonable resource to implement (e.g., multiple cores dedicated solely to noisification).

© Copyright Beneficiaries of the MIKELANGELO Project

However, as we show, even if some of the access made by the attacker is deflected, it is sufficient to reduce the rate of the attack, and at times even eliminate it completely.

The order in which processes and VMs are initiated on the host machine is as follows:

- NM
- Target VM (server + client)
- Attacker VM

This specific order allows the NM to perform its first reconnaissance phase on a bare system, thus learning the baseline performance of the system. Following this initial phase, once the target VM is initiated, the NM is able to perform its second reconnaissance phase, of identifying the caches sets used by the target, which will be the focal point of the NM once the attacker is initiated.

4.3.2.2 Detailed Description

We now turn to describe the various tasks performed by the noisification module, and its possible modes.

Reconnaissance

- Building the memory-cache mapping:
  Upon executing on a bare system, the NM begins 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 [16]. This procedure results in a mapping

  $$\varphi\text{: NM-memory -> LLC-sets}$$

  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

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

  We further assume some arbitrary order on $\varphi^{-1}(S)$, for every LLC-set S. This order allows us to consistently access some k lines of each set, for any k, and in particular, allows us to implicitly access the i'th line corresponding to some set S, in a consistent manner.

- Profiling the baseline cache activity:

© Copyright Beneficiaries of the MIKELANGELO Project

Once the memory-cache mapping is available, the NM performs a prime-and-probe procedure for identifying the baseline cache activity of the system, and gathering activity statistics for each cache set.

This phase results in a table modeling the cache-access profile of a bare system with no additional co-located VMs (neither target nor attacker).

- Identifying target cache sets:
  After the target VM server is running, and a client constantly accesses the server (which causes the server to start performing its decryption procedure repeatedly), the NM performs a prime-and-probe procedure for identifying the active sets used by the target server. This mimics the procedure employed by the attacker when identifying the relevant target cache sets, as described in deliverable D3.4 [16].

  In particular, this procedure results in a table modeling the cache-access profile of the system with an active target VM on top of a bare system. Cache sets which show a distinct activity profile compared to the baseline cache profile, are considered as the active sets used by the target. We denote the collection of sets thus identified by *M*.

Noisification

After having identified the collection *M* of sets which are potentially used by the target server, and given the ordering of the cache lines of each set, the NM is activated by the following procedure, taking two parameters, *M* and *k*, where intensity parameter *k* determines the number of lines accessed by the module in each noisified set.:

```
ActivateNoisification(M,k)
for each cache set S in M
     for i = 1 to k
          access line i of S
     endfor
Endfor
```

4.3.2.3 Evaluation

We have performed multiple simulations of testing the performance of our noisification module, NM, with the attacker module developed within the project, as described in detail in deliverable D3.4. This section provides the results of our evaluation study.

We first discuss some tradeoffs in the parameters of the NM. Recall that NM identifies a collection *M* of cache sets that are potentially used by the target, and which are expected to

© Copyright Beneficiaries of the MIKELANGELO Project

be the cache sets primed and probed by the attacker. Since NM does not know which of these cache sets will actually be accessed by the attacker, nor when the attacker will be accessing them, it is required for the noisification module to introduce noise that affects all of these cache sets. The ability of NM to introduce noise into a collection of cache sets depends on several parameters, namely:

1. Parallelism level of NM, i.e., the number of available cores for NM
2. The number of cache sets to protect, i.e., $|M|$
3. The intensity, or amount of noise, introduced to each protected cache-set, modeled by parameter k.

Ideally, we would like to have $|M|$ as small as possible, and k as large as possible, to ensure the highest level of protection. However, the size of M is determined by the measurements performed by NM, and cannot be controlled directly by NM. It follows that the main parameter which is under the control of NM, with which it can fine-tune the application of noisification to the protected cache sets, is k, i.e., the number of cache lines accessed in each protected cache set. Clearly this parameter should take values between 1 and w, where w is the set-associativity of the LLC in the system. We note that the higher the value of k, the more resource consuming it is for NM to protect a single cache set, which in turn implies a lower rate in which it can traverse the entire collection of cache sets in M. This introduces an inherent tradeoff between the *breadth* of the protection (i.e., the rate in which all sets in M can be noisified), and the *completeness* of the protection (i.e., the amount of noise introduced into each protected cache set).

We note that in our evaluation all tests allowed a single core to be used by NM, hence, no parallelism was employed. This allows to better understand the baseline security performance of NM. As one can expect, allowing NM to use multiple cores can increase the breadth of protection provided by NM.

We now turn to recall some of the parameters used in the implementation of our attack (a complete description is provided in deliverable D3.4). The attacker builds its own memory-to-LLC-sets mapping, and then tests all cache sets, assuming the target is constantly performing decryption using its private key, for some known patterns that represent the conditional statement which the attacker is trying to identify, as described in section 4.1 above. The attacker then identifies the top-5 sets for which the patterns were best. For each of the sets, the attacker attempts to gather 200 samples of (potentially) complete decryptions, each using the entire key. It then performs alignment, and takes majority voting for deciding the value of each bit in the private key.

© Copyright Beneficiaries of the MIKELANGELO Project

We have tested multiple attempts of the attacker to recover the key, with and without NM working. Our system has a 8192 cache sets in the LLC, organized as a 12-way associative cache. In all of our experiments the NM was able to map all 8192 cache sets.

We have allowed the attacker to recover the key for 500 attempts: 20 tests, where in each test the attacker attempts to recover the key 5 times from each of its top 5 cache sets. NM worked with intensity parameters k = 1, 2, 5. Table 1 gives a summary of the performance of NM in the above settings. For each value of k, the table shows the statistics of the size of M (min, max, average, standard deviation), the number of sets mapped by the attacker (min, max, average, standard deviation), the number of attempts in which the attacker was able to extract more than 2 samples (number of such attempts, and the average and standard deviation of the number of samples in those attempts), and the number of successful attempts.

Table 1: Noisification evaluation results

	# sets to protect by NM	#sets mapped by attacker	#attempts with >2 samples + statistics	#successful attempts
Without NM	-	8182-8190 AVG: 8184 STD: 10	85 AVG: 149 STD: 74	62
With NM, k=1	190-358 AVG: 227 STD: 40	8127-8178 AVG: 8159 STD: 15	17 AVG: 5 STD: 2	0
With NM, k=2	217-299 AVG: 258 STD: 24	8087-8187 AVG: 8167 STD: 20	22 AVG: 20 STD: 24	0
With NM, k=5	194-259 AVG: 234 STD: 19	8156-8187 AVG: 8180 STD: 8	27 AVG: 27 STD: 46	0

As can be seen by our results, activating NM results in a drastic decrease in the number of samples extracted by the attacker. Furthermore, the table shows that decreasing the intensity of protecting each cache set does not affect the number of successful attempts. However,

© Copyright Beneficiaries of the MIKELANGELO Project

increasing the intensity implies a lower rate of traversing the entire collection of protected cache sets, which results in the attacker's ability to obtain more useful samples (although insufficiently many to actually recover the private key).

It is important to note that the above results are valid for our specific implementation of the attacker. Alternate implementations may result in different performance. However, the overall insight obtained by our evaluation study is that:

(1) Noisification is an effective method in reducing, and potentially eliminating, the effectiveness of cache side-channel attacks
(2) There is an inherent tradeoff between the number of cache sets to be protected (breadth, as captured by the size of *M*) and the ability to provide full protection to these sets (completeness, as captured by parameter *k*).
(3) The exact level of security can be fine tuned using the intensity parameter *k*.
(4) Potentially, alternate implementations of the attack might necessitate using higher/lower value of *k*.

4.3.3 Future

The current modules designed and implemented as part of SCAM provide the initial design of our proposed monitoring and mitigation solutions. In further developing SCAM, we plan to address the following topics and issues:

(1) SCAM as a security suite
We plan to provide a unified framework for both monitoring and mitigation that is able to support several modes. One mode is the *proactive mode*, in which the mitigation tools are made active at boot time, regardless of any activity discerned by the monitoring module. Another mode we plan to support is *reactive mode*, where mitigation is triggered by the monitoring module as soon as suspicious activity is identified. These two modes apply to both the noisification module, and/or to any additional mitigation techniques to be developed within the framework of SCAM.

(2) Improved implementation
We plan to implement parts of the modules developed so far, as well as future modules, as kernel processes, for improved performance, and better fine tuning of parameters. For example, in the context of noisifcation module, we expect such an implementation to increase the completeness of the protection provided to cache sets (i.e., enabling increasing k without the current toll of the increased number of samples collected by the attacker). This will also serve to verify the adequacy of our approach against other potential implementations of the attacker. We recall that enabling our mitigation module to access all cache lines in a set, consistently, at high-

© Copyright Beneficiaries of the MIKELANGELO Project

rates, will cause the attacker to encounter nothing but cache misses in his attempts to prime-and-probe the LLC. We expect that a kernel-side implementation of some of the components may enable coming close to this ideal.

(3) Fine grained monitoring
We plan to improve our monitoring module by both considering offline approaches for anomaly detection within the monitoring module, in order to provide improved security, as well as by considering alternative implementations of the attacker. This is expected to be especially crucial for the application of our mitigation techniques in reactive mode.

(4) Improved mitigation
We intend to improve our mitigation abilities, by developing stronger noisification mechanisms, which are able to thwart alternate implementations of the attack. Alongside the noisification approach described in the previous sections, we hope to explore the effectiveness, potentially in tandem, of additional mitigation techniques, such as page-remapping, which will perform virtual-to-physical page remapping at run time, thus altering the potential target cache sets.

(5) Integration of SCAM within the MIKELANGELO stack
We will integrate SCAM as an integral part of the MIKELANGELO stack. In particular, we plan to provide a clear API for managing the various parameters in the various modules which are part of SCAM. We will further integrate SCAM, along with the various metrics that affect its performance, within the Snap monitoring overviewing the MIKELANGELO stack as a whole.

(6) Testing and evaluation
We plan to test and evaluate the entire SCAM suite on the testbed used by the MIKELANGELO project, in order to better understand the performance toll imposed by our modules, alongside real-life workloads and scenarios.

© Copyright Beneficiaries of the MIKELANGELO Project

5 Concluding Remarks

sKVM has now reached a level of maturity which contains a working implementation of each of the components; IOcm, vRDMA and SCAM. Each component has been tested individually, and shown to work on the various use cases according to the design of the components. We have shown the progress made during the past year, bringing the basic working prototypes that were available at the end of year 1 into fully functional features, including in-depth evaluation. In the coming year, sKVM will continue to mature and develop in regard to performance to integrate these features together, and become truly usable.

© Copyright Beneficiaries of the MIKELANGELO Project

6 References and Applicable Documents

[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 D2.20 - "The intermediate MIKELANGELO architecture", https://www.mikelangelo-project.eu/wp-content/uploads/2016/07/MIKELANGELO-WP2.20-USTUTT-v2.0.pdf

[4] 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

[5] Har'El, Nadav et al. "Efficient and Scalable Paravirtual I/O System." *USENIX Annual Technical Conference* 26 Jun. 2013: 231-242.

[6] "sysfs - Wikipedia, the free encyclopedia." 2011. 17 Dec. 2015 https://en.wikipedia.org/wiki/Sysfs

[7] The Netperf Homepage. 2002. 17 Dec. 2015, http://www.netperf.org/

[8] The Apache HTTP Server Project. 2009. 17 Dec. 2015, https://httpd.apache.org/

[9] ab - Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.2/programs/ab.html

[10] MIKELANGELO Report D2.16 The First OSv Guest Operating System MIKELANGELO Architecture, https://www.mikelangelo-project.eu/deliverables/deliverable-d2-16/

[11] MIKELANGELO Report D4.1 The First Report on I/O Aspects, https://www.mikelangelo-project.eu/deliverables/deliverable-d4-1/

[12] The C. Macdonell, X. Ke, A. W. Gordon, and P. Lu, "LOW-LATENCY, HIGH-BANDWIDTH USE CASES FOR NAHANNI / IVSHMEM," Forum Am. Bar Assoc., pp. 1–24, 2011.

[13] I. Corporation, "Intel Data Plane Development Kit (Intel DPDK)," no. June, pp. 1–43, 2013.

[14] vhost-user, https://github.com/qemu/qemu/blob/master/docs/specs/vhost-user.txt

[15] Open vSwitch, http://openvswitch.org

[16] 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

[17] 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.

© Copyright Beneficiaries of the MIKELANGELO Project

[18] E. Moscovici and Y. Kuperman, "IOcm IO manager source code repository", https://github.com/mikelangelo-project/dynamic-io-manager
[19] MIKELANGELO Report D4.5 "OSv - Guest Operating System – Intermediate Version", http://www.mikelangelo-project.eu/deliverables/deliverable-d4-5/
[20] IBM MessageSight, http://www.ibm.com/support/knowledgecenter/SSCGGQ
[21] https://linux.die.net/man/8/opensm
[22] Git repository of HyV, https://github.com/zrlio/hyv
[23] RDMA Aware Programming user Manual, https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf
[24] Kernelnewbies.org, "MacVTap", http://virt.kernelnewbies.org/MacVTap

© Copyright Beneficiaries of the MIKELANGELO Project

7 Appendix A: Kernel config file changes

The following is a list of configuration options that must be set when building the Linux kernel for IOcm.

```
CONFIG_VIRTIO_BLK=m
CONFIG_VIRTIO_NET=m
CONFIG_TARGET_CORE=m

CONFIG_MACVLAN=m
CONFIG_MACVTAP=m
CONFIG_TUN=m
CONFIG_BRIDGE=m
CONFIG_STP=m
CONFIG_LLC=m

CONFIG_CAIF=m
CONFIG_CAIF_VIRTIO=m
CONFIG_VHOST_NET=m
CONFIG_VHOST_BLK=m
CONFIG_VHOST_RING=m
CONFIG_VHOST=m

CONFIG_BLK_DEV=y
CONFIG_BLK_DEV_LOOP=m
CONFIG_BLK_DEV_NBD=m
CONFIG_BLK_DEV_LOOP_MIN_COUNT=16
CONFIG_BLK_DEV_CRYPTOLOOP=m
CONFIG_BLK_DEV_RAM=m
CONFIG_BLK_DEV_RAM_COUNT=16
CONFIG_BLK_DEV_RAM_SIZE=4096
```


© Copyright Beneficiaries of the MIKELANGELO Project