KVMIveggur: Flexible, secure, and efficient support for self-service virtual machine introspection

By:
Stewart Sentanoe (University of Passau), Thomas Dangl (University of Passau), and Hans P. Reiser (University of Passau)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2022
July 11-14, 2022

Forensic Science International: Digital Investigation 42 (2022) 301397

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

KVMIveggur: Flexible, secure, and efficient support for self-service virtual machine introspection

Stewart Sentanoe [a, *], Thomas Dangl [a], Hans P. Reiser [a, b]

[a] *University of Passau, Innstr. 43, 94032, Passau, Germany*
[b] *Reykjavík University, Menntavegur 1, 102, Reykjavík, Iceland*

A R T I C L E I N F O

Article history:

Keywords:
Virtual machine introspection
Virtual machine
KVM
Access control

A B S T R A C T

Virtual machine introspection (VMI) has evolved into a widely used technique for purposes such as digital forensics, intrusion detection, and malware analysis. The recent integration of enhanced VMI capabilities into KVM further facilitates the use of VMI. A significant obstacle, however, remains: VMI usually requires highly privileged access to the host system. Existing research prototypes that address this issue either target only the Xen hypervisor, are extremely slow, offer only a subset of the desired functionality, or are hard to deploy in real-life systems. We present our flexible *KVMIveggur* architecture as a novel solution to these challenges. It offers three flavors of isolation (using containers, virtual machines, and network remote access) that all enable access control for secure self-service VMI in cloud environments. It enables the full use of passive and active VMI, supports continuous monitoring also during live VM migration, and can be tailored for low overhead and minimal resource utilization on the host system. The experimental evaluation of our prototype demonstrates the feasibility and the efficiency of our approach and provides detailed insights into the differences between the three flavors.

1. Introduction

Virtual machine introspection (VMI) is the act of monitoring a virtual machine from the outside to gain knowledge about its inner state. First introduced by Garfinkel and Rosenblum (2003), VMI has developed into a set of techniques and mechanisms much more sophisticated by 2021. By now, many applications that were previously implemented as in-guest tools may be deployed outside the virtual machine (Bushouse and Reeves, 2018b). Thereby, these tools can capture great benefits of introspection, such as an untempered view and the inherent isolation of the analysis. The primary applications of VMI include — but are not limited to — digital forensics, intrusion detection, and malware analysis (Dolan-Gavitt et al., 2011; Hebbal et al., 2015).

Among others, a common domain for VMI applications is cloud infrastructures. The industry has seen considerable growth through the widespread adoption of IaaS — a trend that is only expected to continue (Gartner Inc., 2020). However, due to security and privacy concerns, VMI has only seen use in private clouds. The standard approach of running introspection applications directly on the host or on the privileged management domain *Dom0* quickly falls apart for public and/or multi-user clouds due to the lack of isolation and the high attack surface of the introspection interface.

There are proposed general-purpose solutions for multi-tenant clouds such as *CloudVMI* (Baek et al., 2014), *Furnace* (Bushouse and Reeves, 2018a), and *CloudPhylactor* (Taubmann et al., 2016) as well as domain-specific solutions such as *Frost* (Dykstra and Sherman, 2013). Regrettably, public clouds cloud vendors have not yet incorporated such systems. We argue that the reason for this lack of support is that the existing introspection architectures are either limited to specific hypervisors and configurations, have high resource costs and performance overhead, or are inflexible and therefore only applicable to specialized contexts. Furthermore, the solutions typically introduce a high amount of additional complexity to such clouds. Therefore, a suitable access control mechanism for multi-tenant, public clouds must be simple, flexible, and ideally leverage widespread technologies already present in the cloud infrastructure.

Recently, the topic of virtual machine introspection has caught traction with hypervisor and anti-virus vendors (VMware Inc., 2020; Kaspersky Lab, 2020; Bitdefender, 2019) and open-source projects (Lazăr, 2020). The latter have opted to expose their

* Corresponding author.
E-mail addresses: se@sec.uni-passau.de (S. Sentanoe), td@sec.uni-passau.de (T. Dangl), hansr@ru.is (H.P. Reiser).

introspection capabilities to the user as a UNIX socket (introspection socket). This paper examines this interface and argues for an access control scheme using existing paravirtualized device drivers. Our goal is to create an analysis environment for introspection applications that is easy to integrate into existing solutions and that mandates no additional trusted code, thereby reducing the complexity of the overall system. In line with this objective, we propose a solution that increases the efficiency and improve upon the speed of VMI applications.

Therefore, we digest multiple approaches: monitor from a container, from another VM, and over the network. Our first approach is applicable to many existing container solutions. For this paper, we use *Docker* (Docker Inc., 2013) because it is a well-known and widely-used container technology. For this purpose, the existing bind mount feature is repurposed to provision the introspection socket inside the container. We pair each monitored virtual machine with a homogeneous container that is responsible for its introspection. The life cycle of this container is coupled to that of the virtual machine it is monitoring by utilizing existing configuration capabilities of cloud computing platforms such as *OpenNebula* or *OpenStack*.

On the second approach, we aim to extend this access control mechanism to hardware- and paravirtualized monitoring virtual machines. Unlike the implementation for containers, the approach for virtual machines uses the *virtio*-based device drivers. In particular, we adopt *virtio-vsock* (Hajnoczi, 2020), which implements the *vsock* protocol using an emulated *PCIe* device to provide for bi-directional host–guest communication. Intuitively, the expected overhead in both resources and introspection performance will be higher than the containerized equivalent. However, since the required driver is packaged with *QEMU*, this solution requires no additional software to be installed and can run on the same unmodified hypervisor. Thus, it is more suitable for a wider variety of cases.

On the last approach, we expose the introspection socket over the network using prevalent relay tools that are already available on many cloud platforms. We realize this by exposing a TCP socket from the hypervisor over the network and binding it to a UNIX domain socket on the other end for the monitoring application. Thereby, we facilitate the off-loading of some tasks to other networks and even other cloud providers at the cost of newly introduced latency.

We evaluate the respective benefits and drawbacks of the three presented techniques concerning performance, integrity, and robustness. From the obtained results, we derive and present use cases for different introspection applications. Finally, we look into further developments of the approaches and discuss current shortcomings.

The main contributions of this paper are:

1. We propose the repurposing of paravirtualized device drivers to enable access control and isolation for VMI-based monitoring applications.
2. We provide and discuss implementations of the VMI access control and isolation by leveraging *Docker* containers.
3. We reason about introspection applications over the network and demonstrate low overhead implementations that integrate well into existing cloud infrastructures.
4. Using these implementations, we port real-world virtual machine introspection applications, evaluate them in the context of our and other architectures, and discuss their performance.

The rest of the paper is organized as follows: Section 2 presents the relevant background for VMI. Section 3 covers existing approaches toward secure self-service VMI in related work. In Section 4, we discuss the concepts behind our introspection environment and its requirements. In Section 5, we introduce our implementation in detail. Section 6 assesses the use cases of this implementation and their merits regarding performance, robustness, and integrity. Finally, Section 7 concludes the paper.

2. Virtual machine introspection

Virtual machine introspection (VMI) is a method to analyze a guest virtual machine (VM) from the outside with the assistance of the hypervisor (Garfinkel and Rosenblum, 2003). VMI tools have full access to the *state* of a VM, in particular to the main memory as well as the state of vCPU registers and virtual devices. VMI tools can also observe *events* in a VM via interpositioning and intercepting operations such as modifications of vCPU registers and access to specific memory pages.

The techniques used by VMI tools do not require any in-guest agent or cooperation of the monitored machine (Laurén and Leppänen, 2018). Examples of systems built by this principle include *LiveWire* (Garfinkel and Rosenblum, 2003) and *Lares* (Payne et al., 2008). VMI has several advantages: The VMI-based analysis tool is fully isolated from the target VM, protecting it from malicious manipulations. The analysis tool is also not visible from within the target VM, enabling stealthy analysis.

The main challenge of VMI is the semantic gap between data representation in the low-level memory view, i.e., raw binary data, and the high-level interpretation that the guest system in the virtual machine has of that data, e.g., address spaces and data structures (Chen and Noble, 2001). Every VMI application has to solve this problem and reconstruct the high-level view of the guest system. Over the past two decades, researchers have presented many approaches to accomplish this goal (Hebbal et al., 2015; Bauman et al., 2015).

Mechanisms that employ virtual machine introspection can be categorized into two kinds (Jain et al., 2014): *Asynchronous* or *passive* VMI uses external events, e.g., a timer in the introspection application, to periodically analyze the target virtual machine using main memory introspection. In contrast, *synchronous* or *active* mechanisms intercept the monitored virtual machine at specific locations in its control flow to perform context-sensitive analysis. For example, this may occur when the introspection application places a breakpoint in the virtual machine. We aim at supporting and evaluating both kinds of VMI in the architecture proposed in this paper.

VMI requires privileged access to the hypervisor. For example, in Xen-based systems, VMI operations are accessible via hypercalls that usually can be used only by a privileged process on the Dom0. In the KVMi subsystem for KVM, VMI access to a virtual machine is exposed to a monitoring application via a UNIX domain socket on the host system.

In the most common deployments, a VMI application runs in a privileged context that permits VMI access to all virtual machines on the host. We use the term *self-service VMI* for an environment in which a cloud customer can request VMI access to its own VMs from the multi-tenant cloud infrastructure in a secure way. Any system in which a VMI application has VMI access to all virtual machines on a host is not suitable for self-service VMI.

VMI also has a non-negligible performance impact. Several factors contribute to that impact. First, the VMI application causes additional resource consumption (CPU time, main memory, network bandwidth) on the physical host of the target VM. Furthermore, any VMI operation has some impact on the run-time behavior of the target system. Passive VMI examines memory state from the outside, which has no (or essentially negligible if you consider cache and memory bus contention effects) direct impact, but in many cases, a VMI application needs a consistent view of the target state, which requires some way to prevent concurrent modifications. The most common approach is to pause the target VM during analysis. For active VMI, the execution of the target VM is intercepted at selected operations (e.g., through traps to the hypervisor on modifications to vCPU registers or selected memory pages). A VMI application can then handle these intercepted operations synchronously, and during this processing, the vCPU must remain suspended. A crucial factor for VMI performance is the latency of processing such interception events.

Live migration of VMs is a challenge for VMI. Live migration of running VMs from one physical host to another is a standard feature on modern cloud platforms. However, usual live migration implementations migrate only the VM state itself, but not the state of a VMI application or the VMI configuration within the hypervisor. For passive VMI, this usually means that migration of the target VM to a different host simply interrupts the introspection. This disadvantage prevents continuous (uninterrupted) monitoring of a VM. For active VMI, this is even more problematic. For example, if the VMI system has injected breakpoints into the target VM memory and the VM is being migrated away without coordination with the VMI applications, these breakpoints — without the corresponding event handlers — may cause the execution of the target VM to fail on the new host. For this reason, on most systems, VMI is incompatible with live migration.

There are plenty of libraries publicly available that provide implementations for VMI, such as *LibVMI* (Lengyel, 2020) and *LibKVMi* (Bitdefender, 2020). In our work, we employ *LibKVMi* on the *KVM* hypervisor.

3. Related work

This paper aims at providing secure and efficient self-service VMI and systematically explores and compares several flavors on how to achieve this functionality. There are several prior approaches that, to some extent, similarly explore potential ways to provide better access control for VMI operations or enable remote access to VMI in cloud environments.

Win et al. (2014) propose the use of mandatory access control in the hypervisor for managing the access of an external control monitor. Their approach assumes that monitoring uses a hidden monitoring module within the target VM, so it differs from traditional VMI performed entirely from the outside. The proposed solution focuses specifically on the Xen hypervisor. The authors point out that the lack of support for VM migration across hosts is a severe limitation that renders current VMI solutions unsuitable for many real-world environments. However, for their own solution, they also do not yet provide a solution for this problem but list it as potential future work.

CloudPhylactor (Taubmann et al., 2016) follows a similar approach using Xen security modules to implement mandatory access control to VMI operations on the Xen hypervisor. An extension to *CloudPhylactor* called *TwinPorter* (Taubmann et al., 2019) enables synchronized migration of a monitoring VM and a monitored VM, thus supporting live VM migration across hosts. *CloudPhylactor* and *TwinPorter* are tightly coupled to the Xen hypervisor and the *Xen* live migration mechanism. Having a dedicated virtual machine for each VMI application with mandatory access control at the hypercall interface provides the desired isolation. However, these approaches also have the resource and latency cost of a dedicated virtual machine.

CloudVMI (Baek et al., 2014) was an early approach that advocates remote self-service VMI via the network. An RPC server on a cloud host exposes VMI functionality and provides access control mechanisms, and cloud tenants can remotely execute VMI applications, thus enabling a kind of “VMI as a service”. The biggest downside of this approach is the high latency induced by network communication. For passive VMI, the only VMI variant that *CloudVMI* supports, the authors of *CloudVMI* measured an overhead of two orders of magnitude compared to using the native API of *LibVMI*. For active VMI, it can be expected that the performance loss due to the latency is prohibitively high. The reason is that active VMI requires feedback from the VMI application before the execution can continue. In *CloudVMI*, this response is delayed by the round-trip time of the network.

Furnace (Bushouse and Reeves, 2018a) is a cloud VMI framework that uses sandboxing to enable self-service VMI for cloud customers, combining some of the concepts of *CloudPhylactor* and *CloudVMI*. Instead of running VMI applications in isolated virtual machines, it uses sandboxes on the Dom0 of a Xen-based system. The sandbox that runs customer code is implemented with Linux namespaces and uses SELinux and Seccomp-BPF for enforcing access control. Currently, *Furnace* supports only the *Xen* hypervisor, and its open-source implementation is incomplete and has not been updated since the publication of the research paper. Furnace does not address the problem of VMI and migration.

FROST (Dykstra and Sherman, 2013) is a digital forensic tool for the *OpenStack* platform. It aims at providing self-service functionality for digital forensic investigations targeting a user's virtual machine in an *OpenStack* cloud. However, it is limited to providing selected functionality to the user, such as creating a snapshot coupled with integrity hashes. It does not enable the execution of arbitrary VMI applications for such investigations.

None of the existing solutions is the ideal solution for all needs. Most existing approaches are focused on the *Xen* hypervisor and cannot easily be applied in different contexts, in particular on the highly popular *KVM* hypervisor and its *KVMi* extensions. This is because KVM just recently added support for VMI, and on the other hand, Xen had supported VMI for a longer time and can be considered mature enough.

We expect different self-service VMI approaches to be more suitable in different contexts. For example, remote VMI via network might simplify migration, but local lightweight sandboxes might be more efficient than remote VMI or encapsulation in virtual machines. However, there is no prior work that systematically compares all approaches. Most likely, a single “fits-all-situations” solution does not exist. This paper aims to address these problems by presenting a universal secure self-service VMI architecture that offers three options for tenant isolation: a dedicated monitoring VM, an introspection *Docker* container, and a network-based remote introspection interface. The *Docker* approach and the network-based approach achieve the same goal as *CloudPhylactor* but require fewer resources.

4. System design

In the following section, we introduce the requirements for and the design of the *KVMIveggur* architecture.

4.1. Requirements

We design *KVMIveggur* with the following requirements in mind: multiple flavors, self-service introspection, secure access, isolation, and easy integration into cloud management tools.

The architecture has to offer multiple methods for the users to perform introspection on their virtual machine — allowing them to select the method based on their needs in a self-service manner. The method selection should be granular at the VM level, which means that a user operating multiple virtual machines can assign a different flavor to each virtual machine.

It must enforce secure access to the monitoring environment and isolation from other services. To ensure the first property, only an authorized user may access the monitoring environment, i.e., the owner of the virtual machine itself or any users granted access by the owner. For the second property, the environment must not be able to introspect other, unassigned virtual machines.

Furthermore, integration into publicly available management tools must be possible without requiring unreasonable modifications. Therefore, *KVMIveggur* can be adopted straightforwardly by any cloud provider wishing to offer introspection-based services.

4.2. Security model

In our security model, an authorized tenant can request introspection capabilities in one of the three flavors from the cloud provider. We assume the cloud provider can be trusted and has no ill intentions, e.g., selling introspection access to machines of unsuspecting customers.

We further assume that both the host operating system and the type 2 hypervisor are trusted. In particular, our system design does not tolerate attacks on these two entities through VM escapes or other attacks. All other entities, e.g., the monitored and monitoring virtual machine and the docker containers, are considered untrusted in our security model.

Compared to many previous solutions, we decrease the total attack surface. Specifically, our security model addresses two of the most important threats:

1. Malicious, unauthorized access to virtual machines
2. Accidental interference with the infrastructure

For the first threat, it is paramount to ensure that introspection access to the monitored virtual machine is only possible from the dedicated analysis environment of that instance or through remote access. That is to say, it must be clear that no other instance – both from this and other users – can gain access to the introspection API.

Access to the privileged instance is only possible for the authorized user via SSH. The tenants can configure and tailor this environment to their needs. However, he must not be able to influence the permissions to access other virtual machines. Therefore, every introspected virtual machine requires a dedicated introspection environment. Finally, unauthorized access to the privileged instance will only harm the monitored virtual machine. Leveraging VMI, the unauthorized user can do malicious activities to a monitored virtual machine but neither to the host system nor other virtual machines.

With regards to the second threat, we aim to avoid negative effects on the infrastructure. In particular, many previous solutions suffer from availability issues when users of the system accidently deploy incorrect code. Depending on where the introspection program is hosted, the defective code can have far-reaching consequences such as crashing the host or blocking the introspection API for other users. Our architecture does not require trust in the correctness of the deployed code, and its negative effects are limited to the monitored virtual machine and its dedicated introspection environment.

4.3. Architecture

Based on the necessary requirements, we introduce our *KVMIveggur* architecture for self-service VMI clouds. Fig. 1 visualizes its components, their relationships, and the use cases of our introspection solution.

In our design, the hypervisor exposes the VMI capabilities in the form of a UNIX domain socket (UDS). Through controlling the access to this socket, we enforce the access control scheme according to our security model (see Section 4.2), effectively enabling secure introspection for self-service clouds. We provide the user with the flexibility to tailor the introspection environment to their requirements and preferences through our three flavors of isolation:

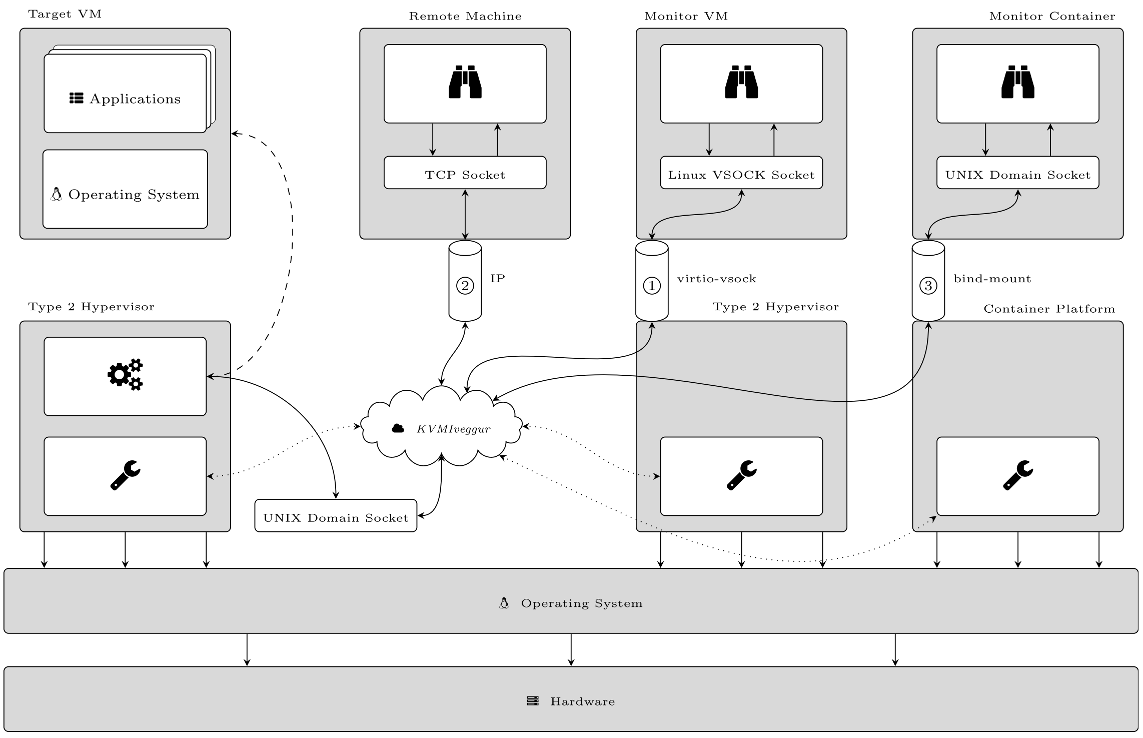

Fig. 1. *KVMIveggur* 's components that are responsible for managing the life cycles of the VM and containers.

① Leveraging *virtio-vsock* (Hajnoczi, 2020, 2021), we transform the UNIX domain socket on the hypervisor into a Linux VSOCK socket. This socket is accessible for the monitoring library inside the monitor virtual machine.
② Using *socat*, we can enact a relay between the UDS and TCP sockets. Through this relay, we can share the introspection capabilities with other machines over the network. Such a machine can then transform it back into a UDS and apply common VMI tooling.
③ Utilizing Docker's *bind-mounts*, we can mount the UNIX domain socket inside the docker container, where the monitoring library may access the socket directly without any levels of indirection.

All of these flavors are easy to integrate and deploy for cloud providers since they rely on technologies already present in many cloud infrastructures and aim to minimize necessary software modifications. Furthermore, an option ①can easily be implemented with support for live migration, as we will see in the next section. During live migration with *KVMIveggur*, both the monitoring and monitored virtual machine is being migrated to another host without interrupting the introspection. These live migration capabilities also aid in simplifying the integration for cloud providers since it allows them to offer introspection-based services on dedicated hosts.

The common denominator of the three flavors is that instead of abstracting away from the interface to the hypervisor, they offer direct access to the native VMI interface. Therefore, applications for *KVMIveggur* are generally much more efficient and easier to port compared to previous introspection architectures.

For the version that is used in this paper, we do not add any security measurement for the data transmission over the network. We assume that the remote machine is also located inside the private network, and the remote machine can only be accessed by the respective owner.

5. Implementation

After introducing the design of our *KVMIveggur* architecture, we elaborate on our implementation for the *OpenNebula* (Milojičić et al., 2011) cloud management tool. Our implementation demonstrates that the approach is practical for public cloud providers. In addition, we provide insights into how the three flavors of isolation were enacted in our system. Our implementation's repository: https://github.com/smartvmi/KVMIveggur.

5.1. On hardware- and paravirtualized machines

To facilitate introspection between two virtual machines, we reuse existing paravirtualized drivers as outlined earlier to reduce obstacles in the way of widespread adoption. We utilize the *VSOCK* protocol to enable the communication between the monitoring virtual machine and the hypervisor.

We begin by deploying a monitoring virtual machine that has *virtio-vsock* enabled:

```
<vsock model="virtio">
<cid auto="no" address="{monitoring cid}"/>
<alias name="{vsock name}"/>
<address type="pci" domain="0x0000" bus="0x00" slot="0
    ↪ x0b" function="0x0"/>
</vsock>
```

Then, we configure the target virtual machine for *vsock*:

```
<qemu:commandline>
<qemu:arg value="-chardev"/>
<qemu:arg value="socket,cid={monitoring cid},port={vsock
port},id=chardev0,reconnect=10"/>
<qemu:arg value="-object"/>
<qemu:arg value="introspection,id=kvmi,chardev=chardev0"
    ↪ />
</qemu:commandline>
```

This setup facilitates a direct bi-directional connection between the introspection tool and the hypervisor.

Furthermore, we enable live migration for such systems by following an approach similar to the *TwinPorter* architecture (Taubmann et al., 2019). However, instead of using a custom driver to notify the monitoring tool of a state transition, we leverage the existing *vsock* driver, which enhances portability and aids in alleviating security concerns. To transfer the actual machines, *KVMIveggur* employs the preexisting live migration protocol in *QEMU*, which eases integration into current cloud infrastructures and tools that build upon it.

5.2. Over the network

We leverage *socat* to forward the monitoring socket, which originally is available only on the hypervisor, to the outside. This forwarding allows us to keep the overhead to a minimum for the other flavors that do not require most of TCP's properties that are specific to network communication.

First, we transform the File-based socket using *socat* into a TCP stream where we also specify the client's IP address:

```
$ socat UNIX-LISTEN:{target VM's UDS location},\
unlink-early,fork TCP:{client's IP address}:\
{target port},fork,end-close
```

Next, on the client side, we transform back the TCP stream into a file-based socket:

```
$ socat TCP-LISTEN:{target port},reuseaddr,reuseport,\
fork UNIX-CONNECT:{UDS new location}
```

This does not violate our access control scheme since we explicitly connect to the configured IP address of the client. According to our security model (see Section 4.2), we assume that the attackers do not gain control over the host machine, which holds this configuration value. Should the attackers get in control of the host, they can directly perform VMI and other operations without passing through our introspection layer.

5.3. On docker containers

We use *Docker bind-mounts* to mount the folder that contains the VMI socket. This is very performant, but it relies on the host machine's file system having a specific directory structure available.[1]

We create the Unix Domain Socket in the designated Docker container at a specific and unique location per virtual machine in the filesystem, which we also mount on the host system:

```
$ docker run <related container options> -v \
<UDS location on the host>:\
<UDS location on the container>
```

Such an approach ensures that the container may only monitor the assigned virtual machine and not the other isolated systems. A user can then connect to the container via *SSH* (Barrett et al., 2001) and perform the monitoring activities as usual.

[1] https://elixir.bootlin.com/linux/v5.14.10/source/net/unix/af_unix.c#L222.

5.4. OpenNebula integration

OpenNebula uses sets of shell scripts to deploy and shutdown a VM inside the designated node. For scenario ①and ②, we add a custom variable to the VM template which enables the owner to specify which VM can be monitored by which VM or from which IP address. For scenario ③, we set two network interfaces on the VM's template where the second interface will be disabled during deployment and assign it to the docker container. In all scenarios, we are able to extract the required information (target VM, target IP, network interfaces and *Docker* mode) from the user's context that *OpenNebula* provides inside a virtual disk alongside the VM's disk. We provide a Python script to handle the monitoring system deployment based on which mode is chosen by the user.

6. Evaluation

In the following, we evaluate the *KVMIveggur* architecture to demonstrate its effectiveness and assess the potential performance overhead that enabling secure self-service VMI has compared to direct access to privileged VMI interfaces. We also aim at providing a systematic comparison of the performance of different flavours of VMI application deployment enabled by *KVMIveggur*. For our evaluation, we employ the following configuration:

NoVMI: No VMI (baseline).

Native: Native VMI as root user on the host (no secure self-service).

Virtio: VMI application within an isolated VM co-located with the target VM, using *virtio-vsock*

NetCoHost: VMI application within an isolated VM co-located with the target VM, using network connections for VMI operations.

NetRemote: VMI application on a remote client using VMI operations over the network.

FromDocker: VMI application in a docker container that runs on the host.

6.1. Benchmark environment

We performed all measurements on an Intel Xeon E3-1230 v5 CPU at 3.40 GHz with hyper-threading enabled as the main system and an Intel Xeon E5-2609 v3 CPU at 1.90 GHz as a secondary system as a migration target. Both systems were equipped with 64 GB of main memory. The operating system was Debian 11 “Bullseye” running Linux kernel 5.4.24. We employed the patch set proposed by *KVMi* (Bitdefender, 2019) and disabled support for huge pages, because they are incompatible with the current *KVMi* implementation. For all other kernel configurations, we adopted the default settings of Debian.

All virtual machines used in the experiment were equipped with one vCPU and 768 MB of RAM. The virtual machines ran Debian 10 with the unmodified, default Linux kernel in version 4.19. They were virtualized using QEMU 4.2.1 and utilizing the processor extensions for hardware-assisted virtualization using KVM.

In addition, we used another system (remote client) that was connected to the hypervisor over Ethernet, when assessing the performance of our approach over the network. The remote client has less than 0.6 ms network latency (measured with *ping*) and 940 Mbps bandwidth (measured with *iperf*) against the hypervisor.

6.2. Application benchmarks

We aim to evaluate our solution in a broad range of applications, which we consider representative of the currently known use cases in academia and industry. Two of these applications are:

1) **Digital forensics:** One of the main areas in which introspection shines is *main memory forensics*. In our evaluation we employ *Volatility 3* (Volatility Foundation, 2009) to perform *passive* monitoring of target virtual machines. This monitoring includes the extraction of information from kernel structures such as currently active processes and network connections.
2) **Deception technology (DT):** Deception Technology (also known as honeypot) is a technique to slow down attackers or distract them from possible targets (Fraunholz et al., 2018). One way to create a DT is by creating a full-fledged system that mimics a production server, but has no production value and is closely monitored. DT also relies on the capability to stay undetected, which is suitable with VMI's design. With *KVMIveggur* on KVM, the monitoring does not require any agent and has better performance than on Xen. We deploy a modified version of *Sarracenia* (Sentanoe et al., 2018), a VMI-based SSH honeypot.

We believe that evaluating these use cases grants insight into the cross-section of VMI mechanisms, which are shared with many real-world applications. Therefore, their evaluation will also be useful to assess other use cases not explicitly stated.

6.2.1. Volatility

To facilitate the use of *Volatility 3* in our architecture, we port *vmifs* — a tool built to expose the main memory of a virtual machine through a *FUSE* file system (Kerrisk, 2018). *Volatility* can then access and analyze the state of the monitored virtual machine through the file system. We evaluated its performance by comparing the execution time of the plugin *lsmod*, which behaves like the Linux program of the same name. Each measurement was performed 50 times. Fig. 2 shows our results.

We can observe comparable performance between the approach of *Docker* containerization (*FromDocker*) and dedicated monitoring virtual machines (*Virtio*). Furthermore, we see the expected slowdown in both remote access approaches (*NetCoHost* & *NetRemote*). We argue that this can be alleviated by avoiding inefficient memory access patterns in the VMI tool itself.

6.2.2. Sarracenia

After porting the *Sarracenia* honeypot to our architecture, we reproduced two of the performance measurements that are explained by the authors of *Sarracenia* on Xen and on KVM using the *Native* approach.

Function tracing overhead: We trace the OpenSSH function *auth_password* under four scenarios: *Baseline:* without any monitoring, *Single Breakpoint:* with one breakpoint at the beginning of the call, *Double Breakpoints:* with breakpoints at the beginning and the end of the call, and *Breakpoint + retval change:* with both breakpoints and a change of the return value. Fig. 3 shows the performance result. Compared with baseline, the overhead on KVM is +4.5%, +13.7%, and +20.6, respectively, for the three scenarios with tracing. On Xen, the overhead is +18.4%, +25.5%, and +28.1%, respectively. Overall, for this use case, VMI on KVM performs better than VMI on Xen.

System performance - System-wide tracing: In this measurement, we trace the *clone* and *exit_group* system call, and the required OpenSSH's functions (that used by *Sarracenia*'s authors) with two scenarios: execute *ls* command and download a file with 2 MB size using *wget*. The results are shown on Table 1. With tracing enabled, running the *ls* command on KVM has better (+154%) than Xen (+213%). On the other hand, Xen performs better (+55%) than KVM (+74%) on the resource intensive use case (*wget* command). This can be explained by KVM requiring more interrupts for its paravirtualized drivers. However, the difference can be considered small.

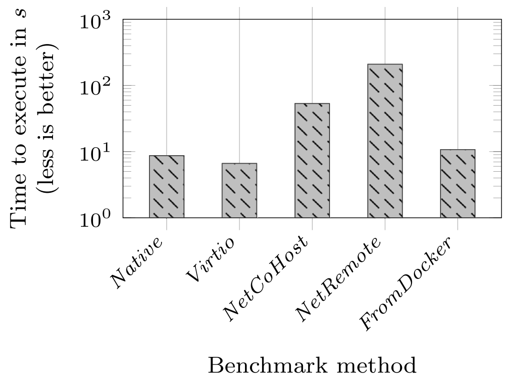

Fig. 2. Execution time of *Volatility 3* plugin *lsmod*

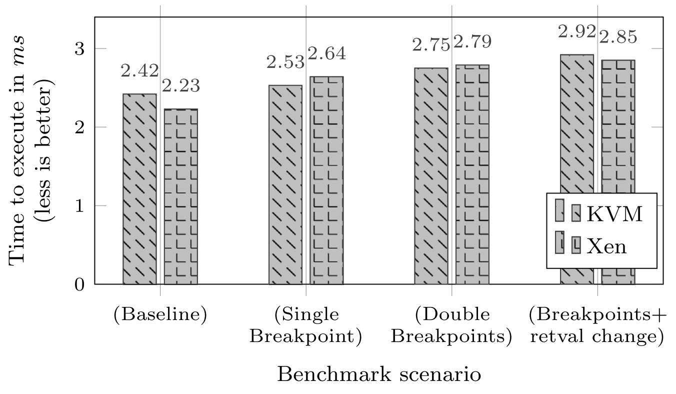

Fig. 3. *Sarracenia*'s function tracing overhead.

Table 1
Sarracenia's system performance with system-wide tracing.

	KVM		XEN	
	Baseline	With Tracing	Baseline	With Tracing
ls	0.15*s*	0.37*s*	0.13*s*	0.40*s*
wget	0.57*s*	0.99*s*	0.65*s*	0.98*s*

6.3. Synthetic benchmarks

Synthetic benchmarks provide insights on the performance impact of the different variants of secure self-service VMI deployment that KVMIveggur supports.

We use the following four benchmarks and metrics to assess the performance differences between these configurations, covering different aspects of the overall architecture:

1. **Introspection performance**: the runtime of a VMI application that extracts the process list from the monitored virtual machine using passive VMI
2. **Performance degradation of the target VM**: the runtime of a computationally heavy application in the monitored virtual machine while being monitored by active VMI
3. **Deployment time**: the time until the monitoring virtual machine and its accompanying analysis environment are ready upon the tenants' request
4. **VMI impact on live migration**: the time until the monitoring virtual machine and its accompanying analysis environment is transferred to another host using live migration

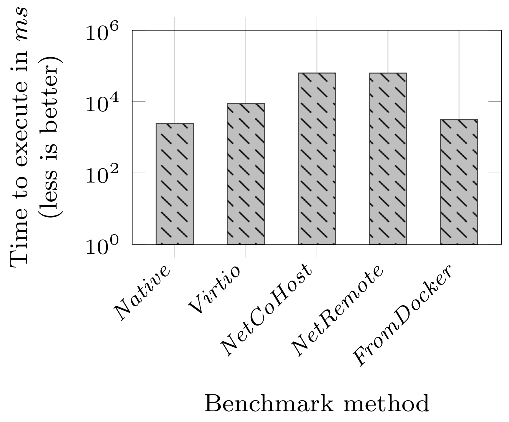

Fig. 4. Synthetic passive benchmarks.

6.3.1. Introspection performance

In this measurement, we use a simple VMI program shipped with *LibVMI* to fetch the process list of a virtual machine. This measurement will show how each method impacts the introspection performance.

By modifying the program, we exclude startup, configuration, and I/O operations from the measurement. Thereby we constrain the evaluation to include the VMI-based read operations only. We repeat these measurements 50 times, each time extracting the process list a total of 5000 times. We also ensure to flush the cache for every iteration to evaluate the memory-reading mechanism directly.

As shown by Fig. 4, method *FromDocker* has clearly the lowest overhead of all secure self-service variants, being only 30% slower than the baseline *Native*. Methods *NetCoHost* and *NetRemote* (over the network) suffer from considerably high overhead (more than 2400%), and method *Virtio* has 263% overhead. From these numbers, *FromDocker* is proven to be the best solution, while network approaches are unfavourable.

6.3.2. Performance degradation of the target VM

Here we assess the performance impact inside the target virtual machine with regards to active introspection mechanisms. Such mechanisms rely on interception points at which the execution of the monitored machine halts to facilitate the analysis at specific locations in the control flow. We are especially concerned about two mechanisms due to their widespread use. These mechanisms are *hyper breakpoints* and the monitoring of writes to the virtualized *control registers*. Specifically, we aim to concentrate our measurements on the monitoring of system call handlers and of scheduler activity inside the virtual machine.

To determine and compare the overhead between the different environments in the first case, we place a *hyper breakpoint* on the system call handler of *getpid*, which is then repeatedly invoked from a user-mode process. The execution duration of this program is then measured and reported in Fig. 5. For the second case, we configure a *VMEXIT* on write access to the *CR3* register in the *VMCS*. Then we calculate the first 1.5 million iterations of Chudnovsky's formula for the approximation of π (Chudnovsky and Chudnovsky, 2004). The time until this calculation completes is measured and depicted in Fig. 5 next to the previous results. Each of the measurements was repeated 50 times, except for *NetRemote* with system call monitoring, which we measured only five times due to the exorbitant overhead of active mechanisms over the network.

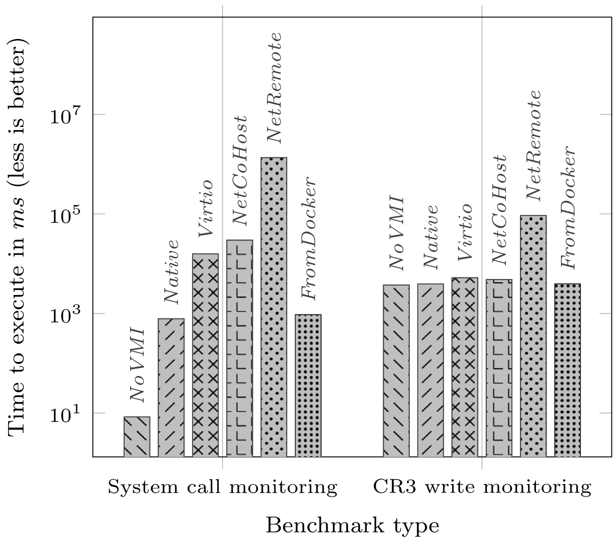

Fig. 5. Consolidated synthetic active benchmarks.

We observe that deploying active introspection mechanisms over the network inflicts a high overhead on the introspected machine. When accounting for latency, this overhead likely rules out its use in any realistic real-world scenario. The remaining environments appear to be well suited for most applications. Across both measurements, the *Docker* environment (*FromDocker*) is comparable to the baseline case of executing the introspection application on the host machine itself (*Native*). The remaining environments incur further performance degradation of more than one order of magnitude. However, depending on the targeted use case, this could still be considered acceptable (see Section 6.4).

6.3.3. Deployment time

We also evaluate the deployment of the monitoring mechanism by observing the start of the analysis environment and its successful connection between the introspection application to the hypervisor. To this end, we inspect the *systemd* journal using *systemd-notify* to extract the timestamps at which these events occurred. Thereby we gain insights into the overhead of instantiating the different solutions. From this, we can judge the suitability for domains where instances require quick deployment, e.g., scalable systems. We repeated each measurement 50 times. We show the obtained results in Fig. 6.

We only address options *Virtio* and *FromDocker* for this part of the evaluation as *Native* does not require a dedicated analysis environment. In the case of options *NetCoHost* and *NetRemote*, the

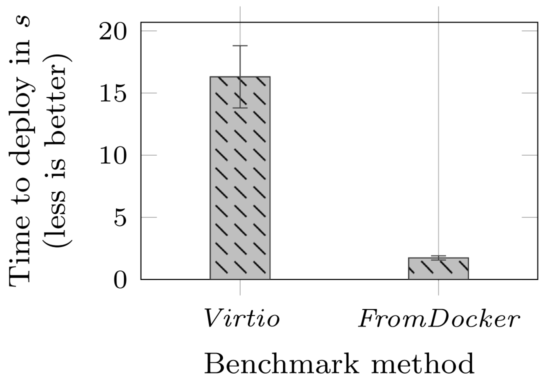

Fig. 6. Deployment of analysis environment.

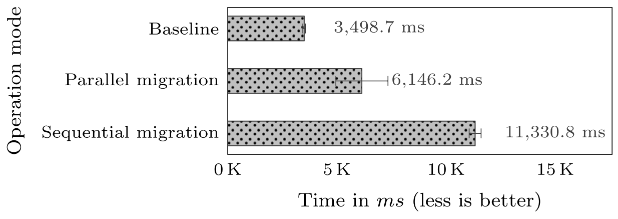

Fig. 7. Live migration of VMs between two distinct hosts.

tenant hosts said environment outside of the infrastructure of the cloud provider, which renders a potential evaluation meaningless. We observe that the *Docker* environment (*FromDocker*) significantly outperforms the *VM/vsock* environment (*Virtio*) by 841%. Regardless, we consider both deployment times acceptable for most domains as the analysis environment usually only needs to be deployed once.

6.3.4. VMI impact on live migration

As described in Section 5.1, our introspection architecture also supports the live migration of virtual machines. To evaluate this mechanism in terms of transfer times, we consider two modi operandi: a sequential transfer, in which we migrate the monitoring virtual machine first followed by the monitored virtual machine, and a parallel transfer, in which we transfer both virtual machines simultaneously. To determine the impact our system may cause, we also determine the baseline transfer times at which we can migrate a single, non-monitored machine. Fig. 7 depicts the results of these measurements with a sample size of 10.

These results appear to be in line with the expectation that both the synchronization of the introspection application for migration and the additional virtual machine contributes to the inflicted draw-down. To further improve performance at this point, we would motivate a reduction of the memory footprint of the virtual machines, especially that of the monitoring machine.

6.4. Robustness and integrity

While all three flavors of isolation can be considered secure under our security model (see Section 4.2), the degree of isolation is certainly varying. To illustrate this fact, we can look at the potential attack surface of an intruder in excess of the introspection API: The monitoring *Docker* container (③) exposes, in essence, the entire system call interface of the host operating system, whereas the dedicated monitoring virtual machine (①) only exposes the minimal interface of the hypervisor. We consider the remote access (②) to be even more isolated since it is not even co-located with any other virtual machine or entity in the cloud. Therefore, its attack surface consists only of the relay tool (*socat*), which is considerably

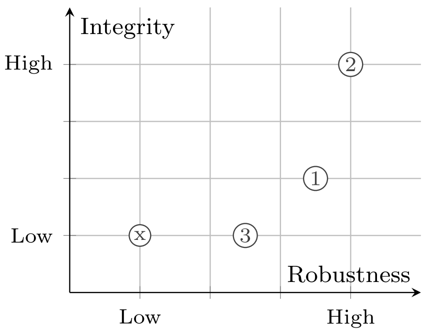

Fig. 8. Relationship between robustness and integrity.

smaller than even the interface of the hypervisor. Of course, one needs to be aware that the decision on this axis involves a risk-reward trade-off: As demonstrated by our evaluation, the higher isolation of the environment is acquired through higher performance degradation and lower introspection performance.

On the other hand, it can be beneficial for some applications, e.g., continuous monitoring, to guarantee the integrity of the analysis environment. Note that our security model typically assumes the cloud providers and their facilities to be trusted. However, discussing the limits of our architecture in this context appears sensible. Attackers that gain access to the host system and the hypervisor have control over the entire architecture and can manipulate any co-located (with the cloud infrastructure) analysis environment to their liking. For example, they can delete or falsify persistent data, e.g., logs that reveal how they managed to gain access. However, they cannot influence the records that were generated through remote access and are stored off-site.

Therefore, we can represent the introduced flavors on a two-dimensional grid where we give robustness on one axis and integrity on the other. Fig. 8 visualizes this relationship whereby robustness is negatively correlated with performance. We consider native VMI without secure-self service (represented with x) as low on both robustness and integrity since it is neither isolated from the host nor does it protect the integrity of analysis results.

7. Conclusion

This paper introduced *KVMIveggur*, an architecture for flexible, secure and efficient self-service virtual machine introspection. It incorporates three distinct methods of achieving access control and isolation: *Docker* containers, dedicated monitoring virtual machines, and remote access over the network. We show that our architecture easily integrates into current cloud platforms and confines security risks by leveraging access facilities of existing paravirtualized device drivers and containerization. To this end, integrated our approach on the publicly available cloud management tool *OpenNebula*.

Furthermore, we have ported existing VMI-based applications such as *vmifs* and the *Sarracenia* honeypot to *KVMIveggur*. We evaluated both our and the *XenAccess* architecture across introspection mechanisms, i.e., passive and active, across application domains, e.g., digital forensics and deception technology, and across our three flavors of isolation. In doing so, we empirically demonstrated the practical applicability of our solution and provided insights into the associated costs and trade-offs involved in offering self-service VMI clouds. Our results indicate that the *Docker* containerization yields superior performance over the other approaches, whereas remote access and the dedicated monitoring virtual machines shine through a high degree of isolation.

Acknowledgement

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — project 361891819 (ARADIA) and the Bundesministerium für Bildung und Forschung (BMBF, German Federal Ministry of Education and Research) — project 01IS21063A-C (SmartVMI).

References

Baek, H.w., Srivastava, A., Merwe, J.V.d., 2014. CloudVMI: virtual machine introspection as a cloud service. In: 2014 IEEE International Conference on Cloud Engineering, pp. 153–158. https://doi.org/10.1109/IC2E.2014.82.
Barrett, D.J., Barrett, D.J., Silverman, R.E., Silverman, R., 2001. SSH, the Secure Shell: the Definitive Guide. O'Reilly Media, Inc.
Bauman, E., Ayoade, G., Lin, Z., 2015. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. ACM Comput. Surv. 48. https://doi.org/10.1145/2775111.
Bitdefender, 2019. Advanced VMI on KVM: a progress report. https://static.sched.com/hosted_files/kvmforum2019/f6/Advanced%20VMI%20on%20KVM%3A%20A%20progress%20Report.pdf. (Accessed 3 May 2021).
Bitdefender, 2020. Libkvmi – KVMi virtual machine introspection library. https://github.com/bitdefender/libkvmi. (Accessed 3 May 2021).
Bushouse, M., Reeves, D., 2018a. Furnace: self-service tenant VMI for the cloud. In: Research in Attacks, Intrusions, and Defenses. Springer International Publishing, pp. 647–669.
Bushouse, M., Reeves, D., 2018b. Hyperagents: migrating host agents to the hypervisor. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. Association for Computing Machinery, New York, NY, USA, pp. 212–223. https://doi.org/10.1145/3176258.3176317.
Chen, P.M., Noble, B.D., 2001. When virtual is better than real [operating system relocation to virtual machines]. In: Proceedings Eighth Workshop on Hot Topics in Operating Systems. IEEE, pp. 133–138.
Chudnovsky, D., Chudnovsky, G., 2004. Approximations and complex multiplication according to Ramanujan. In: Pi: A Source Book. Springer, pp. 596–622.
Docker Inc, 2013. Docker. https://www.docker.com/. (Accessed 20 January 2022).
Dolan-Gavitt, B., Payne, B., Lee, W., 2011. Leveraging Forensic Tools for Virtual Machine Introspection. Georgia Institute of Technology. Technical Report.
Dykstra, J., Sherman, A.T., 2013. Design and implementation of FROST: digital forensic tools for the OpenStack cloud computing platform. Digit. Invest. 10, S87–S95. https://doi.org/10.1016/j.diin.2013.06.010 (the Proceedings of the Thirteenth Annual DFRWS Conference).
Fraunholz, D., Anton, S.D., Lipps, C., Reti, D., Krohmer, D., Pohl, F., Tammen, M., Schotten, H.D., 2018. Demystifying Deception Technology: A Survey arXiv preprint arXiv:1804.06196.
Garfinkel, T., Rosenblum, M., 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. NDSS, pp. 191–206.
Gartner Inc., 2020. Gartner forecasts worldwide public cloud end-user spending to grow 18% in 2021. https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021. (Accessed 30 March 2022).
Hajnoczi, S., 2020. virtio-vsock. http://stefanha.github.io/virtio/. (Accessed 20 January 2022).
Hajnoczi, S., 2021. Features/virtiovsock. https://wiki.qemu.org/Features/VirtioVsock. (Accessed 20 January 2022).
Hebbal, Y., Laniepce, S., Menaud, J.M., 2015. Virtual machine introspection: techniques and applications. In: 2015 10th International Conference on Availability. Reliability and Security, pp. 676–685. https://doi.org/10.1109/ARES.2015.43.
Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R., 2014. SoK: introspections on trust and the semantic gap. In: 2014 IEEE Symposium on Security and Privacy, pp. 605–620. https://doi.org/10.1109/SP.2014.45.
Kaspersky Lab, 2020. Kaspersky security - guest introspection. https://support.kaspersky.com/KSV/4.1/en-US/71371.htm. (Accessed 15 July 2021).
Kerrisk, M., 2018. Fuse - filesystem in userspace (fuse) device. In: Linux Programmer's Manual. https://man7.org/linux/man-pages/man4/fuse.4.html. (Accessed 21 March 2022).
Laurén, S., Leppänen, V., 2018. Virtual machine introspection based cloud monitoring platform. In: Proceedings of the 19th International Conference on Computer Systems and Technologies, pp. 104–109.
Lazăr, A., 2020. KVMi subsystem v10 for KVM. KVM mailing list. https://lore.kernel.org/kvm/20201125093600.2766-1-alazar@bitdefender.com/. (Accessed 20 January 2022).
Lengyel, T.K., 2020. LibVMI: simplified virtual machine introspection. https://github.com/libvmi/libvmi. (Accessed 20 January 2022).
Milojičić, D., Llorente, I.M., Montero, R.S., 2011. OpenNebula: a cloud management tool. IEEE Internet Computing 15, 11–14.
Payne, B.D., Carbone, M., Sharif, M., Lee, W., 2008. Lares: an architecture for secure active monitoring using virtualization. In: 2008 IEEE Symposium on Security and Privacy (Sp 2008). IEEE, pp. 233–247.
Sentanoe, S., Taubmann, B., Reiser, H.P., 2018. Sarracenia: enhancing the performance and stealthiness of SSH honeypots using virtual machine introspection. In: Nordic Conference on Secure IT Systems. Springer, pp. 255–271.
Taubmann, B., Böhm, A., Reiser, H.P., 2019. TwinPorter – an architecture for enabling the live migration of VMI-based monitored virtual machines. In: 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering. TrustCom/BigDataSE), pp. 427–434. https://doi.org/10.1109/TrustCom/BigDataSE.2019.00064.
Taubmann, B., Rakotondravony, N., Reiser, H.P., 2016. CloudPhylactor: harnessing mandatory access control for virtual machine introspection in cloud data centers. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 957–964. https://doi.org/10.1109/TrustCom.2016.0162.
VMware Inc., 2020. VMWare NSX guest introspection. https://docs.vmware.com/en/VMware-NSX-Data-Center-for-vSphere/6.4/com.vmware.nsx.admin.doc/GUID-049EF8ED-224C-4CAF-B6E7-1CD063CCD462.html. (Accessed 20 January 2022).
Volatility Foundation, 2009. Volatility 3: the volatile memory extraction framework. https://github.com/volatilityfoundation/volatility3. (Accessed 20 January 2022).
Win, T.Y., Tianfield, H., Mair, Q., 2014. Virtualization security combining mandatory access control and virtual machine introspection. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 1004–1009. https://doi.org/10.1109/UCC.2014.165.