CHALMERS
UNIVERSITY OF TECHNOLOGY

Automated Virtualization in Digital Forensic and Penetration Testing Work

Bachelor's Thesis in Computer Science and Engineering

ERIC ANDERSSON

Department Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY, UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Degree Project Report

Automated Virtualization in Digital Forensic and Penetration Testing Work

ERIC ANDERSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Automated Virtualization in Digital Forensic and Penetration Testing Work
ERIC ANDERSSON

Examiner:
Jonas Duregård, Department of Computer Science and Engineering

Supervisors:
KVS Prasad, Department of Computer Science and Engineering
Noor Christensen, SecureLink

Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31 772 1000

Department of Computer Science and Engineering
Gothenburg 2019

Automated Virtualization in Digital Forensic and Penetration Testing Work
ERIC ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Virtualization technology has in recent years gained significant popularity in the information technology industry, and despite its widespread use all areas of application have not yet been discovered. This thesis is done by request of the company SecureLink where they want to build a centralized system for the automatic creation and management of standardized virtual machines used in digital forensic and penetration testing work. The aim of this work has been to, based on a given specification, assemble and demonstrate a virtualization software suite for use in this system. The produced solution is referred to as the virtualization stack and uses KVM/QEMU as the hypervisor (the software that creates and runs virtual machines), libvirt to configure the virtual machines, and Vagrant to manage entire virtual environments using single commands. As part of the work virtual machine templates suitable for both digital forensic work and penetration testing have been developed, and workflow automation examples that use the virtualization stack to perform example assignments have been created. The solution is shown to be scalable and modular while allowing a high degree of automation. The presented solution can either be used in its current state or implemented into a larger program that adds additional functionality. The final product meets all the given system specifications except for those relating to standards in digital forensic investigations. Suggestions for further work is to build a front-end used to generate virtual environments according to specifications made by the user instead of using static configuration files, and to add features that meet more of the standards required in digital forensic work.

Keywords: Virtualization, hypervisor, virtual machines, digital forensics, penetration testing.

Automated Virtualization in Digital Forensic and Penetration Testing Work
ERIC ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Sammanfattning

Virtualiseringsteknologi har under de senaste åren vunnit stor mark inom IT-industrin, men trots dess många användningsområden har inte alla tillämpningar funnits än. Detta examensarbete är gjort för företaget SecureLink där de vill bygga ett centraliserat system för automatisk uppstart och hantering av standardiserade virtuella maskiner som används för arbete inom IT-forensik och penetrationstestning. Syftet med detta arbete har varit att, utifrån en given specifikation, sätta ihop och demonstrera en uppsättning av virtualiseringsprogramvara som ska användas i systemet. Den producerade lösningen hänvisas till som en virtualiseringsstack och använder KVM/QEMU som hypervisor (mjukvaran som skapar och kör virtuella maskiner), libvirt för att konfigurera de virtuella maskinerna, och Vagrant för att hantera hela virtuella miljöer via enstaka kommandon. Som del av arbetet har mallar för virtuella maskiner lämpliga för forensikarbete och penetrationstestning framtagits, och exempel på automatisering av arbetsflöden där virtualiseringsstacken används för att lösa demonstrationsuppgifter utvecklats. Lösningen visas vara skalbar och modulär samtidigt som den tillåter en hög grad av automatisering. Den presenterade lösningen can antingen användas i sitt nuvarande tillstånd eller implementeras i en större mjukvarulösning med ytterligare funktionalitet. Slutprodukten möter alla givna systemspecifikationer utom de som är relaterade till gällande regelverk inom forensiskt utredningsarbete. Förslag till fortsatt arbete är att bygga en front-end som används för att generera virtuella miljöer utifrån användarens krav istället för att använda statiska konfigurationsfiler, och att lägga till funktioner som möter fler av kraven på IT-forensiskt arbete.

Nyckelord: Virtualisering, hypervisor, virtuella maskiner, IT-forensik, penetrationstestning.

Acknowledgements

I would like to thank my advisors Noor Christensen from SecureLink and K.V.S. Prasad from Chalmers for their valuable help and feedback during the project. I want to reach out to and thank my other colleagues at SecureLink as well.

I would also like to thank my family, friends and lovely girlfriend for supporting me throughout the entirety my studies and the writing of this thesis.

Last but not least I want to thank all the talented contributors of the open-source projects that made the solution presented in this work possible.

Eric Andersson, Gothenburg, May 2019

Contents

List of Figures

1
Introduction

This chapter introduces the reader to the background and aim of this project. A background, problem statement with goals and delimitations, and an overview of the thesis structure are presented.

1.1 Background

Virtualization technology has in recent years gained significant popularity in the information technology industry through the delivery of dramatic cost savings, operational efficiency and flexibility in areas such as server consolidation, cloud services and data centers [1]. There are multitudes of areas that may in some way benefit from virtualization, and new use cases are constantly being found.

This thesis focuses on virtualization in digital forensic and penetration testing work where virtual machines can be utilized as easily launched and reusable lab environments. This approach has many benefits. Performing for example data extraction on a virtual copy of a hard drive instead of on the real object guarantees that crucial evidence is not accidentally modified, which could impair its usability in a court of law. A hacker that tests a denial of service attack against a virtual instance of a real production system does not actually harm the operation of any company, but has still found out whether there exists any vulnerabilities against the attack in question. It is also worth mentioning the reduced amount of hardware required since one computer can be used to run multiple virtual environments. This is not only cost effective both in terms of equipment expenses and energy usage, but more environmentally sustainable as well.

The work is being done at the request of SecureLink, an IT consulting company specializing in security. SecureLink wants to build a centralized system for automatically creating and running standardized virtual machines used in digital forensic investigative work and penetration testing. This system is going to consist of a server running a suitable hypervisor and software for the creation of virtual machines. The intended workflow is that an employee should be able to connect to the server and start a new virtual machine for a certain project. The employee can then perform the work, gather any results and delete the virtual machine when they are finished. The process should be automated to as large extent as possible and allow employees to quickly start virtual machines based on existing templates for different types of assignments. The system will offload SecureLink's employees by automating repeti-

tive and time consuming tasks that have previously been done manually. It will also make cooperation and data exchange between colleagues simpler, as the virtual machines run on a central entity reachable by everyone. Figure 1.1 shows an overview of the centralized system and demonstrates the planned workflow.

1.2 Problem statement

The aim of this thesis project is to produce a solution that automates the creation and lifecycle management of virtual machines built from existing templates. This means that the focus is on the left half of figure 1.1, that is on building the hypervisor and software suite that will be running on the server. The resulting system is a proof of concept that can either be immediately used or further expanded upon. The intention is that this work will also serve as a guideline should SecureLink choose to implement the suggested solution in a production environment.

To arrive at the anticipated outcome, a number of objectives need to be fulfilled. The fundamental pillar of the solution is the hypervisor and software combination used to create and manage virtual machines. Here a number of technologies and frameworks need to be compared if an educated decision is to be made. After this, standardized templates for virtual machines suitable for different assignments need to be created. Once these goals are completed it is possible to start evaluating the task automation potential of the resulting solution. This is done by using the system to automatically perform example assignments.

1.2.1 Specification

A list of requirements on the system together with desired features were given in a system specification by SecureLink.

1. The solution should work according to the process described in section 1.1 and figure 1.1.
   - Automatically create and manage virtual machines.
   - Run multiple virtual machines simultaneously.
   - Virtual machines should be created from a set of base templates. It should be possible to add new and modify existing templates.
   - Users should be able to remotely connect to the virtual machines.
2. The system should be scalable and modular, allowing it to be built on further should the need arise in the future. As a consequence it is desirable that the solution be based on already existing software to as large extent as possible. Use tried and tested alternatives over homemade solutions.
3. Open-source software is favored over proprietary alternatives.
4. A Debian or Red Hat based Linux distribution is preferred for the environment where the solution is to be run.
5. The system should function according to current standards in digital forensic investigative work (described further in section 2.1).

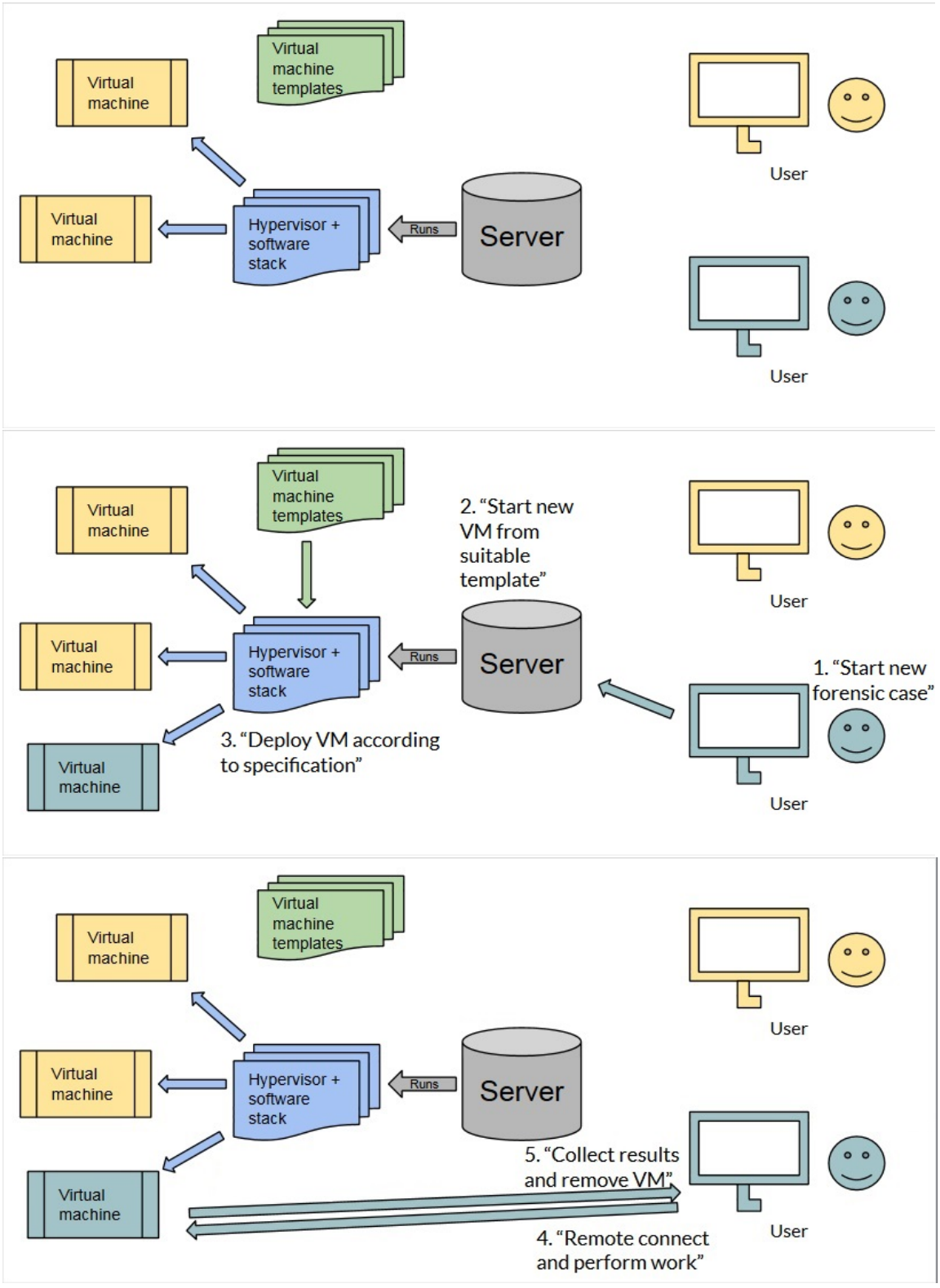

Figure 1.1: The system consists of a server running a suitable hypervisor and software suite for creating and managing virtual machines. A user that wants to perform work connects to the server and launches a virtual machine based on a fitting template. The user performs the work remotely and gathers any results before deleting the virtual machine. When the machine is deleted, the system goes back to the initial state.

6. It should be possible to execute chains of virtual machines where for example a second virtual machine depends on work performed by the first one.

1.2.2 Delimitations

As the aim of this thesis is to build the system for automatic deployment and lifecycle management of virtual machines used in digital forensic and penetration testing work, the server that will be running the solution is not dealt with. This means that the fact that the system will be running on a server is taken into consideration, but no effort is put towards choosing, configuring or setting up that server. No time will be spent on performance tuning of the virtual machines that are used to demonstrate the system. While performance is an important factor that will be considered when deciding on the hypervisor and software combination used, the focus will be on demonstrating the automation potential of the system.

The main focus will be on solving the task of launching virtual machines effectively and with minimal interaction from the user. As such, no time will be spent on building a user interface for the solution that is intended to be used in the final product. If the entire workflow can be performed using a single terminal command, it is seen as an advantage.

1.3 Thesis structure

After the introduction this report is structured as follows: The technical background (chapter 2) gives the reader a basic introduction on digital forensics, penetration testing and virtualization.

The virtualization stack (chapter 3) describes the hypervisor and software solution used to create and run virtual machines in the system. Section 3.1 explains the reasoning used when selecting a hypervisor for the virtualization stack, while sections 3.2-3.4 describe every component of the stack more closely. Section 3.5 provides a proof of concept that demonstrates the solution in action.

Virtual machine template development (chapter 4) describes the process and reasoning used when creating base virtual machine images for use with the virtualization stack. Workflow automation (chapter 5) shows how features of the virtualization stack are used to automate a sample assignment using both one and multiple virtual machines.

In results (chapter 6), the final product is evaluated against the specification given in the introduction. Discussion and conclusions (chapter 7) evaluates the overall project outcome while also giving suggestions for potential further work.

The appendices list the full source code for all files used as examples in this thesis (appendix A), the runtime output from all the examples (appendix B) and finally the methodologies used during the project (appendix C).

2
Technical background

This section aims to give a basic introduction on subjects central to this work. The first two sections describe the fundamentals of digital forensics and penetration testing in order to give the reader an understanding of the functionality required in the system. After that an introduction on virtualization is given.

2.1 IT Forensics

Digital forensic science can be defined as quoted from Gary Palmer at the Digital Forensic Research Workshop [2]:

> "The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations."

As mentioned in the introduction, virtual machines can be utilized as convenient and reusable lab environments for digital forensic work. When handling evidence that needs to be usable in a court of law it is crucial that the original object is not in any way altered, as this could impair its validity. When working with for example data extraction on a hard drive, it is possible to create a virtual clone of the disk on which all operations are performed instead. This guarantees that the state of the original hard drive remains unchanged. This situation is used as an example forensic assignment given by SecureLink. To connect this to the definition of digital forensic science given above, a virtual machine is in this case used in the identification and analysis phases of the work.

> A hard drive infected with malware has been received and the malicious files need to be identified. The following steps describe the work process:
>
> 1. Create a case and generate case number
> 2. Mount hard drive in read-only mode
> 3. Clone into a disk image
> 4. Create and start a virtual machine
> 5. Mount the disk image
> 6. Run a scanning tool

7. Collect the result in an output folder that belongs to the case
8. Shut down and delete the virtual machine and possible temporary storage

Automated deployment of virtual machines can be applied to make several aspects of digital forensic work more effective. The malware detection example assignment is a time-consuming task but large parts of it can potentially be made fully automatic. This same assignment is used in chapter 5 to demonstrate the automation potential of the system built.

A system used to perform digital forensic work needs to function according to SecureLink's digital forensic investigation standards that are used to define what is considered good practice. Requirements given by SecureLink are:

- Source material that is connected to the system is only to be handled in read-only mode.
- Cloned instances of source material should be verified (for example by comparing hash values of the source and clone) after reading to protect against bit errors or manipulation.
- All data should be handled as evidence and be segmented per case. Data generated during the various moments of an assignment should be written to separate storage, segmented to the same case as the source material.
- Every single event must be logged (clone, mount, data processing, data extraction, etc.), together with the user, timestamp, and elapsed time. All serial numbers and pieces of source information are saved.
- Every device must be synced against the same time source (zone, time and date). All time updates must be logged with time and where the time was taken from.
- Every data source should have its own universally unique ID.

2.2 Penetration Testing

In information technology a penetration test refers to a vulnerability assessment method in which an authorized cyber attack is performed on a system, network, piece of equipment or other facility, with the goal of identifying weaknesses and proving how vulnerable the system would be to a real attack [3]. It is a commonly used method to find and mitigate vulnerabilities in a system before attackers are able to exploit them. There exists a variety of approaches and technical tools used to perform penetration testing, but the general process can be simplified into the five following steps [4]:

1. Reconnaissance - Gathering vital preliminary information about the target system.
2. Scanning - The use of technical tools to increase knowledge of the system and find attack vectors.

3. Gaining Access - Using the information gained in the previous steps to take control of one or more devices in the system.
4. Maintaining Access - Taking the steps necessary to be able to keep control of the system and gather as much data as possible.
5. Covering Tracks - Clear any trace of the attack.

To perform these steps penetration testers use a wide variety of tools. Several operating system distributions specially tailored for penetration testing exist. These generally contain pre-installed and configured software necessary for all conceivable tasks. An example of such an operating system distribution is Kali Linux [5]. Penetration testing is a relevant area of study for automated deployment and lifecycle management of virtual machines. Virtualization can be used to set up lab instances of production environments, allowing penetration testing to be performed without interfering with any physical system. A computer with a general-purpose operating system can use virtualization to set up a virtual machine instance running Kali Linux and get access to a full penetration testing tool suite. User experience can be improved in both cases by taking advantage of the functionality offered by a system for automatic virtual machine management.

2.3 Virtualization

Virtualization refers to the process of creating virtual, or software-based representations of objects such as computer hardware platforms, servers, storage devices and network resources. Although the technology has existed since the 1960's, where it was used to logically divide mainframes and allow multiple applications to run simultaneously, the term has broadened and virtualization has gained significant popularity within the information technology industry in recent years. The possibility to simulate hardware functionality and create virtual computer systems has found uses in areas such as server consolidation, cloud services and data centers. The technology has gained huge traction as a result of the substantial cost savings, operational efficiency and flexibility it achieves. Using virtual machines it is possible to simultaneously run multiple operating systems on one physical machine and efficiently divide system resources between the instances. Since any computer architecture can be emulated, it is possible to run operating systems or software built for a certain type of architecture on other types which would normally not be able to do so. Virtual machines are isolated from the underlying hardware which provides both fault tolerance and security. The layer of abstraction offered by virtualization makes virtual machines hardware agnostic, meaning they can without difficulty be moved between host machines. Virtual machines can be saved, copied and moved like regular files [6][7].

This work focuses on hardware virtualization which refers to the creation of virtual machines that act like real computers with their own operating system. The difference is that they run entirely within the computer that performs the virtualization. This computer is often referred to as a *host*, while the virtual machines are referred

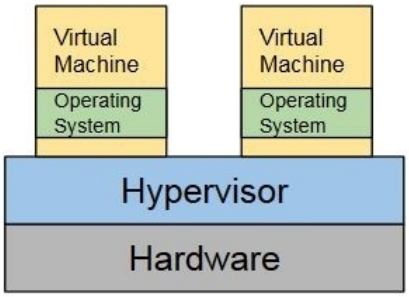

Figure 2.1: Illustration of a computer running a type-1 or native hypervisor. The hypervisor in turn runs two virtual machines with their own operating systems.

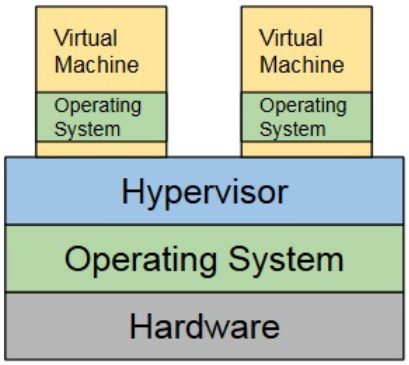

Figure 2.2: Illustration of a computer running a type-2 or hosted hypervisor. The hypervisor in turn runs two virtual machines with their own operating systems.

to as *guests*. The software that creates and runs the virtual machines is called a *hypervisor*. The hypervisor acts as a thin layer of abstraction which decouples virtual machines from the host. It is responsible for creating the virtual environment on which the virtual machines operate and dynamically allocates computing resources to each guest as needed [7]. Hypervisors are generally classified into one of two types [8]. Type-1 hypervisors or native hypervisors run directly on the host computer's hardware where it manages the guest machines. Type-2 hypervisors or hosted hypervisors run on the operating system of a computer like any other software. Figures 2.1 and 2.2 illustrate an example of each hypervisor type. The distinction between the types is not always clear. KVM (Kernel-based Virtual Machine) is an example of a hypervisor that is a Linux kernel module [9]. This means that it effectively makes the host operating system act as a type-1 hypervisor, but due to Linux distributions being general-purpose operating systems, it falls somewhere between the two definitions.

Hypervisors often use virtual machine images or templates to create and configure virtual machines. Using an object-oriented programming analogy one can say that

if a virtual machine instance is an object, then the virtual machine image is its class. Using images makes it possible to conveniently reuse virtual machines, save their state or move them between host computers. Some hypervisors can perform hardware-assisted virtualization in which the hypervisor makes use of special processor instructions created for the purpose of achieving high performance virtualization. Two examples of such virtualization processor instruction extension sets are Intel VT and AMD-V. Using hardware-assisted virtualization it is possible to run virtual machines that function at near native performance [9].

3

The virtualization stack

This chapter describes the hypervisor and software suite used to create and run virtual machines in the system. The solution consists of three components: KVM/QEMU, libvirt and Vagrant. This hypervisor and software combination is referred to as the *virtualization stack*. In the first section the reasoning used when selecting a hypervisor to use in the virtualization stack is explained. The subsequent sections describe each component of the virtualization stack more closely. The final section provides a proof of concept that demonstrates the solution in action.

To summarize, KVM/QEMU is the hypervisor pair that creates the virtual computer architecture and performs hardware-assisted virtualization. Libvirt is an API and collection of software that abstracts away the underlying hypervisors and provides a high level interface to launch and manage individual virtual machines. Vagrant is a command line utility that acts as a wrapper around libvirt, providing another layer of abstraction and an even higher level interface used to automatically manage entire groups of virtual machines. The manner in which each component provides a layer of abstraction on top of the previous one is why the solution is referred to as a virtualization stack. A depiction of the stack using the logotype of each component is shown in figure 3.1.

Figure 3.1: The software used in the virtualization stack. The levels of abstraction are symbolized with the arrow going from left to right.

3.1 Hypervisor selection

There exists a large array of hypervisors made by a variety of vendors, all with their respective strengths and weaknesses. As the hypervisor is the absolute fundamental backbone of a virtualization solution it is important to make an educated decision when choosing which one to use in the system. The chosen hypervisor needs to deliver satisfying performance while simultaneously being supported by virtualization management tools that can be used to automate the deployment and lifecycle management of virtual machines running in the system.

Hypervisors considered are listed below. The list is based on hypervisor recommendations given as a starting point by SecureLink, and findings from searching for popular alternatives online.

- Linux's Kernel-based Virtual Machine (KVM)
- Microsoft Hyper-V
- Oracle VM
- Oracle VirtualBox
- Proxmox VE
- VMWare ESXi
- Xen

Studies that compare the performance of different hypervisors have been carried out by many authors. Hwang et al. [10] perform a component-based performance comparison between Hyper-V, KVM, ESXi and Xen where performance is isolated by resource such as CPU, memory, disk and network. The level of performance isolation is also studied to measure how competing virtual machines may interfere with each other. The resulting conclusion is that the overhead incurred by each hypervisor varies significantly depending on the type of task assigned to it, and that no hypervisor always outperforms the others. W. Graniszewski and A. Arciszewski [11] compare Hyper-V, ESXi, Oracle VM, Virtualbox and Xen, measuring CPU, memory, disk and network performance using standard benchmarking tools. One conclusion is that type-1 hypervisors experience great advantages over type-2 solutions due to their direct access to system resources. ESXi is found to be the winner in performance. Leite et al. [12] conduct a performance evaluation of KVM and Xen for cloud computing usage, and KVM is found to deliver the better results. Alroobaea et al. [13] compare the performance of KVM, Xen and Proxmox VE in areas such as response efficiency, CPU cache and throughput. Memory, disk and application performance is also analyzed. The experiments show that KVM delivers the best performance on most parameters while Xen excels in file system performance and application performance. KVM is suggested to be best suited for applications which are CPU and memory intensive, while database and storage based applications will perform better using Xen.

Among the alternatives only KVM, Proxmox VE and Xen are fully free and open-source. SecureLink states in the system specification (section 1.2.1) that open-source

alternatives should be favored over proprietary counterparts. Although these hypervisors do not always outdo the proprietary alternatives, the performance difference is not dramatic enough that it is necessary to deviate from SecureLink's preferences. Another aspect that is interesting when considering open-source alternatives is the amount of support a project has within the industry. Both KVM and Xen are used in the virtualization solutions of many large vendors, one example being Google Cloud Platform which uses KVM [14]. Amazon Web Services recently abandoned Xen in favor of KVM for its new EC2 hypervisor [15].

Conclusively, the hypervisor chosen for this system is KVM. Out of the open-source options KVM is found to offer the best overall performance according to the examined studies. With its prevalence in the industry KVM is ensured to stay competitive in the future as well.

3.2 KVM/QEMU

KVM (Kernel-based Virtual Machine) is a virtualization solution that is fully integrated into the Linux kernel. It allows the kernel to act as a hypervisor and perform hardware-assisted virtualization with near native performance through the use of special processor instruction set extensions. Every virtual machine is treated as its own Linux process and the default scheduler is used for resource allocation [9][16].

KVM by itself does not perform any emulation. QEMU (Quick EMUlator) is a machine emulator that allows a host machine to emulate the processor architecture of a guest machine. This is used to run operating systems or software built for one type of architecture on another. QEMU uses KVM to perform hardware-assisted virtualization, and together the pair is often referred to as KVM/QEMU [17]. Using KVM/QEMU it is possible to run multiple virtual machines that each has its own virtualized hardware such as a network card, graphics adapter and disk. Figure 3.2 illustrates a computer using KVM/QEMU to run a virtual machine, and how KVM being a part of the Linux kernel is able to interact directly with system resources.

To summarize, QEMU emulates the processor architecture and peripheral devices while KVM performs hardware-assisted virtualization that accelerates this process. Both KVM and QEMU are fully open-source, and KVM is included in QEMU as of version 1.3 [9].

3.3 Libvirt

Libvirt is a collection of software used to conveniently manage hardware virtualization. It includes an API library, a daemon (libvirtd) and a command line tool (virsh), all of which are used to manage virtual machines and their related functionality such as lifecycle operations, storage management and network interface management. XML is used to define and represent virtual machines. A primary goal of libvirt is to provide a singular approach to manage multiple virtualization

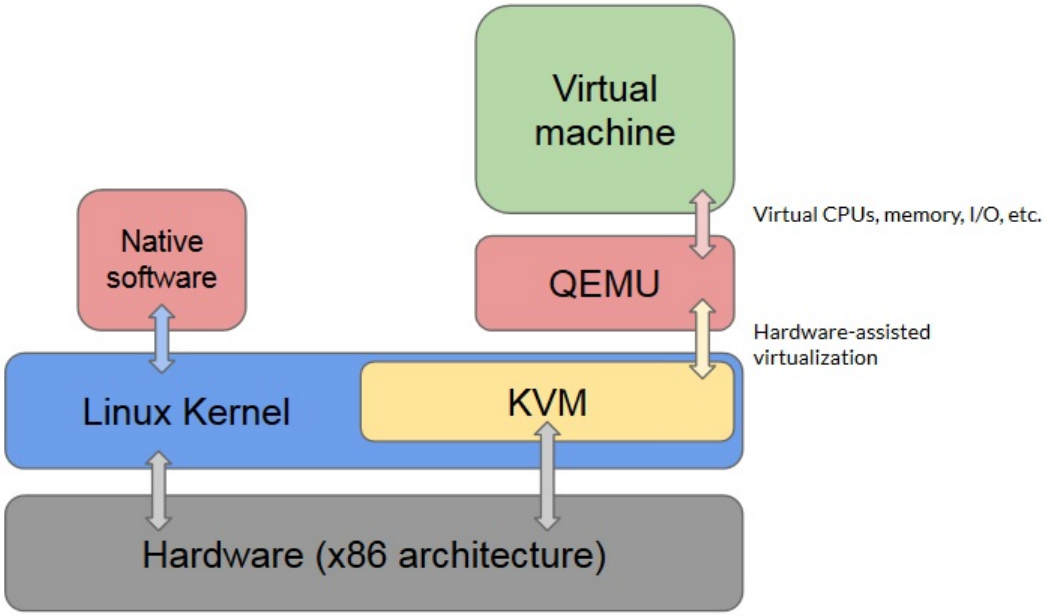

Figure 3.2: Illustration of a computer using KVM/QEMU to run a virtual machine.

providers, and as a consequence the toolkit supports a variety of hypervisors [18]. Simply stated, libvirt abstracts away the underlying hypervisor and provides a higher level interface to launch and manage virtual machines. Figure 3.3 illustrates a scenario where libvirt is used to run a virtual machine with KVM/QEMU. Libvirt simply configures the hypervisor pair.

Libvirt is open-source and developed by Red Hat, both of which it has in common with KVM. Both projects being maintained by the same organization ensures good mutual support between the tools, now and in the future, which makes the pairing of KVM and libvirt for hardware virtualization a natural choice.

3.4 Vagrant

Vagrant is an open-source command line utility for lifecycle management of virtual machines. It has a strong focus on automation, and allows users to build and manage easily configurable, portable, reproducible and disposable virtual environments using a single workflow. The virtual machines are created using existing hypervisor providers supported by Vagrant such as KVM, Virtualbox and VMWare [19]. Put simply, Vagrant acts as a wrapper around a virtualization toolkit (in this case libvirt), providing another layer of abstraction and an even higher level interface used to manage entire groups of virtual machines. Figure 3.4 illustrates a scenario where Vagrant is used to run a virtual machine using libvirt to configure KVM/QEMU.

Virtual environments created with Vagrant are represented using a Vagrantfile. The

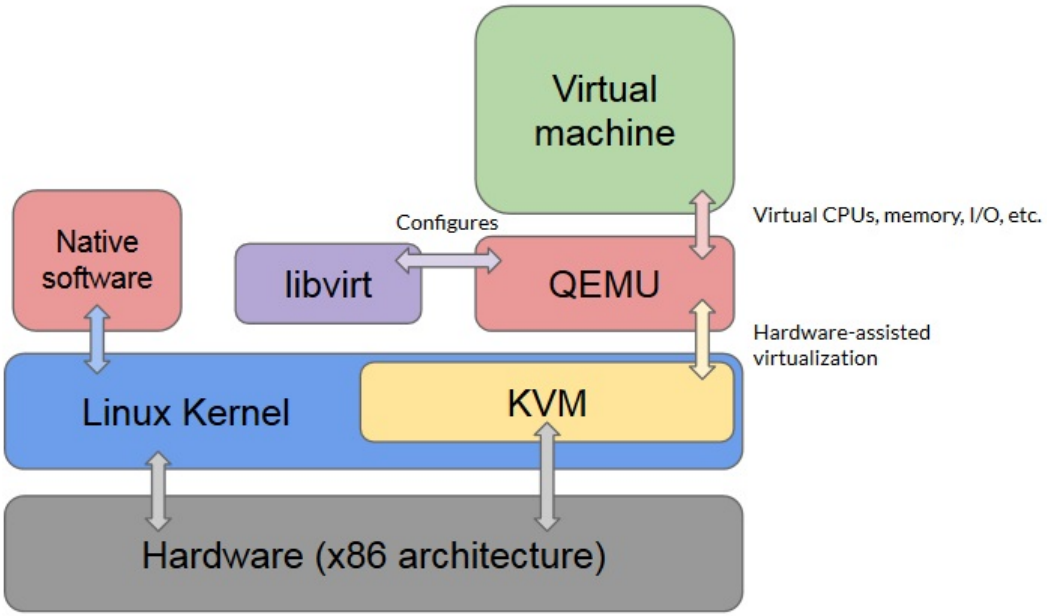

Figure 3.3: Illustration of a computer using KVM/QEMU through libvirt to run a virtual machine.

Vagrantfile defines a set of instructions that Vagrant follows while creating and running the virtual machines. It describes the type of machine used (such as what base image is used and how much virtual memory is allocated) and how the machine is run (such as setting it to execute a certain script after start-up). Vagrantfiles are written in the programming language Ruby [20]. An example of a Vagrantfile can be examined in section 3.5. There, a Vagrantfile used to demonstrate the virtualization stack is shown and the code is explained. The full source code of all Vagrantfiles used in this project can be located in appendix A.

Virtual machines deployed using Vagrant are cloned from base images referred to as boxes. This process is substantially quicker than building each virtual machine from scratch. The box used for a virtual machine is specified in the Vagrantfile for the virtual environment that machine belongs to. A box provides a clean slate starting point for virtual machines deployed using Vagrant, and modifications made to a machine cloned from a box are not reflected onto the box itself. A modified virtual machine instance can be used to create a new box that uses the modified state as a starting point for subsequent virtual machine deployments [21].

Vagrant provides a large array of functionality that can be used to tailor the behavior of a virtual environment. Examples of this are the ability to sync files and folders between host and virtual machine, or defining machine triggers that automatically run code at certain points in the Vagrant execution cycle. Instead of explaining every single available feature in this section, making it huge in the process, functionality that has not been described earlier will be explained in this report when implemented into the system. This way the reader is acquainted with the tool in an

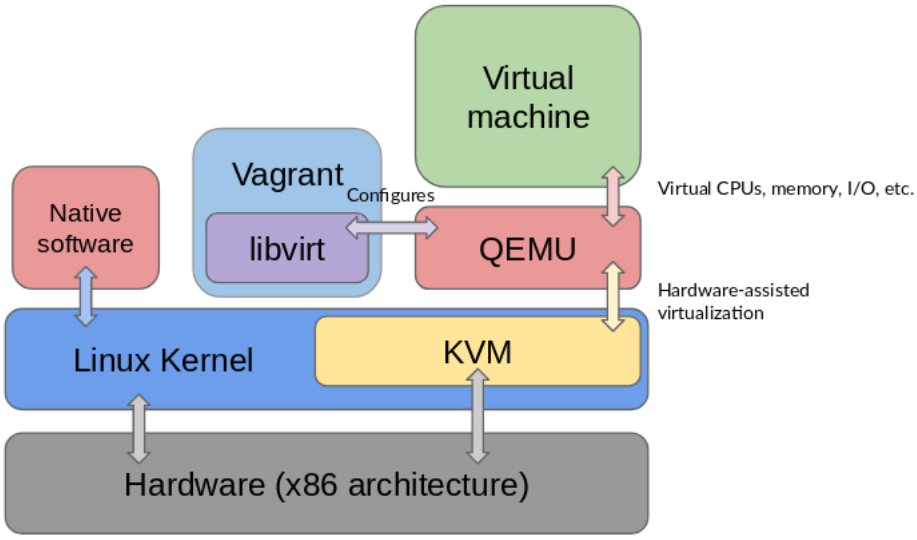

Figure 3.4: Illustration of a computer using KVM/QEMU through libvirt and Vagrant to run a virtual machine. Vagrant simply works as a wrapper around the libvirt toolkit.

easier to digest step by step manner.

Additional features can be added by installing plugins to extend Vagrant. As libvirt is not one of the default providers supported by Vagrant, the plugin Vagrant-libvirt is used to connect Vagrant to the rest of the virtualization stack. This plugin is what allows Vagrant to act as a wrapper that deploys and controls KVM/QEMU virtual machines through the libvirt toolkit [22].

3.5 Proof of concept

To showcase the potential of the KVM/QEMU + libvirt + Vagrant stack, a simple proof of concept was created. This demonstration consists of a Vagrantfile that is used to deploy a single virtual machine and perform some of the high level features offered by Vagrant. The proof of concept mainly focuses on demonstrating the level of automation that is achievable, namely how the configuration and scripts run on the virtual machine are put into action by a single command from the user.

```
1 # This Vagrantfile configures a barebone virtual machine and
  ↪ showcases some of Vagrant's
2 # high level features such as triggers and provisioners by using
  ↪ simple shell scripts with print statements.
3 Vagrant.configure("2") do |config|
4   # VM template.
5   config.vm.box = "generic/debian9"
```

```
6
7     # Trigger that runs before VM deployment.
8     config.trigger.before :up do |trigger|
9       trigger.info = "Running before up trigger."
10      trigger.run = {inline: <-- SHELL
11        echo "This is a trigger that is being run before the VM is
          ↪  created. From here I can move files into the template,
          ↪  etc. After this script, the VM is created and booted up."
12        sleep 10
13      SHELL}
14    end
15
16    # Shell provisioner that runs once VM is deployed
17    config.vm.provision :shell, inline: <-- SHELL
18      echo "Hello world! This output is from a shell script that is
        ↪  run automatically when the VM has been set up. This allows
        ↪  me to run arbitrary code on the deployed VM. I can use this
        ↪  to install software, run programs, etc. The VM can be
        ↪  accessed manually using either the graphical interface
        ↪  provided by Libvirts Virtual Machine Manager, or by using
        ↪  ssh from the command line."
19      sleep 15
20    SHELL
21
22    # Trigger that runs before VM destruction.
23    config.trigger.before :destroy do |trigger|
24      trigger.info = "Running before destroy trigger"
25      trigger.run = {inline: <-- SHELL
26        echo "This is a trigger that is being run before the VM is
          ↪  destroyed. This can be used to transfer results into an
          ↪  output folder when finished, etc. After this script, the
          ↪  VM is shut down and destroyed along with all temporary
          ↪  storage."
27        sleep 10
28      SHELL}
29    end
30  end
```

Listing 3.1: The proof of concept Vagrantfile.

The source code of the used Vagrantfile is shown in Listing 3.1. Line 5 configures what box is used as a base image for the virtual machine created when running the Vagrantfile. In this case a Debian 9.8.0 image from the official catalog of public Vagrant boxes is used.

A highly useful feature of Vagrant is triggers. Triggers allow user-defined commands to be run at certain points of the Vagrant execution cycle. Triggers can be used

to run arbitrary code on the virtual machine or the host machine, which makes it possible to automate many tasks. In this proof of concept the triggers are used to print simple statements to be read by the user. Later examples will use triggers to perform more advanced operations, such as automatically running malware scans on a virtual machine and copying the results to the host machine, or to automatically destroy a virtual machine once its task has been performed. The code in lines 8-14 of the Vagrantfile defines a trigger that is run before the virtual machine is deployed. The trigger runs a simple shell script on the host machine that prints the message:

> "This is a trigger that is being run before the VM is created. From here I can move files into the template, etc. After this script, the VM is created and booted up."

and then pauses the execution for ten seconds to give the user time to read the output. Similarly, lines 23-29 define a similar trigger that runs before the virtual machine is shut down and destroyed.

Provisioning is a feature similar to triggers with the difference that it allows the definition of commands that can be run on the virtual machine repeatedly. Provisioning is done automatically once a virtual machine has been created, and can then be repeated by the user at any time by entering "vagrant provision" on the command line. Vagrant allows provisioning to be done using multiple external tools, but in this example a simple shell script is used. Lines 17-20 of the Vagrantfile define a shell provisioner that, similarly to the earlier described triggers, prints a message to the user and pauses the execution for a couple of seconds.

To summarize, this proof of concept automatically configures and launches a Debian 9.8.0 virtual machine while using triggers and provisioning to run custom-made shell scripts on the host machine and the virtual machine at certain steps of the execution. The file is executed by entering the command "vagrant up" on the command line. The output received when running the Vagrantfile can be seen in appendix B.1.

4

Virtual machine template development

To automate virtual machine deployment, and further down the road automate the workflow of common assignments as well, base templates consisting of machine images appropriate for general digital forensic work and penetration testing are required. The templates are implemented as Vagrant boxes that can be specified in a Vagrantfile to use that specific image as a starting point for a virtual machine. Since the requirements on a template may change over time, the easily customizable Vagrant box is a fitting format. A Vagrant box for use with KVM and libvirt is a .tar file containing [22]:

- The base image file in .qcow2 format (QEMU Copy-On-Write version 2, a disk image format supported by QEMU/KVM [23]).
- Metadata in a .json file (JavaScript Object Notation, a common file format for translating readable text to attribute-value data objects [24]) describing the box image (provider, virtual size, format).
- A Vagrantfile performing default configuration of KVM/QEMU and libvirt specific settings.

The vagrant-libvirt plugin provides the shell script *create_box.sh* that packages a Vagrant box from a given .qcow2 image. The resulting .tar file can then be added to Vagrant and used to create virtual machines. One template for digital forensic work in a Linux environment was created, together with another one for penetration testing purposes. Each custom template is described more closely in their own respective section.

4.1 Linux data extraction

This template is based on a SIFT Workstation virtual machine image. The SIFT Workstation is a free open-source distribution of incident report and forensic tools designed to perform comprehensive digital forensic examinations. SIFT provides the ability to securely examine raw disks, multiple file formats and file systems. Strict guidelines are placed to ensure that evidence is exclusively examined in read-only mode, as to verify that no data is modified [25].

The SIFT Workstation virtual machine image available for download on the official website uses the file format .ova (Open Virtualization Appliance), which to be usable with KVM has to be converted to the earlier mentioned .qcow2 format. As can be observed in Listing 4.1 the .ova file is simply a .tar archive in which a .ovf (Open Virtualization Format, a descriptor file for the packaged virtual machine [26]) and .vmdk (Virtual Machine DisK, an open disk image format [27]) file can be found. The .vmdk format is supported by QEMU and can be converted to a .qcow2 image using a simple terminal script. The process is shown in Listing 4.2.

```
$ file SIFT-Workstation.ova
SIFT-Workstation.ova: POSIX tar archive
$ tar -tf SIFT-Workstation.ova
sift-update.ovf
sift-update.mf
sift-update-disk1.vmdk
sift-update-file1.iso
```

Listing 4.1: SIFT-Workstation.ova is simply a .tar archive containing files used to describe the virtual machine.

```
$ tar -xvf SIFT-Workstation.ova
sift-update.ovf
sift-update.mf
sift-update-disk1.vmdk
sift-update-file1.iso
$ qemu-img convert -O qcow2 sift-update-disk1.vmdk
 ↪  SIFT-Workstation.qcow2
$ ls
sift-update-disk1.vmdk  sift-update.mf   SIFT-Workstation.ova
sift-update-file1.iso   sift-update.ovf  'SIFT-Workstation.qcow2'
```

Listing 4.2: The .vmdk image is extracted and converted to the .qcow2 format.

The *create_box.sh* script provided by vagrant-libvirt was used to package the resulted .qcow2 file into a Vagrant box, making the SIFT Workstation virtual machine template usable with Vagrant. Listing 4.3 shows the shell session used when creating the Vagrant box from the .qcow2 file.

```
$ ./create_box.sh SIFT-Workstation.qcow2
{100}
==> Creating box, tarring and gzipping
./metadata.json
./Vagrantfile
./box.img
Total bytes written: 8985978880 (8.4GiB, 30MiB/s)
==> SIFT-Workstation.box created
==> You can now add the box:
==> 'vagrant box add SIFT-Workstation.box --name SIFT-Workstation'
```

```
$ vagrant box add SIFT-Workstation.box --name SIFT-Workstation
==> box: Box file was not detected as metadata. Adding it
 ↪  directly...
==> Adding box 'SIFT-Workstation' (v0) for provider:
        box: Unpacking necessary files from:
         ↪  file:///home/eric/ragebed/sift/SIFT-Workstation.box
==> box: Successfully added box 'SIFT-Workstation' (v0) for
 ↪  'libvirt'!
```

Listing 4.3: Shell session where a SIFT Workstation Vagrant box is created from a .qcow2 file.

4.2 Linux penetration testing

The template intended for penetration testing work is based on the latest official Kali Linux virtual machine image. Kali is a popular open-source Debian-based Linux distribution maintained by Offensive Security Ltd., which is specifically designed for penetration testing purposes [5]. As such, it comes with a large array of common vulnerability assessment and exploitation tools preinstalled.

The Kali Linux virtual machine image available for download on the official Offensive Security website uses the .ova file format. As was the case with the SIFT Workstation image built in the preceding section, this .ova file needed to be converted to the .qcow2 format to be used with KVM as hypervisor. The same process as in section 4.1 was used to package the Kali Linux virtual machine image into a Vagrant box. The shell session used when creating the box is shown in Listing 4.4 below.

```
$ tar -xvf kali-linux-2019.1-vbox-amd64.ova
Kali-Linux-2019.1-vbox-amd64.ovf
Kali-Linux-2019.1-vbox-amd64-disk001.vmdk

$ qemu-img convert -O qcow2
 ↪  Kali-Linux-2019.1-vbox-amd64-disk001.vmdk
 ↪  Kali-Linux-2019.1-vbox-amd64-disk001.qcow2

$ ./create_box.sh Kali-Linux-2019-amd64-disk001.qcow2
{80}
==> Creating box, tarring and gzipping
./metadata.json
./Vagrantfile
./box.img
Total bytes written: 11356221440 (11GiB, 31MiB/s)
==> Kali-Linux-2019-amd64-disk001.box created
==> You can now add the box:
```

```
==>   'vagrant box add Kali-Linux-2019-amd64-disk001.box --name
 ↪  Kali-Linux-2019-amd64-disk001'

$ vagrant box add Kali-Linux-2019-amd64-disk001.box --name
 ↪  Kali-Linux-2019-amd64-disk001
==> box: Box file was not detected as metadata. Adding it
 ↪  directly...
==> box: Adding box 'Kali-Linux-2019-amd64-disk001' (v0) for
 ↪  provider:
        box: Unpacking necessary files from:
         ↪  file:///home/eric/ragebed/kali/Kali-Linux-2019-amd64-disk001.box
==> box: Successfully added box 'Kali-Linux-2019-amd64-disk001'
 ↪  (v0) for 'libvirt'!
```

Listing 4.4: Shell session where a Kali Linux Vagrant box is created from a .qcow2 file.

5
Workflow automation

With the virtualization stack to launch and control virtual environments finished, and base virtual machine templates appropriate for typical intended work of the system built, it now is possible to demonstrate the automation potential of the system. This section focuses on the implementation of a number of demonstration cases where the virtualization stack is used to launch one or more virtual machines based on the developed templates, which then automatically performs an example task given by SecureLink.

These demonstration cases use plain shell scripts to achieve their tasks. This has been done for simplicity. When implementing the virtualization stack into a real system it would be a better alternative to use one of the multiple programming language bindings that exist for libvirt and Vagrant. This topic is discussed further in chapter 7.

5.1 Malware detection

This workflow automation showcase is based on the example forensic assignment in section 2.1 where an infected hard drive is scanned in order to identify and analyze the malicious files. It is assumed that the hard drive has been cloned into a disk image that is located in project folder. This showcase thus automates steps 4 to 8 in the example assignment given in section 2.1. Using one command to start the process and one command to end it, the work described in the steps below is performed:

1. Create and start a SIFT Workstation virtual machine
2. Attach and mount the disk image
3. Run a ClamAV scan on the disk
4. Write the scan result and copy any detected malware to an output folder
5. Copy the output folder to the project directory on the host machine
6. Detach the disk image
7. Shut down and delete the virtual machine

The source code of the Vagrantfile used can be seen in Listing 5.1. Longer comments have been omitted, and the shell scripts used in the triggers are explained but not listed in this section. The full source code along with all shell scripts can be examined in appendix A.2.

```
1  Vagrant.configure("2") do |config|
2    # VM template.
3    config.vm.box = "SIFT-Workstation"
4
5    # Vagrant plugins.
6    config.vagrant.plugins = ["vagrant-libvirt", "vagrant-scp"]
7
8    # Libvirt configuration.
9    config.vm.provider :libvirt do |libvirt|
10     libvirt.memory = 4096
11   end
12
13   # Trigger that runs after VM deployment.
14   config.trigger.after :up do |trigger|
15     trigger.run = {path: "attach_disk_image.sh"}
16     trigger.run_remote = {path: "scan_disk_image.sh"}
17   end
18
19   # SSH configuration.
20   config.ssh.username = 'sansforensics'
21   config.ssh.password = 'forensics'
22   config.ssh.insert_key = true
23
24   # Trigger that runs before VM destruction.
25   config.trigger.before :destroy do |trigger|
26     trigger.run = {path: "detach_disk_image_and_copy_output.sh"}
27   end
28 end
```

Listing 5.1: The malware detection Vagrantfile.

Line 3 configures the box used as a base image for the virtual machine launched when running the Vagrantfile. The box used is the SIFT Workstation template created in section 4.1, which among a large array of useful tools for digital forensic work, has the malware detection software ClamAV (Clam AntiVirus, an open-source antivirus engine [28]) preinstalled.

A list of Vagrant plugins required to run the file is stated in line 6. If a plugin in the list is not installed, Vagrant will try to automatically download it. In this case the plugins vagrant-libvirt (adds libvirt as a Vagrant provider, see 3.4) and vagrant-scp (allows Vagrant to move files between host and virtual machine using Secure Copy) are used.

The code in lines 9-11 state libvirt-specific options. Inside this block it is possible to, from within the Vagrantfile, specify lower level configuration options that affect how libvirt will create and manage the virtual machine. Examples of such options are the amount of virtual memory and cores allocated to the virtual machine, the type of

disk bus to be emulated, which hypervisor to use and more. A list of all possible options can be found in [22]. In this example only the amount of memory has been set.

Lines 14-17 define a trigger that is automatically executed once the virtual machine has been created. This trigger runs two shell scripts. The first script *attach_disk_image.sh* is run on the host machine and uses libvirt's *virsh attach-disk* command to attach a disk image located in the project folder to the virtual machine. The second script *scan_disk_image.sh* is run on the virtual machine and mounts the attached disk and then runs a ClamAV malware detection scan on it.

Lines 20-22 configure how Vagrant accesses the virtual machine using SSH (Secure Shell). This section is required as the SIFT Workstation image requires login credentials to be accessed, and Vagrant uses SSH to perform tasks such as running scripts on the virtual machine. Having passwords hard-coded into the Vagrantfile is not desirable from a security perspective, but was done in this demonstration for the sake of simplicity. The password used is the default password set on any SIFT Workstation image downloaded from the official source, and no sensitive information is stored in the base template. It would not be difficult to store passwords separately and require some form of authentication to fetch it for use by the Vagrantfile.

Finally, the code in lines 25-27 defines a trigger that is automatically executed before the virtual machine is destroyed. This trigger runs the shell script *detach_disk_image_and_copy_output.sh* on the host machine. The script uses the plugin vagrant-scp to copy the result report and any malware found during the malware detection scan to an output folder in the project directory on the host machine. Libvirt's *virsh detach-disk* command is then used to detach the disk image from the virtual machine. When the script finishes, the virtual machine is shut down and deleted.

The procedure was tested by creating a disk image on which a trojan downloaded from a collection of live malware samples (the repository can be found in [29]) was intentionally planted. The Vagrantfile was then run using that disk image. The planted malware was successfully found by the scanning tool and copied to the host machine. The output received when running the Vagrantfile along with the resulting scan report and contents of the output folder are all listed in appendix B.2.

5.2 Virtual machine chaining

In requirement #6 of the system specification (section 1.2.1) it is requested that the solution should make it possible to execute chains of virtual machines. Some assignments may require multiple virtual machines to perform a task. Let V1 and V2 be two virtual machines. V2 depends on the output of V1 to perform its work, and thus waits until V1 is finished before starting and carrying out its task. This procedure is what is referred to as virtual machine chaining.

The purpose of this section is to demonstrate that virtual machine chaining is possible using the virtualization stack. Two examples where the virtualization stack is used to deploy chains of virtual machines are shown. The first example is a barebone demonstration where two virtual machines are created and destroyed in sequence. The second example is an extension of the malware detection workflow automation showcase of section 5.1 in which a second virtual machine uses the output from the first one to analyse the detected malware.

5.2.1 Virtual machine chaining example

This example demonstrates a very simple instance of virtual machine chaining. Two virtual machines "first" and "second" are defined. When run, the example automatically starts and stops two virtual machines in sequence. By running the shell script *start.sh* the first virtual machine is created. Once it has finished it is destroyed and second is started. Second also destroys itself automatically. The purpose of this showcase is to show how multiple virtual machines with their own behavior and properties can be defined using a single Vagrantfile. The source code of the Vagrantfile used in the example is shown in Listing 5.2. Similarly to earlier examples, longer comments have been omitted. The full Vagrantfile can be examined in appendix A.3.

```
1  Vagrant.configure("2") do |config|
2    # Vagrant plugins.
3    config.vagrant.plugins = ["vagrant-libvirt"]
4
5    # First VM configuration.
6    config.vm.define "first" do |first|
7      # VM template.
8      first.vm.box = "SIFT-Workstation"
9
10     # libvirt configuration.
11     first.vm.provider :libvirt do |libvirt|
12       libvirt.memory = 4096
13     end
14
15     # SSH configuration.
16     first.ssh.username = "sansforensics"
17     first.ssh.password = "forensics"
18     first.ssh.insert_key = true
19
20     # Trigger that runs after VM deployment.
21     first.trigger.after :up do |trigger|
22       trigger.info = "Running afterup trigger"
23       trigger.info = "Destroying first"
24       trigger.run = {inline: "vagrant destroy first -f"}
25     end
26   end
```

```
27
28    # Second VM configuration.
29    config.vm.define "second", autostart: false do |second|
30      # VM template.
31      second.vm.box = "SIFT-Workstation"
32
33      # libvirt configuration.
34      second.vm.provider :libvirt do |libvirt|
35        libvirt.memory = 4096
36      end
37
38      # SSH configuration
39      second.ssh.username = "sansforensics"
40      second.ssh.password = "forensics"
41      second.ssh.insert_key = true
42
43      # Trigger that runs after VM deployment.
44      second.trigger.after :up do |trigger|
45        trigger.info = "Running afterup trigger"
46        trigger.info = "Destroying second"
47        trigger.run = {inline: "vagrant destroy second -f"}
48      end
49    end
50  end
```

Listing 5.2: The virtual machine chaining example Vagrantfile. This Vagrantfile defines two virtual machines that are executed in sequence.

The big difference compared to earlier examples is that this Vagrantfile defines two virtual machines, "first" and "second". First is configured in lines 6-26 while lines 29-49 configure second. Each virtual machine has its own configuration that decides for example which Vagrant box to use and which triggers should be run. Defining multiple virtual machines in one Vagrantfile makes it possible to execute them simultaneously or in sequence using a single command. In this case the two virtual machines are identical and do nothing else than destroying themselves once deployed. The output received when running the Vagrantfile is listed in appendix B.3.

5.2.2 Malware detection and analysis

This workflow automation showcase is an extension of the task performed in section 5.1 and uses two virtual machines to first identify malware on an infected hard drive, and then analyze the detected files. The first virtual machine performs the same steps as listed in section 5.1, while the second virtual machine uses the output from the first virtual machine to write a hex dump on the detected malware that is then copied to an output folder located in the project folder on the host machine.

Although the hex dump could simply be done by the first machine, the purpose of this showcase is to demonstrate the possibility to, using the virtualization stack, create chains of virtual machines that work together to complete a practical task.

While experimenting with virtual machine chaining a problem was encountered related to running multiple scripts inside a single trigger. Vagrant does not always execute the scripts in the same order as they are defined in the Vagrantfile. When two scripts are supposed to be run in sequence, a first one that performs some operation on the virtual machine and a second one that destroys the virtual machine, it can happen that the script that destroys the virtual machine runs first even though it is written after the first script in the trigger. It is unclear whether this is a bug in Vagrant or a quirk resulting from how the triggers are parsed. To mitigate the problem, this chaining example only runs a single script in each trigger. If more than one script needs to be run in sequence they are called from within that script.

The source code of the Vagrantfile used in the showcase is shown in Listing 5.3. Similarly to earlier examples, longer comments have been omitted and the functionality of shell scripts that are run by triggers are explained but not listed in this section. The full Vagrantfile along with all shell scripts can be examined in appendix A.4.

```
1  Vagrant.configure("2") do |config|
2    # Vagrant plugins.
3    config.vagrant.plugins = ["vagrant-libvirt", "vagrant-scp"]
4
5    # First VM configuration.
6    config.vm.define "first" do |first|
7      # VM template.
8      first.vm.box = "SIFT-Workstation"
9
10     # libvirt configuration.
11     first.vm.provider :libvirt do |libvirt|
12       libvirt.memory = 4096
13     end
14
15     # SSH configuration.
16     first.ssh.username = "sansforensics"
17     first.ssh.password = "forensics"
18     first.ssh.insert_key = true
19
20     # Trigger that runs after VM deployment.
21     first.trigger.after :up do |trigger|
22       # Attach disk image to VM, mount disk and perform malware
       ↪  scan. Destroy first.
23       trigger.run = {path: "after_up_first.sh"}
24     end
25
```

```
26      # Trigger that runs before VM destruction.
27      first.trigger.before :destroy do |trigger|
28        # Detach disk image from VM, copy output folder to host,
          ↪  deploy second.
29        trigger.run = {path: "before_destroy_first.sh"}
30      end
31    end
32
33    # Second VM configuration.
34    config.vm.define "second", autostart: false do |second|
35      # VM template.
36      second.vm.box = "SIFT-Workstation"
37
38      # libvirt configuration.
39      second.vm.provider :libvirt do |libvirt|
40        libvirt.memory = 4096
41      end
42
43      # SSH configuration.
44      second.ssh.username = "sansforensics"
45      second.ssh.password = "forensics"
46      second.ssh.insert_key = true
47
48      # Trigger that runs after VM deployment.
49      second.trigger.after :up do |trigger|
50        # SSH into second and run the script analyse_malware.sh
51        trigger.run = {path: "after_up_second.sh"}
52      end
53
54      # Trigger that runs before VM destruction.
55      second.trigger.before :destroy do |trigger|
56        # Move output files to host machine
57        trigger.run = {path: "before_destroy_second.sh"}
58      end
59    end
60  end
```

Listing 5.3: The malware detection and analysis Vagrantfile.

This Vagrantfile defines two virtual machines, "first" and "second". First is configured in lines 5-31 while lines 34-59 configure second. Much of the code is similar to that of the malware detection showcase explained in 5.1, and the reader is advised to refer to that example for the blocks that are not explained in this section.

Lines 21-24 define a trigger that is run after first is created and started up. This trigger runs the shell script *after_up_first.sh* on the host machine which first uses libvirt's *virsh attack-disk* command to attach a disk image located in the project

folder to the virtual machine. The Vagrant command *vagrant ssh* is then used to access the virtual machine and run the shell script *mount_and_scan_disk.sh*. This script mounts the attached disk and runs a ClamAV malware detection scan on it. After this, first initiates its own deletion.

The code in lines 27-30 defines a trigger that is run before first is destroyed. This trigger runs the shell script *before_destroy_first.sh* on the host machine. The script uses the plugin vagrant-scp to copy the result report and any malware found during the malware detection scan to an output folder in the project directory. Libvirt's *virsh detach-disk* command is then used to detach the disk image from the virtual machine. When the script finishes, first is shut down and deleted, and second is created and started up.

Lines 49-52 define a trigger that is run after second is created and started up. This trigger runs the shell script *after_up_second.sh* on the host machine which uses the Vagrant command *vagrant ssh* to access the virtual machine and run the shell script *analyse_malware.sh*. This script analyses the content of the output folder created by the first virtual machine, writing a hex dump of every malware file that was detected during the scan. After this, second initiates its own deletion.

Finally, the code in lines 55-58 defines a trigger that is run before second is destroyed. This trigger runs the shell script *before_destroy_second.sh* on the host machine which uses vagrant-scp to copy the hex dumps to an output folder in the project directory. When the script finishes, second is shut down and deleted.

This process was tested in the same manners as the procedure in section 5.1. The first virtual machine successfully found the malware planted on the disk image, and copied the scanning report and detected malware to the host. The second virtual machine successfully used the output from the first virtual machine to perform a hexdump on the malware which was then copied to the host as well. The output received when running the Vagrantfile along with the resulting scan report, copied malware and hexdump are all listed in appendix B.4.

6
Results

In this chapter the solutions described in The virtualization stack (chapter 3), Virtual machine template development (chapter 4) and Workflow automation (chapter 5) are presented and evaluated as one final product. Since the proof of concept presented in section 3.5 and the workflow automation examples in chapter 5 are used to demonstrate the capabilities of the virtualization stack, this chapter primarily focuses on evaluating the solution against the requirements given in the system specification of section 1.2.1.

The KVM/QEMU + libvirt + Vagrant virtualization stack is the proposed solution that can be implemented into SecureLink's centralized system for automatic creation and lifecycle management of standardized virtual machines for use in digital forensic investigative work and penetration testing. The solution makes it possible to deploy entire virtual environments using a single command. Multiple related or unrelated virtual machines are able to run simultaneously in a seamless manner. The virtual machines are all created from base templates intended for different areas of work. It is simple to both add new templates and modify existing ones. The state of a running virtual machine can at any time be saved and used as a new template, leaving the older version unaffected. The entire process used to start up virtual machines is fully automated, and once a machine is up and running users can connect to it remotely. The scripting potential of the solution affects not only the creation and destruction of virtual machines, but allows entire workflows to be automated as well. Together these properties meet all the conditions given in requirement #1 of the system specification. Figure 6.1 illustrates how the virtualization stack can be used to achieve a process that is virtually identical to that depicted in Figure 1.1 which described the system's intended workflow.

Requirement #2 accounts for the desired scalability and modularity of the system. One request is that the solution be based on existing software rather than homemade solutions to as large extent as possible. The virtualization stack is comprised of tried and trusted software with widespread industry usage. The solution is modular in its essence as each component simply adds a layer of abstraction above the rest of the stack. A variety of programming language bindings exist for both libvirt and Vagrant. This makes it uncomplicated to build further upon the virtualization stack or implement the solution into a larger system.

Every component of the virtualization stack is free and open-source. This meets requirement #3 which states that open-source alternatives should be favored over

proprietary software. Both the virtual machine templates developed in 4.1 and 4.2 use open-source operating system distributions as well. Requirement #4 requests that the environment used should be a Debian or Red Hat based Linux distribution. All development has been performed on a computer running a Debian Buster installation, guaranteeing that the solution is compatible with such a system.

Requirement #5 states that the end system should function according to current standards in digital forensic investigative work. In its current state the solution does not meet this requirement. Although it is possible to set rules to handle all disks and images connected to the solution in read-only mode, the remaining requirements have not been fully implemented. This topic is discussed further in chapter 7.

The possiblity to perform virtual machine chaining was requested in requirement #6. Section 5.2 describes the implementation of examples where the virtualization stack is used to perform chaining. The results from the examples prove that the virtualization stack is able to perform virtual machine chaining without issues. Some problems were run into when running scripts in environments with multiple virtual machines, but they were mitigated as explained in section 5.2.2.

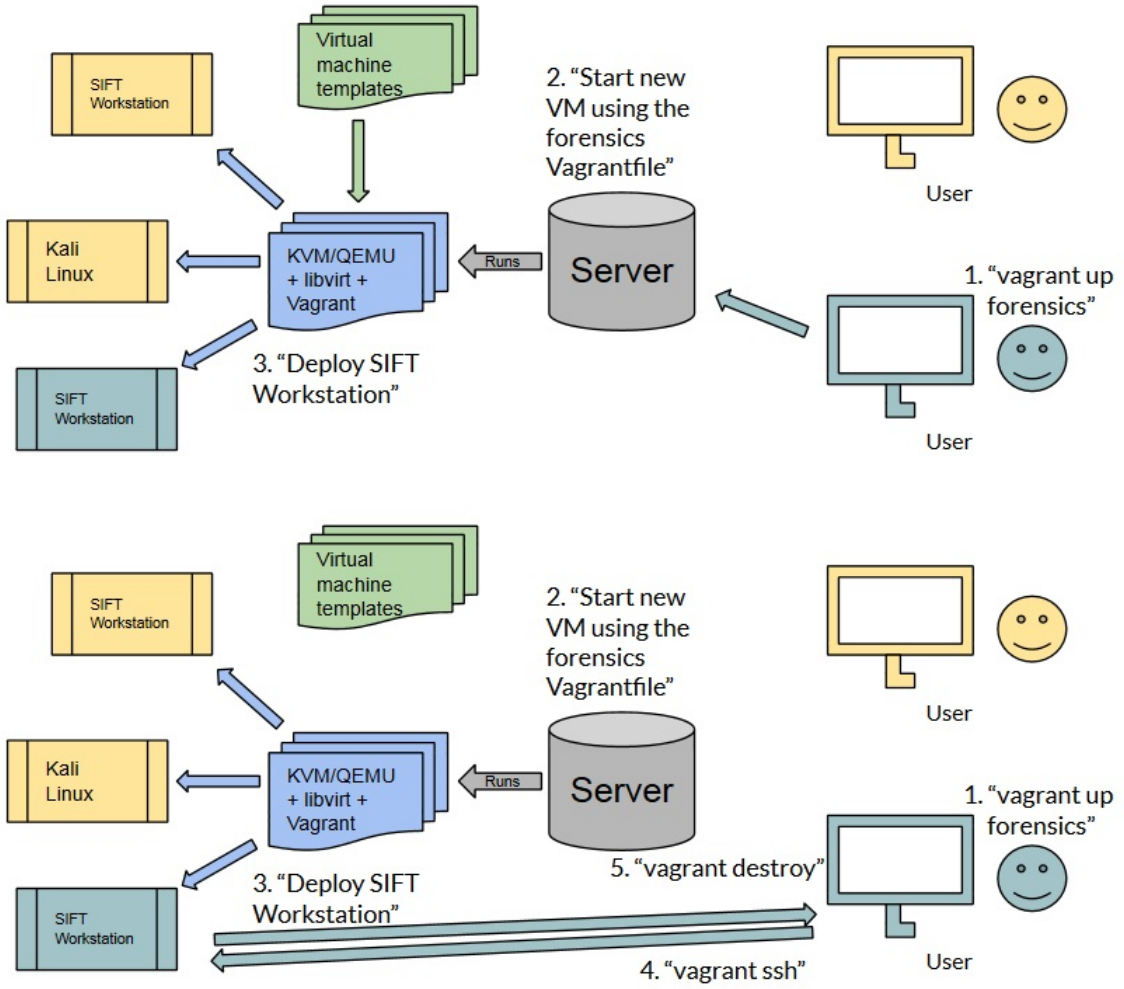

Figure 6.1: A user that wants to perform work connects to the server and launches a virtual machine based on a fitting template using a single command. Remote connecting to or deleting the virtual machine takes a single command respectively as well. All parts of the workflow can be scripted.

7
Discussion and conclusions

In this chapter the overall project outcome, that is the final product and knowledge gained, is discussed and conclusions are made. Areas where the solution meets the system specification and those where it does not are analysed while providing reasons why. Suggestions regarding possible further work are given as well.

The aim of this work has been to produce a solution that automates the creation and lifecycle management of virtual machines built from existing templates. To arrive at this outcome, a hypervisor and virtualization software suite was assembled and evaluated, standardized virtual machine templates suitable for different assignments were developed, and a number of workflow automation examples that demonstrate the features of the solution were created. A system specification was given in the initial stages of the project. Over the course of time this specification naturally changed, with additional requirements being added as the project proceeded. Virtual machine chaining is an example of a requirement that was not included in the introductory specification, but materialized as the functionality offered of the virtualization stack was demonstrated to colleagues at SecureLink. In its final state the solution is able to fully meet every requirement in the system specification except for the one relating to digital forensic investigation standards.

The presented solution is a software suite for virtualization that can be used to launch and manage virtual machines used in digital forensic and penetration testing work in SecureLink's planned centralized system. The KVM/QEMU + libvirt + Vagrant solution is scalable, modular, and allows for a high degree of automation through the application of user-defined scripts. The virtualization stack can either be used in its current state by creating Vagrantfiles tailored for certain assignments similar to the workflow automation examples in chapter 5, or implemented into a larger software solution with expanded functionality. Since multiple programming language bindings exist for the components of the virtualization stack, all features that have been utilized in the examples can be performed programmatically rather than via direct user interaction or macros. By building additional layers of software above the virtualization stack the system can be expanded to perform tasks such as user authentication, or further meeting forensic standards by generating unique ID's for every project and storing case data and virtual machine output in a structured manner. The intention is that whichever path is chosen, this work will function as a guideline should the solution be implemented in a production environment.

A suggestion is to, instead of using static Vagrantfiles, build a front-end where users

can specify what operations should be performed by the virtual machine(s) and then have a suitable Vagrantfile be generated and executed by the system. A collection of generalized and modular scripts for common operations performed in assignments can be written, and from this users can combine operations freely to create complex workflows. When new functionality is needed, new scripts can simply be added to the collection. This relieves users from having to write custom Vagrantfiles in Ruby for every new type of assignment. The configurations used to generate Vagrantfiles can be saved and reused as well.

In its current state, the solution does not meet the majority of the standards on digital forensic investigative work that were given. This is partially due to the fact that these requirements were not defined in a concrete manner by SecureLink before roughly half of the project time had passed. As the area of digital forensic science is relatively young, no official industry go-to standards exist as of yet. The example requirements listed in section 2.1 are based on how digital forensic work is performed at SecureLink, which in turn is based on practical experience of the field, such as which investigation methodologies have led to results that held up in a court of law. It is fully possible to set rules that handle all disks connected to the solution in read-only mode, but the other requirements were harder to verify. The most important aspect of the forensic standards is that every single event related to the investigation is logged in one way or another so that it is possible to keep a chain of custody. It should be clear that the functionality offered by the solution is able to achieve this task. Logging can be performed by incorporating calls to some logging utility from the scripts that are run during the virtual machine execution cycle. A rudimentary version could be incorporated into the scripts themselves by simply appending for example a tuple of the form (case ID, user, event, time) to a log file in the project folder as a result of every meaningful operation that the virtual machine performs. Although more work could have been spent on making the solution meet current standards in digital forensic work, building a system that fulfills every single requirement is almost enough work to warrant an entire project of its own. The high degree of customization offered by the virtualization stack, demonstrated in the various examples, shows that it is entirely possible to meet these standards through further work on the system.

Conclusively, virtualization is an interesting and practical technology that despite its widespread use in the information technology industry has not yet had all its areas of application discovered. In this thesis hardware virtualization is used to provide virtual computer environments for use in digital forensic and penetration testing work, and a software solution to automate this process has been produced. The reduced amount of hardware necessary to perform such work as a consequence of virtualization is not only more monetarily sustainable, but environmentally so as well. It is also worth noting the ethical aspects of this project. Penetration testing is used for vulnerability mitigation in computer systems, while digital forensic investigations collect evidence of cybercrime. Both areas help making the digital society a safer place and hopefully the solution delivered in this work will do its part in this regard.

Bibliography

[1] VMWare, Inc., "A Performance Comparison of Hypervisors", 2007. [Online]. Available: https://www.vmware.com/pdf/hypervisor_performance.pdf, Accessed on: 2019-01-29.

[2] G. Palmer, "A Road Map for Digital Forensic Research". Technical Report DTR-T0010-01, DFRWS, November 2001. Report from the First Digital Forensic Research Workshop (DFRWS).

[3] K M. Henry, "Introduction to Penetration Testing", in Penetration Testing: Protecting Networks and Systems. Cambridgeshire, United Kingdom: IT Governance Publishing, 2012.

[4] R. Corey, "Summarizing The Five Phases of Penetration Testing", 2015. [Online]. Available: https://www.cybrary.it/2015/05/summarizing-the-five-phases-of-penetration-testing/, Accessed on: 2019-02-22.

[5] "Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution", 2019. [Online]. Available: https://www.kali.org/, Accessed on: 2019-04-03.

[6] VMWare, Inc., "What is virtualization technology and virtual machines?", 2019. [Online]. Available: https://www.vmware.com/solutions/virtualization.html, Accessed on: 2019-02-22.

[7] C.D. Graziano, "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project", MSc thesis, Electrical and Computer Engineering, Iowa State University, Ames, The United States, 2011. [Online]. Available: https://lib.dr.iastate.edu/etd/12215, Accessed on: 2019-01-29.

[8] G.J. Popek, R.P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures", Commun. ACM, vol. 17, no. 7, pp. 412-421, Jul, 1974. doi:10.1145/361011.361073, [Online]. Available: ACM Digital Library, https://dl.acm.org/, Accessed on 2019-01-29.

[9] KVM, "Kernel Virtual Machine", 2019. [Online]. Available: https://www.linux-kvm.org, Accessed on: 2019-02-14.

[10] J. Hwang, T. Wood, S. Zeng, "A Component Based Performance Comparison of Four Hypervisors", in IFIP/IEEE International Symposium on Integrated Network Management, 2013. [Online]. Available: https://www.researchgate.net/publication/242105480_A_Component_Based_Performance_Comparison_of_Four_Hypervisors, Accessed on: 2019-02-03.

[11] W. Graniszewski, A. Arciszewski, "Performance analysis of selected hypervisors (Virtual Machine Monitors - VMMs)", International Journal of Electronics and Telecommunications, vol. 62, no. 3, pp. 231-236, Sep 2016. doi:10.1515/eletel-2016-0031, [Online]. Available:

http://www.ijet.pl/index.php/ijet/article/view/10.1515-eletel-2016-0031, Accessed on: 2019-02-03.

[12] D. Leite, M. Peixoto, M. Santana, R. Santana, "Performance Evaluation of Virtual Machine Monitors for Cloud Computing", in 13th Symposium on Computing Systems, 2012. [Online]. Available: https://www.researchgate.net/publication/236669181_Performance_Evaluation_of_Virtual_Machine_Monitors_for_Cloud_Computing, Accessed on: 2019-02-03.

[13] A. Algarni, M.R. Ikbal, R. Alroobaea, A. Ghiduk, F. Nadeem, "Performance Evaluation of Xen, KVM, and Proxmox Hypervisors", International Journal of Open Source Software and Processes, vol. 9, no. 2, Apr 2018. doi:10.4018/IJOSSP.2018040103, [Online]. Available: https://www.researchgate.net/publication/327482365_Performance_Evaluation_of_Xen_KVM_and_Proxmox_Hypervisors, Accessed on: 2019-02-03.

[14] Google, "Google Compute Engine FAQ", 2019. [Online]. Available: https://cloud.google.com/compute/docs/faq, Accessed on: 2019-05-02.

[15] Amazon, "Amazon EC2 Web FAQs", 2019. [Online]. Available: https://aws.amazon.com/ec2/faqs/#compute-optimized, Accessed on: 2019-05-02.

[16] KVM, "FAQ - KVM", 2019. [Online]. Available: https://www.linux-kvm.org/page/FAQ, Accessed on: 2019-02-14.

[17] QEMU, "QEMU", 2019. [Online]. Available: https://wiki.qemu.org, Accessed on: 2019-03-02.

[18] Libvirt - Virtualization API, "FAQ - Libvirt: What is libvirt?", 2019. [Online]. Available: https://wiki.libvirt.org/page/FAQ#What_is_libvirt.3F, Accessed on: 2019-03-02.

[19] HashiCorp, "Introduction to Vagrant", 2019. [Online]. Available: https://www.vagrantup.com/intro/index.html, Accessed on: 2019-04-12.

[20] HashiCorp, "Vagrant Documentation", 2019. [Online]. Available: https://www.vagrantup.com/docs/, Accessed on: 2019-04-12.

[21] HashiCorp, "Getting started - Boxes", 2019. [Online]. Available: https://www.vagrantup.com/intro/getting-started/boxes.html, Accessed on: 2019-04-12.

[22] vagrant-libvirt, "Vagrant provider for libvirt", GitHub repository, 2019. [Online]. Available: https://github.com/vagrant-libvirt/vagrant-libvirt, Accessed on: 2019-04-12.

[23] M. McLoughlin, "The QCOW2 Image Format", 2008. [Online]. Available: https://people.gnome.org/ markmc/qcow-image-format.html, Accessed on: 2019-04-24.

[24] ECMA International, "The JSON Data Interchange Syntax", 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, Accessed on: 2019-04-24.

[25] SANS Digital Forensics, "SIFT Workstation Overview", 2019. [Online]. Available: https://digital-forensics.sans.org/community/downloads#overview, Accessed on: 2019-04-24.

[26] DMTF, "Open Virtualization Format Specification", 2010. [Online]. Available: https://www.dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf, Accessed on: 2019-04-24.
[27] VMWare, Inc., "Virtual Disk Format 5.0", 2011. [Online]. Available: https://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf, Accessed on: 2019-04-24.
[28] Cisco, "ClamAV", 2019. [Online]. Available: https://www.clamav.net/, Accessed on: 2019-04-25
[29] greg5678, "Malware-Samples", GitHub repository, 2017. [Online]. Available: https://github.com/greg5678/Malware-Samples, Accessed on: 2019-04-25.

A

Source code

This appendix presents the unabridged source code of every Vagrantfile and script used in the examples in this report.

A.1 Proof of concept

Vagrantfile:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# This Vagrantfile configures a barebone virtual machine and showcases some of Vagrant's
# high level features such as triggers and provisioners by using simple shell scripts with print statements.
Vagrant.configure("2") do |config|
  # VM template.
  config.vm.box = "generic/debian9"

  # Trigger that runs before VM deployment.
  config.trigger.before :up do |trigger|
    trigger.info = "Running before up trigger."
    trigger.run = {inline: <<-SHELL
      echo "This is a trigger that is being run before the VM is created. From here I can move files into the template, etc. After this script, the VM is created and booted up."
      sleep 10
    SHELL}
  end

  # Shell provisioner that runs once VM is deployed
  config.vm.provision :shell, inline: <<-SHELL
```

```
21      echo "Hello world! This output is from a shell script that is
        ↪ run automatically when the VM has been set up. This allows
        ↪ me to run arbitrary code on the deployed VM. I can use this
        ↪ to install software, run programs, etc. The VM can be
        ↪ accessed manually using either the graphical interface
        ↪ provided by Libvirts Virtual Machine Manager, or by using
        ↪ ssh from the command line."
22      sleep 15
23    SHELL
24
25    # Trigger that runs before VM destruction.
26    config.trigger.before :destroy do |trigger|
27      trigger.info = "Running before destroy trigger"
28      trigger.run = {inline: <-- SHELL
29        echo "This is a trigger that is being run before the VM is
          ↪ destroyed. This can be used to transfer results into an
          ↪ output folder when finished, etc. After this script, the
          ↪ VM is shut down and destroyed along with all temporary
          ↪ storage."
30        sleep 10
31      SHELL}
32    end
33  end
```

A.2 Malware detection

Vagrantfile:

```
1   # -*- mode: ruby -*-
2   # vi: set ft=ruby :
3
4   # This Vagrantfile configures a VM running SIFT that performs an
    ↪ automated malware scan on an external disk image and then
    ↪ copies the scanning report and all detected malware to the
    ↪ host computer.
5   #
6   # Vagrant triggers are used to run scripts on the host computer
    ↪ or the VM at certain points in the workflow. Features of both
    ↪ Vagrant and libvirt are used to complete the task. To avoid
    ↪ cluttering the Vagrantfile, the scripts can be found in
    ↪ separate files.
7   #
8   # This example demonstrates the functionality offered by the
    ↪ virtualization stack being used in a realistic context.
9   Vagrant.configure("2") do |config|
```

```
10    # Configuration options used are documented and commented
      ↪  below. For a complete reference, please see the online
      ↪  documentation at
11    # https://docs.vagrantup.com and
      ↪  https://github.com/vagrant-libvirt/vagrant-libvirt.
12
13    # VM template.
14    config.vm.box = "SIFT-Workstation"
15
16    # Vagrant plugins.
17    config.vagrant.plugins = ["vagrant-libvirt", "vagrant-scp"]
18
19    # Libvirt configuration.
20    config.vm.provider :libvirt do |libvirt|
21      libvirt.memory = 4096
22    end
23
24    # Trigger that runs after VM deployment.
25    config.trigger.after :up do |trigger|
26      trigger.run = {path: "attach_disk_image.sh"}
27      trigger.run_remote = {path: "scan_disk_image.sh"}
28    end
29
30    # SSH configuration.
31    config.ssh.username = 'sansforensics'
32    config.ssh.password = 'forensics'
33    config.ssh.insert_key = true
34
35    # Trigger that runs before VM destruction.
36    config.trigger.before :destroy do |trigger|
37      trigger.run = {path: "detach_disk_image_and_copy_output.sh"}
38    end
39  end
```

attach_disk_image.sh:

```
1  #!/usr/bin/env bash
2
3  echo "Attaching disk image.img"
4  sudo virsh attach-disk sift_default --source
   ↪  /home/eric/ragebed/vagrantvms/sift/images/image.img --target
   ↪  vdb
```

scan_disk_image.sh:

```
1  #!/usr/bin/env bash
2
3  echo "Mounting disk image.img"
```

```
cd ../..        # script runs in /home/sansforensics, we want to
    ↪ be in the root folder for this
sudo mkdir /media/diskimage
sudo mount /dev/vdb1 /media/diskimage

echo "Running ClamAV scan on the disk and writing output to
    ↪ var/tmp/vagrant_output"
# update virus database
echo "forensics" | sudo -S freshclam
mkdir /var/tmp/vagrant_output
echo "forensics" | sudo -S bash -c "clamscan -ri
    ↪ --copy=var/tmp/vagrant_output /media/diskimage >>
    ↪ var/tmp/vagrant_output/clamavscanresult.txt"

sudo chmod 777 -R /var/tmp/vagrant_output        # Ugly hack, but
    ↪ vagrant scp had problems copying the malware file to host
```

detach_disk_image_and_copy_output.sh:

```
#!/usr/bin/env bash

echo "Moving output files to host"
vagrant scp :/var/tmp/vagrant_output .

echo "Detaching disk image.img"
sudo virsh detach-disk sift_default --target vdb
```

A.3 Virtual machine chaining example

start.sh:

```
#!/usr/bin/env bash

vagrant up first
vagrant up second
```

Vagrantfile:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# This Vagrantfile configures a barebone VM chaining example. Two
    ↪ VMs, named first and second, are defined and configured.
    ↪ Vagrant up will deploy first while second waits. After first
    ↪ is destroyed, it will start up second.
#
# This example demonstrates how chains of virtual machines can be
    ↪ started using a single command.
Vagrant.configure("2") do |config|
```

```
8   # Configuration options used are documented and commented
    ↪ below. For a complete reference, please see the online
    ↪ documentation at
9   # https://docs.vagrantup.com and
    ↪ https://github.com/vagrant-libvirt/vagrant-libvirt.
10
11  # Vagrant plugins.
12  config.vagrant.plugins = ["vagrant-libvirt"]
13
14  # First VM configuration.
15  config.vm.define "first" do |first|
16    # VM template.
17    first.vm.box = "SIFT-Workstation"
18
19    # libvirt configuration.
20    first.vm.provider :libvirt do |libvirt|
21      libvirt.memory = 4096
22    end
23
24    # SSH configuration.
25    first.ssh.username = "sansforensics"
26    first.ssh.password = "forensics"
27    first.ssh.insert_key = true
28
29    # Trigger that runs after VM deployment.
30    first.trigger.after :up do |trigger|
31      trigger.info = "Running afterup trigger"
32      trigger.info = "Destroying first"
33      trigger.run = {inline: "vagrant destroy first -f"}
34    end
35  end
36
37  # Second VM configuration.
38  config.vm.define "second", autostart: false do |second|
39    # VM template.
40    second.vm.box = "SIFT-Workstation"
41
42    # libvirt configuration.
43    second.vm.provider :libvirt do |libvirt|
44      libvirt.memory = 4096
45    end
46
47    # SSH configuration
48    second.ssh.username = "sansforensics"
49    second.ssh.password = "forensics"
50    second.ssh.insert_key = true
```

```
51
52      # Trigger that runs after VM deployment.
53      second.trigger.after :up do |trigger|
54        trigger.info = "Running afterup trigger"
55        trigger.info = "Destroying second"
56        trigger.run = {inline: "vagrant destroy second -f"}
57      end
58    end
59  end
```

A.4 Malware detection and analysis

ragebed.sh:

```
1  #!/usr/bin/env bash
2
3  vagrant up first
4  vagrant destroy first -f
5
6  vagrant up second
7  vagrant destroy second -f
```

Vagrantfile:

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  # This Vagrantfile defines and configures two VMs running SIFT.
   ↪ The first VM performs an automated malware scan on an
   ↪ external disk image and then copies the scanning report and
   ↪ all detected malware to the host computer. The first VM
   ↪ destroys itself, and in the process starts up the second VM
   ↪ which automatically imports said malware and starts malware
   ↪ analysis software.
5  #
6  # This example is an expansion of the single machine automated
   ↪ scan and demonstrates VM chaining being used in a realistic
   ↪ context.
7  Vagrant.configure("2") do |config|
8    # Configuration options used are documented and commented
     ↪ below. For a complete reference, please see the online
     ↪ documentation at
9    # https://docs.vagrantup.com and
     ↪ https://github.com/vagrant-libvirt/vagrant-libvirt.
10
11   # Vagrant plugins.
12   config.vagrant.plugins = ["vagrant-libvirt", "vagrant-scp"]
13
```

```
14  # First VM configuration.
15  config.vm.define "first" do |first|
16    # VM template.
17    first.vm.box = "SIFT-Workstation"
18
19    # libvirt configuration.
20    first.vm.provider :libvirt do |libvirt|
21      libvirt.memory = 4096
22    end
23
24    # SSH configuration.
25    first.ssh.username = "sansforensics"
26    first.ssh.password = "forensics"
27    first.ssh.insert_key = true
28
29    # Trigger that runs after VM deployment.
30    first.trigger.after :up do |trigger|
31      # Attach disk image to VM, mount disk and perform malware
        ↪ scan. Destroy first.
32      trigger.run = {path: "after_up_first.sh"}
33    end
34
35    # Trigger that runs before VM destruction.
36    first.trigger.before :destroy do |trigger|
37      # Detach disk image from VM, copy output folder to host,
        ↪ deploy second.
38      trigger.run = {path: "before_destroy_first.sh"}
39    end
40  end
41
42  # Second VM configuration.
43  config.vm.define "second", autostart: false do |second|
44    # VM template.
45    second.vm.box = "SIFT-Workstation"
46
47    # libvirt configuration.
48    second.vm.provider :libvirt do |libvirt|
49      libvirt.memory = 4096
50    end
51
52    # SSH configuration.
53    second.ssh.username = "sansforensics"
54    second.ssh.password = "forensics"
55    second.ssh.insert_key = true
56
57    # Trigger that runs after VM deployment.
```

```
58      second.trigger.after :up do |trigger|
59        # SSH into second and run the script analyse_malware.sh
60        trigger.run = {path: "after_up_second.sh"}
61      end
62
63      # Trigger that runs before VM destruction.
64      second.trigger.before :destroy do |trigger|
65        # Move output files to host machine
66        trigger.run = {path: "before_destroy_second.sh"}
67      end
68    end
69  end
```

after_up_first.sh:

```
1  #!/usr/bin/env bash
2
3  echo "Attaching disk image.img"
4  sudo virsh attach-disk chaining_demo_first --source
   ↪ /home/eric/ragebed/vagrantvms/chaining_demo/images/image.img
   ↪ --target vdb
5
6  vagrant ssh first --command ./../../vagrant/mount_and_scan_disk.sh
```

mount_and_scan_disk.sh:

```
1  #!/usr/bin/env bash
2
3  echo "Mounting disk image.img"
4  cd ../..          # script runs in /home/sansforensics, we want to
   ↪ be in the root folder for this
5  sudo mkdir /media/diskimage
6  sudo mount /dev/vdb1 /media/diskimage
7
8  echo "Running ClamAV scan on the disk and writing output to
   ↪ var/tmp/vagrant_output"
9  # update virus database
10 echo "forensics" | sudo -S freshclam
11 mkdir /var/tmp/vagrant_output
12 echo "forensics" | sudo -S bash -c "clamscan -ri
   ↪ --copy=var/tmp/vagrant_output /media/diskimage >>
   ↪ var/tmp/vagrant_output/clamavscanresult.txt"
13
14 sudo chmod 777 -R /var/tmp/vagrant_output        # Ugly hack, but
   ↪ vagrant scp had problems copying the malware file to host
```

before_destroy_first.sh:

```
1  #!/usr/bin/env bash
2
```

```
3 echo "Moving output files to host"
4 vagrant scp first:/var/tmp/vagrant_output ./vagrant_output_first
5
6 echo "Detaching disk image.img"
7 sudo virsh detach-disk chaining_demo_first --target vdb
```

after_up_second.sh:

```
1 #!/usr/bin/env bash
2
3 vagrant ssh second --command ./../../vagrant/analyse_malware.sh
```

analyse_malware.sh:

```
1 #!/usr/bin/env bash
2
3 echo "Analysing output from first machine..."
4 cd ../../vagrant/vagrant_output_first        # script runs in
  ↪ /home/sansforensics, we want to be in the rsynced vagrant
  ↪ folder
5 sudo mkdir ../../var/tmp/vagrant_output_second
6
7 # Perform hexdump on all files in folder
8 for filename in * ; do
9   if [ $filename == *.txt ]; then        # except .txt files
10    continue;
11  fi
12
13  echo "Performing hexdump on $filename and saving output to
    ↪ ../../var/tmp/vagrant_output_second/hexdump_$filename.txt"
14  echo "forensics" | sudo -S bash -c "xxd $filename >
    ↪ ../../var/tmp/vagrant_output_second/hexdump_$filename.txt"
15 done
16
17 sudo chmod 777 -R ../../var/tmp/vagrant_output_second        # Ugly
  ↪ hack, but vagrant scp had problems copying the malware file
  ↪ to host
```

before_destroy_second.sh:

```
1 #!/usr/bin/env bash
2
3 echo "Moving output files to host"
4 vagrant scp second:/var/tmp/vagrant_output_second
  ↪ ./vagrant_output_second
```

B

Runtime output

This appendix presents the runtime output from every example used in the report. The contents of certain files and folders are listed as well in cases where they are relevant to the example.

B.1 Proof of concept

Terminal output:

```
eric@eric:~/ragebed/vagrantvms/presentation_demo$ vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Running action triggers before up ...
==> default: Running trigger...
==> default: Running beforeup trigger.
    default: Running local script: triggerbeforeup.sh
    default: "This is a trigger that is being run before the VM is
    ↪ created. From here I can move files into the template, etc.
    ↪ After this code, the VM is created and booted up."
==> default: Checking if box 'generic/debian9' version '1.9.2' is
↪ up to date...
==> default: Creating image (snapshot of base box volume).
==> default: Creating domain with the following settings...
==> default:  -- Name:              presentation_demo_default
==> default:  -- Domain type:       kvm
==> default:  -- Cpus:              2
==> default:  -- Feature:           acpi
==> default:  -- Feature:           apic
==> default:  -- Feature:           pae
==> default:  -- Memory:            2048M
==> default:  -- Management MAC:
==> default:  -- Loader:
==> default:  -- Nvram:
==> default:  -- Base box:          generic/debian9
==> default:  -- Storage pool:      default
==> default:  -- Image:
↪ /var/lib/libvirt/images/presentation_demo_default.img (32G)
==> default:  -- Volume Cache:      default
==> default:  -- Kernel:
```

```
==> default:  -- Initrd:
==> default:  -- Graphics Type:     vnc
==> default:  -- Graphics Port:     -1
==> default:  -- Graphics IP:       127.0.0.1
==> default:  -- Graphics Password: Not defined
==> default:  -- Video Type:        cirrus
==> default:  -- Video VRAM:        256
==> default:  -- Sound Type:
==> default:  -- Keymap:            en-us
==> default:  -- TPM Path:
==> default:  -- INPUT:             type=mouse, bus=ps2
==> default: Creating shared folders metadata...
==> default: Starting domain.
==> default: Waiting for domain to get an IP address...
==> default: Waiting for SSH to become available...
    default:
    default: Vagrant insecure key detected. Vagrant will
     ↪ automatically replace
    default: this with a newly generated keypair for better
     ↪ security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's
     ↪ present...
    default: Key inserted! Disconnecting and reconnecting using new
     ↪ SSH key...
==> default: Configuring and enabling network interfaces...
==> default: Rsyncing folder:
 ↪ /home/eric/ragebed/vagrantvms/presentation_demo/stuff/ =>
 ↪ /home/vagrant/vagrantfolder
==> default: Running provisioner: shell...
    default: Running: /tmp/vagrant-shell20190515-30079-ppgof9.sh
    default: "Hello world! This output is from a shell script that
     ↪ is run automatically when the VM has been set up. This
     ↪ allows me to run arbitrary code on the deployed VM. I can
     ↪ use this to install software, run programs, etc. The VM can
     ↪ be accessed manually using either the graphical interface
     ↪ provided by Libvirts Virtual Machine Manager, or by using
     ↪ ssh from the command line."
eric@eric:~/ragebed/vagrantvms/presentation_demo$ vagrant destroy
 ↪ -f
==> default: Running action triggers before destroy ...
==> default: Running trigger...
==> default: Running beforedestroy trigger
    default: Running local script: triggerbeforedestroy.sh
```

```
    default: "This is a trigger that is being run before the VM is
     ↪ destroyed. This can be used to transfer results into an
     ↪ output folder when finished, etc. After this code, the VM
     ↪ is shut down and destroyed along with all temporary
     ↪ storage."
==> default: Removing domain...
```

B.2 Malware detection

Terminal output:

```
eric@eric:~/ragebed/vagrantvms/sift$ vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Creating image (snapshot of base box volume).
==> default: Creating domain with the following settings...
==> default:  -- Name:              sift_default
==> default:  -- Domain type:       kvm
==> default:  -- Cpus:              1
==> default:  -- Feature:           acpi
==> default:  -- Feature:           apic
==> default:  -- Feature:           pae
==> default:  -- Memory:            4096M
==> default:  -- Management MAC:
==> default:  -- Loader:
==> default:  -- Nvram:
==> default:  -- Base box:          SIFT-Workstation
==> default:  -- Storage pool:      default
==> default:  -- Image:
 ↪ /var/lib/libvirt/images/sift_default.img (100G)
==> default:  -- Volume Cache:      default
==> default:  -- Kernel:
==> default:  -- Initrd:
==> default:  -- Graphics Type:     vnc
==> default:  -- Graphics Port:     -1
==> default:  -- Graphics IP:       127.0.0.1
==> default:  -- Graphics Password: Not defined
==> default:  -- Video Type:        cirrus
==> default:  -- Video VRAM:        9216
==> default:  -- Sound Type:
==> default:  -- Keymap:            en-us
==> default:  -- TPM Path:
==> default:  -- INPUT:             type=mouse, bus=ps2
==> default: Creating shared folders metadata...
==> default: Starting domain.
==> default: Waiting for domain to get an IP address...
==> default: Waiting for SSH to become available...
    default:
```

```
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's
     ↪  present...
    default: Key inserted! Disconnecting and reconnecting using new
 ↪  SSH key...
==> default: Configuring and enabling network interfaces...
==> default: Rsyncing folder: /home/eric/ragebed/vagrantvms/sift/
 ↪  => /vagrant
==> default: Running action triggers after up ...
==> default: Running trigger...
    default: Running local script: attach_disk_image.sh
    default: Attaching disk image.img
    default: Disk attached successfully
    default: Running: /tmp/vagrant-shell20190515-29046-1wwmifv.sh
    default: Mounting disk image.img
    default: Running ClamAV scan on the disk and writing output to
 ↪  var/tmp/vagrant_output
eric@eric:~/ragebed/vagrantvms/sift$ vagrant destroy -f
==> default: Running action triggers before destroy ...
==> default: Running trigger...
    default: Running local script:
 ↪  detach_disk_image_and_copy_output.sh
    default: Moving output files to host
    default: Warning: Permanently added '192.168.121.63' (ECDSA) to
 ↪  the list of known hosts.
    default: Detaching disk image.img
    default: Disk detached successfully
==> default: Removing domain...
```

clamavscanresult.txt:

```
/media/diskimage/nothingsuspicioushere/02ab39d5ef83ffd09e3774a67b
783bfa345505d3cb86694c5b0f0c94980e5ae8: Unix.Trojan.DDoS_XOR-1
 ↪  FOUND
/media/diskimage/nothingsuspicioushere/02ab39d5ef83ffd09e3774a67b
783bfa345505d3cb86694c5b0f0c94980e5ae8: copied to
 ↪  'var/tmp/vagrant_output/02a
b39d5ef83ffd09e3774a67b783bfa345505d3cb86694c5b0f0c94980e5ae8'

----------- SCAN SUMMARY -----------
Known viruses: 6529230
Engine version: 0.99.4
Scanned directories: 136
Scanned files: 1140
Infected files: 1
Data scanned: 46.98 MB
Data read: 47.62 MB (ratio 0.99:1)
```

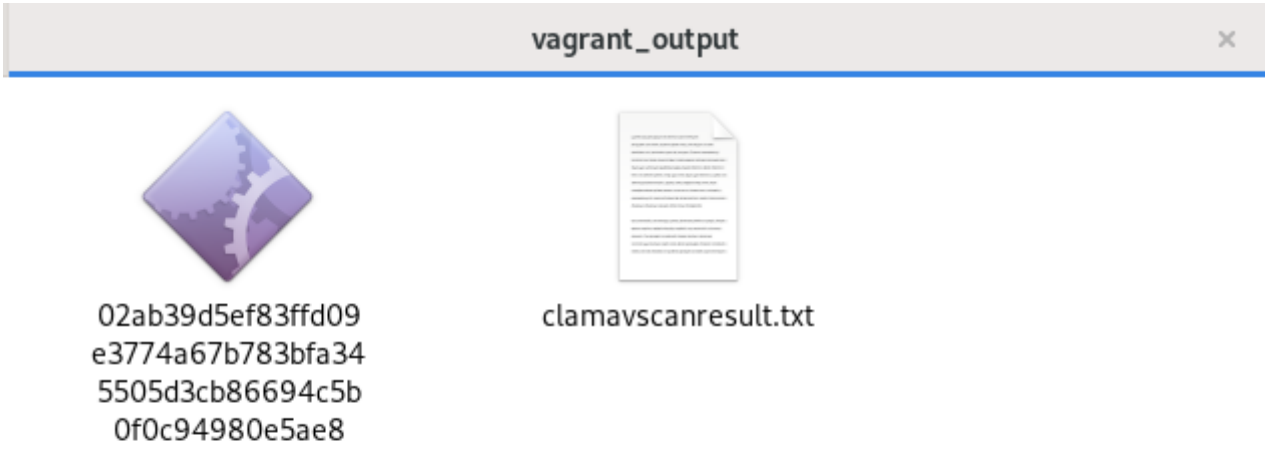

Figure B.1: Contents of the folder vagrant_output after execution.

```
Time: 20.076 sec (0 m 20 s)
```

B.3 Virtual machine chaining example

Terminal output:

```
eric@eric:~/ragebed/vagrantvms/chaining_barebones$ ./start.sh
Bringing machine 'first' up with 'libvirt' provider...
==> first: Creating image (snapshot of base box volume).
==> first: Creating domain with the following settings...
==> first:  -- Name:              chaining_barebones_first
==> first:  -- Domain type:       kvm
==> first:  -- Cpus:              1
==> first:  -- Feature:           acpi
==> first:  -- Feature:           apic
==> first:  -- Feature:           pae
==> first:  -- Memory:            4096M
==> first:  -- Management MAC:
==> first:  -- Loader:
==> first:  -- Nvram:
==> first:  -- Base box:          SIFT-Workstation
==> first:  -- Storage pool:      default
==> first:  -- Image:
 ↪  /var/lib/libvirt/images/chaining_barebones_first.img (100G)
==> first:  -- Volume Cache:      default
==> first:  -- Kernel:
```

```
==> first:  -- Initrd:
==> first:  -- Graphics Type:     vnc
==> first:  -- Graphics Port:     -1
==> first:  -- Graphics IP:       127.0.0.1
==> first:  -- Graphics Password: Not defined
==> first:  -- Video Type:        cirrus
==> first:  -- Video VRAM:        9216
==> first:  -- Sound Type:
==> first:  -- Keymap:            en-us
==> first:  -- TPM Path:
==> first:  -- INPUT:             type=mouse, bus=ps2
==> first: Creating shared folders metadata...
==> first: Starting domain.
==> first: Waiting for domain to get an IP address...
==> first: Waiting for SSH to become available...
    first:
    first: Inserting generated public key within guest...
    first: Removing insecure key from the guest if it's present...
    first: Key inserted! Disconnecting and reconnecting using new
 ↪  SSH key...
==> first: Configuring and enabling network interfaces...
==> first: Rsyncing folder:
 ↪  /home/eric/ragebed/vagrantvms/chaining_barebones/ => /vagrant
==> first: Running action triggers after up ...
==> first: Running trigger...
==> first: Destroying first
    first: Running local: Inline script
    first: vagrant destroy first -f
    first: ==> first: Removing domain...
Bringing machine 'second' up with 'libvirt' provider...
==> second: Creating image (snapshot of base box volume).
==> second: Creating domain with the following settings...
==> second:  -- Name:              chaining_barebones_second
==> second:  -- Domain type:       kvm
==> second:  -- Cpus:              1
==> second:  -- Feature:           acpi
==> second:  -- Feature:           apic
==> second:  -- Feature:           pae
==> second:  -- Memory:            4096M
==> second:  -- Management MAC:
==> second:  -- Loader:
==> second:  -- Nvram:
==> second:  -- Base box:          SIFT-Workstation
==> second:  -- Storage pool:      default
==> second:  -- Image:
 ↪  /var/lib/libvirt/images/chaining_barebones_second.img (100G)
```

```
==> second:  -- Volume Cache:      default
==> second:  -- Kernel:
==> second:  -- Initrd:
==> second:  -- Graphics Type:     vnc
==> second:  -- Graphics Port:     -1
==> second:  -- Graphics IP:       127.0.0.1
==> second:  -- Graphics Password: Not defined
==> second:  -- Video Type:        cirrus
==> second:  -- Video VRAM:        9216
==> second:  -- Sound Type:
==> second:  -- Keymap:            en-us
==> second:  -- TPM Path:
==> second:  -- INPUT:             type=mouse, bus=ps2
==> second: Creating shared folders metadata...
==> second: Starting domain.
==> second: Waiting for domain to get an IP address...
==> second: Waiting for SSH to become available...
    second:
    second: Inserting generated public key within guest...
    second: Removing insecure key from the guest if it's present...
    second: Key inserted! Disconnecting and reconnecting using new
     ↪ SSH key...
==> second: Configuring and enabling network interfaces...
==> second: Rsyncing folder:
 ↪ /home/eric/ragebed/vagrantvms/chaining_barebones/ => /vagrant
==> second: Running action triggers after up ...
==> second: Running trigger...
==> second: Destroying second
==> second: Running local: Inline script
==> second: vagrant destroy first -f
==> second: ==> second: Removing domain...
```

B.4 Malware detection and analysis

Terminal output:

```
eric@eric:~/ragebed/vagrantvms/chaining_demo$ ./ragebed.sh
Bringing machine 'first' up with 'libvirt' provider...
==> first: Creating image (snapshot of base box volume).
==> first: Creating domain with the following settings...
==> first:  -- Name:              chaining_demo_first
==> first:  -- Domain type:       kvm
==> first:  -- Cpus:              1
==> first:  -- Feature:           acpi
==> first:  -- Feature:           apic
==> first:  -- Feature:           pae
==> first:  -- Memory:            4096M
```

```
==> first:  -- Management MAC:
==> first:  -- Loader:
==> first:  -- Nvram:
==> first:  -- Base box:          SIFT-Workstation
==> first:  -- Storage pool:      default
==> first:  -- Image:
 ↪  /var/lib/libvirt/images/chaining_demo_first.img (100G)
==> first:  -- Volume Cache:      default
==> first:  -- Kernel:
==> first:  -- Initrd:
==> first:  -- Graphics Type:     vnc
==> first:  -- Graphics Port:     -1
==> first:  -- Graphics IP:       127.0.0.1
==> first:  -- Graphics Password: Not defined
==> first:  -- Video Type:        cirrus
==> first:  -- Video VRAM:        9216
==> first:  -- Sound Type:
==> first:  -- Keymap:            en-us
==> first:  -- TPM Path:
==> first:  -- INPUT:             type=mouse, bus=ps2
==> first: Creating shared folders metadata...
==> first: Starting domain.
==> first: Waiting for domain to get an IP address...
==> first: Waiting for SSH to become available...
    first:
    first: Inserting generated public key within guest...
    first: Removing insecure key from the guest if it's present...
    first: Key inserted! Disconnecting and reconnecting using new
 ↪  SSH key...
==> first: Configuring and enabling network interfaces...
==> first: Rsyncing folder:
 ↪  /home/eric/ragebed/vagrantvms/chaining_demo/ => /vagrant
==> first: Running action triggers after up ...
==> first: Running trigger...
    first: Running local script: after_up_first.sh
    first: Attaching disk image.img
    first: Disk attached successfully
    first: Mounting disk image.img
    first: Running ClamAV scan on the disk and writing output to
 ↪  var/tmp/vagrant_output
==> first: Running action triggers before destroy ...
==> first: Running trigger...
    first: Running local script: before_destroy_first.sh
    first: Moving output files to host
    first: Warning: Permanently added '192.168.121.94' (ECDSA) to
 ↪  the list of known hosts.
```

```
    first: Detaching disk image.img
    first: Disk detached successfully
==> first: Removing domain...
Bringing machine 'second' up with 'libvirt' provider...
==> second: Creating image (snapshot of base box volume).
==> second: Creating domain with the following settings...
==> second:  -- Name:              chaining_demo_second
==> second:  -- Domain type:       kvm
==> second:  -- Cpus:              1
==> second:  -- Feature:           acpi
==> second:  -- Feature:           apic
==> second:  -- Feature:           pae
==> second:  -- Memory:            4096M
==> second:  -- Management MAC:
==> second:  -- Loader:
==> second:  -- Nvram:
==> second:  -- Base box:          SIFT-Workstation
==> second:  -- Storage pool:      default
==> second:  -- Image:
 ↪  /var/lib/libvirt/images/chaining_demo_second.img (100G)
==> second:  -- Volume Cache:      default
==> second:  -- Kernel:
==> second:  -- Initrd:
==> second:  -- Graphics Type:     vnc
==> second:  -- Graphics Port:     -1
==> second:  -- Graphics IP:       127.0.0.1
==> second:  -- Graphics Password: Not defined
==> second:  -- Video Type:        cirrus
==> second:  -- Video VRAM:        9216
==> second:  -- Sound Type:
==> second:  -- Keymap:            en-us
==> second:  -- TPM Path:
==> second:  -- INPUT:             type=mouse, bus=ps2
==> second: Creating shared folders metadata...
==> second: Starting domain.
==> second: Waiting for domain to get an IP address...
==> second: Waiting for SSH to become available...
    second:
    second: Inserting generated public key within guest...
    second: Removing insecure key from the guest if it's present...
    second: Key inserted! Disconnecting and reconnecting using new
     ↪  SSH key...
==> second: Configuring and enabling network interfaces...
==> second: Rsyncing folder:
 ↪  /home/eric/ragebed/vagrantvms/chaining_demo/ => /vagrant
==> second: Running action triggers after up ...
```

```
==> second: Running trigger...
    second: Running local script: after_up_second.sh
    second: Analysing output from first machine...
    second: Performing hexdump on
     ↪  02ab39d5ef83ffd09e3774a67b783bfa345505d3cb86694c5b0f0c94980e5ae8
     ↪  and saving output to
     ↪  ../../var/tmp/vagrant_output_second/hexdump_02ab39d5ef83ffd09e3774
    a67b783bfa345505d3cb86694c5b0f0c94980e5ae8.txt
==> second: Running action triggers before destroy ...
==> second: Running trigger...
    second: Running local script: before_destroy_second.sh
    second: Moving output files to host
    second: Warning: Permanently added '192.168.121.197' (ECDSA) to
     ↪  the list of known hosts.
==> second: Removing domain...
```

clamavscanresult.txt:

```
/media/diskimage/nothingsuspicioushere/02ab39d5ef83ffd09e3774a67b
783bfa345505d3cb86694c5b0f0c94980e5ae8: Unix.Trojan.DDoS_XOR-1
 ↪  FOUND
/media/diskimage/nothingsuspicioushere/02ab39d5ef83ffd09e3774a67b783
bfa345505d3cb86694c5b0f0c94980e5ae8: copied to
 ↪  'var/tmp/vagrant_output/
02ab39d5ef83ffd09e3774a67b783bfa345505d3cb86694c5b0f0c94980e5ae8'

----------- SCAN SUMMARY -----------
Known viruses: 6529230
Engine version: 0.99.4
Scanned directories: 136
Scanned files: 1140
Infected files: 1
Data scanned: 46.98 MB
Data read: 47.62 MB (ratio 0.99:1)
Time: 34.401 sec (0 m 34 s)
```

hexdump_02ab39d5ef83ffd09e3774a67b783bfa345505d3cb86694c5b0f0c94980e5ae8.txt:

```
                          .
                          .
                          .
00090a20: 6e00 0000 2200 0300 0063 616c 6c5f 676d  n..."....call_gm
00090a30: 6f6e 5f73 7461 7274 0063 7274 7374 7566  on_start.crtstuf
00090a40: 662e 6300 5f5f 4354 4f52 5f4c 4953 545f  f.c.__CTOR_LIST_
00090a50: 5f00 5f5f 4454 4f52 5f4c 4953 545f 5f00  _.__DTOR_LIST__.
00090a60: 5f5f 4548 5f46 5241 4d45 5f42 4547 494e  __EH_FRAME_BEGIN
00090a70: 5f5f 005f 5f4a 4352 5f4c 4953 545f 5f00  __.__JCR_LIST__.
00090a80: 6474 6f72 5f69 6478 2e35 3331 3300 636f  dtor_idx.5313.co
```

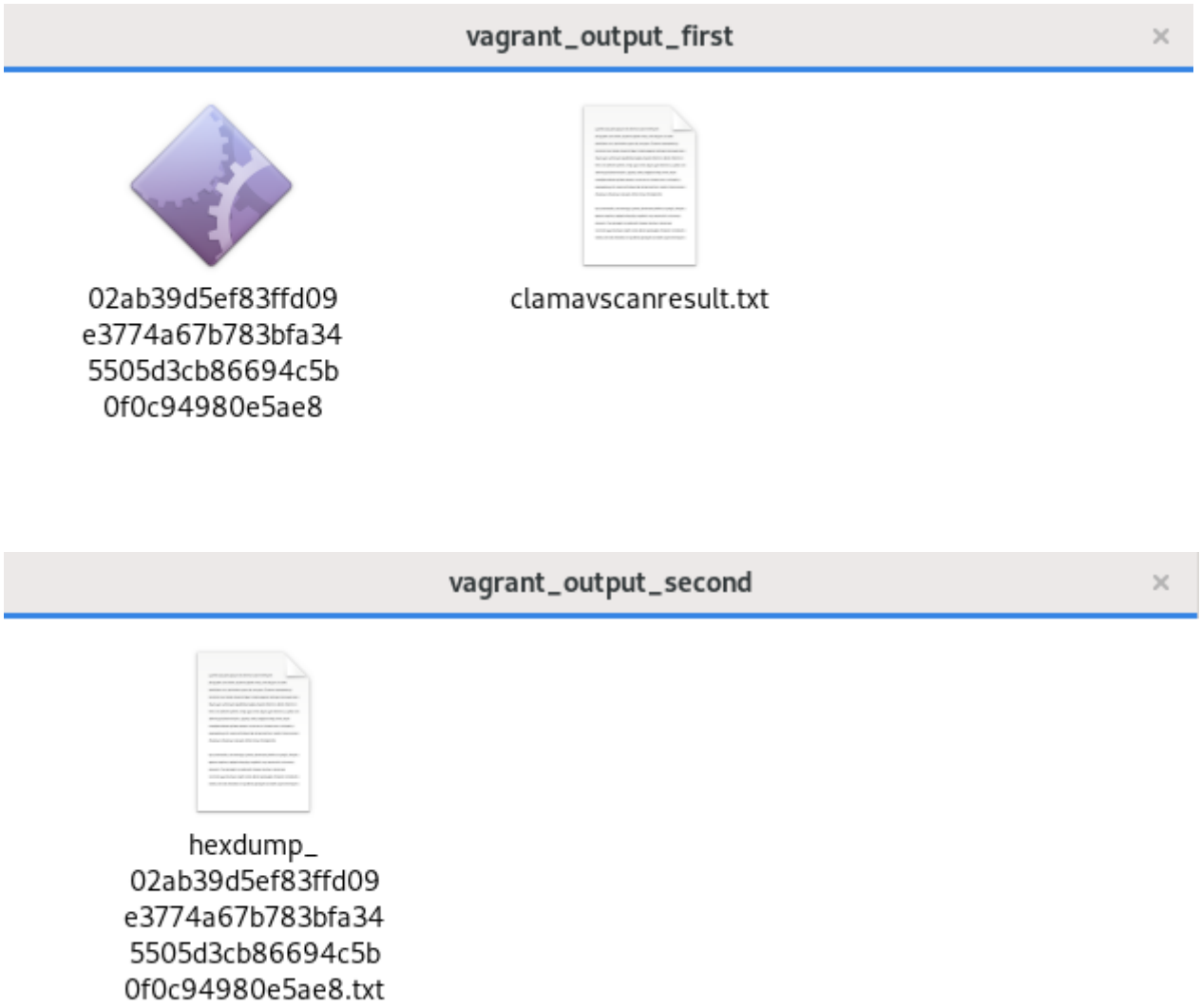

Figure B.2: Contents of the folders vagrant_output_first and vagrant_output_second after execution.

```
00090a90: 6d70 6c65 7465 642e 3533 3131 005f 5f64  mpleted.5311.__d
00090aa0: 6f5f 676c 6f62 616c 5f64 746f 7273 5f61  o_global_dtors_a
00090ab0: 7578 006f 626a 6563 742e 3533 3732 0066  ux.object.5372.f
00090ac0: 7261 6d65 5f64 756d 6d79 005f 5f43 544f  rame_dummy.__CTO
00090ad0: 525f 454e 445f 5f00 5f5f 4652 414d 455f  R_END__.__FRAME_
00090ae0: 454e 445f 5f00 5f5f 4a43 525f 454e 445f  END__.__JCR_END_
00090af0: 5f00 5f5f 646f 5f67 6c6f 6261 6c5f 6374  _.__do_global_ct
00090b00: 6f72 735f 6175 7800 6175 746f 7275 6e2e  ors_aux.autorun.
00090b10: 6300 6372 6333 322e 6300 656e 6372 7970  c.crc32.c.encryp
00090b20: 742e 6300 6578 6563 7061 636b 6574 2e63  t.c.execpacket.c
00090b30: 0062 7569 6c64 6e65 742e 6300 6869 6465  .buildnet.c.hide
00090b40: 2e63 0068 7474 702e 6300 6b69 6c6c 2e63  .c.http.c.kill.c
00090b50: 006d 6169 6e2e 6300 7072 6f63 2e63 0073  .main.c.proc.c.s
00090b60: 6f63 6b65 742e 6300 7463 702e 6300 7468  ocket.c.tcp.c.th
00090b70: 7265 6164 2e63 0066 696e 6469 702e 6300  read.c.findip.c.
00090b80: 646e 732e 6300 7374 6163 6b5f 6361 6368  dns.c.stack_cach
00090b90: 655f 6c6f 636b 005f 4c5f 6c6f 636b 5f32  e_lock._L_lock_2
00090ba0: 3200 7374 6163 6b5f 7573 6564 005f 4c5f  2.stack_used._L_
00090bb0: 756e 6c6f 636b 5f32 3438 005f 4c5f 6c6f  unlock_248._L_lo
00090bc0: 636b 5f32 3737 005f 4c5f 756e 6c6f 636b  ck_277._L_unlock
00090bd0: 5f33 3235 0073 7461 636b 5f63 6163 6865  _325.stack_cache
00090be0: 0069 6e5f 666c 6967 6874 5f73 7461 636b  .in_flight_stack
00090bf0: 0073 7461 636b 5f63 6163 6865 5f61 6374  .stack_cache_act
00090c00: 7369 7a65 005f 4c5f 6c6f 636b 5f37 3430  size._L_lock_740
00090c10: 005f 4c5f 756e 6c6f 636b 5f38 3637 005f  ._L_unlock_867._
                          .
                          .
                          .
```

C
Methods

This appendix gives a description of the methodology used during the project. The work has been done with employees from SecureLink acting as advisors. The project was divided into the following phases:

1. Study on hypervisors and virtualization toolkits and frameworks
2. Practical experience of use cases
3. Virtualization stack proof of concept
4. Template development
5. Workflow automation

The project was started by performing a general study on hypervisors and virtualization toolkits and frameworks available on the market. This time in phase #1 was spent getting theoretical knowledge about virtualization and looking at some of the suggestions for a starting point given by SecureLink along with other alternatives. Phase #2 was spent investigating an example use case given by SecureLink in order to get a better idea of what the system should be able to do. With more knowledge regarding the requirements on the system, it was possible to start phase #3. Here a hypervisor + software combination was assembled and a proof of concept demonstrating its features was developed. In phase #4 virtual machine templates suitable for digital forensics and penetration testing work were developed for use with the virtualization stack. Phase #5 was dedicated to showcasing the automation potential of the solution by using the virtualization stack and virtual machine templates to automatically perform example assignments.

C.1 Information gathering

Relevant information has primarily been gathered using online search engines and university library resources to look for scientific papers and websites using keywords related to the project. To provide further reading material, the sources of discovered work have often been examined as well. A considerable amount of technical documentation has been studied when researching and implementing software for the virtualization system. Here [9], [18] and [20] have been of particular relevance.

C.2 Development

Every design decision in the system has been taken using the following course of action:

Study -> Experimentation -> Implementation

The study step refers to the process of gathering information regarding possible solutions and getting a good view of feasible alternatives. The experimentation step consists of testing combinations of alternatives and comparing them according to parameters such as performance, availability of desired functionality and the ease of which one may build further upon the solution. Finally, the solution found to be the best fit is chosen to be integrated into the system, and the actual implementation part may start. During the development it has been useful to regularly apply a reflective loop to contemplate the state of the system:

A: What does the system do?
B: What should it do?
A->B: How to get there?

Since the system should be easy to expand upon further, existing solutions for required functionality has been used to as large extent as possible. The virtualization stack was built by combining hypervisor software with compatible virtualization toolkits and frameworks. After that virtual machine templates suitable for relevant tasks were developed in order to provide a working environment for the task automation. A proof of concept for the system was developed by using the functionality available the virtualization stack to deploy a virtual machine created from such a template and show off its abilities.

Languages used in the development were Ruby and Bash. All development has been done on a computer with a Debian Buster installation. SecureLink has experience working with several Debian and Red Hat derived distributions, and expressed the desire that one should be used to run the solution.

C.3 Version control

To have backup of project files available and perform the development in a structured manner, version control was implemented using a private repository in the web-based Git service GitHub. The master branch contained finalized features while a separate development branch was used for work in progress. Feature branches forked from development were used to test and experiment with new functionality (such as virtual machine templates or workflow automation examples) before implementing them through merging with the development branch. When enough advancement had been made to warrant it (for example when moving on to a new project phase), the master branch was updated with the progress in the development branch.