Automated testing of firmware installation and update scenarios for peripheral devices

DAG REUTERSKIÖLD
Automated testing of firmware installation and update scenarios for peripheral devices

DAG REUTERSKIÖLD
Abstract

This research presents an approach to transition from manual to automated testing of hardware specific firmware. The manual approach for firmware testing can be repetitive and time consuming. A significant proportion of the time is spent on cleaning and re-installing operating systems so that old firmware does not interfere with the newer firmware that is being tested. The approach in this research utilizes virtual machines and presents an automation framework. One component of the automation framework is an application to imitate connected peripheral devices to bypass hardware dependencies of firmware installers. The framework also consists of automation and pipeline scripts with the objective to execute firmware installers and detect errors and abnormalities in the installation and updating processes. The framework can run on locally hosted virtual machines, but is most applicable using cloud hosted virtual machines, where it is part of a continuous integration that builds, downloads, installs, updates and tests new firmware versions, in a completely automated manner.

The framework is evaluated by measuring and comparing execution times with manually conducted installation and updating tests, and the result shows that the framework complete tests much faster than the manual approach. Another evaluation in terms of scalability is made where speedup is measured. It is however concluded that scalability does not necessarily depend on the performance and resource allocation of systems, but rather how well it can be expanded and used for future devices and functionalities.
Sammanfattning


## Contents

### 1 Introduction
- 1.1 Background .................................................. 1
- 1.2 The principal’s interest ...................................... 1
- 1.3 Problem statement ............................................. 2
  - 1.3.1 Research Question ....................................... 2
  - 1.3.2 Research Methodologies ................................. 2
  - 1.3.3 Scope of Study ............................................ 4
- 1.4 Thesis outline .................................................. 4

### 2 Background
- 2.1 History of Virtualization ................................. 6
- 2.2 Hypervisors .................................................. 8
- 2.3 Virtualization approaches ............................. 9
  - 2.3.1 Full virtualization ..................................... 9
  - 2.3.2 Paravirtualization ................................... 9
  - 2.3.3 Hardware-layer virtualization ...................... 10
  - 2.3.4 Operating system level virtualization .......... 11
  - 2.3.5 Desktop Virtualization ............................... 12
- 2.4 Drivers in windows ........................................ 12
- 2.5 Eye tracking ................................................ 14
- 2.6 The Tobii architecture .................................. 15
  - 2.6.1 Manual eye tracking installation tests .......... 17
- 2.7 Related work ............................................... 17
- 2.8 Summary .................................................... 19

### 3 Method
- 3.1 Choice of virtualization software .................... 21
- 3.2 Virtual Machine setup .................................. 22
- 3.3 Device Spawning .......................................... 23
3.4 Automation framework ............................................. 25
  3.4.1 Automated Installation ........................................ 26
  3.4.2 Automated Installation testing ............................... 27
  3.4.3 Updating ......................................................... 28
  3.4.4 Server hosted integration ...................................... 30
3.5 Evaluation ........................................................... 32
  3.5.1 Evaluation on a client hosted environment ................. 33
  3.5.2 Evaluation in the server hosted environment ............... 34

4 Results and discussion .............................................. 35
  4.1 Results from client hosted environment ...................... 35
  4.2 Results from server hosted environment ...................... 39
  4.3 Discussion ......................................................... 40
  4.4 Social Sustainability and Ethics .................................. 43

5 Conclusions .......................................................... 44
  5.1 Concluding remarks ............................................... 44
  5.2 Validity of the results ........................................... 45
  5.3 Future works ....................................................... 45

Bibliography ............................................................ 46
Chapter 1

Introduction

1.1 Background

Virtualization is in its core, a creation of a virtual rather than a physical version of a computer resource. By making one physical computer function as two or more virtual computers, multiple operating systems can be installed and function on one single machine [1]. Today, almost any computing environment can run multiple virtual machines, that is, independent virtual computer systems, at the same time. Additionally, almost all common computer systems are compatible to be abstracted and run as virtual machines (VMs)[2]. There are several popular technologies for virtualization, including server, database, storage and desktop virtualization. This research mainly focuses on desktop virtualization which essentially implies separating an Operating System (OS) from its normal physical computer and run the VM as an isolated and controllable application using virtualization software such as VMWare, Hyper-V, and VirtualBox[3]. This approach is often referred to as Local Desktop Virtualization (LDV). Desktop virtualization can also be achieved by using cloud services such as AWS or Azure, which allow users to run VMs remotely but have them appear as if they are running on the user’s local computer. This approach can be referred to as Remote Desktop Virtualization (RDV).

1.2 The principal’s interest

When a peripheral device, say a USB-drive, is connected to a Windows-based OS, the Windows Registry and the event log files will be updated. Windows will look for drivers that aids the communication between the USB-drive and the OS in the Windows Driver Store [4]. Devices in Windows leave traces
such as time stamps, Instance ID:s and other data, that is rarely removed. These traces can interfere with newer versions of firmware, even if the old firmware of the devices has been uninstalled. It is therefore common to test device firmware on clean machines, that is, computers with a newly installed OS that has no memory of devices or old installation files. This type of verification approach is often used by software developers and Quality Assurance (QA) testers at Tobii, the host company of this research. Tobii is one of the leading high-technology companies that focus on gaze based interaction and eye tracking devices. Eye tracking is a technology that measures where an individual is looking at a given time, and what sequence an individual’s eyes are moving from one position to another. The technology comes in many forms and integrations, and has countless of possibilities and appliances.

However, as the number of distributions of eye trackers are increasing, the current approach of conducting manual testing is becoming increasingly difficult. This is especially true for hardware specific software (device firmware), that is mainly deployed and installed using drivers, drivers that in some ways are dependent on “clean” OSs. Re-installing and setting up clean environments for each integration and for every new firmware build is time consuming. Instead of having physical clean machines, developers and testers can use VMs for installation and deployment scenarios of their devices.

1.3 Problem statement

The main focus of this research is to investigate and evaluate the possibilities to automate the process of firmware installation, updating and testing, using virtual machines. The desired outcome is an automated framework that allows developers and testers to install, update and test device firmware in an efficient and scalable way, without being bound to specific hardware.

1.3.1 Research Question

*How to transition from manual to automated testing of firmware installation and update scenarios for peripheral devices?*

1.3.2 Research Methodologies

Figure 1.1 illustrates an overview of the research process used in this study. The study is categorized as an applied research as it involves answering a question with specific circumstances, related to particular situations [5]. The study
aims to solve real world problems by developing an automated framework for actual scenarios. The host company provides test-cases so that the automated framework can be applied with real firmware of peripheral devices.

Figure 1.1: Research Methodology

The data collection is partially done by conducting semi-structured interviews with developers and QA testers from the host company, and observations on the manual approach of firmware installation testing. Experiments on eye tracking devices for both automated installation and update scenarios are conducted as a part of the data collection of the research. The research method is of qualitative character with a small but sufficient enough data set where data collection continues until saturation and when a reliable research result is reached. To draw conclusions and establish what is true and what is false, an inductive approach of reasoning is used to formulate theories and propositions with explanations from observations and patterns. In order to evaluate the implemented framework in terms of scalability and to give an indication of the practical usage of the framework, time is the main metric of measurement.
1.3.3 Scope of Study

The research will focus on devices, drivers and firmware for Microsoft Windows 10. Other OSs might be compatible with the framework but will not be taken into account due to time limitations of the research.

Firmware installation can work in many different ways depending on both structure of the peripheral device, and also the methods used by the developers of the firmware. The implementation in this research is only tested to work on Tobii Experience Eye tracking software. However, the methods and approaches presented in this report should be comprehensive enough to function as a guideline for developers, researchers or anyone whom has an interest in creating automation tools for any type of peripheral device.

Conducting tests on a newly installed OS is in many ways an ideal environment for installing firmware, which is not the most probable scenario for real world users. Users tend to have unique configurations of their OS and installed programs, which might affect how a firmware is installed. The goal of the automated framework is therefore to give an clear indication of how firmware should install and update in a general scenario, and not how it will perform for each real world installation.

1.4 Thesis outline

The report has the following structure:

- **Chapter 2 - Background** - The chapter provides the necessary background information for understanding the thesis and presents state-of-the art research and tools used to conduct the research.

- **Chapter 3 - Methods** - The chapter focuses on the technical aspects used to address the research question. The chapter aims to answer what was done and how it was done. The chapter also brings up how the framework is evaluated.

- **Chapter 4 - Results and Discussion** presents the results of the research, divided in to two sections, one evaluation with Local Desktop Virtualization (LDV) and one evaluation with Remote Desktop Virtualization (RDV). The chapter also presents a discussion of the research, where the results are interpreted and the pros and cons of the research are discussed.
• *Chapter 5 - Conclusions* - The chapter summarizes the conclusions, the validity discussion and brings up future implications of the research.
Chapter 2

Background

The following chapter includes the pre-study of the thesis. The purpose of this chapter is to contextualize and justify the research question and provide the necessary background for understanding the thesis. The chapter will present state-of-the-art research and tools used to conduct the research.

2.1 History of Virtualization

In 1961, IBM introduced the world to Time Sharing [2], which was the first driving force behind virtualization. Time Sharing means that multiple users can access an OS at the same time, without the different users knowing about one another. In 1964, the IBM CP-40 was released which allowed the first virtual environment with a virtual memory. Multiple versions of the CP-40 and its successor CP-67 were released in the late 1960’s and early 1970’s. With each release came better performance which allowed an increasing number of VMs with more virtual memory.

Virtualization research and development halted in 1974 after a release of IBM’s System/370 with support for multiple OSs, as well as a firmware modification specifically developed for virtualization called VM-assist [6]. It was not until the late 1980’s, with the rise of the Internet that there was a new need for virtualization. In 1887, a program called SoftPC [7] was released for UNIX machines, including the Mac II, which allowed MS-DOS software to run on UNIX, something that was not possible before. SoftPC worked by emulating the PC by a simulation of the integrated circuit chips. Throughout the 1990’s, multiple important software based virtualization tools were released, including Virtual DOS machine, Virtual PC 1.0 and VMware [2].

VMware was first released in 1999 and is by many considered to be the
first commercial virtualization platform for the major CPU architecture x86 [2]. The patented infrastructure could initially only be installed on desktop computers and was called VMware Workstation. The product was later released on server platforms and could either run on top of Linux or Windows, or, on bare metal, that is, without any OS, allowing better performance for the VMs. This virtualization approach is called Hardware-layer virtualization and is an approach that will be brought up later in this chapter.

In 2005 and 2006, the two major x86 CPU vendors Intel and Advanced Micro Devices, or AMD, introduced hardware-assisted support to virtualization [8]. Previously, x86 virtualization had to be done by binary translation of the virtual environment through virtualization software. Now, with the help of hardware architectural extensions, a new less privileged execution mode allowed direct execution of parts of the virtual environments code. The initial release of hardware-assisted support did however not outperform the binary translated virtualization [9] in efficiency and throughput. It was not until the second generation of hardware-assisted support, when hardware support for virtualizing memory was provided, that virtualization started to become what it is today.

The early method of Time-sharing from the 1960s was a breakthrough for computer technology and changed the way computers and servers handles resources. Many technical advances have happened since then, but the core definition of virtualization remains, "to enable a computing environment to run multiple independent systems at the same time" [1]. Using virtualization can be beneficial in many ways depending on purpose, some of the key features are:

- **Multiple operating systems** – Users can have multiple, either different or identical operating systems, running at the same time, simultaneously. This can be beneficial when, for instance, dealing with compatibility issues [1].

- **Portability** – Once configured, a VM is not bound by the machine (or its OS) it is created on. Users are not restricted by location or physical packaging and a VM can easily be imported on multiple platforms[1].

- **Infrastructure consolidation** – Virtualization can substantially lower computational resources by balancing loads between physical machines and VMs [2]. Using multiple VMs on a one physical machine can substantially reduce hardware resources. This is a widely used concept in cloud computing, enterprise servers and networks.
• Security – Because of differences in privilege levels, a VM is in many ways more isolated and does not have the same control over the physical machine it is running on [10]. This makes resources on the machine harder to access for malicious attackers and can therefore be a usable asset in computer security.

• Testing – A VM uses virtual hard disks that the OS is installed on. Once installed, the configurations can be woken up, backed up and transported, but also encapsulated in a particular state[2]. In the state, or snapshot as it is sometimes referred to as, users can experiment and do testing freely. If something goes wrong, if, for instance, a program installed incorrectly, users can revert back to the snapshot and start over. This enables flexible debugging and also allows the user to do risky operation since there is always the option to restore to a backed-up state.

2.2 Hypervisors

The term hypervisor has been around since the CP-projects of IBM [11]. Hypervisor can be described as the additional software layer that runs in between the hardware and OS. The hypervisor is what creates and monitors the virtual hardware execution environment and manages the VMs, and is often referred to as the Virtual Machine Monitor (VMM) [10]. The computer system on which the hypervisor is running VMs on is called a host. The VMs are referred to as guests and were originally defined as efficient, isolated duplicates of real machines. In 1974, Popek and Goldberg defined VMM:s by three characteristics: ”First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources”[12].

A VM running one application should, under normal circumstances, have no access to an applications running in another VM. The same goes for a VM trying to access applications or files from the host machine. Usually nonetheless, the hypervisor allows access through a shared directory and this type of approach allows users to transfer files between the host and the VM. Because of isolation, installing firmware on a guest will not affect the host OS, which is why VMs and hypervisors are valuable resources when it comes to firmware testing.
2.3 Virtualization approaches

There are different approaches and levels of virtualization. Sahoo, Mohapatra and Lath [10] classifies them as:

2.3.1 Full virtualization

The concept of full virtualization is shown in Figure 2.1. The hypervisor runs on top of the host OS and the guest OSs will run on virtual hardware provided by the hypervisor [7]. These kinds of hypervisors are classified as Type-2 hypervisors, or hosted hypervisors [13]. To achieve true full virtualization, all CPU operations done by the guest have to be reproduced by the hypervisor [7]. Because of the structure of the x86, this is both challenging and impractical. Therefore, to be considered a full virtualization, the level of virtualization needs to be so high that the VM provides a robust enough reproduction of the underlying hardware which allows the OSs of the guest to run without modification. For the hypervisor, this means providing the guest with a virtual memory, virtual devices and even virtual Bios. I/O devices are managed and shown to the guest machines with the help of imitation of physical devices through emulation, done by the hypervisor [14]. Each interaction with the guest machine, such as mouse movement or a key press is captured from the host and translated by the hypervisor. The interaction is then mimicked in the guest machine to appear as "real" interactions done by a physical device. In general, applications that perform much I/O through the hypervisor will have high performance penalties [15].

2.3.2 Paravirtualization

The affix "para" comes from the Greek and means "beside" or "alongside" [14]. Paravirtualization, refers to the communication alongside the hypervisor and the guest OS. In short, the guest OS is able to speak directly to the hypervisor. Paravirtualization involves modifying the OS kernel of the guest to permit instructions and kernel operations such as memory management, interrupt handling and time keeping, that in full virtualization would need to be binary translated. The idea is to "blur" the line between the guest OS and the hypervisor to increase performance through reducing the content switching. This is mainly done by automatically trapping privileged and sensitive calls in the hypervisor. Unlike in full virtualization where the guest OS is not aware of it being a VM [10], paravirtualization relies on the idea that most tasks
are executed in the virtual domain, but some operations, that are difficult to run in a virtual environment, are executed directly in the host domain. Since paravirtualization relies on deep OS kernel modifications, it is not as portable and flexible as full virtualization, and not every OS is compatible with such modifications.

### 2.3.3 Hardware-layer virtualization

In this approach, the hypervisor runs directly on hardware, monitoring, controlling and synchronizing the guest OSs access to hardware resources. These kind of hypervisors are referred to as Type-1 hypervisors, Bare-metal hypervisors, or native hypervisors, and runs on a separated level, under the guest OS (Figure 2.2). Hardware-layer virtualization is mainly used on server virtualization which is an extensive area and comes in many forms to meet multiple sets of requirements [2]. In its core, server virtualization relies on the masking of server resources to gain performance by making multiple, isolated VMs appear as independent instances of hardware [10]. Multiple virtual servers are sharing the workload instead of having one server doing all the work. This usually leads to efficiency and higher utilization of the hardware’s resources.

Hardware-layer virtualization should not be mistaken with hardware-assisted virtualization, which was described in the first section of this chapter as performance improving support from CPU vendors to remove the need for binary translation [9].
Hardware virtualization is not the same as hardware emulation or simulation. Emulation is the concept of an system that aims to act and behave as another system. In the book Advanced Server Virtualization [2], the authors describes emulation as a "sophisticated impersonation" where "Instructions are interpreted from the executing environment into instructions that the real, underlying environment understands". An emulator relies only on software and mimics the hardware and has high performance penalty compared to virtualization that has to run with only minor decreases in speed and performance. A simulator accepts predefined inputs and provides predefined outputs. Simulation is a less sophisticated interpreter that is, in this context, mainly used in hardware design and prototyping.

2.3.4 Operating system level virtualization

Operating system level virtualization is a virtualization method that differs from previously mentioned approaches, since it does not involve any hypervisors [3]. In this approach, the OS is modified to isolate instances of multiple OSs on a single host machine. Since there is no instruction trapping in the hypervisor, this approach can be beneficial in terms of performance and flexibility. However, since all instances run on the same host, every operation is executed on the same kernel. Thus, if one kernel crashes or is compromised, all instances will be compromised. In recent years, a variant of OS level virtualization has gained a lot of popularity using software containers [16]. The approach is often referred to as containerization and is a more lightweight virtualization approach[17]. Platforms like Docker[16] uses containers to virtualize at the OS level, allowing "protected portions of the OS". Compared to
hypervisor virtualization, containers do not need a guest OS. Instead, Docker
runs isolated individual applications or services. Multiple containers running
on the same OS are not aware that they are sharing resources since they have
their own abstracted networking layer level, and this approach is often used to
bypass compatibility issues.

2.3.5 Desktop Virtualization

Desktop virtualization is a generic term for hypervisors where the guest OS is
running on top of another OS. In a research paper called Virtualization tech-
niques and technologies: State-of-the-art [3], the authors separate desktop vir-
tualization in two main variants, Local (Client-Hosted) desktop virtualization
and Remote (Server-Hosted) Desktop Virtualization. Local desktop virtuali-
zation (LDV) is the concept of a guest OSs that runs above a host OS locally
on a personal computer with the help of a hypervisor software such as Ora-
acle’s open-source VirtualBox or VMware [16]. Remote Desktop Virtualiza-
tion (RDV) involves connecting and running a VM remotely. The idea allows
users to run applications remotely but have them appear as if they were running
on the user’s local computer. Historically this type of virtualization has only
been available for server OSs, however, there are some solutions that lets you
run virtual desktop OSs, remotely. This is done by using cloud services such
as Azure [18] or Amazon Web Services (AWS)[19]. Azure is a cloud comput-
ing platform created by Microsoft for development, testing, deployment and
data management. Azure Virtual Machines is a service which allows users to
create and interact with VMs remotely, without physical hardware that runs
it. AWS offers a similar service integrated in their platform called Amazon
WorkSpaces which differs in for instance some functionality and pricing[20].

2.4 Drivers in windows

A computer driver is a software component for communication with different
parts of a computer system [4]. This could for instance be an application that
needs to read data from a plugged in device such as a printer, a keyboard, or an
eye tracker. The application would call an OS function, and the OS would call
a function implemented by the driver. The driver has the logic to get data and
will return the data to the OS, which then will return it to the application. There
are often several drivers involved in handling such requests, layered in a driver
stack. Some of the drivers in the stack are there just to manipulate the request
and then pass it along to drivers lower in the stack, hence, a driver does not
have to be associated with a hardware device [21]. Microsoft Windows drivers can therefore be separated into two categories: User-mode drivers and Kernel-mode drivers [22]. User-mode drivers typically function as bridges between Kernel-mode drivers, applications or other OS components. A Kernel-mode driver is part of the *executive*, which is the OS components that controls processes and threads, security, I/O and Plug-and-Play. Plug-and-Play is a combined hardware and software configuration that enables computer systems to recognize hardware changes without intervening with the users. Users can add, remove and switch devices, both USB and compatible hardware from a computer systems without having to reconfigure or change anything other than an occasional reboot. Instead of loading a driver set and determine which devices they support, Windows inspect the devices that are present and then loads and calls the drivers for those devices. Drivers in Windows are written in C or C++ and follow different models depending on its objective. Some drivers are classified as Intermediate drivers [21] and can follow methods like the Device filter driver model or the Software driver model. Other drivers are classified as Highest-level drivers and follow file system models. Every driver package, regardless of classification and model has to have an INF-file and a catalog (.cat) file. An INF-file [23] has a specific structure and follows a single set of syntax rules and should be updated with a new firmware launch. An example of the INF-file structure for a mouse device driver installation can be found in figure 2.3. Each section of the INF-file is either system-defined or

![Figure 2.3: Structure of an INF-file, provided by Microsoft [23]](image)

manufacturer defined and can consist of information about version numbers,
destination directives, HID:s, pointers from one section to another section, or other information that Windows needs during installation. Other files, such as catalog files are also part of driver packages and are used to verify that the package was not altered after it was published using cryptographic hashes of each file in the driver package.

## 2.5 Eye tracking

Eye tracking is in its core a technology that measures where an individual is looking at, a given time, and what sequence an individual’s eyes are moving from one position to another. The technology has multiple applications in both scientific and commercial areas. Some of the applications, mentioned in the study *Eye Tracking for everyone* [24] are human computer interaction techniques, medical diagnoses, psychological studies and computer vision. Making a computer aware of where a user is looking can determine a user’s presence, focus, attention, consciousness, drowsiness or other types of mental states. It can also be used to enable hands-free interaction by taking advantage of where a user is looking, and use that information as a “pointer” on the screen. This enables interactions with computers and other type of devices when a user does not want or simply does not have the ability to use a keyboard and mouse as the input form. It can enhance experiences and the technology can aid people to communicate and use computers effectively in ways that historically has not been achievable. The essentials of the *Tobii* eye tracking technology [25] works in the following way:

- The eye tracker consists of cameras, illuminators and in some cases, processing hardware to interpret image streams
- The illuminators create a reflection pattern on the eyes using near infrared light.
- The cameras captures high-frame-rate images of the users eyes
- The images are processed with machine learning and other mathematical algorithms to determine the user’s eye position and estimated projected gaze point on the screen
- An application layer adds user interaction and enables user aware software functions for both *Tobii applications* and third-party uses
The focus of this research lies in the application layer together with the hardware associated software packages. For each eye tracker, different types of firmware and drivers are needed, and in order to make sure that a device installed correctly, tests has to be made. Next section describes in short, the structure of the eye tracking firmware.

2.6 The Tobii architecture

In the following section, one of the software structures for one of the distributions that is available on the market is described. Because of an NDA agreement with Tobii, no device distributions will be mentioned by name and will simply be referred to as Device 1 and so on. The software consists of five main components (Figure 2.4), the Base driver package, the Extension Driver Package, the Middleware driver package, the Tobii Experience application and the Hello Driver Package. The base driver, extension driver and the Middleware driver are all "under the hood"-components that either execute during install or in the background, post installation. The Hello driver is a bridge to the integrated Windows biometric security functionality that determines the identity of a user, making it possible to log in to Windows devices using a Tobii eye tracker. The construction of Tobii Hello is different from the rest of the stack and is not in the scope of this research. The Tobii Experience application is where the user sets up an eye tracking profile and can customize settings and features. The application also provides information about the state of the device and will tell the user if the eye tracker is not functioning properly. The experience application is part of most of the Tobii distributions will appear as a software component in the Device Tree found in Windows Device Manager.

Every physical Tobii eye tracking device has a VID and a PID. PID stands for product identifier and VID stands for vendor identifier. The USB hub driver in Windows will create a hardware ID, or HID, based on the PID and the VID. The HID is what Windows will try to match with already existing drivers in the Driver Store, and determine if a driver needs to be installed or not. The firmware can in general be installed in two ways, either by Windows update and Microsoft store, or by an offline installer. Both of them essentially work in the same way and consists of the same files, but are triggered differently. The Windows Update software is triggered, if enabled, by Microsoft and can happen when a user runs Windows Update, when a device is plugged in, or if a critical update is pushed out to the devices. The offline installer has to be obtained and triggered by the user manually. When an installation is triggered, the exact same scripts and files will execute in the same order for both the
The first thing that will run when a Tobii firmware installation is triggered is the extension driver. The extension driver is a user-mode driver and its one and only purpose is to deploy the other components, among them, the base and Middleware driver package. The base driver consists of two components, the HID Driver, which communicates with the eye tracker functionality supported by Windows, and the Platform Runtime Service. A service is essentially a small application that executes in the background of Windows. Services handles service functions and are the running communicators between a devices and the OS. The Platform Runtime Service is specific for every device distribution and also provides specific functionality for every device while the Tobii Service is comprehensive for all distributions and part of the Middleware driver package. A service can be in states such as Running, Paused or Stopped. Both Platform Runtime Service and Tobii Service has to be running in order for the eye tracker to even be turned on.

If a user runs an installation without having a device plugged in, all the scripts will run and the Driver Store will be updated. However, it is only when offline and the windows update method.

Figure 2.4: Driver architecture, provided by Tobii
a device is plugged in that windows will try to match the assigned hardware ID to a driver package in the Driver Store, start the services and create the software components. It is therefore central to use the intended hardware when testing and verifying that drivers and firmware were correctly installed.

2.6.1 Manual eye tracking installation tests

Two semi-structured interviews were done with the software Quality Assurance (QA) testers at Tobii. When a new eye tracking firmware package has been developed, it is referred to as a candidate bundle and the bundle will either be tested on a completely clean machine with a re-installed OS, or on a factory image. A factory image is a version of an OS that a computer with an integrated eye tracker will have when it is shipped from the factory to the consumers. An already released firmware is called a public bundle. When asked how update testing is handled, one of the interviewees answered that "If we have a candidate for a bundle, we see that it is set up on a clean machine [...] When we set up the update scenarios, we first clean the machine and install the latest public bundle, and then we update to the new candidate that we have". When asked why they use clean machines, the other interviewees answered that "We have to re-install it [the OS] because we cannot clean the drivers". Running tests on a machine that has had other Tobii firmware installed can yield false positives and old firmware can interfere with the newly installed firmware. Sometimes the QA testers skip to clean the machines, but only if the package is still in development phase and not a release candidate. When asked how long a clean-up takes, one of the QA testers said that in "best case, it takes half an hour to clean it, worst case, it takes several hours, depending on how fast Windows installs". Regarding using VMs, one of the QA testers explained that they "have considered it. It is a very good way to test more efficiently, and fast. The reason we have not transitioned to it is because we are stuck in our old processes and simply have not had the time to review how we could apply virtual testing".

2.7 Related work

There is an absence of studies that explicitly concerns the area of automated testing in virtual environments. There have however been research conducted in virtualization, automated testing and test evaluation separately.

Virtualization research often resolves around large networks, high performance computers or security. One example is a study called Scalability of
VM Provisioning Systems [15] conducted in 2016 by MIT Lincoln Laboratory. The study describes a methodology for accurately measuring the start-up time on high performance computers. High performance computers (HCP) uses virtualization for performance gains and the study compared the start up time difference of three open-source cloud management frameworks of VMs. The measurements in the study consisted of calculating the difference between start time and final time from a web server log after launching an increasing set of VMs. The results of the study yield that even with almost identical hardware, there is a 10x performance difference between the fastest and slowest cloud management framework. The authors conclude that using virtualization "is a compelling option for certain workloads with unique or unusual requirements". The research gives some valuable insight in terms of scientific method when measuring performance, but research arrangement differs a lot from this study since, for instance: the underlying virtualization technology used for High Performance Computers is very different from "regular" virtualization such as desktop virtualization; the study evaluates start-up time of a VM and does not involve any actual tests in VMs, in which this study aims to do.

Another related study in terms of virtualization and testing is called A Simultaneous Application of Combinatorial Testing and Virtualization as a Method for Software Testing [26]. The objective of the study was to improve the process of testing pair-wise interactions using a strategy called Orthogonal Array Testing with the help of virtualization. The study includes test cases that involved verifying the compatiblity of web applications in browsers and its support of JavaScript, cookies, ActiveX in different OSs. The authors conclude that using virtualization reduced testing resources such as time, required hardware and software configurations. Even if the test case and the test arrangement is very different form what is conducted in this research, the goal to reduce testing resources by using virtualization is similar.

One study related to automated testing called A Comparative Study of Manual and Automated Testing for Industrial Control Software [27] compares manually and automatically generated tests. The objective of the study was to investigate if both types of tests achieve similar code coverage, that is, if both tests have the same degree of code execution. The authors conclude that automated tests achieve the same code coverage as manually created test in only a fraction of the time. The study has some similarities with this study, but compares manual and automated testing, and not automated testing in a “regular” and virtual environment. The study also focuses on industrial costs which is not in the scope of this project.

In 2018, O’Shea et al. published a case study called A virtualized test au-
The goal of the study was to assess the performance and scalability of a web-based test automation framework, across multiple virtualized environments. The study differentiates virtualization and containerization and compares five test environments including a Physical machine, a single VM, four VMs running in parallel, one single container-optimized VM:s and four Container-optimized VM:s. The applications under tests were web-based applications for data center environments, and the test cases considered in the study focused on functional black box testing. The results from the study yields that the physical machine outperform the rest of the environments in both stability and time performance but does not "represent the best possible use of resources and lack in terms of operational flexibility". The research conducted by O'Shea et al. differs from this study in both structure and objective. The concept of doing automated tests in virtual environments is however similar, though web-based applications tests differs a lot in construction compared to firmware installation tests.

Another study called Automated Scalability Testing of Software as a Service [28] compares automated scalability test frameworks, all working with different types of systems and having distinct pros and cons. The study aims to precisely quantify scalability and suggests three scalability evaluation techniques. The first one compares the aggregated performance of a system while receiving different workloads at the same time as resources are being scaled. The second metric is speedup, which, as previously mentioned, is the improvement in speed of a fixed size execution task when more resources are accessible. The third metric is degradation which is "obtained by increasing the workload for a fixed system architecture".

2.8 Summary

This chapter describes the fundamental concepts of the underlying theory regarding virtualization, software drivers, eye tracking and automated testing. The chapter starts with a brief summary of the historically most important advances in virtualization to today. In order for the reader to fully grasp the study, the most important methods and approaches to virtualization are described, including Hypervisors, full virtualization, Paravirtualization, Hardware-layer virtualization, OS level virtualization and desktop virtualization. The chapter also describes some of the state-of-the-art tools that can be used in different scenarios in virtualization. Furthermore, the chapter briefly investigates what a driver is and how Windows handles drivers. The chapter describes eye track-
ing appliances and the procedure of manually testing eye tracking installation and updating.

There is an absence of studies that explicitly concerns the area of automated testing of device installation and updates in virtual environments. There exists research in performance of virtualization as well as research in testing using virtualization, but not in the area of hardware dependent testing. The purpose of this study is to fill that gap by suggesting a new method to automated testing of device installation that has not been investigated adequately or can be achieved with the current state-of-the art tools.

One of the studies called *Automated Scalability Testing of Software as a Service* mentioned in the related research section described a method of evaluating a software service using three types of scalability metrics. One of the metrics, speedup, will lay as a foundation to the evaluation of the method presented in the next chapter.
Chapter 3

Method

The following chapter describes the implementation and evaluation of the framework. The first section justifies the choice of virtualization software used for the research. The next section briefly describes how a VM can be configured in order to make it compatible with the implemented automation framework. Furthermore, the chapter continues by describing all separate components in the automation framework presenting the implementation procedure. The chapter ends with a description of how the framework was evaluated.

3.1 Choice of virtualization software

In a study called *Virtualization Technologies and Cloud Security: Advantages, Issues, and Perspectives* [29], the five most widespread x86 virtualization hypervisors that support full virtualization are listed.

<table>
<thead>
<tr>
<th>X86 hypervisor</th>
<th>Open source</th>
<th>Hypervisor type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen</td>
<td>Y</td>
<td>Native</td>
</tr>
<tr>
<td>KVM</td>
<td>Y</td>
<td>Hosted</td>
</tr>
<tr>
<td>VMWare ESX</td>
<td>N</td>
<td>Native</td>
</tr>
<tr>
<td>Hyper-V</td>
<td>N</td>
<td>Native</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>Y</td>
<td>Hosted</td>
</tr>
</tbody>
</table>

Out of the five X86 hypervisor presented in the table, two of them, VirtualBox and Hyper-V fit as a choice for Hypervisor for this research. VirtualBox was chosen as one of the candidates because the software is free of charge, open source, and compatible with almost any OS, including Windows 10 as
both a host and a guest. Hyper-V was chosen as the second candidate because its overall exhaustive Windows integration and because runs natively on Windows machines. Hyper-V was the final choice of virtualization software, mainly because of the following factors; it is not open source but is included in all Windows 10 editions besides Windows 10 Home. Hyper-V also has native support of both Azure and automation and the configuration management framework PowerShell, something that VirtualBox can support, but not without modification or use of external plug-in programs. When it comes to other functionalities, VirtualBox and Hyper-V are very similar. The main reason for choosing Hyper-V was however the performance gains it can yield compared to its peers. When developing and conducting tests with VMs, the goal is to come as close to reality as possible, that is, as close to executing software or conducting tests on a physical device rather than on a virtual one. Hyper-V is a hosted, or in other words a type-1 hypervisor[30] and relies on an approach where the host OS provides management features and drivers for hardware, but where the host and the guests run side by side (Figure 3.1). The guest and the host run in different partitions where the parent partition contains the host OS and where the child partitions are created and managed by the host, but run beside the parent. This means that a VM run directly on the computer’s hardware, making the difference between a virtualized and a physical environment smaller compared to the other hypervisor alternatives.

### 3.2 Virtual Machine setup

In most cases, VMs has to be configured in order to function and execute properly. There are many ways to initialize VMs, one could for instance allocate memory and forward an OS installation disk and install the OS on the allocated memory. Both Random Access Memory (RAM) and disk storage can be set to be fixed size or dynamically growing for virtual machines. Another approach of setting up a VM is to clone an instance of a physical computer system to a compatible hypervisor format using tools like the Microsoft utility called Disk2vhd [32] which can be used to create a Microsoft’s VM Disk, a VHD. A user can then use the snapshots of the volumes on the physical device and run them in compatible hypervisors.

For this research, a Windows 10 Enterprise-image provided by Microsoft was installed on a VM and a snapshot (or checkpoint as Microsoft refers to them) was created. The checkpoint was created after the following changes on the guest machine.
• Disable Windows privilege level called User Account Control

• Set PowerShell Execution Policy to Unrestricted

• Update Test framework called Pester for PowerShell

• Reboot machine and create checkpoint

With a working VM, and a preconfigured checkpoint that could be reverted to, implementation of the automated test framework was started.

### 3.3 Device Spawning

Since Windows requires hardware detection to fully install and run hardware related firmware, a C++ command line application was implemented. The purpose of the application is to spawn the required hardware as software components. The application is similar to a User-mode driver but only appears as a connected device when the application is running, rather than forward already present devices or drivers. The application can spawn a device and
make it appear as a Generic Software Device in the Device Tree, and closing
the application would 'disconnect' the device.

The application has two starting states, a user driven and a parameter driven
state (Figure 3.2). The user driven state happens when the application is started
without any parameters and will prompt the user to choose from a number of
predefined device names with correlating HID:s, or be asked to input a "custom"
device name and a HID. The parameter driven state use the device name
and a HID provided either by a script or by a command line argument.

![Figure 3.2: User driven starting state of Device Spawner](image)

When the application is presented with a name and a HID, either from the
user or parameter state, a function called CreateVirtualDevice (Figure
3.3) will be called. The function includes multiple packages, including
swdevice.h, which is a Windows-API for developing software devices. In
order to create a device, an object called createInfo is initialized which
includes information such as the Device Description, the size of the structure,
the instance ID, and the HID that PnP (Plug-and-Play) uses to describe the
device. Most of the members of createInfo are set to NULL, simply be-
cause they are not necessary for the purpose of the application. The object is
then passed on to the Microsoft SwDeviceCreate function as a parameter,
joined by the name of the device, an address to an empty device object, and
the root path. The root path is the ID of the device that is the parent of the
software device. In this case, the root path was set to HTREE/ROOT/0 which
is highest up in the device tree, and one of the places where the OS will look
for matching ID:s with the content of the Windows Driver Store.

The SwDeviceCreate function returns a handle that represents the de-
vice. If the parameter CreateResult of the handle is set to SUCCESS, the
software device spawned and the Device Spawner will output a success indi-
cation. The CreateResult will return false in scenarios when the HID is
invalid or already exists in the device tree, or if the Device Spawner is executed without administration privileges. Other approaches to pass the hardware dependency of the firmware installation were considered, including emulating devices with the help of the hypervisor. This approach was dismissed since it would be restricted to one specific hypervisor, harder to automated and not have the capability to run on a regular system to fulfill other purposes, beyond firmware installation with virtualization.

3.4 Automation framework

The automation framework consists of the device spawner, an installation and verification script and, an update and verification script. The scripts running on the VM were created in PowerShell. The scripts follows a model called Arrange-Act-Assert which is a common pattern for organizing test code [33] and is based on separating the unit test code section in three methods; Arrange, Act and Assert. Arrange is the part of the test that involves setting up objects and data that should be tested into the right state. Act runs the function that is being tested and the assert checks that that the act step was executed successfully. The automation script handles all installation and verification steps with the help of Pester[34]. Pester is a unit testing framework integrated in PowerShell. A pester script can consist of one or multiple test blocks, called Describe, where each test block can consist of assertion blocks, called It.
These two types of blocks hold the arrange and act sections and the code in an *It-block* block will set itself ‘unfulfilled’ if the expectation stated in the test is not met. The expectation of the test is called a *Should command*, which is the Assert section of the code. Each *It-block* is timed by Pester and will output the time and the result of the test (Passed/Failed).

To run an automated installation, the script has to be provided with a parameter that indicates what distribution it is testing, since the structure of the distributions can vary. Each distribution has its own configurations that states what name and HID the spawned device or devices should have, what services that should run after a successful installation, and in which INF-files the automation script should extract data from. Data collected from the INF-files, such as version numbers and HIDs should correlate with the data of the installed or updated components post installation, found in the Device Tree. The arrange step of the automation test sets all the data from the configuration file and the INF file in the beginning of the script.

### 3.4.1 Automated Installation

The following section describes each test section of the automation script:

**Admin prerequisition** - The first action that the script does after the arrange step is to check that the executing test script has administrations privileges of the OS. If this assert step returns false, a reconfiguration of the VM is needed since the Device Spawner demands administration privileges to operate and spawn devices.

**Verify installer folder** - The second tests checks that there exists a firmware folder that will hold all installation files, including a setup executable, INF-files, and other configuration files.

**Attempt to Spawn Devices** - The next action that is done is that the script calls the Device Spawner in the parameter driven state to spawn the device with a HID declared in the configuration file.

**Verify Spawned Devices** - This act-section will verify that a device has been spawned correctly by using a PowerShell command that gets all devices known to Plug-and-Play (in the Device Tree) and, with the help of an *Should-block*, check that the status of the spawned device is set to ‘OK’.
Adding certificates - Because of how Windows handles unsigned firmware, some certificates has to be added in order to bypass a consent message that can pop up when running the setup. The consent message will pause the installer and only continue if a user presses 'OK', which is not ideal when running automated scripts. The certificates could have been added during the preconfiguration of the VM, however, since certificates expire, adding them for each new iteration of the VM will require less configuration once they expire.

Starting the installer - This action starts a 'quiet' state of the firmware installer that will, depending on the distribution, install a certain number of drivers and applications. Each of the packages described in section 2.6 should be installed by the setup file. Executing this file, either manually or with a script, is the same as Windows Update triggering a package that was downloaded automatically when a user plugged in an eye tracker or booted up a newly installed computer with an integrated eye tracker for the first time. A PowerShell command called `Get-WindowsDriver` was added to count the number of installed drivers after the installed finished.

3.4.2 Automated Installation testing

The installation testing follows the same patterns as the automated installer and will run after the setup returned an indicator that there are no more installation jobs. The verification tests will do the following:

Extension driver verification - Depending on the distribution, the installer package can come with an extension driver, such as Tobii Experience. The name and in which INF-file the version number of the Extension driver exists in is defined in the configuration file. Since INF files follow a syntax, versions numbers are easily accessible and can be fetched in a one line PowerShell command. The driver will, when a device is plugged in, appear as a software component in the device tree. The script first verifies that the component is present, and then collects its version number from Plug-and-Play. The script will then compare the two version numbers to see that they are identical in a `Should-block`.

Verified Platform Runtime Service status - The platform Runtime service should be present and running after a device was installed or updated. The name of the Runtime Service can be found in both the configuration and the INF-file, and there exists a PowerShell command called
Get-Service that will, if the service exists, return the state of a given service. Because of how scheduling works in Windows, there might be a delay of when the service starts which demand a control flow statement that would end as soon as the Service was spawned. An Should-block checked that the status of the service was set to Running. A timer was also implemented to make sure that the script continued running after a certain amount of time.

**Fetch Runtime Service version** - The Runtime Service version is not available with a common PowerShell command and is instead extracted from a specific string in the log files located in the Windows Program Data. The Runtime Service version is not included in the INF-file or in the configuration file so the verification consisted of a Should-block that checked that there exists a version number.

**Verified Tobii Service status** - The time it takes for Tobii Service (the overall service application for every distribution) to spawn will depend on the Windows scheduler, just like for the Runtime service. A control flow statement and a timeout is therefore needed in order to wait for the service to spawn and an Should-block to check the that the status was set to Running and continue after a certain amount of time.

**Verify Middleware** - When Tobii Service is running, the Middleware package should be installed, and some registry keys should be added to Windows containing data such as names and version numbers. A Powershell command called Get-ItemProperty, followed by the directory of the registry keys collects the version number of the currently installed Middleware. If no version number is returned, an Should-block catches that something went wrong during installation.

If all of these sections returns positive It-blocks an installation executed successfully meaning that all necessary components that should be present to the OS, are present.

### 3.4.3 Updating

The automated updating script expands on the installation script and will continue in a new Describe-block, but with the already arranged variables and parameters from the previous Describe-block. There is no specific update executable, only the same kind of installer, but with a higher firmware version number and with updated components. Since the spawned device and
the certificates are still present, there is no need to repeat these steps. However, since all the components are re-installed, each verification done in previous block has to be done again. The update script includes the following sections:

**Verified update installation folder** - Same process as before, check that there exist a folder that holds the new installation files.

**Run installer** - The installer is executed in a 'quiet' state, same as before.

**Fetch update extension driver** - This *It-block* will fetch the extension driver version using the same method as in the previous *Describe-block* and compare the version present in Plug-and-Play with the version number located in the INF-files.

**Compare extension driver versions** - This *It-block* will compare the version number of the extension driver collected in the first installation, with the current extension driver version. This is the first occurrence of an update verification scenario. In the previous *It-block* it was verified that the installed application corresponds to the application stated in the INF-file. In this step, it is verified that the current application version number is the same or higher as the previous application version number. The reason the version number can be the same without failing the test case is simply because not all components in a package are updated for every new firmware. The test will however fail if the newly installed version number is lower than the previous version.

**Verified Platform Runtime Service status** - The Runtime service status should be set to *Running*.

**Fetch Runtime Service version** - The version number of this service is fetched from the log files. Since all the components are re-installed even if they are of the same version, there should now consist two version number strings in the log files, one from the first install and one for the update. If there are two, there is no need to compare the versions with one another.

**Tobii Service status** - The *Tobii Service* should be set to *Running*.

**Compare Middleware versions** - The Middleware version is found by fetching a registry value, and in the same way the extension driver versions were compared, the Middleware versions of the previously installed and the updated one are compared.
Check Buildnumber and Version number - One last test is made to check to see that the build and version number of the installer is correctly set. If the build number was updated between the old and new installer, the version number should be higher. If the build numbers between the old and new installer are the same, the version number should be the same as well. If one of these two conditions are false, a Should-block catches the fault and the test case fails.

To automate the process of running the framework with Hyper-V, a provisioning script was created. The PowerShell command `Start-VM` is used to start a preconfigured VM (section 3.2) in Hyper-V. The firmware and the framework are transferred from the host to the guest using the `Copy-VMFile` command and the `Expand-archive` is used to unpack the framework. The automation scripts is then triggered with `Invoke-Command` and as soon as the framework is done executing, the VM is terminated.

### 3.4.4 Server hosted integration

To fully automate the process of installation, updating and testing, the framework was integrated in a server hosted environment using remote desktop virtualization. The automation server tool Jenkins was used to handle the pipeline and communicate with VMs as well as handling reporting. Jenkins handles continuous integrations which means that the Jenkins manages a chain of actions that host, compile, monitor and test code in an automated fashion. Microsoft Azure was used as the underlying cloud computing service for the automation framework, and the communication between Jenkins and Azure was handled by a Azure VM Agents plugin for Jenkins. The first thing that was done to set up the server hosted environment was to create the `Windows-10-image`. One approach of exporting an already configured VHD file from Hyper-V was tested, but due to performance and some compatibility issues, this approach was abandoned. The image used was created with the help of a Windows 10 base-image provided by Microsoft from the Azure Market Place, and was modified with the help of the imaging tool `Packer`[35]. `Packer` creates images with pre-configured OSs and installs software and uses JSON configuration files that describes the base image. A provisioning tool for `Packer` called `Ansible` was invoked to install packages and utilities. Utilities like `Java 8` and `Git` were installed to make Jenkins and version handling work on the machine. `Ansible` also handled some of the configurations mentioned in section 3.2 that were needed to run the framework, such as installing `Pester`, and change the
User Account Control. After the provisioning was done, Packer created and uploaded the image to Azure.

On Azure’s side, the virtualization of the image is managed with the help of hypervisors that run multiple VMs on servers, handled by fabric controls[36]. Fabric controls are distributed applications that allocates services and monitors the server and the services in the data centers. Each fabric controller is connected to a cloud orchestration software. In this research, the Azure VM Agents plugin was used for communication with the cloud orchestration API and configured the agent that was running the image. The configuration specifies factors like size of VM, type of hard disks, max input/output operations per second (IOPS), and number of cores available to the VM. A pre-defined plan for agent resources supplied by Azure called Standard_D3_v2 was used in this research. The retention policy of the VM, that is the policy that handles what to do with the VM after it is used was set so that it would only run once.

Two Jenkins jobs were created for each device distribution that was tested, one for automated installation, and one for automated installation and updating. Each Jenkins job executed with the help of a Jenkins file containing the following operations:

Initialized libraries and git repository - Jenkins libraries were added as well as directories for the git repository that holds the framework.

Set build number - A new build number for the Jenkins job was set.

Request agent - A request was sent to Azure via the Azure VM Agents plugin. Azure would spawn a VM from the source image, and the VM would be assigned to the Jenkins job. For each job, a VM would have a randomly generated name, an IP:adress, and a work space on located on its virtual hard drive.

Checkout - The automation script repository was cloned to the work space of the VM.

Download latest builds of firmware and device spawner - Depending on the device distribution of the job, the latest development build of the firmware was downloaded for installation testing. For the update scenario, the latest released build was downloaded to be the firmware that the frameworks updates from. The latest build of the device spawner was also downloaded and added to the work space of the VM.

Execute framework scripts - Either the installation or update automation scripts was executed on the VM. The Pester results from the tests were out-
putted in the Jenkins console and written to an `xml-file` that Jenkins would read and interpret as Passed, Failed, Skipped, Pending or Inconclusive.

**Terminate VM** - The Azure VM Agents plugin would send a kill signal to Azure that terminates the VM as soon as the tests were completed.

Each job was scheduled to run once every 24 hours at midnight. The VMs handled by Azure were fully isolated meaning that all Jobs would run at the same time, in parallel. Since the IP:address were available, it was also possible to manually connect to the VM using Remote Desktop Protocol to for instance debug the framework, or run custom installation and update scenarios.

### 3.5 Evaluation

To verify that the automated framework returns the correct outputs, unit test were conducted by manually running the automation scripts in a VM, with Hyper-V. Each section was verified so that every output correlated with the state of the installed or updated firmware in the VM. For some sections, mutated states were created by for instance removing or changing firmware files, or stopping services to verify that negative states return failed test outputs and that positive states returned passed test outputs.

The framework was evaluated both using local desktop virtualization and remote desktop virtualization. These two virtualization technologies will hereafter be referred to as the client hosted and the server hosted environments. In each environment, four distributions of eye trackers were tested for both installation scenarios, and installation and update scenarios. The main metric of measurement was the time it took for a virtual environment to initialize, for the framework to execute from start to finish, and for the VM to terminate. The time data was analyzed and compared with the estimated time of manual testing, described in section 2.6.1.

Scalability was evaluated by measuring the speedup of the framework. Speedup was described in the related research, as increasing the resources available and measure how the performance improves. Speedup is often used to measure scalability in parallel systems [37] and is calculated with the following formula:

$$ S(n) = \frac{T_1}{T_n} $$

Where $n$ is the number of available processors, $T_1$ is the elapsed time of execution when running on a single processor, and $T_n$ is the elapsed time of
execution when \( n \) processors are available. The increasing resources in this research will be virtual processors and Random-access memory (RAM) available to the VM. Due to limitations in resource management of the server hosted environment, speedup was only measured for the client hosted set up.

### 3.5.1 Evaluation on a client hosted environment

The specifications for the desktop computer used to evaluate the framework can be found in Table 3.2.

<table>
<thead>
<tr>
<th>OS</th>
<th>Windows 10 Enterprise, 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>i7-7700HQ 2.80GHz, 8 Cores</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB</td>
</tr>
<tr>
<td>GPU</td>
<td>Intel HD Graphics 630, GeForce GTX 1050</td>
</tr>
</tbody>
</table>

To evaluate the framework in terms of time, a PowerShell `StopWatch` was used. The timer started at the beginning of the provisioning script and ended as soon as the automation script stopped executing. To collect data to calculate the speedup of the framework, nine checkpoints were created in Hyper-V. The checkpoints were based on the same state, but differed in how much resources were made available to the VM. The number of processors and RAM can be found in Table 3.5.1.

<table>
<thead>
<tr>
<th>Checkpoint name</th>
<th>Processors</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C1 ) (1</td>
<td>2)</td>
<td>1</td>
</tr>
<tr>
<td>( C2 ) (1</td>
<td>4)</td>
<td>1</td>
</tr>
<tr>
<td>( C3 ) (1</td>
<td>8)</td>
<td>1</td>
</tr>
<tr>
<td>( C4 ) (2</td>
<td>2)</td>
<td>2</td>
</tr>
<tr>
<td>( C5 ) (2</td>
<td>4)</td>
<td>2</td>
</tr>
<tr>
<td>( C6 ) (2</td>
<td>8)</td>
<td>2</td>
</tr>
<tr>
<td>( C7 ) (4</td>
<td>2)</td>
<td>4</td>
</tr>
<tr>
<td>( C8 ) (4</td>
<td>4)</td>
<td>4</td>
</tr>
<tr>
<td>( C9 ) (4</td>
<td>8)</td>
<td>4</td>
</tr>
</tbody>
</table>

A provisioning script was created such that each of the four distributions were tested on each of the nine checkpoints. Because of long execution times and
the overall time limitation of the research, the number of executions for each scenario and device distributions was not based on a confidence interval. The number was based on how many executions that would finish during a time period of 48 hours, which was considered as a reasonable amount of time for saturation to occur. An average for each checkpoint for each specific distribution was calculated as well as the standard deviation from the same test scenarios. 18 installation speedup scenarios and 18 updating speedup scenarios were calculated from the data where half of them measured the CPU speedup and half measured RAM speedup. An average speedup was also calculated for each checkpoint. The results of the speedup calculations should not only yield how scalable the framework is, but also imply a suitable balance of resources and performance.

3.5.2 Evaluation in the server hosted environment

The integrated clock function managed the time measurement and the timer started as soon as an Azure VM is assigned to the Jenkins job. The resources allocated for the VM from the pre-defined Azure plan Standard_D3_v2 can be found in Table 3.4.

<table>
<thead>
<tr>
<th>OS</th>
<th>Windows 10 Enterprise, 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.4 GHz Intel Xeon® E5-2673 v3, 4 Virtual CPUs</td>
</tr>
<tr>
<td>RAM</td>
<td>14GB</td>
</tr>
</tbody>
</table>

The speedup calculations described in the previous subsection should give an indication if the level of allocated resources in the cloud environment is too low, too high or sufficient.
Chapter 4

Results and discussion

The following chapter holds the result of the research. The first section contains the results from the executions on the client hosted environment and presents the speedup of the framework. The second section presents the time results of the framework running on a server hosted environment. The last section of the chapter contains the discussion of the research where the results are interpreted and where pros and cons of the research are discussed.

4.1 Results from client hosted environment

Unit tests results indicates that the automation scripts functioned accordingly to the intention of the framework. The output of the framework showed that the process of installing, updating and verifying installation and updating could be done in an automated fashion.

20 executions for each checkpoint for each device distribution finished during the 48 hour period, bringing the total number of executions to 720. The average time for running the framework on a client hosted environment can be found in Table 4.1 for installation and Table 4.2 for updating scenarios. The results indicates that there is a time gain in using the automated framework on a client hosted environment compared to conducting manual tests. In section 2.6.1 it was stated that only the process of manually cleaning an environment before running tests took, at best, about 30 minutes which is longer than any of the scenarios presented in both tables. The results also indicate that there was a time difference depending on the allocated resources for the framework, both for installation and updating scenarios.
The results found in Table 4.1 and Table 4.2 are, for each distribution and checkpoint, based on 20 executions of the same scenarios (10 for installation and 10 for updating). The standard deviation from these scenarios can be found in Figure 4.1 and Figure 4.2. The results show that the standard deviation is higher with less allocated resources, and that the standard deviation in general is higher for updating scenarios compared to installation scenarios.
The CPU speedup of the framework is presented in Table 4.3 for installation and Table 4.4 for updating scenarios. The horizontal lines on the x-axis in both tables represent the change in the number of allocated processors. The results in the table show that the speedup ratios in the two upper thirds of both tables are higher than the speedup ratios in the lower thirds of the tables. This indicates that there is a higher time gain going from one to two processors, or one to four processors, compared to going from two to four processors.
### Table 4.3: CPU speedup ratio, installation scenario

<table>
<thead>
<tr>
<th>Checkpoint</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>Avg. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 -&gt; C4</td>
<td>4,040</td>
<td>3,520</td>
<td>3,783</td>
<td>3,857</td>
<td>3,800</td>
</tr>
<tr>
<td>C2 -&gt; C5</td>
<td>3,768</td>
<td>3,343</td>
<td>3,304</td>
<td>2,990</td>
<td>3,351</td>
</tr>
<tr>
<td>C3 -&gt; C6</td>
<td>3,143</td>
<td>3,076</td>
<td>3,386</td>
<td>2,988</td>
<td>3,148</td>
</tr>
<tr>
<td>C1 -&gt; C7</td>
<td>5,499</td>
<td>4,282</td>
<td>4,249</td>
<td>4,728</td>
<td>4,690</td>
</tr>
<tr>
<td>C2 -&gt; C8</td>
<td>4,445</td>
<td>3,900</td>
<td>3,765</td>
<td>3,983</td>
<td>4,023</td>
</tr>
<tr>
<td>C3 -&gt; C9</td>
<td>3,938</td>
<td>3,501</td>
<td>3,545</td>
<td>3,446</td>
<td>3,607</td>
</tr>
<tr>
<td>C4 -&gt; C7</td>
<td>1,341</td>
<td>1,301</td>
<td>1,099</td>
<td>1,263</td>
<td>1,251</td>
</tr>
<tr>
<td>C5 -&gt; C8</td>
<td>1,180</td>
<td>1,167</td>
<td>1,140</td>
<td>1,332</td>
<td>1,204</td>
</tr>
<tr>
<td>C6 -&gt; C9</td>
<td>1,253</td>
<td>1,138</td>
<td>1,047</td>
<td>1,153</td>
<td>1,148</td>
</tr>
</tbody>
</table>

### Table 4.4: CPU speedup ratio, updating scenario

<table>
<thead>
<tr>
<th>Checkpoint</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>Avg. speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 -&gt; C4</td>
<td>2,649</td>
<td>2,167</td>
<td>2,082</td>
<td>1,914</td>
<td>2,203</td>
</tr>
<tr>
<td>C2 -&gt; C5</td>
<td>3,335</td>
<td>2,747</td>
<td>3,092</td>
<td>1,816</td>
<td>2,747</td>
</tr>
<tr>
<td>C3 -&gt; C6</td>
<td>3,109</td>
<td>2,526</td>
<td>2,094</td>
<td>2,294</td>
<td>2,506</td>
</tr>
<tr>
<td>C1 -&gt; C7</td>
<td>5,155</td>
<td>3,398</td>
<td>3,334</td>
<td>3,157</td>
<td>3,761</td>
</tr>
<tr>
<td>C2 -&gt; C8</td>
<td>4,437</td>
<td>3,511</td>
<td>3,309</td>
<td>2,929</td>
<td>3,546</td>
</tr>
<tr>
<td>C3 -&gt; C9</td>
<td>4,438</td>
<td>3,110</td>
<td>2,749</td>
<td>2,719</td>
<td>3,254</td>
</tr>
<tr>
<td>C4 -&gt; C7</td>
<td>1,902</td>
<td>1,597</td>
<td>1,563</td>
<td>1,642</td>
<td>1,676</td>
</tr>
<tr>
<td>C5 -&gt; C8</td>
<td>1,330</td>
<td>1,278</td>
<td>1,127</td>
<td>1,614</td>
<td>1,337</td>
</tr>
<tr>
<td>C6 -&gt; C9</td>
<td>1,427</td>
<td>1,231</td>
<td>1,402</td>
<td>1,185</td>
<td>1,311</td>
</tr>
</tbody>
</table>

Table 4.5 and Table 4.6 holds the result of the RAM speedup for installation and updating scenarios. The horizontal lines on the x-axis in these tables represents the allocated RAM going from 2 GB to 4 GB in the upper third of the table, going from 2 GB to 8 GB in the middle part of the table, and going from 4 GB to 8 GB in the lower third of the table. The results in the tables shows that there is little to no time gain in increasing the allocated RAM when running the framework. This was true for both installation and updating scenarios.
4.2 Results from server hosted environment

The average time of 20 executions (10 for installation and 10 for updating) of running the framework on the server hosted environment for every distribution can be found in Table 4.7. Since resource allocation management for the server hosted environment was limited, no speedup was calculated.
The results in the table shows, just like for the client hosted environment, that there is a time gain using the automated framework in the server hosted environment compared to conducting manual tests. The results in the table also shows that running the framework on the server hosted environment is faster than running it with one processor on the client hosted environment. It is in most cases slower than running the framework with two or more processors in the client hosted environment.

### 4.3 Discussion

The results showed that it is possible to use VMs for installing, updating and testing hardware specific software in an automated matter. The implemented framework is much faster than conducting manual tests, both when running the framework on a client and a server hosted environment. When comparing the results between the two tested environments, aside from running the framework on one processor, the client based environment performs better than the server based environment, that is, if all tests would run sequentially. This could have many explanations, one assumption is that the start up time for the server hosted environment was longer than for the client hosted environment. The underlying virtualization technology for the integrations differs in many ways, which will affect the overall performance of the VMs. Another explanation could be the way both integrations handled obtaining the framework and the firmware. The client hosted environment used a PowerShell command to transfer files from the host to the guest, which was a fairly fast approach. The server hosted environment obtained the files by downloading them from an internal network which, in general, was a slower approach and could have affected the total execution time.

The calculated standard deviations of the tests were high, especially for the executions that ran on one processor. The reason for this is that for some executions, the scheduled tasks of starting different services after installation and updating took a long time or simply did not start at all. There was an implemented timeout that would make the test continue after a certain amount
of waiting time. The waiting time for services to start is one of the main explanations for the overall bad performance of the first three checkpoints. The underlying reason for this could be that some of the tasks scheduled by Windows needed to be completed before starting the services, and that the required computational power for finishing these tasks were simply too low.

The CPU speedup calculations for the client hosted environment shows that there was a significant difference in running the framework on two processors instead of one. Even though the speedup ratio is higher when going from one to four CPUs, the low ratios when going from two to four CPUs indicates that the crucial performance increase occurs with the two allocated CPUs. The highest measured speedup of that scenario can be found in Table 4.3 where $D1$ had a speedup ratio of 4.040 when going from one processor to two processors (with the same amount of allocated RAM). The theoretical speedup limit of parallel systems states that doubling the amount of processors should not reduce the execution time with more than half. The reasoning for the high speedup ratio for the framework is not that it defies the laws of parallelism, but rather that it performs poorly when running on one processor, most likely due to the services not starting because of scheduling in the OS. The CPU speedup ratio when increasing the number of processors from two to four shows a small execution time gain for both installation and updating scenarios. The reason for this could be parallelism in the OS or in the installers. In terms of scalability, these results suggest that there could be a small reduction in execution time if more CPU power was added. The RAM speedup ratio shows no correlation between increasing the RAM and reducing execution time.

In terms of automation, the server hosted environment is very self reliant, partially because it is scheduled to run once every 24 hours, but also because it will automatically download the most recent builds, following the practices of continuous integration. The results from each run is published for developers and QA testers so that unexpected installation and updating behaviour is found and dealt with. The time measurements of the server hosted environment started as soon as a VM provided by Azure became available. The time in between the request to Azure and a response with an available VM is not included in the results. The published results from the continuous integration yielded that including the response period would have affected the results and that the time period could vary significantly.

Another approach for a more reliable result is to test the framework without virtualization, similar to how tests were done in a study [17], mentioned in section 2.7. Tests conducted on a physical computer with a clean OS can give
a baseline in terms of speed of the framework. The results of the test could be analyzed to see if there are some factors in the installation scenarios that differs in virtualized environments.

Ideal would be to also evaluate the scalability on the server hosted environment in the same way speedup was measured for the client hosted environment. It would however require a lot of reconfiguration that was not feasible because of the time limitation in this research. Even if the client hosted environment executed faster than the server hosted environment, the differences are modest, which could imply that the speedup of the Server hosted environment would have scaled in a similar manner. Another shortcoming of the research is that too few executions finished during the 48 hour period. To achieve a 95% confidence interval, population size and standard deviation calculations demanded an average of 27 executions for each scenario and device, for both installation and updating scenarios, which was not the case in this research. In terms of scalability measurement, this research is limited to only one metric, speedup. For a more reliable result, the two other metrics, aggregated performance and degradation presented in section 2.7 should also have been used to evaluate the scalability of the framework. Because of these shortcomings, the speedup results should only be seen as an indication of how well the framework scales, and not as a baseline for similar research.

Another important aspect to consider are the monetary costs that the cloud services comes with. Cloud services often charge customers each second that the system is running, and the price depends on the type of the service and how much resources that are allocated. The speedup results from the client hosted environment indicated that the framework could run with less allocated resources in terms of both CPU and RAM. Reducing the resources with half of the number of CPUs (from 4 to 2) and half of the allocated RAM (from 14 GB to 7 GB) should yield similar time results, but would, according to the pricing page of Azure[38], reduce monetary costs in half.

Speedup proved to be a useful metric when deciding resource allocation to minimize monetary cost. But in terms of scalability, the metric is not so comprehensive. The true scalability of the framework can not be measured with a metric and is perhaps not affected by adding more resources to reduce execution time. The true scalability lies in the automation and in the ability to easily add new device distributions to the framework. The scalability lies in being able to add more test cases as the firmware is further developed with new functionalities and features. Since the automation is set to run at midnight and since VMs in Azure will run in parallel without affecting one another, having low execution time is perhaps not as important as having stable continuous
integrations and correct outputs in the correct scenarios.

4.4 Social Sustainability and Ethics

Automating work that otherwise would have needed manual labour can be beneficial in many aspects. It is however important to be aware of the limitations of automated tests and that they are not as comprehensive or flexible as manual tests. It is crucial to assure the quality and the correctness of the tests by validation and verification. Delivering functional and well tested firmware and products is especially important for eye tracking, since some users may depend on the solutions for basic communication.
Chapter 5

Conclusions

The following chapter holds the conclusions, validity discussions and future works of the research.

5.1 Concluding remarks

This research aimed to answer the following question:

How to transition from manual to automated testing of firmware installation and update scenarios for distributed eye tracking devices?

This research has presented an approach to automate the process of installing, updating and verifying an installation of firmware using virtual machines. The approach uses an implemented framework that consists of components such as an application that mocks connected devices, automation- and pipeline scripts. The framework can run on desktop machines but is most applicable in a server hosted environment where it is part of a continuous integration that builds, downloads, installs, updates and tests the latest firmware version in a completely automated manner. The evaluation of the framework was done by measuring and comparing the time it took for the framework to execute, with conducting manual installation and updating test. Results showed that the framework completes test much faster than the manual approach. speedup was also used as part of the evaluation. Even though the measured scalability shows some tendency of scaling, increasing the allocated resources to reduce execution times might not be the most important aspect of scalability. The true scalability lies in the continuous integration and the ability to add and change device distributions in a simple and fast manner. The key value of this research is that testers and developers now can configure and run test remotely, without
being bound to specific hardware.

5.2 Validity of the results

Experiments were performed on real firmware provided by the principal of this research. The firmware is not available to the public which is a concern for the validity of this research, since the experiments cannot be replicated by an external source. To address this issue, the experiment parameters and overall structure of the firmware has thoroughly been described so that the experiments can be reproduced on similar firmware and devices.

The conclusion validity in this type of research is signified by the relation between treatment and the outcome, and the ability to assert if the conclusions, based on the obtained results from the data, are correct[39]. The main threats of the conclusion validity can either be; missed or unperceived relationships in the data, or, observing non-existent relationships in the data. To reflect conclusion validity in this research, two approaches of automated testing was implemented, tested and compared. Both of them produced similar results. When it comes to the results based on the speedup, shortcomings of the experiments are brought up in the discussion section of the research. Because of the shortcomings, the speedup results should not be considered a baseline for similar research and only give an indication of how well the framework scales.

5.3 Future works

The framework could be expanded such that the device spawner not only makes distributions appear as they are connected devices, but that it actually simulates signals from the device, such as eye movement. This would make tests more comprehensive and move the framework to another area of testing. One initial approach to this expansion could be to have physical eye tracking hardware accessible remotely to the VMs through network sharing of serial devices.

Another expansion could be to have multiple Azure versions of Windows available in the server hosted integrations. This would make the test cases broader and represent real world users and their configurations more accurately.
Bibliography


[31] Stephen Hemminger. “Hyper-V support in DPDK”. In: *DPDK Summit Userspace*.


