RUHR UNIVERSITÄT BOCHUM

RUHR-UNIVERSITÄT BOCHUM

Evaluation of Current Virtual Machine Detection Methods

Phillip Kemkes

Bachelor's Thesis – September 2, 2019.
Chair for System Security.

1st Supervisor:	Prof. Dr. Thorsten Holz
2nd Supervisor:	M.Sc. Moritz Contag
Advisor:	M.Sc. Kevin Mayer

hgi SYSSEC

Abstract

Malicious software is commonly analysed dynamically by executing it in virtual systems and evaluating its behaviour. Therefore, malware developers try to detect these artificial environments to bypass analysis. The goal of this bachelor's thesis is to create an overview of the most commonly used methods, evaluate their distribution over a big set of presumably malicious binaries and create a statement about the prevalence of Virtual Machine (VM) detection today. This is achieved by utilising a dynamic analysis environment to automatically execute the binaries and test them for known behaviour used to detect virtual systems. The conclusion is that there exists a vast amount of methods to detect a VM and that they are proven to be used in at least 2.77 % of analysed binaries. It is highly plausible that an even higher amount of malware actually uses these methods, which is still open for further analysis. This is an interesting result as today's computer systems are more and more virtualised with the rise of cloud computing services like *Amazon Web Services (AWS)* or *Microsoft Azure*, because malware would lose potential victims using these systems when employing VM detection methods.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Official Declaration

Hereby I declare that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure that this paper has been written solely on my own. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

Datum / Date	Unterschrift / Signature

Contents

1 Introduction

This chapter gives a general outline for this thesis and introduces the topic of VM detection. It describes the motivation for this work in Section 1.1. Related work that has been carried out is discussed and put into context of this thesis in Section 1.2. Also in Section 1.3 the contributions of this thesis are described. Finally the organisation of this thesis is presented in Section 1.4.

1.1 Motivation

Software can be analysed for their potential malicious intention in multiple ways. One method is static analysis in which a binary sample is examined by reviewing it without executing, for example with reverse engineering. This is a tedious job, which can be slowed down or impeded by using various obfuscation techniques [26, 29, 31]. It should be noted that any analysed binary will be referred to as *sample* in this thesis without regard of its possible malicious or benign background.

To overcome these challenges, another approach is dynamic analysis. This is performed, in contrast to the static procedure, by executing the binary in a controlled and monitored environment, observing its behaviour. Actual malicious actions are easier to detect if they are executed, becauce it is harder to hide them. All obfuscation methods have to be disbanded at some point to tell the system what to do. Ransomware[1] for instance has to edit user files in order to encrypt them which can hardly be done without the Windows Application Programming Interface (API) and system calls. Those are monitored in most dynamic analysis systems. Another example are command-and-control servers that have to be reached via some sort of network communication. This can not be done without a network interface which can be monitored as well.

One method to perform dynamic analysis is to run samples in VMs and inspect its behaviour from outside using techniques like Virtual Machine Introspection (VMI). Unfortunately, by doing this a new problem arises. A VM can never perfectly emulate a real system. There are always some aspects that show that the current instance is only a virtualised. Examples for this are timing discrepancies in comparison to a real machine [2]. These and other indications of virtual systems can

[1]Ransomware encrypts data on a victim's system and demands ransom in exchange for the decryption key.

sometimes be checked. Indicators are for instance the existence of certain drivers, device names or registry keys, referring to specific virtual machine managers [4, 12, 18, 19].

Malware developers are aware of this and therefore implement methods to detect virtual machines and evade analysis. One example is ransomware *Locky*. It uses the Read Time-Stamp Counter (`RDTSC`) instruction to count the number of cycles the system performed since the last restart. Locky combines two of those instructions to measure the cycles taken to perform two certain Windows API calls. If the amount is longer than expected, Locky assumes it is executed in a virtual environment. It stops its malicious behaviour and changes into a dormant state [27].

Another example is banking trojan *Dridex*. It checks the value of the registry key `HKLM\System\ControlSet001\Services\Disk\Enum` for strings like "VMWARE" or "VBOX". If present, the malware hides its malicious intent and goes dormant or crashes the system. Similar behaviour with a different registry key has also been observed on *Nymain downloader* [27].

1.2 Related Work

Several studies exist on finding new methods for detecting and evading virtual environments and sandboxes. These can be grouped into different categories such as detecting system artefacts [4], performing timing attacks [2] and conducting reverse turing tests [7, 21].

Besides finding new methods, researchers have developed automated analysis systems to detect evasive malware. For this generally two approaches exist. Some works focus on detecting known methods by analysing the execution traces of malware [18] or by looking for suspicious instructions, such as `CPUID` or `RDTSC` [2].

Other works concentrate on finding, but not specifically identifying, evasive behaviour by executing malware on multiple systems. They then compare the actions performed at different levels on virtual and on non-virtual systems. If a noticeable difference in execution at instruction level [16], system call level [1] or in the system state, for example changes in files, registry keys, processes, services or network activity [19] is found, the sample is classified as *evasive* but generally not further inspected.

Also some open-source virtual machine detection tools exist to test a virtual system on transparency of its artificial nature [20, 24]. These employ and demonstrate a great amount of methods, however they do not issue a statement with respect to the prevalence and reliability of these methods.

1.3 Contribution

This thesis provides a detailed overview about various VM detection and analysis evasion methods explaining their technical background and their functionality. It then creates an evaluation of the prevalence and distribution of most of these methods in today's malware by analysing 50.000 samples.

1.4 Organisation of this Thesis

In this section a brief overview of the structure of this thesis is given. In Chapter 2 the technical background is described. Also all researched VM detection methods are presented with their usage, functionality and reliability. In Chapter 3 the design goals, assumptions and definitions for this thesis are discussed. Furthermore, the implementation of the analysis system is described as well as which methods are detected by it. Following this, in Chapter 4 the limitations of the approach are discussed. Also tests conducted to analyse the reliability of the results of the implementation are presented as well as the results of the evaluation of the big set of samples. Lastly, in Chapter 5 the created results are reviewed and a conclusion is drawn. Finally, any possible future work resulting from this thesis is discussed.

2 Technical Background

This chapter provides technical background for this thesis. Initially, from a high-level perspective, the concept of dynamic malware analysis and the relevance of Windows API calls, system calls and instructions are discussed in Section 2.1. In Section 2.2 some background about VMs and their components that are utilised by dynamic malware analysis is described. Afterwards, Section 2.3 discusses the functionality of the VMI implementation *VMIProgram* that is used by *G DATA Software AG* internally. It is utilised for the dynamic analysis performed in scope of this thesis. Lastly, in Section 2.4 various methods for detecting and evading VMs and analysis systems are described.

2.1 Dynamic Malware Analysis

To analyse malware, one approach is to execute the sample and inspect its behaviour. This is generally described as dynamic malware analysis. As with most technologies in computer science, automation is highly desirable with this method. Keeping the necessity of human interaction as low as possible allows for higher efficiency. Analysts carry this out by using virtualised environments. These systems can easily and quickly be reset to their initial state. This is necessary, because to create results as deterministically as possible an analysis has to always start with the identical, clean state.

The behaviour of a sample can be characterised by its use of various types of function calls, system calls and machine instructions during execution. The methods to monitor these are discussed in the Sections 2.1.1 to 2.1.4.

2.1.1 Windows API Calls

An Application Programming Interface (API) is a collection of functions provided by an underlying or external system to be employed by developers to create programs running on that system. The Windows API enables interaction with system resources like files, devices and many more. This allows a more standardised high-level abstraction for the developer to perform actions on the system that would otherwise require deep and complex knowledge of the underlying, system-specific functions. For example the Windows API call `RegOpenKey` allows the developer to access a specified registry key with only one simple function. The Windows registry is an internal database for storing setting and configuration information.

All Windows API functions are available through Dynamic-Link Libraries (DLLs). They are included in the Windows Operating System (OS). To perform most actions in the Windows system a developer needs to access system resouces. That is generally only possible by using the Windows API. This accounts for any developer disregarding their potential malicious intentions. Therefore, it is desirable to trace the use of said functions. This can be achieved in multiple ways, which are discussed in Section 2.1.3. Hereafter the shorter term *API call* always refers to the action of calling a function from the Windows API.

2.1.2 System Calls

System calls provide an interface for programs to request actions to be performed by the system's kernel. API calls often facilitate system calls for example to create processes, read files, set timers and much more. Those interactions can solely be performed by the kernel due to security reasons.

In the Windows environment these are normally not directly called by the developer. Nonetheless, it is possible to do so. Malware developers can use this fact to circumvent analysis systems that only track API calls. How to trace system calls is discussed in the next section as well.

2.1.3 Function Tracing

One method to achieve function tracing is hooking. This is a method for intrusively intercepting the usage of certain functions with custom code for logging amongst other things. This can be achieved for example by altering the original instructions in the code of the function in memory so that before executing the original code the system executes a jump to a custom function. This would then log the use of the original function, execute any overwritten instructions and jump back to the previous code. One of the first works implementing this method was done in 1999 by Hunt and Brubacher [14]. Another method to implement function hooks would be to use compiler flags, but this is only feasible if the source code is available and the functions can be newly compiled, which is generally not the case for API and system calls.

Function hooking works for API calls as well as system calls, as long as the hooking program is executed with high enough privileges. This is necessary as it needs to access memory not owned by it. The downside of this method is that it is highly intrusive. This makes it easy to detect these methods as shown by demonstration tools and scientific research [20, 22, 24].

In contrast to this, another method to trace API calls and system calls is VMI. This is a generally non-intrusive approach outside the analysed system. VMI is discussed in detail in Section 2.3.

2.1.4 Instruction Tracing

A complete trace of all used instructions of a malware sample is technically hard to create dynamically. One of the few works on this is the analysis framework *Cobra* [30]. It operates similar to a debugger and runs in the kernel space of the analysed system. It can be used to dynamically execute and analyse samples on instruction level.

Another work on analysing instructions is using *DSD-Tracer* [18]. It combines static and dynamic analysis by statically disassembling the sample to create an instruction trace. It combines this with a behavioural profile created during dynamically executing it using the *Bochs* emulator.

A simpler but less complete approach in contrast to the aforementioned methods would be to only trace instructions that cause VM exits using VMI. These are specific to virtual environments and are described in Section 2.2.2.

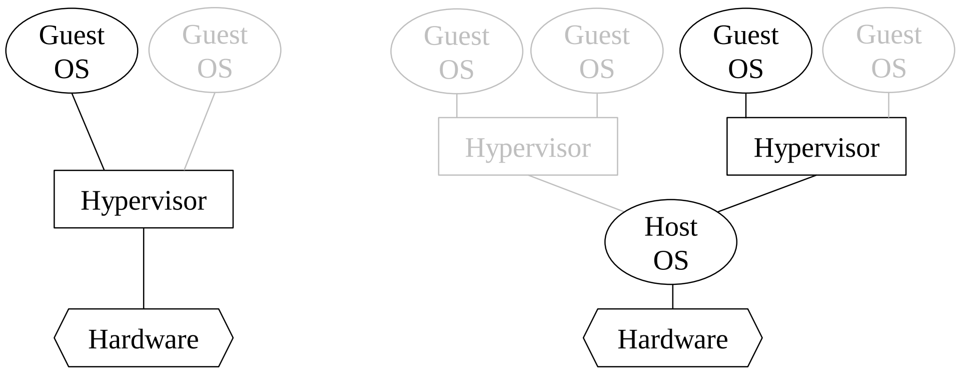
2.2 Virtual Machines

Virtual environments provide many advantages for dynamic malware analysis like the potential for efficient automation and resetting to an initial clean state. A VM has to be managed by a hypervisor, which is described in Section 2.2.1. The hypervisor handles the creation, reset and execution of the VM. Especially when instructions that produce VM exits are used. These cause the VM to pause the execution of the current thread and consign the control flow to the hypervisor. VM exits and their relevance to dynamic malware analysis is discussed in Section 2.2.2.

2.2.1 Hypervisor

A hypervisor, or Virtual Machine Manager (VMM), manages one or multiple VMs. It is used as an abstraction between the hardware and the VM. There are two types of hypervisors as shown in Figure 2.1. Type-1 or native hypervisors on one hand are directly implemented on the hardware and can freely distribute resources to the VMs. This provides greater efficiency and speed as there is no system between hypervisor and hardware. An example for this type is the *Xen* hypervisor.

A type-2 or hosted hypervisor on the other hand is installed onto a host OS. It is easier to install and more intuitive to use. But it is far less efficient due to the fact that the hypervisor itself needs to be managed by the underlying host OS like any other normal program does. An example for a type-2 hypervisor is *VirtualBox*.



(a) Type-1 native.

(b) Type-2 hosted.

Figure 2.1: The two different types of hypervisors.

2.2.2 VM Exits

When *Intel* Virtualization Technology (VT) is enabled, Intel's Virtual Machine Extensions (VMXs) are used. These are additional instructions used to manage VMs. VMX have two modes of operation: VMX non-root and VMX root operation. Typically, the guest system is running in VMX non-root operation, while the hypervisor runs in VMX root operation. Transitioning from VMX root to VMX non-root operation is performed using VM entries, the opposite is achieved with VM exits.

Certain instructions[1] cause a VM exit when used in VMX non-root operation. If this is the case, the thread is halted and the control flow is passed to the hypervisor. Two examples are the instructions `CPUID` and `RDTSC`. `CPUID` returns general information about the Central Processing Unit (CPU) of the system. `RDTSC` is the instruction used to read the automatically incremented timestamp counter register `TSC` which stores the passed CPU cycles since the system was started.

[1]A full list can be found in Intel's software developer manuals [15].

Executing instructions that cause VM exits creates more overhead and therefore consumes more time than on a real system. Therefore, they are susceptible to be used for detecting a VM with timing based attacks which are described in depth in Section 2.4.2.

2.3 Virtual Machine Introspection

This section describes technical background of VMI, a technique to dynamically analyse the runtime state of a VM. It is implemented in an internal project of G DATA Software AG called VMIProgram, which is utilised by this work for tracing API calls and system calls.

2.3.1 Events

The VMI implementation used in this work is based on an event-based approach to detect whenever the behaviour of the VM has to be analysed. There are three types of events used for this: register, memory and interrupt events.

Register events are triggered whenever certain predefined registers are changed. This is for example used for the CR3 register, of which the value is unique for every active process. It changes to the value of the currently active process. Also every change in Model Specific Registers (MSRs) like `FS` and `GS` is monitored, as here the address to the Thread Information Block (TIB) is stored. This is required to determine the currently active thread, so that it is possible to successfully monitor functions through multiple threads.

Memory events are triggered by manipulating the access rights of certain memory regions. Whenever a region is tried to be read, written or executed and the corresponding rights are not enabled on that area, a page fault occurs and the hypervisor has to handle it. This is utilised by VMIProgram to trace API function calls as described in Section 2.3.2.

Interrupt events can be triggered for example by using the debugger interrupt `Int 3`. This is utilised for tracing system calls as described in Section 2.3.3.

2.3.2 Tracing Windows API Calls

The API call tracing produces a log of all issued calls. It also logs the input and return values of certain calls that need to be configured beforehand. This is achieved by utilising the VMI memory events caused if the `NX` bit is set to 1 on the corresponding page in the Extended Page Tables (EPT). This declares a page as *not* *executable*, therefore if code from a marked region is tried to be executed, an exception is raised. The EPT is Intel's implementation of nested paging, the mechanism the hypervisor uses for managing the translation of memory pages between the guest and the physical memory.

VMIProgram controls when an exception is to be raised by setting either code regions of the currently monitored process or of the system libraries as not executable. For this it defines two states of access rights of the memory for each process, the *system DLL view* and the *program view* as shown in Figure 2.2.

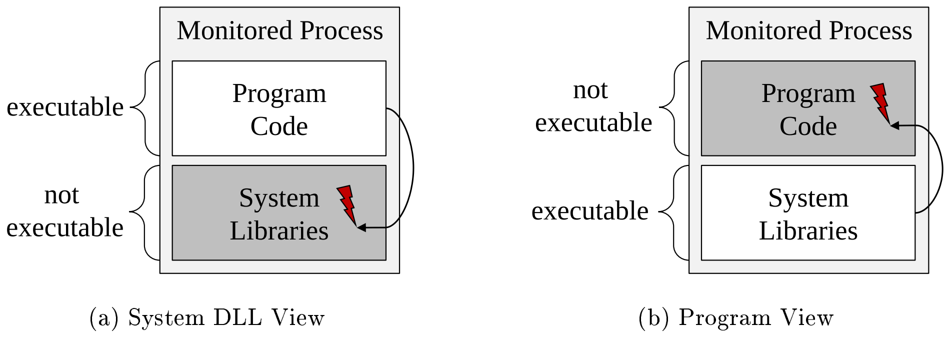


Figure 2.2: The two states of memory access rights defined in VMIProgram. The red lightning bolt indicates a page fault.

A monitored process always starts in system DLL view so that whenever it tries to execute code in the system libraries to call an API function, an exception is raised. This exception is then handled by VMIProgram. It inspects the address of the code to execute and deduces the used function from it. If information about the input parameters of the function was previously defined in the VMIProgram, it extracts these parameters from registers and stack, dereferences pointers if necessary and logs the use of the function. It then changes the access rights configuration to the program view and hands the control back to the VM. Afterwards, the code of the program is set to *not executable*. Now the API call is executed and when it returns to the original process code, another exception is raised. Again VMIProgram handles this by extracting the return values from registers and stack, changing the access rights configuration back to system DLL view and handing control back to the VM.

2.3.3 Tracing System Calls

In contrast to the non-intrusive approach of tracing API calls by leveraging the access rights, system calls are traced in an invasive manner. For this the system call `KiSystemService` is used. This is the kernel function that takes the parameters which were previously pushed onto the stack by the calling program. It then calls the system call with these parameters.

As `KiSystemService` is a central function when using system calls, VMIProgram intercepts this. It replaces a line of code in the beginning with the debugger instruction `Int 3`. This causes an interrupt, which has to be handled by the hypervisor and then respectively by VMIProgram. It extracts the parameters, logs the system call and replaces the interrupt with the original instruction. It then issues a single-step, which instructs the CPU to only execute the next instruction and hold afterwards. Then VMIProgram restores the instruction that was just executed with the `Int 3` interrupt again and hands the control back to the VM. The latter does not notice that the call has ever been intercepted.

If the VM tries not to execute but to read the code of `KiSystemService`, VMIProgram intercepts this by using the memory access rights to create an interrupt. It then replaces the `Int 3` interrupt with the original instruction so that the VM does not notice that it was manipulated. After the read instruction is executed, VMIProgram restores the `Int 3` interrupt.

2.4 Virtual Machine Detection and Evasion Methods

This section describes various methods to detect or evade virtual analysis systems that utilise VMs and their characteristics. These methods can be grouped into three categories. When malware is scanning for system artefacts, it tries to find traces left by hypervisors in the guest system. These indicators are described in detail in Section 2.4.1. Malware can also try to use the timing differences between virtualised and non-virtualised systems as shown in Section 2.4.2.

The last group describes malware that utilises interaction tests to determine if the system is used by a human. These can be classified as reverse turing tests, as the roles of a classic turing test are changed in this case. The computer and not the human is the judge of the humanity of the subject. This is similar to *CAPTCHAs*, commonly employed by websites to protect against automation. These tests are described in Section 2.4.3.

It should be mentioned that the results of these methods often indicate the presence of a VM with different degrees of probability. It is possible that a positive result of some methods could also be caused by irregularities of a non-virtualised system. Therefore, malware could execute multiple methods to detect a VM. Henceforth methods that indicate the presence of a VM with a high probability are called *reliable* methods, while methods that only offer a low probability are called *unreliable*. For example a method that finds an obvious string left in the system by a hypervisor is reliable, while a method only depending on disk size differences is considered unreliable.

Most code examples in this section are written in simplified C++, as some necessary and preliminary components like type casts, variable declarations and function calls are omitted to increase the readability and focus on the VM detection method. These examples are also influenced by the open source VM detection demonstration tools *al-khaser* and *paranoid fish* (*pafish*) [20, 24].

2.4.1 Detecting System Artefacts

Hypervisors often generate system artefacts that can be detected. These can be very low-level for example the addresses stored in certain registers. But some artefacts can also be very high-level, like the existence of telltale registry keys or services. In this section various detection methods utilising these system artefacts are discussed.

Low-Level Detection

The Interrupt Descriptor Table (IDT), Local Data Table (LDT) and Global Descriptor Table (GDT) registers can be inspected as low-level indicators. The IDT stores information about the correct response to interrupts and exceptions. The LDT is unique for every process and stores various types of descriptors. The GDT exists only once for the whole OS. It stores for example descriptors of memory segments like the code and data regions.

The base addresses to these tables are stored in their respective registers IDTR, LDTR and GDTR. As these have to be different on the guest system in contrast to the host system, they generally have different addresses. In the past this has been separately observed by Rutkowska and by Klein on older VMWare and *VirtualPC* systems [17, 28]. To access these registers the user space instructions `SIDT`, `SLDT` and `SGDT` are sufficient. The effectiveness of this method can be questioned though, as this is dependent on the implementation by the hypervisor. Quist and Smith have already shown the unreliability of the IDT and GDT methods about two years later [25]. Nonetheless this is still implemented in the VM detection demonstration tool al-khaser.

Very similar to the aforementioned method is Omella's approach to check the Task Register (TR) [23]. This register is only available on 32-bit infrastructure and stores a pointer to the Task State Segment (TSS) of the current task. In his studies the value returned from the respective user space instruction `STR` was different on VMs than on non-virtual machines. The TSS has to be virtualised as well and therefore differs with the address returned on a non-virtualised system.

The way VMWare's I/O port is communicating with the hypervisor can be used as another detection method. To issue a command the register `EAX` has to be set with a specific value. `EBX` needs to hold the command parameters. `ECX` then has to hold a command number and `EDX` has to be set to the corresponding Command Line Tools port number.

After filling the parameters the instruction `IN` has to be used to receive the output of the issued command. If this is done on a VMWare system, the specific value, which was previously stored in `EAX`, is stored in register `EBX` afterwards. This can be checked for and used as an indicator for the VMWare VM. This process is shown in an assembly example in Listing 2.1. It has been observed on malware *Necurs* as well as the use of the undocumented `VPCEXT` instruction, which is only implemented in the VirtualPC hypervisor. If this does not generate an exception, it was clearly executed on a VirtualPC VM [27].

```
mov    eax, 564D5868h ;magic number, this equals to 'VMXh'
mov    ebx, FFFFFFFFh ;command parameters
mov    ecx, 0Ah       ;command number
mov    edx, 5658h     ;hypervisor port number, this equals to 'VX'
in     eax, dx
cmp    ebx, 564D5868h ;if EBX holds the value 'VMXh', this is running on a VMWare VM
```

Listing 2.1: Assembly example detecting a VMWare VM using VMWare's I/O port.

Network and Device Based Detection

A more high-level method is to check information about the network adapters like names and Media Address Control (MAC) addresses. For both of these the two API calls `GetAdaptersInfo` and `GetAdaptersAddresses` are usable. In Listing 2.2 a code example using `GetAdaptersInfo` is shown. Both functions return a linked list in either `AdapterInfo` or `AdapterAdresses`. Here the malware can check for example for adapter names like "VMWare" or for MAC addresses starting with `08:00:27` for VirtualBox VMs or with `00:16:3E` for Xen VMs.

```
GetAdaptersInfo(pAdapterInfo, &ulOutBufLen);
while (pAdapterInfo)
{
    if (StrCmpI(pAdapterInfo->Description, "VMWare") == 0
        || (pAdapterInfo->AddressLength == 6 && memcmp(pAdapterInfo->Address, "\x00\x16\x3E",
            3) == 0))
    {
        // If either the name or the MAC address matches, this is running on a VM
        bIsVM = true;
        break;
    }
    // If VM is not found, cycle through list until nullptr
    pAdapterInfo = pAdapterInfo->Next;
}
```

Listing 2.2: Simplified code example checking for a VMWare network adapter name and Xen MAC address using `GetAdaptersInfo`.

Alternatively, malware can check for MAC addresses using Windows Management Instrumentation (WMI) queries. WMI is Microsoft's implementation of Web-Based Enterprise Management (WBEM), which is utilised to gather management information from systems with a standardised method. The query structure is similar to common database query languages like Structured Query Language (SQL). It is possible to check for MAC addresses using the WMI query in Line 1 of Table 2.1 at the end of this section. To issue this and any other WMI query the API call `IWbemLocator::ExecQuery` is used.

Other indicators are potentially existing network shares that are custom to certain hypervisors. These can be checked using the API call `WNetGetProviderName` to see if this returns the provider name "VirtualBox Shared Folders" for the network type `WNNC_NET_RDR2SAMPLE`.

Malware can also check for general device names that indicate a VM. This can be achieved by using the Windows API function `CreateFile` or the system call `NtCreateFile`. If it does not return an error when using it with device names typical for VMs, the check has detected the virtualised system. A simplified code example can be found in Listing 2.3. This can also be achieved by using the WMI query in Line 2 of Table 2.1 at the end of this section.

```
hFile = CreateFile("\\\\.\\HGFS", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL);
if (hFile != INVALID_HANDLE_VALUE)
{
   bIsVM = true;
}
```

Listing 2.3: Simplified code example checking for a VMWare device name using `CreateFile`.

Disk Property Based Detection

As another system artefact, the disk can be analysed in multiple ways. Most virtual analysis systems only get assigned a small amount of disk space to keep the hardware costs as low as possible. Therefore, malware can check the size and conclude that, if it is smaller than a certain amount, it is running in a VM. The definition of that threshold depends solely on the malware's implementation. The size of the disk can be checked using the API calls `DeviceIoControl` and `GetDiskFreeSpace` and the system call `NtDeviceIoControl`, see Listing 2.4. The disk size can also be checked by using the WMI query in Line 3 of Table 2.1 at the end of this section.

Besides the size also the disk's hardware ID can be checked for any strings indicating a VM like "vbox" or "virtual". This can be done by using the API call `SetupDiGetDeviceRegistryProperty`, see Listing 2.5.

```c
// Define a minimum hard disk size of 80 GB in bytes.
minHardDiskSize = (80ULL * (1024ULL * (1024ULL * (1024ULL))));
// Get disk size of current disk.
GetDiskFreeSpaceEx(NULL, NULL, &totalNumberOfBytes, NULL);
if (totalNumberOfBytes.QuadPart < minHardDiskSize)
{
    bIsVM = true;
}
```

Listing 2.4: Simplified code example checking the disk space using `GetDiskFreeSpaceEx`.

```c
// Create a handle to all present devices.
hDevInfo = SetupDiGetClassDevs((LPGUID)&GUID_DEVCLASS_DISKDRIVE, 0, 0, DIGCF_PRESENT);
// Enumerate through all devices.
for (i = 0; SetupDiEnumDeviceInfo(hDevInfo, i, &DeviceInfoData); i++)
{
    // Get the hardware ID and save it into the buffer.
    SetupDiGetDeviceRegistryProperty(hDevInfo, &DeviceInfoData, SPDRP_HARDWAREID,
        &dwPropertyRegDataType, (PBYTE)buffer, dwSize, &dwSize)
    // If the buffer contains any of these strings, the system is most likely a VM.
    if ((StrStrI(buffer, "vbox") != NULL) ||
        (StrStrI(buffer, "vmware") != NULL) ||
        (StrStrI(buffer, "qemu") != NULL) ||
        (StrStrI(buffer, "virtual") != NULL))
    {
        bIsVM = true;
        break;
    }
}
```

Listing 2.5: Simplified code example checking the disk's hardware ID using `SetupDiGetDeviceRegistryProperty`.

It can also be extracted from the disk if the system was booted from a Virtual Hard Disk (VHD) container. This is a common file format for virtual disks. This can easily be achieved by using the Windows API call `IsNativeVhdBoot` which returns `true`, if the system was booted from a VHD container.

System Memory Based Detection

Apart from the disk size, the system memory size can also be inspected as an indicator for a VM, as it is often kept as small as possible to save hardware costs. It can be checked using the API call `GlobalMemoryStatus`, see Listing 2.6.

Another method to check the system memory size is to use any of the WMI queries in Lines 4 to 6 in Table 2.1 at the end of this section.

```
// Define a minimum system memory size of 1 GB in bytes.
ullMinRam = (1024LL * (1024LL * (1024LL * 1LL)));
// Create a buffer structure that is going to hold the information.
MEMORYSTATUSEX statex = {0};
// Fill the buffer.
GlobalMemoryStatusEx(&statex);
if (statex.ullTotalPhys < ullMinRam)
{
    bIsVM = true;
}
```

Listing 2.6: Simplified code example checking the system memory size using `GlobalMemoryStatusEx`.

CPU Property Based Detection

The most direct and low-level method to see, what CPU the current OS is running on, is to use the instruction `CPUID` with any of the following input parameters in register `EAX`:

- `EAX = 0x0`

  This returns the highest implemented calling parameter for `CPUID` on this system in `EAX`. It also returns the CPU's manufacturer ID string in `EBX`, `ECX` and `EDX`. The last fact can be checked, as it can show the presence of a virtual system. If the system is running with a Xen hypervisor for example, the returned ID string is "XenVMMXenVMM". This is also true for most hypervisor manufacturers.

- `EAX = 0x1`

  This returns general information about the CPU in `EAX` and its feature flags in `EBX`, `ECX` and `EDX`. Relevant for detecting virtual systems is the highest bit in `ECX`. This is the *hypervisor present* flag. It is set to 1 if the system is running with a hypervisor.

- `EAX = 0x40000000`

  This is an input parameter reserved for hypervisors. Most hypervisors return the hypervisor vendor string to this, similar to `EAX = 0x0` in `EBX`, `ECX` and `EDX`.

- `EAX = 0x80000002, EAX = 0x80000003, EAX = 0x80000004`

  If supported, this returns the processor brand string in `EAX`, `EBX`, `ECX` and `EDX`. To get the full 48 byte string the three input values have to be used consecutively. A legit Intel CPU for example would return "Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz", while the emulator Qemu returns "QEMU Virtual CPU".

- `EAX = 0x8FFFFFFF`

  If `CPUID` with `EAX = 0x0` showed that the current CPU is manufactured by AMD, in some models this input value is implemented as an easteregg. It returns the string "IT'S HAMMER TIME" in `EAX`, `EBX`, `ECX` and `EDX`. This would somewhat disprove the presence of a VM, though it is only implemented on AMD's K7 and K8 CPUs. The Bochs emulator for example did not implement that easteregg as shown by Ferrie [8].

It is also common to assign only one CPU core to the VM to save hardware costs. This can be checked by malware for example by using the API calls `GetSystemInfo` and `GetNativeSystemInfo`. These functions fill a `SYSTEM_INFO` structure of which the property `dwNumberOfProcessors` can be accessed. If this is equal to one, it is an unreliable indicator for a VM. The WMI query in Line 7 of Table 2.1 at the end of this section can also be used to detect the amount of available CPU cores by accessing the property `NumberOfCores`.

Besides this, more CPU features can be analysed to detect a VM. Using the query in Line 8 of Table 2.1 at the end of this section to check the processor ID, the resulting value equals to `NULL` if checked on a VirtualBox system. Therefore, this can be used as an indicator as well.

The amount of processor cores can also be checked without any API or system call. It can be achieved with one of the intrinsic functions `__readfsdword` or `__readgsqword` depending on the processor architecture, i. e. if it is a 32-bit or a 64-bit system.

An intrinsic function is, in contrast to other functions, not stored in a library, but built into the compiler. It substitutes the call with a predefined set of instructions. In this case, it is used to read the TIB, which holds information about the currently active thread on a Windows system. On a 32 bit system it is stored in the `FS` register, on a 64 bit system in the `GS` register respectively. At offsets `0x30` in `FS` and `0x60` in `GS` the pointer to the Process Environment Block (PEB) is stored.

The PEB is a data structure in Windows that holds various information about the process and the system environment, which is mostly used by the OS. At offset `0x64` on a 32 bit system and at offset `0xB8` on a 64 bit system it holds a pointer to the value of the field `NumberOfProcessors`, which states the amount of CPU cores. This concludes to a similarly possible check as aforementioned, see Listing 2.7.

```
// On a 32 bit system:
ulNumberProcessors = __readfsdword(0x30) + 0x64;
if (*ulNumberProcessors < 2)
{
    bIsVM = true;
}
// On a 64 bit system:
ulNumberProcessors = __readgsqword(0x60) + 0xB8;
if (*ulNumberProcessors < 2)
{
    bIsVM = true;
}
```

Listing 2.7: Simplified code example checking the amount of CPU cores by reading the PEB.

With API call `GetPwrCapabilities`, another aspect of the CPU can be checked to identify a VM. It returns the available power states for the system. Most virtual systems do not support the power states S1 to S4, which represent sleep and hibernate states. They also usually do not support thermal zones. A code example showing the usage of `GetPwrCapabilities` can be found in the following Listing 2.8.

```
GetPwrCapabilities(&powerCaps);
// If the system does not support power state S1 to S4 and neither thermal zones, it is most
     likely a VM.
if (((powerCaps.SystemS1 | powerCaps.SystemS2 | powerCaps.SystemS3 | powerCaps.SystemS4) ==
     false) && (powerCaps.ThermalControl == false))
{
    bIsVM = true;
}
```

Listing 2.8: Simplified code example checking the power capabilities of the CPU using `GetPwrCapabilities`.

Similar to this, the current temperature and the amount of CPU fans can be checked using the WMI queries in Lines 9 and 10 of Table 2.1 at the end of this section. This has been observed to be implemented by the Remote Access Trojan (RAT) malware *GravityRAT* by Fortuna [9].

DLL and Firmware Based Detection

It is also possible to check the currently loaded DLLs for any indicators of a sandbox system like the module `sbiedll.dll` used by *Sandboxie*. This can be checked with the Windows API call `GetModuleHandle` and a DLL as input. If that does not return `NULL`, it means that this DLL is not present in the system. It is a reliable indicator that the malware is running in a sandbox system.

GetProcAddress on the other hand can be used to detect the availability of normally disabled legacy API calls as shown in a blog post [11]. If this is for example used with kernel32.dll as DLL and wine_get_unix_file_name as function to be searched for and it returns a valid address, the presence of the *Wine* emulator is proven. This can also be used for other legacy and undocumented calls like IsNativeVhdBoot or NtQueryLicenseValue.

System artefacts can also be found in the file system. Malware can use the Windows API call GetFileAttributes as well as the system call NtQueryAttributesFile to check if certain files or directories are present in a system, see Listing 2.9.

```
// Check for one of many possible files indicating a VM.
dwAttrib = GetFileAttributes("C:\system32\drivers\VBoxMouse.sys");
// If file exists and is not a directory, this is running on a VirtualBox VM.
if ((dwAttrib != INVALID_FILE_ATTRIBUTES) && (dwAttrib & FILE_ATTRIBUTE_DIRECTORY))
{
    bIsVM = true;
}
```

Listing 2.9: Simplified code example checking the existence of a file using GetFileAttributes.

Another method to detect VMs is to search for certain strings in firmware tables. This can be achieved with the API calls EnumSystemFirmwareTables and GetSystemFirmwareTable as shown in the simplified code example in Listing 2.10.

```
// Get all firmware tables and write their names into a buffer.
tableSize = EnumSystemFirmwareTables("ACPI", tableNames, 4096);
tableCount = tableSize / 4;
// Enumerate through all table names
for (DWORD i = 0; i < tableCount; i++)
{
    // Try to find certain strings in that firmware table.
    GetSystemFirmwareTable("ACPI", tableNames[i], firmwareTable, 4096);
    if ((find_str_in_data("VirtualBox", firmwareTable)
    || (find_str_in_data("BOCHS", firmwareTable))
    {
        bIsVM = true;
    }
}
```

Listing 2.10: Simplified code example checking firmware tables for certain strings using GetSystemFirmwareTable.

Processes and Services Based Detection

It is also possible to check the currently running processes for any that indicate the presence of a VM. This can be achieved by using the API calls CreateToolhelp32Snapshot, Process32First and Process32Next as shown in the simplified code example in Listing 2.11. If any processes like `vboxservice.exe` of VirtualBox or `xenservice.exe` of Xen are present, it is a reliable indicator for a VM.

```
// Create snapshot of all running processes and threads.
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
// Get first process and compare the name. This is necessary before using Process32Next.
Process32First(hSnapshot, &pe32);
if (StrCmpI(pe32.szExeFile, "xenservice.exe") == 0)
{
    bIsVM = true;
}
// Enumerate through all other processes and compare the name.
while (Process32Next(hSnapshot, &pe32))
{
    if (StrCmpI(pe32.szExeFile, "xenservice.exe") == 0)
    {
        bIsVM = true;
        break;
    }
}
```

Listing 2.11: Simplified code example checking currently running processes for a Xen specific process using `CreateToolhelp32Snapshot`.

Analysing the currently active processes can also be achieved with the WMI query in Line 11 of Table 2.1 at the end of this section. This was was also found by Hund in a malware sample utilising Microsoft *Word* macros [13].

Besides processes, also the currently registered services can be checked by using the API calls `OpenSCManager` and `EnumServicesStatus`. The returned array of services can be analysed for any strings that indicate a VM like for example “VBoxSF” for VirtualBox or “vmmemctl” for VMWare.

Registry Key Based Detection

The general existence or the value of certain registry keys can also be indicators of a VM. This can be achieved with the use of the API calls `RegOpenKey` and `RegQueryValue` or respectively with the system calls `NtOpenKey` and `NtQueryValueKey`, see Listing 2.12.

Miscellaneous Detections

It is also possible to check the names of currently existing windows using the API call `FindWindow`. If a window with class “VBoxTrayToolWndClass” or with name “VBoxTrayToolWnd” can be found, it is a reliable indicator for a VirtualBox VM.

```
// Check if key exists and can be opened. If so, this is a VM.
if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, "HARDWARE\\ACPI\\DSDT\\VBOX__", NULL, KEY_READ, &
    hkResult) == ERROR_SUCCESS)
{
   bIsVM = true;
}
// Open key and inspect value. If it contains "VBOX", this is a VM.
RegOpenKeyEx(HKEY_LOCAL_MACHINE, "HARDWARE\\Description\\System", NULL, KEY_READ, &hkResult);
RegQueryValueEx(hkResult, "SystemBiosVersion", NULL, NULL, lpData, &cbData);
if (StrStrI(lpData, "VBOX") != NULL)
{
   bIsVM = true;
}
```

Listing 2.12: Simplified code example checking registry keys and values using `RegOpenKeyEx` and `RegQueryValueEx`.

The undocumented API call `NtQueryLicenseValue` with "Kernel-VMDetection-Private" as input in the parameter `LicenseValue` is a VM detection method built into Windows. If this returns anything else but zero, a VM is detected.

Besides that, malware can use the WMI queries in Lines 12 to 15 of Table 2.1 to analyse the probability of the presence of a VM by analysing the system log, the Basic Input/Output System (BIOS) serial number and the model and manufacturer information found in class `Win32_ComputerSystem`.

The path of the executable file of the current process can also be checked using the Windows API function `GetModuleFileName` with `NULL` as handle input. Analysis systems sometimes rename the executable of the malware or have specific file paths inside the VM. If this contains any strings like for example "sandbox", "virus" or even the hash value of its own binary, the malware can detect this and react accordingly. Strictly speaking, this is not only a method for VM detection, but it can generally detect dynamic analysis systems. In most cases though, as described in Section 2.1, these employ VMs.

Table 2.1: WMI queries described in Section 2.4.1. *Property* denotes which property has to be inspected. *Check* denotes which check has to be performed to identify the possible presence of a VM.

#	**Query** – `SELECT * FROM ...`	**Property**	**Check**
Network and Device Based Detection			
1	`Win32_NetworkAdapterConfiguration`	`MACAddress`	Any MAC addresses that indicate a VM?
2	`Win32_PnPEntity`	`DeviceId`	Any devices that indicate a VM?
Disk Property Based Detection			
3	`Win32_LogicalDisk`	`Size`	Disk size too small for a non-virtualised system?
System Memory Based Detection			
4	`Win32_MemoryArray`	`EndingAddress`	System memory size too small for a non-virtualised system?
5	`Win32_MemoryDevice`		
6	`Win32_PhysicalMemory`	`Capacity`	
CPU Property Based Detection			
7	`Win32_Processor`	`NumberOfCores`	Equals to 1?
8	`Win32_Processor`	`ProcessorID`	Holds the value `NULL`?
9	`MSAcpi_ThermalZoneTemperature`	`CurrentTemperature`	Equals to zero?
10	`Win32_Fan`	–	Returns a class with zero fans?
Processes and Services Based Detection			
11	`Win32_Process`	`Name`	Any processes that indicate a VM?
Miscellaneous Detections			
12	`Win32_NTEventlogFile WHERE FileName = 'System'`	`Sources`	If the log is not empty, check for sources like “vboxvideo” or “VBoxVideoW8”.
13	`Win32_BIOS`	`SerialNumber`	Values like “VMWare” or “Xen”?
14	`Win32_ComputerSystem`	`Model`	Values like “HVM domU”?
15		`Manufacturer`	Values like “innotek GmbH”?

2.4.2 Timing Attacks

Besides directly accessing system artefacts as described in the previous section, the timing differences between a physical and a virtual machine when using certain functions and instructions can be used to detect the latter. But also the fact that analysis systems often use only a limited runtime for each sample and that this runtime is kept as low as possible can be abused. This is done to be able to analyse more samples in a given time period. Malware can stall the execution of malicious code to exceed the runtime in analysis systems. This does not affect their effectiveness on a real victim's system though, as these are often kept alive much longer, but their malicious intent can be hidden from analysis systems that way. Methods to achieve these timing attacks are described in this section.

Timing attacks also have low- and high-level approaches similar to detecting system artefacts. The instruction `RDTSC` can be used to measure the current time. This returns the CPU cycles passed since start of the system. With two consecutively issued instructions the time used for anything in between can be calculated. This is especially interesting for instructions that cause the VM to hand over the control to the hypervisor with a VM exit. This costs more time than when executed in a non-virtualised system which does not cause a VM exit. A code example showing this method with the instruction `CPUID` can be found in Listing 2.13.

```
// Measure elapsed CPU cycles before and after usage of possible VM exit with CPUID using
    intrinsic functions.
tsc1 = __rdtsc();
__cpuid(cpuInfo, 0);
tsc2 = __rdtsc();
// If CPUID took more than 1000 CPU cycles, a VM exit was most probably caused, what means
    that this is running on a VM
if (tsc2 - tsc1 > 1000)
{
    bIsVM = true;
}
```

Listing 2.13: Simplified code example demonstrating the use of `RDTSC` to measure time caused by a possible VM exit.

It has also been observed that the ransomware Locky uses a similar method with the two API calls `GetProcessHeap` and `CloseHandle` [10]. On a non-virtualised system `CloseHandle` should be at least 10 times faster than `GetProcessHeap`. On a VM this ratio is reversed. It might take at least 10 times longer because of the way the TIB is virtualised. Locky uses this to detect a virtual environment as shown in the code example in Listing 2.14. This method is not completely reliable as it sometimes yields false positives, meaning it sometimes declares non-virtualised systems as VMs. Locky's developers tried to mitigate that at least by a small portion by performing this test ten times consecutively and only accepting the positive result, if it was returned every time. False positives are not highly critical to this type of malware though, as its developers distribute it as much as possible. It would be more desirable to stagger the analysis as much as possible rather than losing a few potential victims of the ransomware.

```
tsc1 = __rdtsc();
GetProcessHeap();
tsc2 = __rdtsc();
CloseHandle(0);
tsc3 = __rdtsc();
// If <time taken by CloseHandle> is at least ten times bigger than <time taken by
    GetProcessHeap>, this is most probably running on a VM.
if (tsc3 - tsc2) / (tsc2 - tsc1) > 10)
{
    bIsVM = true;
}
```

Listing 2.14: Simplified code example comparing the cycles taken by `GetProcessHeap` and `CloseHandle` using `RDTSC`.

Sometimes analysis systems can try to circumvent long delays caused by malware by forcing the continuation of the execution. This can for example be achieved by returning earlier than expected from the use of the API call `Sleep`. Malware can try to detect these differences to identify an analysis system by inspecting the cycles passed with `RDTSC`. If the calculated amount of cycles does not fit the expected delay time, the altered behaviour is detected. Instead of `RDTSC` the API call `GetTickCount` can be used in the same manner. This of course only works if `RDTSC`'s or `GetTickCount`'s return values are not altered as well.

If the `Sleep` function was altered by reducing the time to wait, it is possible to still create a long delay by using multiple short `Sleep`s consecutively in a loop. This is also possible with any other function that takes up some computation time. This of course can be countered again by the analysis system with various methods altering the original control flow of the malware, but this would exceed the scope of this thesis.

Developers of analysis systems can also try to harden their analysis system against timing attacks by making certain API calls for creating delays not available or change their behaviour. This in turn can be checked by malware. The VM detection demonstration tool al-khaser checks the availability of the undocumented function `NtDelayExecution`. If this returns a negative result, the malware can assume it is running inside an analysis system. al-khaser also tries to create timers using both `SetTimer` and `timeSetEvent`. If these API calls return `NULL` as ID numbers for the timers, the malware can assume the functionality is suppressed. Malware can also create an event that can be waited for using the API call `CreateEvent`. If this returns `NULL`, the function was most probably altered.

Similarly, it is also possible to create timers using `CreateWaitableTimer` with the corresponding functions `SetWaitableTimer` and `WaitForSingleObject`. Also `CreateTimerQueue` with `CreateTimerQueueTimer` can be used. If any of those functions behave differently than expected, the malware can assume it is being dynamically analysed. But if any of the aforementioned functions did work in their intended way, the malware can freely create delays at the beginning of the execution to circumvent its analysis completely with its often short execution time.

Another possibility to create a delay is to send an Internet Control Message Protocol (ICMP) echo request using the API call `IcmpSendEcho` as observed by Cisco's Talos Intelligence Group [3]. For this first a handle has to be created using `IcmpCreateFile`. If this returns an error, it can be assumed the function has been blocked by an analysis system. But if it works, the echo request can then be sent and a timeout can be specified. The function waits either for a response or for the time specified to run out. With using an Internet Protocol (IP) address that never responds, the malware waits similarly to the aforementioned functions to create a delay.

2.4.3 Reverse Turing Tests

Malware can also perform tests to check if an actual human is using the system it is running on. This section describes various checks to achieve this. One simple method is to analyse how long the system has already been running. In automated analysis systems a VM is often booted with the malware starting as soon as possible. After the analysis, the system's state is usually reverted and the system is reset to fall back onto a clean state. The API function `GetTickCount` provides functionality to check the time since boot. The malware can for example check, if the returned time is greater than twelve minutes and only then execute its malicious code as pafish demonstrates in its implementation.

It can also check if the current user name is something generic or obvious like "malware" or "virus" by utilising the Windows API function `GetUserName`. Malware can also use the function `GetCursorPos`. This returns the current position of the cursor. By doing this consecutively with a delay in between, it can check if the mouse is moving at all or if it always rests in the same position, which would be an indicator for an automated system.

The possibilities for these checks are manifold and dependent on the creativity of the malware developer. In contrast to other checks, which depend on the design of virtual systems, checks utilising reverse turing tests are based on the often chaotic human behaviour. This generates way more different indicators. However these indicators are not as reliable as the ones based on implementation details. Possible checks are file names, contents of documents or images in the user's folder, the configured desktop wallpaper or the last modified date on files.

3 Design and Implementation

This chapter describes the setting of the technical environment for this thesis. Firstly, in Section 3.1 the general design goals that have to be achieved by the implementation are described. Required assumptions and definitions are then discussed in Section 3.2. The methods monitored by the implemented analysis system and their classification are shown in Section 3.3. Afterwards, the general technical software environment in which the analysis system is implemented is described in Section 3.4. Carried out modifications of the hypervisor are presented in Section 3.5. Similarly all changes made to the configuration of the VMI implementation VMIProgram are discussed in Section 3.6. Also in Section 3.7 the custom tool for analysing the raw data created by the analysis system is introduced. Lastly, the implementation of a VM detection sample with known source code that is needed for evaluation purposes is discussed in Section 3.8.

3.1 Design Goals

The overall goal of this thesis is to analyse the prevalence of VM detection methods using dynamic malware analysis relying on VMI. To achieve this, it is necessary to employ automated execution and analysis of samples.

Furthermore, to achieve representative results, it has to be possible to execute and analyse a high amount of samples in a given timespan. Therefore, it is essential for this implementation to offer scalability. The more resources are utilised for the analysis, the higher the amount of analysed samples should be.

Also the implementation is required to generate correct, comparable and deterministic results. This is necessary to create a statistical overview of the VM detection methods used and its prevalence.

3.2 Assumptions and Definitions

For this thesis it is assumed that the employed components of the do not have any security issues, which would yield the possibility for malware to escape the environment of the VM. Otherwise, this violates the design goals of correctness and determinism.

It could manipulate the results of the analysis. In addition, it is assumed that every analysed sample is executed on a clean system.

Furthermore, it is assumed that the employed VMI implementation VMIProgram is secure and functions without errors. It can detect any API calls and system calls that are issued by the sample and its child processes. Also it is assumed that any component that is used for the implementation of the environment described in Section 3.7 works as intended.

The aforementioned assumptions are necessary in achieving the goals of correctness, comparability and determinism as stated in Section 3.1. Furthermore, any analysed binary will be referred to as *sample* without regard of its potential malicious or benign background. All aspects of a sample that indicate that it uses any kind of method to detect a virtual system is defined as an *indicator*.

Also only simplified names for API calls are noted throughout the thesis and the several suffixes are omitted, as there sometimes exist different variations of the calls, for example `CreateEventA`, `CreateEventExA`, `CreateEventW` and `CreateEventExW`. `A` and `W` denote different string encodings and `Ex` means that the function call is extended with additional arguments. In all cases this is not important to our analysis and all different variations are traced.

3.3 Monitored Methods and Classification

All indicators have to be classified into different degrees of obviousness to create comparable results. These are defined as *strong*, *moderate* and *weak*. A table with all monitored indicators and their regarding degree can be found in Table 3.1. A complete overview with full lists of monitored input values can be found in Table B.1 in the appendix.

Some methods, like querying the value of a registry key, where the queried value name contains “VBox” or “VMWare” as sub-string are obviously used to detect a VM. Others, like retrieving the amount of milliseconds passed since the system was started, can be used for VM detection. But calls like this are more commonly used for other reasons like simple time checks.

As a general approach for classification of the obviousness, the following aspects are important. If an indicator shows any direct reference to a hypervisor manufacturer, it is classified as strong. This could be obvious sub-strings of input values or system artefacts that can be attributed to a hypervisor, like the network type of VirtualBox' shared folders. If an indicator accesses something that can be used for VM detection, but it is unclear whether any conclusions are drawn from its return value, it is only considered as moderate. An example for this is accessing the system's firmware table. If the indicator can only be used for VM detection when executed in a specific way, like consecutively calling a function, but is not accessing any clear system artefacts, it is considered weak. Examples for this are using `GetTickCount` to measure the time taken for certain actions or using `CreateEvent` to create an object that can be waited for with `WaitForSingleObject`. This is due to the fact that the analysis system of this thesis can not differentiate between the special and any other use of these functions. It does not fully analyse the control flow of the sample. Also if the probability is high that the indicator is used for a different reason that VM detection, like checking the disk size, which is often done by normal software installers, it is considered as weak.

If any timing attack related calls are used, the minimum delay time for declaring it as an analysis evasion method with moderate obviousness is defined as five minutes. This is presumably long enough to stall some dynamic analysis systems. All timespans shorter than this are only considered as a weak indicator. It should be noted though that this hard limit is not very reliable since it is not based on any studies regarding the common analysis time in dynamic malware analysis systems. It is also problematic that there could be benign reasons for long delays that are not related to VM detection or analysis evasion.

One special case is monitored. As described in Section 2.4.2, Locky tries to detect the presence of a VM by comparing the execution time of the two API calls `GetProcessHeap` and `CloseHandle`. It measures the timing differences using the instruction `RDTSC`. It is not possible to implement tracing of the latter, this is further discussed in Section 3.5. Therefore, the only possible way to detect this method is to detect the direct consecutive use of the two API calls. This unfortunately yields a relatively high probability for false positives. It could be a complete coincidence that this situation occurs. Therefore, it is only considered a moderate indicator.

Finally, certain methods described in Section 2.4 can not be monitored due to technical limitations of the approach and overcoming these are out of the scope of this thesis which is further discussed in Sections 3.5 and 4.1.

3.4 Target Software Environment

For this work the dynamic malware analysis environment of G DATA Software AG is used. Due to compatibility limitations with other systems, Microsoft's Windows 10 Version 1511 is used as guest OS on the virtual machines. The hypervisor Xen is employed in version 4.6.1. This relatively old version is necessary with the *LibVMI* library version 0.12 used by G DATA's VMIProgram. The latter is the component that implements the VMI functionality to trace API calls and system calls during execution of malware samples. Furthermore multiple servers can be managed by the dynamic malware analysis environment simultaneously. This fact contributes to the goal of scalability of the analysis.

Table 3.1: Obviousness for all monitored methods. If an indicator is defined with an input value in Table B.1, the obviousness degree stated here is only applicable when these input values are found. If multiple degrees are defined for one method, the correct degree is also defined by the corresponding input parameters.

Indicator	**Obviousness**
API Calls	
`CreateEvent`	−
`CreateFile`	+/ ∘
`CreateTimerQueue`	−
`CreateTimerQueueTimer`	∘
`CreateToolhelp32Snapshot`	∘
`CreateWaitableTimer`	−
`DeviceIoControl`	−
`EnumServicesStatus`	∘
`EnumSystemFirmwareTables`	∘
`FindWindow`	+
`GetAdaptersAddresses`	∘
`GetAdaptersInfo`	∘
`GetCursorPos`	∘
`GetDiskFreeSpace`	−
`GetFileAttributes`	+
`GetModuleFileName`	∘
`GetModuleHandle`	+
`GetNativeSystemInfo`	∘
`GetPwrCapabilities`	∘
`GetSystemFirmwareTable`	∘
`GetSystemInfo`	∘
`GetTickCount`	−
`GetUserName`	∘
`GlobalMemoryStatus`	∘
`Icmp6CreateFile`	−
`Icmp6SendEcho2`	∘
`IcmpCreateFile`	−
`IcmpSendEcho`	∘
`IcmpSendEcho2`	∘
`IsNativeVhdBoot`	+
`IWbemServices::ExecQuery`	∘
`NtDelayExecution`	∘
`NtQueryLicenseValue`	+
`OpenSCManager`	∘
`Process32First`	∘
`Process32Next`	∘
`RegOpenKey`	+/ ∘
`RegQueryValue`	+/ ∘
`SetTimer`	∘
`SetWaitableTimer`	∘
`SetupDiGetDeviceRegistryProperty`	∘
`Sleep`	∘
`timeSetEvent`	∘
`WaitForSingleObject`	∘
`WNetGetProviderName`	+
System Calls	
`NtCreateFile`	+/ ∘
`NtDeviceIoControlFile`	−
`NtOpenKey`	+/ ∘
`NtQueryValueKey`	∘
`NtQueryAttributesFile`	+
Instructions	
`CPUID`	+/ ∘/ −
Specials	
`GetProcessHeap` & `CloseHandle`	∘

+ : strong indicator   ∘ : moderate indicator   − : weak indicator

It would be possible to update the guest OS's version if needed. Though there are potential compatibility issues with the used LibVMI version. Resolving these would have exceeded the scope of this thesis as it also is not specifically necessary.

The implementation employs the *INetSim* service that simulates commonly used network services for the samples, as they are not granted access to the internet. This service always responds with simple messages to all requests from various protocols like HTTPS, SMTP, FTP and many more.

Furthermore, with every sample a new VM is started with the identical initial state. After a predefined time the execution is aborted and the resulting data is collected. This setup and the INetSim service help reaching the goals of correctness, comparability and determinism.

The analysis of the resulting tracing data from the dynamic analysis environment is performed by an application developed for this thesis using the .NET framework 4.7.2. All functionality can also easily be ported to .NET Core and is developed using parallel execution of multiple threads. Therefore the goal of portability and scalability of this aspect is fulfilled.

3.5 Modifications to the Hypervisor

As stated in Section 2.4, the instructions `CPUID` and `RDTSC` can be used to detect VMs as these instructions cause a VM exit. This yields the possibility for the hypervisor to trace these instructions, which is not yet implemented in VMIProgram. The simplest approach to enable this tracing is to modify Xen's sourcecode. The modified code for `CPUID` can be found in Listing 3.1. In this case, it reacts whenever the VM exit handler detects the usage of `CPUID` with any relevant input values. It outputs the following into Xen's kernel log: the used instruction, a timestamp, the ID of the VM, the input register value and the corresponding `CR3` value. This is then evaluated in the analysis. The `CR3` value in combination with the VM-ID enables correlating processes to issued instructions. After printing, the original handling of the instruction is continued.

Tracing `RDTSC` is implemented similarly as can be seen in Listing 3.2. In this case also the VM exit handler was modified to print information about the usage of the instruction into Xen's kernel log.

This log is then read by the analysis environment and added to the results. Unfortunately the output of the `RDTSC` tracing fills the buffer of Xen's kernel log faster than it can be read, because the instruction is traced for every process of the VM. This results in an output approximately every ten microseconds. Even when the buffer's size is increased as much as possible, the log can not be read and cleared in time before new messages overwrite old ones.

```
case EXIT_REASON_CPUID:
+       switch ( regs->eax )
+       {
+       case 0x0:
+       case 0x1:
+       case 0x40000000:
+       case 0x80000000:
+       case 0x80000001:
+       case 0x80000002:
+       case 0x80000003:
+       case 0x80000004:
+       case 0x8FFFFFFF:
+           printk(XENLOG_INFO "vmx_CPUID: TIME=%015ld ID=%05u EAX=0x%lX CR3=0x%lX .\n", NOW()
    , (unsigned int)v->domain->domain_id, regs->eax, v->arch.hvm_vcpu.hw_cr[3]);
+           break;
+       default:
+           break;
+       }
        is_pvh_vcpu(v) ? pv_cpuid(regs) : vmx_do_cpuid(regs);
        update_guest_eip(); /* Safe: CPUID */
        break;
```

Listing 3.1: Added code to Xen to trace the usage of `CPUID`. Lines starting with a `+` have been added.

```
case EXIT_REASON_RDTSC:
+       printk(XENLOG_INFO "vmx_RDTSC: TIME=%015ld ID=%05u CR3=0x%lX .\n", NOW(), (unsigned
    int)v->domain->domain_id, v->arch.hvm_vcpu.hw_cr[3]);
        update_guest_eip(); /* Safe: RDTSC, RDTSCP */
        hvm_rdtsc_intercept(regs);
        break;
```

Listing 3.2: Added code to Xen to trace the usage of `RDTSC`. Lines starting with a `+` have been added.

A possibility to circumvent this would be to limit the output only to processes that are monitored by VMIProgram. However this is out of scope of this thesis due to the complexity of Xen. Consequently, the code in Listing 3.2 used to trace `RDTSC` is omitted. `CPUID` is used far less frequently so that its tracing can be performed as described as above.

3.6 Configuration of VMIProgram

The *VMIProgram* traces the use of all API calls as described in Section 2.3. It registers which function is used by inspecting the called address. This is done with a database that has been created by analysing the exported functions of the DLLs of the guest OS. However the call `IWbemLocator::ExecQuery` that is used to execute WMI queries as described in Section 2.4.1, is not a named exported function and therefore is not stored in that database. This has been revealed by testing with a known sample. As VMIProgram outputs the address in that case, it is possible to add it manually to the database and trace the function.

VMIProgram also defines of which functions the input parameters are to be extracted with a configuration file. Here all data types have to be defined. All functions that are described in Section 3.3 have to be configured here.

3.7 VM Detection Indicator Analysis

VMIProgram and the modified Xen generate a raw trace of API calls, system calls and `CPUID` instructions. This raw data is filtered for the predefined indicators described in Section 3.3. This is achieved by scanning the text files for the function name. If that is present the definition of the indicator is important. Either the plain use of the call implies possible VM detection or the input parameters have to be checked. In the case of Locky's method utilising the timing differences between the calls `GetProcessHeap` and `CloseHandle` as described in Section 2.4.2, the consecutive use of those two calls has also to be logged as an indicator.

After the files are analysed, the results are then summarised, showing the prevalence of each method and describing which sample used which methods to potentially detect a VM.

3.8 Custom VM Detection Sample

For evaluation and testing purposes, a sample with known behaviour that performs VM detection is necessary. With this it is possible to identify false negative results. A sample performing all methods described in Section 2.4 has been developed by extending the VM detection demonstration tool al-khaser.

4 Evaluation

This chapter evaluates the results with respect to correctness. Section 4.1 introduces limitations uncovered during this work. Furthermore the hardware environment is described in Section 4.2. Tests with a known sample, benign samples and samples known to utilise VM detection have been carried out. The results of these tests are evaluated in Sections 4.3, 4.6 and 4.7. The selection of the samples for these tests is described in Section 4.4. Also the most efficient runtime for the analysis of each sample is evaluated in Section 4.5. Lastly the results of testing with a big set of randomly selected, current samples is discussed in Section 4.8.

4.1 Known Limitations

To analyse the prevalence of VM detection methods, a VMI implementation was used as described in Sections 2.3 and 3.4. This has the capability of tracing API calls and system calls thoroughly. Its design however has some limitations that are discussed in this section.

As described in Section 2.1.4, tracing of CPU-instructions has been performed in other works by either modifying the kernel to employ a debugger or by statically analysing the binary and comparing this trace to a dynamic behaviour trace. These approaches created a reliable instruction trace that could for example be searched for methods that use the `VPCEXT` instruction. The employed VMI implementation VMIProgram however is not capable of tracing instructions at all. The only traced instruction in this thesis is `CPUID`, implemented by modifying the Xen hypervisor as `CPUID` causes VM exits. `RDTSC` could not be monitored, even though it also causes VM exits, as it was not feasible to implement this in the scope of this thesis, see Sections 2.2.2 and 3.5.

Due to this fact, all VM detection methods that employ other instructions can not be detected. These are mostly low-level approaches as described in Section 2.4, for example the methods that utilise the IDT, LDT and GDT. Also the VMWare I/O port method, the usage of the `VPCEXT` instruction and the analysis of the PEB using the `FS` and `GS` registers will go unnoticed.

Furthermore, it is also not possible to monitor any VM detection methods that are utilising process injection to perform their actions within another process. Due to a limitation of VMIProgram, the process injected into would not be monitored. Only the process of the analysed sample and its children are monitored. Therefore, a sample employing methods like this is falsely classified as one that does not perform VM detection at all, producing false negative results to some extent.

Another limitation is that only the plain usage of API calls, system calls and the `CPUID` instruction is traced. The analysis system only checks the interaction with the OS, but does not inspect code residing purely in the sample's code region. It does not detect the interaction with the return value of the methods like string comparisons. Therefore, it is often inconclusive if a call to a monitored function is actually used for VM detection. This concludes that only the strong indicators can effectively issue a statement about the usage of VM detection. All other indicators only give a hint about the degree of probability of VM detection.

Furthermore, even though VMIProgram detects all API and system calls issued by an analysed sample, the analysis of the resulting trace can only search for the previously known and defined VM detection methods as described in Section 3.3. Therefore, it is out of scope of this thesis to find usages of new methods or those that have been overlooked when configuring VMIProgram. It rather analyses the prevalence of already known methods. Unfortunately as stated for example in Section 2.4.3, there certainly exist a lot of possibilities to employ VM detection that have not yet been found and are completely dependant on the creativity of the malware developer.

4.2 Hardware Environment

The implementation described in Chapter 3 facilitates scalability. It is employed on four different servers to increase the amount of samples that can be analysed in a given timespan. Those servers are provided by G DATA Software AG and only differ in the CPU specifications described in Table 4.1. Otherwise, all servers are equipped with 12 Double Data Rate 3 (DDR3) Dual In-line Memory Modules (DIMMs) of Random Access Memory (RAM) with a total size of 64 GigaByte (GB) and a speed of 1.333 GigaHertz (GHz).

The different processors most certainly do not affect the analysis in a way that is relevant to this thesis, as they are all Intel processors facilitating the Intel VT for virtualisation. This enables the use of VMX for all systems. On every server one core is reserved for `Domain0`, as this is the privileged host VM on Xen systems. Xen refers to its VMs as *domains* and on `Domain0` of all employed servers is the dynamic malware analysis environment implemented. The other cores are used for the VMs needed for the analysis, one core for each VM. This results in a total amount of 36 simultaneously running VMs.

Table 4.1: Differences in CPU specifications of the four used servers.

	CPU		
Server	Type	Amount of Cores	Max Turbo Frequency
0	Intel® Xeon® X5670	12	3.333 GHz
1	Intel® Xeon® L5520	8	2.48 GHz
2	Intel® Xeon® L5520	8	2.48 GHz
3	Intel® Xeon® X5650	12	3.06 GHz

4.3 Validation with a Known Sample

To validate that the analysis yields no false negative detection results, it is tested with a sample of which the source code is known. This sample is based on the VM detection demonstration tool al-khaser. Additionally, it is extended with all methods described in Section 2.4 that had not yet been implemented.

The runtime of the analysis was increased as much as necessary to detect all delays created with function calls, because the analysis system does not modify any delays. All methods described in Section 3.3 were found with the analysis. The following methods were not found due to the limitations described in Section 4.1:

- Methods detecting differences in returned addresses from the instructions `SIDT`, `SLDT`, `SGDT` and `STR`.
- Communication with VMWare's I/O port.
- Analysis of the response to VirtualPC's undocumented `VPCEXT` instruction.
- Evaluation of the amount of processor cores by inspecting the PEB using the instruction `__readgsword`.
- Analysis of the time taken for the `CPUID` instruction using the `RDTSC` instruction.

These are the only false negative results of this validation which was expected due to the aforementioned reasons.

4.4 Selection of Samples

To select samples to be used in this thesis, the following steps were performed. The goal was to gather samples as recent as possible while still having a certain amount of randomness in the selection. For this, all samples that G DATA received from their various input sources in the timespan between the 27.06.2019 at 00:00 AM and

08.07.2019 at 15:30 PM have been selected. These sources are for example G DATA's business partners but also the samples sent in by their customers. This resulted in a total of 1,036,965 samples.

For the main analysis a subset of 50,000 samples is selected randomly and is henceforth referred to as the *main-set*. This amount has been chosen to not exceed the time frame of two weeks. This is based on the runtime of 12 minutes for each sample, which is determined in Section 4.5.

G DATA themselves perform generic detection of VM detection methods. This is not done by analysing any API calls but rather by searching for certain strings or instructions in the sample's process memory right before its execution is finished. For this the pattern matching tool *YARA* developed by *VirusTotal* is used. Checking for the results of these memory analyses on all selected samples resulted in a subset of 1,304 samples that returned a positive result to G DATA's detection and will be henceforth referred to as *vm-detection-set*.

For determining the best runtime for the samples a test is performed comparing different runtimes on the same set of samples, as described in Section 4.5. This set is created by selecting 100 random samples of the main-set and 100 different random samples of the vm-detection-set. This resulted in a set of 200 samples with unique hash values and will henceforth be referred to as *runtime-set*.

Also for evaluating the amount of false positive results a set of benign samples is selected. These are samples that are known to have no malicious intention. G DATA define this using the certificates created by trusted manufacturers and using sources like *AV-TEST*, which sell these samples. For this 100,000 samples received in the timespan between 08.07.2019 00:00 AM and 10.07.2019 04:45 AM were selected. For the test a random subset of 200 samples is then taken, which will from now on be referred to as *benign-set*.

4.5 Evaluating the Most Efficient Sample Runtime

The analysis system executes each sample for a specific timespan. When this runs out, the analysis is stopped and the created results are collected. Generally, it is expected that any VM detection methods are executed at the beginning of the sample run, assuming the whole run is not hold up by the sample with a delay.

Ideally, a run should execute as long as the sample performs any actions, but it is impossible to know when the sample is finished. This would require to solve the halting problem. On one hand, the higher the runtime, the less samples can be analysed in a given timespan. But on the other hand, the higher the runtime, the higher is the probability to find more VM detection methods used by the sample.

It is desirable to analyse as many samples as possible. Therefore a compromise has to be found. To determine the most suitable runtime, a test comparing the results of different timespans on the same set of samples is conducted. For this the 200 samples from the runtime-set was used. When running the test it showed that 11 samples were not possible to execute. This was probably due to those being corrupted or not compatible file types like `PDF` or `DOCX`.

For this test the runtimes 3 minutes, 6 minutes, 9 minutes, 12 minutes and 15 minutes per sample have been selected. Only strong indicators reliably show the intention of VM detection. Therefore, only these are taken into account for comparing the results. Those are displayed in Figure 4.1. As can be seen in it, there is no big difference between the five configurations, but 12 and 15 minutes both yielded the best results. It should be noted that the test with 6 minutes showed fairly better results than the test with 9 minutes runtime. This indicates that the analysis system might not be perfectly deterministic, but this could also be due to the chaotic nature of malware. It is possible that the samples behave randomly depending on some minor aspects of the system. It seems that two samples in the 9-minute-test did not execute in the same way as they did in the 6-minute-test.

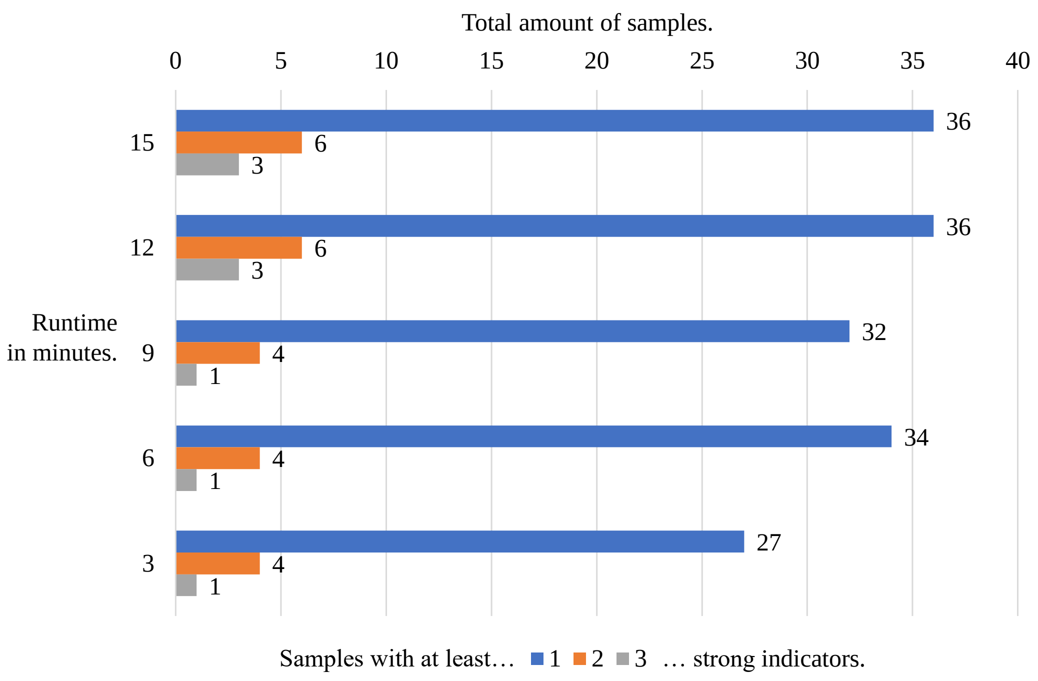


Figure 4.1: The total amount of samples from the runtime test-set that used at least one, two or three different strong indicators.

Another aspect of quality for this test run is the overall amount of indicators found, disregarding the level of obviousness. In Table 4.2, the different amounts of indicators found with each test run are shown. The same difference between 6 and 9 minutes as can be seen in Figure 4.1 can also be found here. It is possible that some samples behaved differently in the two test runs. Also the 15 minute test run created less results than the 12 minute test run, which also shows that the samples do not seem to behave deterministic in every execution.

Table 4.2: The total amounts of different indicators found in all samples of each test run in the runtime test.

Runtime in Minutes	**Amount of Found Indicators**
15	1249
12	1269
9	1189
6	1197
3	1189

As a conclusion, 12 minutes seem to be the best compromise between somewhat complete results and a high amount of samples that can be analysed in the scope of this thesis.

4.6 Validation with Benign Samples

To analyse the false positive rate, first, four benign samples have been manually selected and examined.

- A custom sample that only performs a simple, constant console output and nothing else, created with C++.

  SHA256 value:
  `4079619f6bfb0d543a19a27a8164392b21b22270ec895d72ced50d90976b2d88`

- The Windows calculator with version number 10.0.17134.1 taken from a Windows 10 Version 1803 Build 17134.950 system. It has been manually assured that this runs on the Windows 10 Version 1511 provided by the VMs without errors.

  SHA256 value:
  `284674a806bcbe692c76761baaf21327638de0c7135bfb06953648be7d661fbd`

- The installer for the text editor Notepad++ for 32bit x86 systems of version 7.7.1.

  SHA256 value:
  `6787c524b0ac30a698237ffb035f932d7132343671b8fe8f0388ed380d19a51c`

- The not customised TeamViewer QuickSupport binary that can be executed without installing to enable remote access to the graphical user interface of a system. The binary with version 14.4.2669.0 was used.

  SHA256 value:
  `b2e3946fde991d991ef28c181be540031acc99771347d03e7ac7cdca6992c28f`

Apart from the manually selected samples also the 200 samples of the benign-set were analysed with a runtime of 12 minutes. Ideally no indicator returns a positive result in any of these samples. This is somewhat unlikely though as is it possible that VM detection is even performed by benign samples to act differently on virtualised systems. Nonetheless, it should only rarely occur as benign software would usually want to function similarly on any system it is executed on. As there is no feasible way to reliably identify samples that do not perform VM detection, this is best possible way to inspect false positives.

4.6.1 Results of Analysing the Manually Selected Benign Samples

The results of the manually selected samples are discussed in this section. An overview of all found VM detection methods is given in Table 4.3.

- **Custom Sample**
  As expected, not a single indicator returned a positive result.

- **Windows Calculator**
  Two indicators return positive results: `CPUID` with the input values `0x0` and `0x1`. It is uncertain why the calculator uses this instruction, but as Section 4.8 shows, this is a very common occurrence and might be used for simple compatibility checks with the CPU.

- **Notepad++ Installer**
  This returned one positive, moderate indicator, the API call `GetModuleFileName` with input `NULL`. This returns the full path to the sample's binary. It is possible that the installer performs certain checks on itself before starting the installation. Besides that, the API call `GetTickCount` and the instruction `CPUID` with inputs `0x0` and `0x1` have been found.

- **TeamViewer's QuickSupport binary**
  This returned two positive moderate indicators, the two API calls `GetModuleFileName` with input `NULL` and `GetSystemInfo`. Also the weak indicators `CreateEvent`, `GetTickCount` and `CPUID` with inputs `0x0` and `0x1` have been found. This probably has also been done to perform a self check and for compatibility reasons.

Table 4.3: Overview of all found VM detection methods when analysing the four manually selected benign samples.

Indicator	**Custom Sample**	**Windows Calculator**	**Notepad++ Installer**	**TeamViewer's QuickSupport binary**
Moderate				
`GetModuleFileName`	×	×	✓	✓
`GetSystemInfo`	×	×	×	✓
Weak				
`CPUID, EAX = 0x0`	×	✓	✓	✓
`CPUID, EAX = 0x1`	×	✓	✓	✓
`GetTickCount`	×	×	✓	✓
`CreateEvent`	×	×	×	✓

✓: The indicator is detected. ×: The indicator is not detected.

These results show that the analysis system does not yield any consistent false positives. One sample did not show the use of any indicator. Therefore the reason for the high occurrence of the instruction `CPUID` described in Section 4.8 is not because of an error in the system as could be suspected. This validation showed that a sample that definitely does not use this instruction also correctly created only negative indicators for that instruction.

Also only a few moderate and no strong indicators were found. This shows that the analysis system yields somewhat reliable results. But it should be noted that this also shows that moderate and weak indicators might not be reliable for finding VM detection.

4.6.2 Results of Analysing the 200 Randomly Selected Benign Samples

The results of analysing the 200 randomly selected samples of the benign-set are discussed here. Of those only 192 samples were runnable. Only three samples used at least one strong indicator. One of these three samples is *Avast SZFLocker* in version 1.0.189.0, which is a decryptor tool for the ransomware *SZFLocker*. Also the file search utility called *UltraFileSearch Lite* in version 3.4.0.13329 used at least one strong indicator. The remaining sample is the *Kaspersky Virus Removal Tool* in version 15.0.19.0. Two of those samples are connected to malware which might explain the use of VM detection methods. The search tool is opening registry keys that are typical for the emulator Wine and the hypervisor VirtualBox. Why this is done is unclear.

In Table B.2 in the appendix a full overview of the distribution of all indicators can be found. Here, it is obvious that the indicators `GetCursorPos`, `GetModuleFileName`, `GetNativeSystemInfo`, `GetSystemInfo`, `CreateEvent`, `CPUID` with input values `0x0` and `0x1` and Locky's timing trick are not reliable indicators for samples using VM detection. These returned positive results in various benign samples, where it should be unlikely that VM detection is performed. Therefore, these indicators are assumed to be false positives.

4.7 Validation with VM Detecting Samples

In this section it is tested how well the analysis performs on samples on which G DATA's memory analysis showed that they have aspects of VM detection. For this the 1,304 samples of the vm-detection-set have been analysed. It has to be noted that G DATA's method for analysing the samples is vastly different to the method used in this work. It is possible that code is analysed that is stored in the memory, but that has never been and might never be executed in the control flow of the sample. While G DATA's analysis shows that the sample has the capability of performing VM detection methods, this work only detects executed methods. Therefore, it is expected that some samples do not show any VM detection behaviour even if G DATA's analysis returned positive results.

Of the 1,304 analysed samples, only 1,287 were executable. The other 17 samples were either corrupted binaries or not compatible with our analysis system. This is unexpected, as G DATA's analysis has positive detections for all 1,304 samples. This means that it was possible to execute the 17 samples in the past. Even with multiple attempts the samples are consistently not runnable in the implemented analysis system. The reason for this is unclear, but is also not relevant for the results of this test.

As before, first, we only inspect the usage of strong indicators, as these are the only reliable indicators for VM detection. These results can be found in Figure 4.2. This showed that 370 samples used at least one obvious indicator of VM detection, which are 28.7% of all executable samples. Four samples even used ten different strong indicators.

As can be seen here, a significantly higher amount of samples is found to be using strong indicators in comparison to the test using benign samples. This shows that the analysis system can successfully identify samples using obvious VM detection methods.

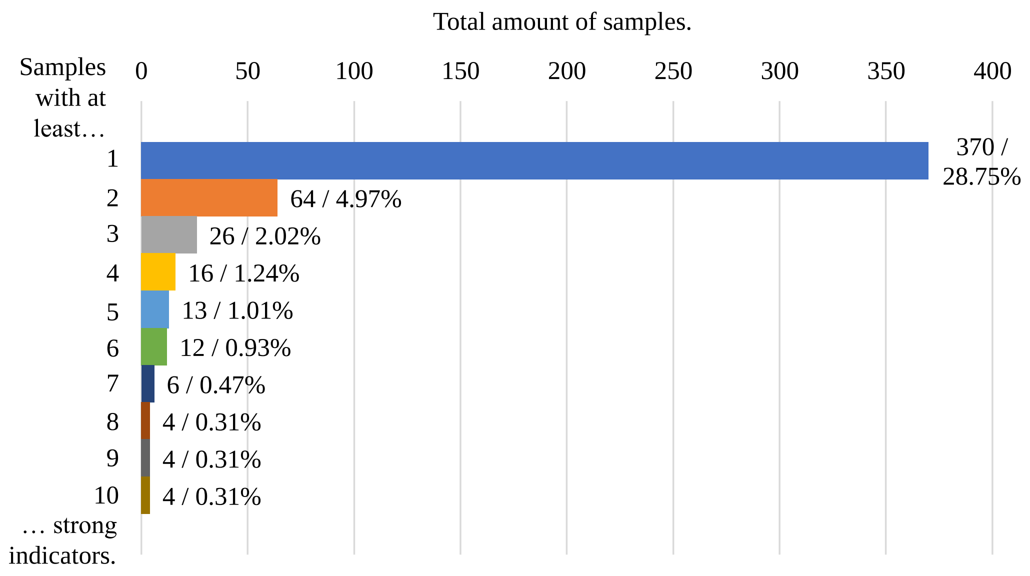


Figure 4.2: The amount of samples in the vm-detection-set that used from at least one up to at least ten different strong indicators.

In Table B.3 in the appendix a complete overview of all found indicators is shown to display the prevalence of each indicator in the vm-detection-set. Here some irregularities stand out and should be addressed. The methods `GetModuleFileName` and `CPUID` with inputs `0x0` and `0x1` were used by almost every sample. As this is the vm-detection-set, on one hand it is possible that these methods were actually used for VM detection. But on the other hand the same results are found in all other tests. Some other methods were only rarely or not used at all, for example the API calls `IsNativeVhdBoot` and `NtQueryLicenseValue`, which shows that these might be not reliable or outdated and are therefore not used in current VM detecting samples.

In Figure 4.3 the distribution of the found strong indicators is depicted. For this all strong indicators that returned a positive result were taken into account. The prefix `AC` denotes API calls, `SC` denotes system calls and `I` denotes instructions. It should be noted that any system calls that were issued by the corresponding API calls are ignored for this figure. This would otherwise distort the ratio between API calls that utilise system calls and those that do not. It shows that `GetProcAddress` is used at least twice as often as any other strong API call. These are mostly calls using either `wine_get_unix_file_name` and `wine_get_version`. It is possible but not clear if this might be used more often for benign reasons than for VM detection.

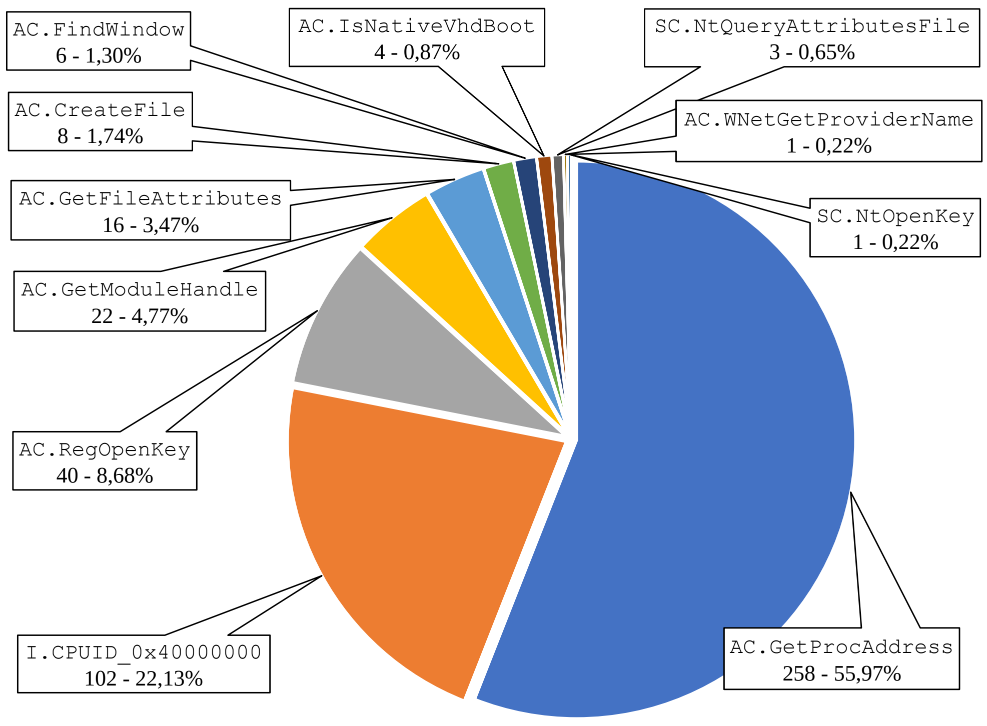


Figure 4.3: The distribution of used strong indicators in the test with the vm-detection-set.

4.8 Conducting Analysis with Random Samples

Lastly the analysis using the 50,000 samples of the main-set is performed. Of this set, 4,463 samples (8.9 %) were not valid binaries compatible with our analysis system. As any input from G DATA's sources is tried to be executed on the analysis system and is not filtered beforehand, it is possible that either corrupted files or incompatible file type, for example `PDF` or `DOCX`, were used. Therefore, only 45,537 samples could be analysed.

Figure 4.4 shows that 1,263 (2.77 %) of all analysed samples used at least one strong method of VM detection, while 469 (1.03 %) used at least two strong methods. Also in Table B.4 in the appendix the overall prevalence of each method can be found. As stated in Section 4.1, it is hard to say which methods are used for VM detection and which are not when inspecting the moderate and weak indicators. But the results show that VM detection is still prevalent in current malware. The high amount of moderate indicators suggests that the actual prevalence of VM detection is higher than found here. Especially, since many low-level methods could not be monitored. It should also be noted that G DATA's memory analysis only classified 74 samples of the main-set to be using VM detection methods. Interestingly, our analysis only found 20 of those samples, while we detected 1.243 different samples using at least one strong indicator.

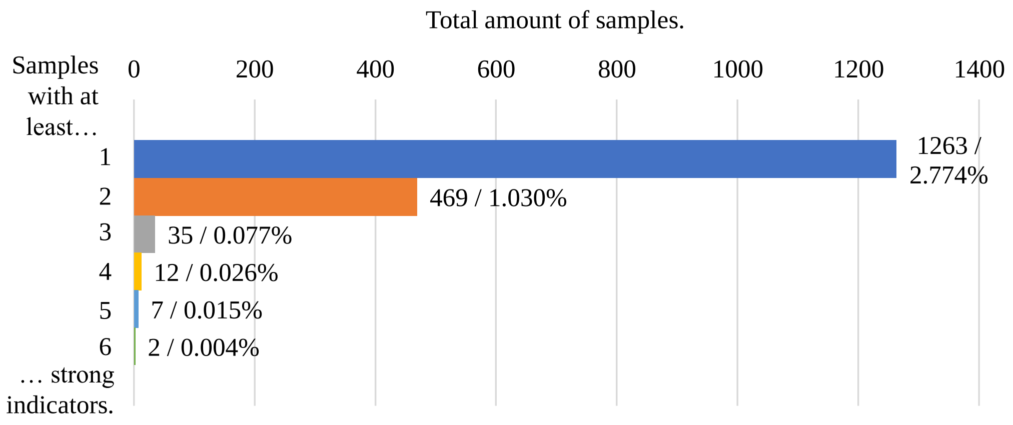


Figure 4.4: The amount of samples from the main-set that used from at least one up to at least six different strong indicators.

Inspecting the results further in Figure 4.5, it should be noted that, in contrast to the analysis of the vm-detection-set, the API call `GetModuleHandle` in 310 samples was found more frequently than `GetProcAddress` in 116 samples. In the previous test `GetProcAddress` was used about ten times more often than `GetModuleHandle`. This could show that the 258 samples from the vm-detection-set using `GetProcAddress` were genuinely doing this to detect a VM. The figure also shows that the most prevalent strong VM detection indicator is the use of the instruction `CPUID` with input value `0x40000000`. This suggests that it is most common to use low-level methods like `CPUID` to detect a VM. The relatively high usage of `CPUID` with input values `0x80000002`, `0x80000003` and `0x80000004` in at least 786 samples also supports this statement, even though these are only moderate indicators.

It should also be noted that neither in the vm-detection-set, nor in the main-set any usages of `NtQueryLicenseValue`, `IcmpSendEcho`, `Icmp6SendEcho` and `Icmp6CreateFile` could be found. This indicates that these methods are only very rarely used in current malware, if at all. The instruction `CPUID` with input value `0x8FFFFFFF` was also never used, but this is most certainly the case, because our system is using Intel processors. Malware probably checks with other `CPUID` input values like `0x0` beforehand, if the CPU is manufactured by AMD. Only if that is the case, the AMD easteregg is checked for.

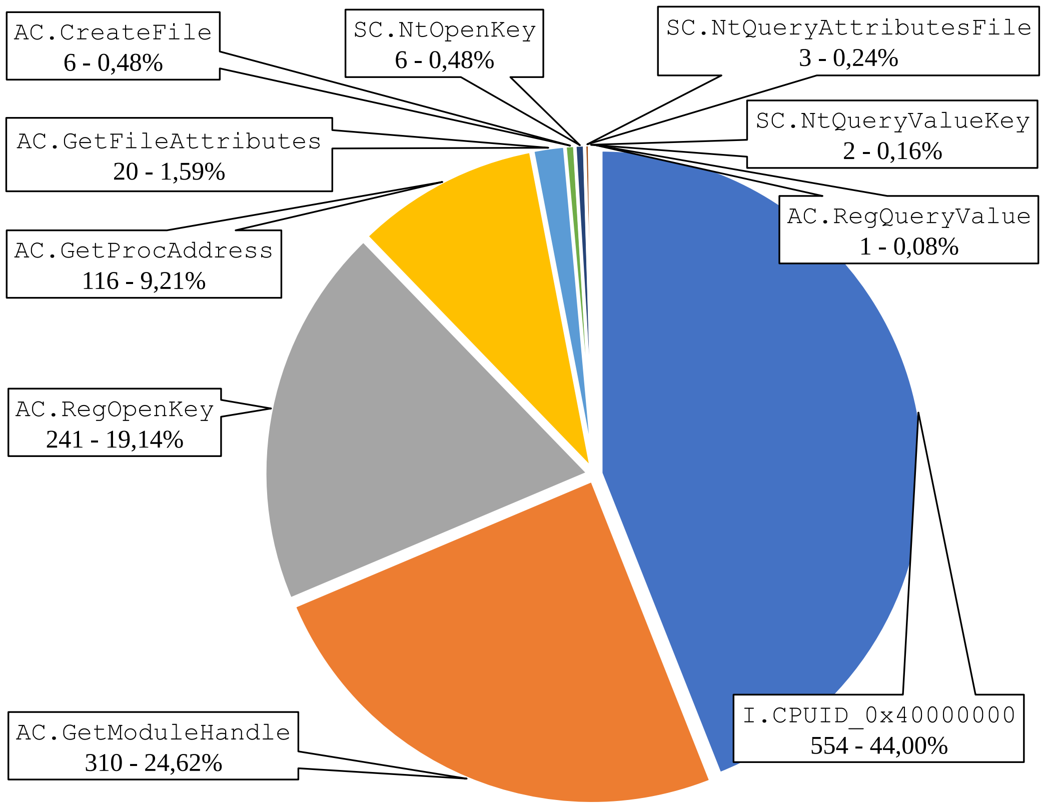


Figure 4.5: The distribution of used strong indicators in the main-set.

The indicators `GetModuleFileName`, `CreateEvent`, `GetTickCount` or the Locky timing trick are found in at least 30 % of the samples. It is unlikely that such a great amount of samples utilise VM detection when other, more reliable indicators are less present. These methods have also been found to be most certainly false positives, see Section 4.6. This shows that these indicators, in contrast to previous classifications, are weak.

5 Conclusion

This chapter summarises the found results and discusses conclusions drawn from them in Section 5.1. In Section 5.2 future work resulting from this thesis is discussed, approaching possible countermeasures to harden VMs against VM detection. Another aspect discussed is improving the analysis to create more insightful and reliable results.

5.1 Summary

This thesis has found that VM detection is still prevalent in current malware. At least 2.8 % of analysed samples are found to utilise methods that clearly indicate the use of VM detection. The methodology of this work has shown to have some limitations and can not identify most low level methods. It can also not deduce the true intention behind a lot of methods. The true prevalence of VM detection might be significantly higher. It is likely that malware more often uses low-level than high-level methods as these are less *noisy* than obvious API calls in combination with input values like registry keys attributed to certain hypervisors. Nonetheless, this work showed that it could be worthwhile to further investigate the usage of VM detection methods in current malware.

This is interesting as today's computer systems are more and more virtualised with the rise of cloud computing services like AWS or Microsoft *Azure* as current market predictions show [5]. It is reasonable to assume that malware detecting and therefore evading these would lower their spreading. It is questionable if VM detection will still be relevant in the future if the growth of cloud and other virtualised systems further increases.

5.2 Future Work

In this section open questions resulting from this work are discussed. The first aspect is the hardening of virtual systems to decrease transparency of their virtual nature. It is desirable that malware can not detect the VMs and that their behaviour in an analysis environment is identical to when they are running in a victim's system. Secondly, ideas about improving the analysis to create more reliable and insightful results are described.

5.2.1 Possible Countermeasures

This thesis showed that some aspects of the system can easily be checked to identify a VM. These can be very low-level like checking memory addresses of system information structures. They can also be high-level like checking for the existence of certain network shares or registry keys.

Some changes with instructions causing VM exits could be handled by the hypervisor. The return values of `CPUID` could be altered so that they replicate those of a non-virtualised system. Also the return values of `RDTSC` could be changed so that the time passed seems shorter that it actually was. Although, this would require a complex implementation as the values need to be determined for every context. All changes have to be made with caution, because some system aspects rely on true information about the CPU like the architecture or similar. Changes made here could create problems with the compatibility of programs running on the system or even of the OS. It could be possible to only change the return value for the processes monitored by VMI by inspecting the value of the `CR3` register. This would reduce the risk of creating unforeseen problems with the system when changing these values.

Similarly it should be investigated if it is possible to hide system artefacts like registry keys, network devices, MAC addresses or processes by altering these values in the OS. This also could yield problems in the operation of the system as some parts of the hypervisor could depend on those system artefacts. As before, VMI could potentially also be employed to only alter the returned values of for example the API call `RegOpenKey` or the system call `NtOpenKey`. These changes could then only be applied to VM detection methods of the monitored processes.

Also timing attacks could tried to be circumvented by disabling or actively shortening delays. This could be problematic to implement without it being noticeable. Furthermore, it should be tried to create an analysis system that seems as if a human is actually using it by for example creating real files with plausible names, automatically moving the cursor to mimic user interaction or having the system's uptime at a believable amount. Of course, it is very hard to do this perfectly and some aspects will potentially always be detectable.

5.2.2 Improving the Analysis

This work showed that simply detecting the use of API calls, system calls and instructions causing VM exits with their corresponding input values does not provide very thorough and reliable results. Some improvements should be made for future analyses.

When staying with the technical approach of VMI, at first all instructions creating VM exits should be traced. As described in Section 4.1, this work does not provide the detection of the instruction `RDTSC`. This would have been too complex for the scope of this thesis. This should be implemented in future approaches so that all aspects of possible VM detection can be found.

The analysis could also create a full trace of operations, potentially by including other methods like static analysis similar to the approach with DSD-Tracer. Using this to analyse the handling of the return values, it would be clear if malware checks for certain VM-indicating aspects. It would also be beneficial to implement a similar technique to the approach of Comparetti et al's *Reanimator* to not only analyse executed but also dormant functionalities [6]. It statically analyses a sample for previously found, malicious functionalities from other malware samples. Furthermore a method detecting process injection should be implemented so that malware using this technique is detected as well and therefore rendering the analysis as complete as possible.

Additionally, it would be desirable to have a non-virtualised system that is otherwise identical to the VM to compare the behavioural profile of the sample on this system with the virtualised one, similar to the approach of Lindorfer et al [19]. Using this, not only predefined methods but also yet undiscovered indicators of VM detection could be found.

Combining all these approaches would create an ideal analysis environment that could potentially detect any VM detection method. Unfortunately, this is highly complex as all components have to be synchronized and orchestrated. All compared systems have to be identical so that the behaviour of the malware is not affected by any aspect of the system and all resulting data need to be stored and analysed. This would presumably require a lot of computing power.

When analysing malware, the analyst is always at an arms race with the malware's developers. It is virtually impossible for any side to be completely sure to either detect and prevent everything or to be fully undetected on a victim's system.

A Acronyms

API Application Programming Interface

AWS Amazon Web Services

BIOS Basic Input/Output System

CPU Central Processing Unit

DDR3 Double Data Rate 3

DIMM Dual In-line Memory Module

DLL Dynamic-Link Library

EPT Extended Page Tables

FTP File Transfer Protocol

GB GigaByte

GDT Global Descriptor Table

GDTR GDT Register

GHz GigaHertz

HTTPS Hyper Text Transfer Protocol Secure

ICMP Internet Control Message Protocol

IDT Interrupt Descriptor Table

IDTR IDT Register

IP Internet Protocol

LDT Local Data Table

LDTR LDT Register

MAC Media Address Control

MSR Model Specific Register

OS	Operating System
PEB	Process Environment Block
RAM	Random Access Memory
RAT	Remote Access Trojan
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
TIB	Thread Information Block
TR	Task Register
TSS	Task State Segment
VHD	Virtual Hard Disk
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Manager
VMX	Virtual Machine Extension
VT	Virtualization Technology
WBEM	Web-Based Enterprise Management
WMI	Windows Management Instrumentation

B Classification and Prevalence of Monitored VM Detection Methods

Table B.1: Overview of all monitored API calls, system calls and instructions. The first column shows the simplified name (without suffixes). The second column shows the parameters that included to check if the function indicates VM detection. The third column respectively shows the values that are looked for in said parameters. The last column shows the classification of the method as described in Section 3.3.

Function	**Parameters**	**Values**	**Obviousness**
API Calls			
`CreateEvent`	–	–	weak
`CreateFile`	`lpFileName`	`"VBoxMiniRdrDN", "VBoxGuest", "VBoxTrayIPC", "HGFS", "vmci"`	strong
		`"\C:", "\PhysicalDrive0"`	moderate
`CreateTimerQueue`	–	–	weak
`CreateTimerQueueTimer`	`DueTime`	$\geq$ 5 minutes	moderate
`CreateToolhelp32Snapshot`	`dwFlags`	flag `TH32CS_SNAPPROCESS`	moderate
`CreateWaitableTimer`	–	–	weak
`DeviceIoControl`	`dwIoControlCode`	`IOCTL_DISK_GET_LENGTH_INFO`	weak
`EnumServicesStatus`	–	–	moderate
`EnumSystemFirmwareTables`	–	–	moderate
`FindWindow`	`lpClassName`, `lpWindowName`	`"VBoxTrayToolWndClass", "VBoxTrayToolWnd"`	strong
`GetAdaptersAddresses`	–	–	moderate
`GetAdaptersInfo`	–	–	moderate
`GetCursorPos`	–	–	moderate
`GetDiskFreeSpace`	`lpDirectoryName`, `lpRootPathName`	`NULL`	weak

Function	Parameters	Values	Obviousness
API Calls			
`GetFileAttributes`	`lpFileName`	`"VBoxMouse.sys", "VBoxGuest.sys", "VBoxSF.sys", "VBoxVideo.sys", "vboxdisp.dll", "vboxhook.dll", "vboxmrxnp.dll", "vboxogl.dll", "vboxoglarrayspu.dll", "vboxoglcrutil.dll", "vboxoglerrorspu.dll", "vboxoglfeedbackspu.dll", "vboxoglpackspu.dll", "vboxoglpassthroughspu.dll", "vboxservice.exe", "vboxtray.exe", "VBoxControl.exe", "virtualbox guest additions", "vmmouse.sys", "vmhgfs.sys", "vm3dmp.sys", "vmci.sys", "vmhgfs.sys", "vmmemctl.sys", "vmmouse.sys", "vmrawdsk.sys", "vmusbmouse.sys", "VMWare", "sample.exe", "malware.exe"`	strong
`GetModuleFileName`	`hModule`	`NULL`	moderate
`GetModuleHandle`	`lpModuleName`	`"avghookx.dll", "avghooka.dll", "snxhk.dll", "sbiedll.dll", "dbghelp.dll", "api_log.dll", "dir_watch.dll", "pstorec.dll", "vmcheck.dll", "wpespy.dll", "cmdvrt64.dll", "cmdvrt32.dll"`	strong
`GetNativeSystemInfo`	–	–	moderate
`GetProcAddress`	`lpProcName`	`"wine", "IsNativeVhdBoot"`	strong
		`"NtQueryLicenseValue"`	moderate
`GetPwrCapabilities`	–	–	moderate
`GetSystemFirmwareTable`	–	–	moderate
`GetSystemInfo`	–	–	moderate
`GetTickCount`	–	–	weak
`GetUserName`	–	–	moderate
`GlobalMemoryStatus`	–	–	moderate

Function	**Parameters**	**Values**	**Obviousness**
API Calls			
`Icmp6CreateFile`	–	–	weak
`Icmp6SendEcho2`	`Timeout`	$\geq$ 5 minutes	moderate
`IcmpCreateFile`	–	–	weak
`IcmpSendEcho`	`Timeout`	$\geq$ 5 minutes	moderate
`IcmpSendEcho2`	`Timeout`	$\geq$ 5 minutes	moderate
`IsNativeVhdBoot`	–	–	strong
`IWbemServices::ExecQuery`	`strQuery`	`"Win32_Processor", "Win32_LogicalDisk", "Win32_BIOS" "Win32_ComputerSystem", "MSAcpi_ThermalZoneTemperature" "Win32_Fan", "Win32_PnPEntity", "Win32_NetworkAdapterConfiguration" "Win32_NTEventlogFile", "Win32_NetworkAdapter", "Win32_Process" "Win32_MemoryArray", "Win32_MemoryDevice", "Win32_PhysicalMemory"`	moderate
`NtDelayExecution`	–	–	moderate
`NtQueryLicenseValue`	`LicenseValue`	`"Kernel-VMDetection-Private"`	strong
`OpenSCManager`	–	–	moderate
`Process32First`	–	–	moderate
`Process32Next`	–	–	moderate

Function	Parameters	Values	Obviousness
API Calls			
`RegOpenKey`	`lpSubKey`	`"VirtualBox", "VBox", "Virtual Machine"` `"VMware", "Wine"`	strong
		`"\Target Id 0\Logical Unit Id 0"` `"SystemBiosVersion", "VideoBiosVersion"` `"SystemBiosDate", "SystemManufacturer"` `"SystemProductName", "Services\Tcpip\Linkage"` `"HARDWARE\Description\System"` `"ControlSet001\Control\SystemInformation"`	moderate
`RegQueryValue`	`lpSubKey,` `lpValueName`	`"VirtualBox", "VBox", "Virtual Machine"` `"VMware", "Wine"`	strong
		`"\Target Id 0\Logical Unit Id 0"` `"SystemBiosVersion", "VideoBiosVersion"` `"SystemBiosDate", "SystemManufacturer"` `"SystemProductName", "Services\Tcpip\Linkage"` `"HARDWARE\Description\System"` `"ControlSet001\Control\SystemInformation"` `"Identifier"`	moderate
`SetTimer`	`uElapse`	$\geq$ 5 minutes	moderate
`SetWaitableTimer`	`lpDueTime`	$\geq$ 5 minutes	moderate
`SetupDiGetDeviceRegistryProperty`	–	–	moderate
`Sleep`	`dwMilliseconds`	$\geq$ 5 minutes	moderate
`timeSetEvent`	`uDelay`	$\geq$ 5 minutes	moderate
`WaitForSingleObject`	`dwMilliseconds`	$\geq$ 5 minutes	moderate
`WNetGetProviderName`	`dwNetType`	`WNNC_NET_RDR2SAMPLE`	strong

Function	**Parameters**	**Values**	**Obviousness**
System Calls			
`NtCreateFile`	`ObjectAttributes`	`"VBoxMiniRdrDN", "VBoxGuest", "VBoxTrayIPC", "HGFS", "vmci"`	strong
		`"\C:", "\PhysicalDrive0"`	moderate
`NtDeviceIoControlFile`	`dwIoControlCode`	`IOCTL_DISK_GET_LENGTH_INFO`	weak
`NtOpenKey`	`ObjectAttributes`	`"VirtualBox", "VBox", "Virtual Machine" "VMware", "Wine"`	strong
		`"\Target Id 0\Logical Unit Id 0" "SystemBiosVersion", "VideoBiosVersion" "SystemBiosDate", "SystemManufacturer" "SystemProductName", "Services\Tcpip\Linkage" "HARDWARE\Description\System" "ControlSet001\Control\SystemInformation"`	moderate
`NtQueryAttributesFile`	`ObjectAttributes`	`"VBoxMouse.sys", "VBoxGuest.sys", "VBoxSF.sys", "VBoxVideo.sys", "vboxdisp.dll", "vboxhook.dll", "vboxmrxnp.dll", "vboxogl.dll", "vboxoglarrayspu.dll", "vboxoglcrutil.dll", "vboxoglerrorspu.dll", "vboxoglfeedbackspu.dll", "vboxoglpackspu.dll", "vboxoglpassthroughspu.dll", "vboxservice.exe", "vboxtray.exe", "VBoxControl.exe", "virtualbox guest additions", "vmmouse.sys", "vmhgfs.sys", "vm3dmp.sys", "vmci.sys", "vmhgfs.sys", "vmmemctl.sys", "vmmouse.sys", "vmrawdsk.sys", "vmusbmouse.sys", "VMWare", "sample.exe", "malware.exe"`	strong

Function	**Parameters**	**Values**	**Obviousness**
System Calls			
`NtQueryValueKey`	`ValueName`	`"SystemBiosVersion", "VideoBiosVersion"` `"SystemBiosDate", "SystemManufacturer"` `"SystemProductName", "Identifier"`	moderate
Instructions			
`CPUID`	`EAX`	`0x0`	weak
		`0x1`	weak
		`0x40000000`	strong
		`0x80000002, 0x80000003, 0x80000004`	moderate
		`0x8FFFFFFF`	strong
Special			
`GetProcessHeap` & `CloseHandle`	–	–	moderate

Table B.2: Amount of samples in the benign-set that were found to use each individual indicator.

Method	**# Found Indicators**
API Calls: Strong	
`CreateFile`	1 / 0.5%
`FindWindow`	0 / 0.0%
`GetFileAttributes`	0 / 0.0%
`GetModuleHandle`	0 / 0.0%
`GetProcAddress`	1 / 0.5%
`IsNativeVhdBoot`	0 / 0.0%
`NtQueryLicenseValue`	0 / 0.0%
`RegOpenKey`	0 / 0.0%
`RegQueryValue`	0 / 0.0%
`WNetGetProviderName`	0 / 0.0%
API Calls: Moderate	
`CreateFile`	7 / 3.7%
`CreateTimerQueueTimer`	0 / 0.0%
`CreateToolhelp32Snapshot`	3 / 1.6%
`EnumServicesStatus`	0 / 0.0%
`EnumSystemFirmwareTables`	0 / 0.0%
`GetAdaptersAddresses`	0 / 0.0%
`GetAdaptersInfo`	2 / 1.0%
`GetCursorPos`	37 / 19.3%
`GetModuleFileName`	137 / 71.4%
`GetNativeSystemInfo`	47 / 24.5%
`GetProcAddress`	0 / 0.0%
`GetPwrCapabilities`	0 / 0.0%
`GetSystemFirmwareTable`	1 / 0.0%
`GetSystemInfo`	57 / 29.7%
`GetUserName`	7 / 3.79%
`GlobalMemoryStatus`	16 / 8.3%
`IcmpSendEcho`	0 / 0.0%
`Icmp6SendEcho`	0 / 0.0%
`IWbemServices::ExecQuery`	0 / 0.0%
`NtDelayExecution`	0 / 0.0%
`OpenSCManager`	1 / 0.5%
`Process32First`	3 / 1.6%
`Process32Next`	3 / 1.6%
`RegOpenKey`	1 / 0.5%
`RegQueryValue`	1 / 0.5%
`SetTimer`	0 / 0.0%
`SetWaitableTimer`	0 / 0.0%
`SetupDiGetDeviceRegistryProperty`	0 / 0.0%
`Sleep`	0 / 0.0%
`timeSetEvent`	0 / 0.0%
`WaitForSingleObject`	3 / 1.6%
API Calls: Weak	
`CreateEvent`	58 / 30.2%
`CreateTimerQueue`	0 / 0.0%
`CreateWaitableTimer`	1 / 0.5%
`DeviceIoControl`	0 / 0.0%
`GetDiskFreeSpace`	0 / 0.0%
`GetTickCount`	58 / 30.2%
`IcmpCreateFile`	0 / 0.0%
`Icmp6CreateFile`	0 / 0.0%
System Calls: Strong	
`NtCreateFile`	1 / 0.5%
`NtOpenKey`	1 / 0.5%
`NtQueryAttributesFile`	0 / 0.0%
`NtQueryValueKey`	0 / 0.0%
System Calls: Moderate	
`NtCreateFile`	7 / 3.7%
`NtOpenKey`	3 / 1.6%
`NtQueryValueKey`	2 / 1.0%
System Calls: Weak	
`NtDeviceIoControlFile`	0 / 0.0%
Instructions: Strong	
`CPUID – 0x40000000`	0 / 0.0%
`CPUID – 0x8FFFFFFF`	0 / 0.0%
Instructions: Moderate	
`CPUID – 0x80000002/3/4`	8 / 4.2%
Instructions: Weak	
`CPUID – 0x0`	187 / 97.4%
`CPUID – 0x1`	187 / 97.4%
Specials: Moderate	
`GetProcessHeap` & `CloseHandle`	72 / 37.0%

Table B.3: Amount of samples from the vm-detection-set that were found to use each individual indicator.

Method	# Found Indicators
API Calls: Strong	
`CreateFile`	8 / 0.6%
`FindWindow`	6 / 0.5%
`GetFileAttributes`	16 / 1.2%
`GetModuleHandle`	22 / 1.7%
`GetProcAddress`	258 / 20.0%
`IsNativeVhdBoot`	4 / 0.3%
`NtQueryLicenseValue`	0 / 0.0%
`RegOpenKey`	40 / 3.1%
`RegQueryValue`	0 / 0.0%
`WNetGetProviderName`	1 / 0.1%
API Calls: Moderate	
`CreateFile`	279 / 21.7%
`CreateTimerQueueTimer`	2 / 0.2%
`CreateToolhelp32Snapshot`	118 / 9.2%
`EnumServicesStatus`	3 / 0.2%
`EnumSystemFirmwareTables`	1 / 0.1%
`GetAdaptersAddresses`	43 / 3.3%
`GetAdaptersInfo`	152 / 11.8%
`GetCursorPos`	83 / 6.4%
`GetModuleFileName`	1,202 / 93.4%
`GetNativeSystemInfo`	522 / 40.6%
`GetProcAddress`	0 / 0.0%
`GetPwrCapabilities`	0 / 0.0%
`GetSystemFirmwareTable`	24 / 1.9%
`GetSystemInfo`	313 / 24.3%
`GetUserName`	308 / 23.9%
`GlobalMemoryStatus`	162 / 12.6%
`IcmpSendEcho`	0 / 0.0%
`Icmp6SendEcho`	0 / 0.0%
`IWbemServices::ExecQuery`	1 / 0.1%
`NtDelayExecution`	5 / 0.4%
`OpenSCManager`	68 / 5.3%
`Process32First`	119 / 9.2%
`Process32Next`	115 / 8.9%
`RegOpenKey`	238 / 18.5%
`RegQueryValue`	229 / 17.8%
`SetTimer`	15 / 1.2%
`SetWaitableTimer`	2 / 0.2%
`SetupDiGetDeviceRegistryProperty`	4 / 0.3%
`Sleep`	11 / 0.9%
`timeSetEvent`	1 / 0.1%
`WaitForSingleObject`	248 / 19.3%
API Calls: Weak	
`CreateEvent`	605 / 47.0%
`CreateTimerQueue`	6 / 0.5%
`CreateWaitableTimer`	5 / 0.4%
`DeviceIoControl`	6 / 0.5%
`GetDiskFreeSpace`	21 / 1.6%
`GetTickCount`	697 / 54.2%
`IcmpCreateFile`	0 / 0.0%
`Icmp6CreateFile`	0 / 0.0%
System Calls: Strong	
`NtCreateFile`	8 / 0.6%
`NtOpenKey`	38 / 3.0%
`NtQueryAttributesFile`	16 / 1.2%
`NtQueryValueKey`	0 / 0.0%
System Calls: Moderate	
`NtCreateFile`	277 / 21.5%
`NtOpenKey`	379 / 29.4%
`NtQueryValueKey`	299 / 23.2%
System Calls: Weak	
`NtDeviceIoControlFile`	1 / 0.1%
Instructions: Strong	
`CPUID` – `0x40000000`	102 / 7.9%
`CPUID` – `0x8FFFFFFF`	0 / 0.0%
Instructions: Moderate	
`CPUID` – `0x80000002/3/4`	27 / 2.1%
Instructions: Weak	
`CPUID` – `0x0`	1,259 / 97.8%
`CPUID` – `0x1`	1,259 / 97.8%
Specials: Moderate	
`GetProcessHeap` & `CloseHandle`	413 / 32.1%

Table B.4: Amount of samples from the main-set that were found to use each individual indicator.

Method	# Found Indicators
API Calls: Strong	
`CreateFile`	6 / 0.0%
`FindWindow`	0 / 0.0%
`GetFileAttributes`	20 / 0.0%
`GetModuleHandle`	310 / 0.7%
`GetProcAddress`	116 / 0.3%
`IsNativeVhdBoot`	0 / 0.0%
`NtQueryLicenseValue`	0 / 0.0%
`RegOpenKey`	241 / 0.5%
`RegQueryValue`	1 / 0.0%
`WNetGetProviderName`	0 / 0.0%
API Calls: Moderate	
`CreateFile`	1,553 / 3.4%
`CreateTimerQueueTimer`	79 / 0.2%
`CreateToolhelp32Snapshot`	5,455 / 12.0%
`EnumServicesStatus`	196 / 0.4%
`EnumSystemFirmwareTables`	26 / 0.1%
`GetAdaptersAddresses`	42 / 0.1%
`GetAdaptersInfo`	1,516 / 3.3%
`GetCursorPos`	3,305 / 7.3%
`GetModuleFileName`	30,449 / 66.9%
`GetNativeSystemInfo`	6,854 / 15.1%
`GetProcAddress`	14 / 0.0%
`GetPwrCapabilities`	15 / 0.0%
`GetSystemFirmwareTable`	137 / 0.3%
`GetSystemInfo`	7,835 / 17.2%
`GetUserName`	2,384 / 5.2%
`GlobalMemoryStatus`	5,610 / 12.3%
`IcmpSendEcho`	0 / 0.0%
`Icmp6SendEcho`	0 / 0.0%
`IWbemServices::ExecQuery`	11 / 0.0%
`NtDelayExecution`	87 / 0.2%
`OpenSCManager`	1,007 / 2.2%
`Process32First`	5,433 / 11.9%
`Process32Next`	5,420 / 11.9%
`RegOpenKey`	676 / 1.5%
`RegQueryValue`	632 / 1.4%
`SetTimer`	73 / 0.2%
`SetWaitableTimer`	10 / 0.0%
`SetupDiGetDeviceRegistryProperty`	27 / 0.1%
`Sleep`	295 / 0.6%
`timeSetEvent`	0 / 0.0%
`WaitForSingleObject`	372 / 0.8%
API Calls: Weak	
`CreateEvent`	13,833 / 30.4%
`CreateTimerQueue`	11 / 0.0%
`CreateWaitableTimer`	308 / 0.7%
`DeviceIoControl`	2 / 0.0%
`GetDiskFreeSpace`	4 / 0.0%
`GetTickCount`	17,615 / 38.7%
`IcmpCreateFile`	20 / 0.0%
`Icmp6CreateFile`	0 / 0.0%
System Calls: Strong	
`NtCreateFile`	6 / 0.0%
`NtOpenKey`	247 / 0.5%
`NtQueryAttributesFile`	15 / 0.0%
`NtQueryValueKey`	3 / 0.0%
System Calls: Moderate	
`NtCreateFile`	1,571 / 3.4%
`NtOpenKey`	2,143 / 4.7%
`NtQueryValueKey`	934 / 2.1%
System Calls: Weak	
`NtDeviceIoControlFile`	0 / 0.0%
Instructions: Strong	
`CPUID` – `0x40000000`	554 / 1.2%
`CPUID` – `0x8FFFFFFF`	0 / 0.0%
Instructions: Moderate	
`CPUID` – `0x80000002`	793 / 1.7%
`CPUID` – `0x80000003/4`	786 / 1.7%
Instructions: Weak	
`CPUID` – `0x0`	44,810 / 98.4%
`CPUID` – `0x1`	44,811 / 98.4%
Specials: Moderate	
`GetProcessHeap` & `CloseHandle`	17,747 / 39.0%

List of Figures

List of Tables

List of Listings

Bibliography

[1] Davide Balzarotti et al. "Efficient Detection of Split Personalities in Malware." In: *NDSS*. 2010.

[2] Michael Brengel, Michael Backes, and Christian Rossow. "Detecting Hardware-Assisted Virtualization". In: *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[3] Edmund Brumaghin et al. *CCleanup: A Vast Number of Machines at Risk.* `https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html`. 2017. (Visited on 07/22/2019).

[4] Xu Chen et al. "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 177–186.

[5] Cameron Coles. *Cloud Market in 2018 and Predictions for 2021.* `https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/`. 2018. (Visited on 08/31/2019).

[6] Paolo Milani Comparetti et al. "Identifying Dormant Functionality in Malware Programs". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 61–76.

[7] Wenrui Diao et al. "Evading Android Runtime Analysis Through Detecting Programmed Interactions". In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2016, pp. 159–164.

[8] Peter Ferrie. "Attacks on More Virtual Machine Emulators". In: *Symantec Technology Exchange* 55 (2007).

[9] Andrea Fortuna. *Malware VM detection techniques evolving: an analysis of GravityRAT.* `https://www.andreafortuna.org/2018/05/21/malware-vm-detection-techniques-evolving-an-analysis-of-gravityrat/`. 2018. (Visited on 07/30/2019).

[10] Nicholas Griffin. *Locky returned with a new Anti-VM trick.* `https://www.forcepoint.com/blog/x-labs/locky-returned-new-anti-vm-trick`. 2016. (Visited on 07/19/2019).

[11] Hexacorn. *Detecting Wine via internal and legacy APIs.* `http://www.hexacorn.com/blog/2016/03/27/detecting-wine-via-internal-and-legacy-apis/`. 2016. (Visited on 08/06/2019).

[12] Thorsten Holz and Frederic Raynal. "Detecting honeypots and other suspicious environments". In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. IEEE. 2005, pp. 29–36.

[13] Ralf Hund. *Word macro uses WMI to detect VM environments*. `https://www.vmray.com/cyber-security-blog/word-macro-detects-vm-environments/`. 2016. (Visited on 08/12/2019).

[14] Galen Hunt and Doug Brubacher. "Detours: Binary Interception of Win32 Functions". In: *Proceedings of the 3rd USENIX Windows NT Symposium*. 1999.

[15] Intel. "Intel® 64 and IA-32 Architectures Software Developer Manuals". In: *Volume 3C: Chapter 25 - VMX Non-Root Operation* 3C (2019).

[16] Min Gyung Kang et al. "Emulating emulation-resistant malware". In: *Proceedings of the 1st ACM workshop on Virtual machine security*. ACM. 2009, pp. 11–22.

[17] Tobias Klein. *scoopy doo - VMware Fingerprint Suite (Archived)*. `https://web.archive.org/web/20070102213035/http://www.trapkit.de/research/vmm/scoopydoo/README`. 2003. (Visited on 07/18/2019).

[18] Boris Lau and Vanja Svajcer. "Measuring virtual machine detection in malware using DSD tracer". In: *Journal in Computer Virology* 6.3 (2010), pp. 181–195.

[19] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. "Detecting Environment-Sensitive Malware". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 338–357.

[20] LordNoteworthy. *al-khaser*. `https://github.com/LordNoteworthy/al-khaser`. 2019.

[21] Najmeh Miramirkhani et al. "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 1009–1024.

[22] Jon Oberheide. *Detecting and Evading CWSandbox*. (2008-01-15) `https://jon.oberheide.org/blog/2008/01/15/detecting-and-evading-cwsandbox/`. 2008.

[23] Alfredo Andres Omella. "Methods for Virtual Machine Detection". In: *Grupo S21sec Gestión SA* (2006).

[24] Alberto Ortega. *Paranoid Fish*. `https://github.com/a0rtega/pafish`. 2016.

[25] Danny Quist, Val Smith, and Offensive Computing. "Detecting the Presence of Virtual Machines using the Local Data Table". In: *Offensive Computing* (2006).

[26] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. "Camouflage in Malware: From Encryption to Metamorphism". In: *International Journal of Computer Science and Network Security* 12.8 (2012), pp. 74–83.

[27] Thomas Roccia et al. “Malware Evasion Techniques and Trends”. In: *McAfee Labs Threats Report June 2017.* (2017). Available: `https://www.mcafee.com/enterprise/de-de/assets/reports/rp-quarterly-threats-jun-2017.pdf`, pp. 8–32.

[28] Joanna Rutkowska. *Red Pill... or how to detect VMM using (almost) one CPU instruction (Archived).* `https://web.archive.org/web/20070226215307/http://invisiblethings.org/papers/redpill.html`. 2004. (Visited on 07/18/2019).

[29] Monirul I Sharif et al. “Impeding Malware Analysis Using Conditional Code Obfuscation.” In: *NDSS.* 2008.

[30] Amit Vasudevan and Ramesh Yerraballi. “Cobra: Fine-grained Malware Analysis using Stealth Localized-executions”. In: *2006 IEEE Symposium on Security and Privacy (S&P’06).* IEEE. 2006, 15–pp.

[31] Ilsun You and Kangbin Yim. “Malware Obfuscation Techniques: A Brief Survey”. In: *2010 International conference on broadband, wireless computing, communication and applications.* IEEE. 2010, pp. 297–300.