Detection of Anti-VM Protections Used by Malware

Master Thesis

Master in *Sciences and Technologies*,
Specialty in *Computer Science*,
Track *Cryptology and Computer Security*.

Author
Elouan Charbonnier <elouan.charbonnier@etu.u-bordeaux.fr>

Supervisor
Pierre Mondon <pierre.mondon@eshard.com>
Lionel d'Hauenens <lionel.dhauenens@eshard.com>

Tutor
Emmanuel Fleury <emmanuel.fleury@u-bordeaux.com>

February 3, 2026

Declaration of authorship of the document

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of the Master in *Sciences and Technologies*, Specialty in *Mathematics* or *Computer Science*, Track *Cryptology and Computer Security*, is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Date and Signature

Acknowledgments

I would first like to thank my supervisors Pierre Mondon and Lionel d'Hauenens for their guidance, advice, and support throughout my internship.

I also thank the entire eShard team for their warm welcome and valuable help during this period.

Finally, I thank Emmanuel Fleury, my internship supervisor, for the opportunity to complete this internship at eShard.

Abstract

To analyze malware, using a disposable and isolated environment allows experimenting with the aforementioned software without taking risks. For this purpose, we use virtual machines that, when needed, can be reset to a previous state (snapshot), often isolated from any network or vulnerable machine. Thus, every malware developer takes this into account and integrates into their product so-called anti-VM techniques. These are techniques that detect the presence of virtual machines and modify the software's behavior to make its analysis more difficult.

This thesis focuses on these methods, their operation, and how to counter them. The first part explains the principles of virtualization and emulation as well as the use of the QEMU emulator. The second part explains a number of anti-VM protections based on Windows APIs and assembly instructions. Then, the third part explains how to counter these techniques using virtual machine configuration, hooks, and WinDbg. Finally, the fourth part shows the use of these techniques in two concrete cases: the PoC (Proof of Concept) malware `al-khaser` and a banking Trojan named *QakBot*.

Contents

Introduction . . . **1**

I Virtualization and Emulation . . . **2**

I.1 Definitions . . . **3**
I.1.1 Virtual Machine . . . 3
I.1.2 Hypervisor . . . 3
I.1.2.1 Types of Hypervisors . . . 3
I.1.3 Virtualization . . . 4
I.1.3.1 Full Virtualization . . . 4
I.1.3.2 Paravirtualization . . . 5
I.1.3.3 Hardware-Assisted Virtualization . . . 5
I.1.4 Emulation . . . 6

I.2 QEMU . . . **7**
I.2.1 Definition and Usage Modes . . . 7
I.2.1.1 General Presentation . . . 7
I.2.1.2 Usage Modes . . . 7
I.2.2 Translator Operation . . . 9
I.2.2.1 Instruction Translation . . . 9
I.2.2.2 Usefulness of a Translator . . . 9
I.2.2.3 Translation Block . . . 10
I.2.3 TCG, QEMU's Translator . . . 10
I.2.4 Component Emulation . . . 14
I.2.4.1 Memory Management Unit (*MMU*) . . . 14
I.2.4.2 Processor, Memory Regions, and Hard Drive . . . 14

II Anti-Virtualization/Anti-Emulation Techniques . . . **15**

II.1 Windows API . . . **16**
II.1.1 Hardware Interaction . . . 16
II.1.1.1 Screen Resolution . . . 16
II.1.1.2 Storage Size . . . 17
II.1.1.3 Memory Size . . . 18
II.1.1.4 MAC Address . . . 18
II.1.1.5 Number of CPUs . . . 20
II.1.1.6 Power Capabilities . . . 21
II.1.2 Artifact Interaction . . . 22
II.1.2.1 Services . . . 22
II.1.2.2 `RegKey` . . . 24
II.1.3 Human Interaction . . . 25
II.1.3.1 Mouse Movement . . . 25
II.1.3.2 Window Change . . . 26

II.2 Machine Instructions . . . **27**
II.2.1 `cpuid` . . . 27
II.2.1.1 Hypervisor Bit . . . 28
II.2.1.2 Hypervisor ID String . . . 28
II.2.2 `rdtsc` . . . 29
II.2.3 `sidt` . . . 30
II.2.3.1 `IDT` . . . 30

II.2.3.2 Technique Explanation . . . 30

II.3 Use of These Techniques in Malware . . . 32

III Countermeasure Methods . . . 34

III.1 Configuration Modification . . . 35
III.1.1 Increase Hard Drive Size . . . 36
III.1.2 Modify the Number of CPUs and Amount of RAM . . . 36
III.1.3 Increase Screen Resolution . . . 36
III.1.4 Modify the MAC Address . . . 36
III.1.5 Modify the Information that `cpuid` Will Retrieve . . . 36

III.2 *Hooks* of Windows API Functions . . . 37
III.2.1 DLL Injection . . . 37
III.2.2 Installing a Hook . . . 37
III.2.2.1 Explanation of Inline Hooking . . . 37
III.2.2.2 Example Code . . . 37
III.2.3 Setting Up Substitutes . . . 38

III.3 Apply a Patch . . . 40
III.3.1 Setting Up the Environment . . . 40
III.3.1.1 On the Target Machine . . . 40
III.3.1.2 On the Debugging Machine . . . 40
III.3.2 Entering the Context of a Process . . . 40
III.3.3 Finding the Entry Point of an Executable . . . 42
III.3.4 Example: Bypassing the `SIDT` Technique . . . 43

IV Practical Cases . . . 45

IV.1 *Al-Khaser* . . . 46
IV.1.1 Anti-VM Techniques for QEMU . . . 46
IV.1.1.1 `ACPI` . . . 47
IV.1.1.2 `SMBIOS` . . . 47
IV.1.2 Anti-VM Techniques for Hyper-V . . . 48
IV.1.3 General Anti-VM Techniques . . . 49
IV.1.3.1 *Windows Management Instrumentation* (WMI) . . . 49
IV.1.3.2 User Input . . . 49

IV.2 *QakBot* . . . 51
IV.2.1 Sample . . . 51
IV.2.2 Analysis . . . 51
IV.2.2.1 `sub_4033fc()` \ `avm_in_vmware_version()` . . . 52
IV.2.2.2 `sub_40349a()` \ `avm_in_vmware_memory_size()` . . . 53
IV.2.2.3 `sub_4035b6()` \ `avm_devices_check()` . . . 53
IV.2.2.4 `sub_40385e()` \ `avm_process_check()` . . . 54
IV.2.2.5 `sub_40336e()` \ `call_cpuid()` . . . 55
IV.2.3 Analysis Result . . . 55

Conclusion . . . 56

Appendices . . . 58
Annex 1 — Script permettant de retrouver le code assembleur de la traduction du code ARM . . . 58
Annex 2 — Fonction `SERVICES_NAME_LIST` et `belong_to` — Détection des services liés à des machines virtuelles . . . 59
Annex 3 — Liste de clefs de registre standard pour les gestionnaires de machines virtuelles . . . 60
Annex 4 — Fonction `qemu_firmware_ACPI` du maliciel `al-khaser` . . . 61
Annex 5 — Fonction `get_system_firmware` — récupération des tables micrologicielles . . . 63
Annex 6 — Fonction `enumerate_object_directory` — Permet de retrouver la liste des noms des objets présent dans un répertoire du gestionnaire d'objets. . . . 64

Annex 7 — Fichier `Generic.h` de `al-khaser` — En-tête contenant toutes les fontions d'anti-VM génériques. 65
Annex 8 — Fonction `ExecWMIQuery` — Permet d'exécuter une requête WMI. 66
Annex 9 — Fonctions `DetectSnxhkWindow` et `EnumWindows_find_match_class` - Teste la présence d'une fenêtre windows ayant un certains nom de classe 67
Annex 10 — Fonction `sub_403ef7` — Implémente un certains nombre de protection anti-VM. 68
Annex 11 — Fonction `sub_403ef7` renommée — Implémente un certains nombre de protection anti-VM. 69
Annex 12 — Listes des descriptions et noms de classe testées par *QakBot* 70
Annex 13 — Syntaxe de la fonction `SetupDiGetDeviceRegistryPropertyA` 71
Annex 14 — Fonction `avm_devices_check` 72
Annex 15 — Listes des processus testés par *QakBot* 74
Annex 16 — Fonction `avm_process_check` 75
Annex 17 — Fonction `cpuid_call()` 77

Bibliography **79**

Introduction

The development of malware requires in-depth knowledge of the target (API, architecture, etc.), but also an understanding of analysis methods. The objectives of malware vary, but the main ones are data exfiltration, remote control, or data encryption as part of a ransom demand. These objectives require persistence within a system, which increases their exposure to analysis tools.

The more traces a program leaves, the easier it is to understand its functioning and protect against it. For this reason, it is common for analysts to create isolated environments specialized in the study of malware (sandboxes), equipped with analysis tools that facilitate understanding of their operation.

To counter these analyses, many malware programs implement mechanisms that allow detection of these analysis environments. These techniques, called anti-VM, are essential for large-scale software. These mechanisms can lead to different behaviors for the software, but in most cases, these techniques lead to putting the software to sleep or modifying its behavior into something benign. For example, the Trojan horse *Blackmoon*[1] takes the form of an innocuous application. During its execution, it employs anti-VM and anti-debug methods to determine whether it should continue executing or return a message explaining that this application cannot be run in a virtual machine.

There are many possible methods to detect a virtualized (or emulated) environment, the most common being assembly instructions that allow querying the processor, hypervisor, or key system components. Some methods are reserved for certain systems; for example, the Windows API provides access to many system parameters such as mouse position, screen resolution, storage size, etc. As we will see, this information can be used to detect the presence of a virtual machine.

However, it is important to note that these protections can also backfire on their developers. Indeed, most companies tend to have a fleet of virtual machines to manage a variety of different tasks. Although these are virtual machines, not all are dedicated to malware analysis and detection. Furthermore, anti-VM techniques with too strict criteria will lead the malware to avoid machines with older configurations. Thus, it is important to know what limits to set for these mechanisms. Although very useful, it will be shown that these anti-VM protections remain much less widespread than other techniques, and when they are implemented, they tend to be quite rudimentary.

In this thesis, we will present the principles of virtualization and emulation, present commonly used anti-VM techniques, and show how to counter them. We will also present two concrete cases of malware using these anti-virtualization (or anti-emulation) methods.

[1] More commonly called *KrBanker*, the name *Blackmoon* comes from a debug string found in the first discovered sample of this malware.

Part I

Virtualization and Emulation

Chapter I.1

Definitions

I.1.1 Virtual Machine

A virtual machine, or guest machine, is a logical environment that imitates or reproduces the behavior of a physical machine. Generally, a virtual machine is created through a virtualization process, relying on a hypervisor responsible for managing and isolating guest systems[1]. However, by abuse of language, we sometimes also designate as "virtual machine" an emulated environment, i.e., entirely simulated by software, without resorting to a hypervisor or hardware virtualization extensions. In this case, the host system completely reproduces a new set of "emulated" components for the guest machine.

I.1.2 Hypervisor

A hypervisor is *software* that allows the creation and management of virtual machines. It is an indispensable element of virtualization, which provides, for each guest, a set of resources (processors, memory, peripherals).

I.1.2.1 Types of Hypervisors

There are two types of hypervisors [1]:

- Type 1 (*bare-metal*): The hypervisor is installed on the hardware and communicates directly with it.

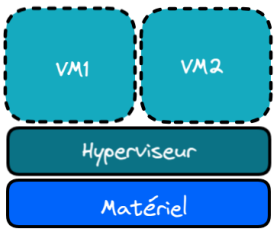

Figure I.1.1: Logical representation of a type 1 hypervisor

- Type 2 (*hosted*): The hypervisor will communicate with the hardware through the host operating system.

[1] We can also mention language execution environments such as the JVM for Java which, despite the designation "virtual machine," do not exactly fall under virtualization.

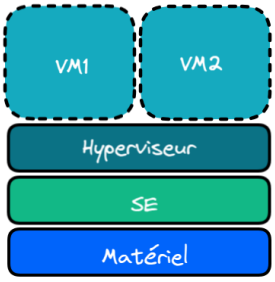

Figure I.1.2: Logical representation of a type 2 hypervisor

It is theoretically possible to use each of the two types of hypervisors for the different virtualization modes that we will present.

I.1.3 Virtualization

Virtualization is a set of techniques allowing hardware or software resources to be isolated to execute separate environments (virtual machines). As mentioned previously, virtualization generally relies on a hypervisor and, by extension, can also be associated with software emulation. In this section, we will focus on virtualization in its strictest sense. Three types of virtualization are distinguished:

- full virtualization: complete virtualization of physical hardware by the hypervisor.
- paravirtualization: modification of the guest so it is aware of the hypervisor's existence.
- hardware-assisted virtualization[2]: virtualization through hardware extensions (Intel VT-x, AMD-V) helping the hypervisor.

In the following sections, we will examine the operation of each technique in more detail.

I.1.3.1 Full Virtualization

In this virtualization mode, the guest system is not aware of its virtualization. The hypervisor provides it with a virtualized version of the physical hardware. This guarantees compatibility with the guest system without needing any modification.

To properly isolate guest systems, the hypervisor intercepts communications between the guest and physical hardware (for example, privileged instructions such as CPU access), then translates them via hardware virtualization mechanisms to ensure compatibility with non-virtualized physical hardware. During translation, the hypervisor must, for each of these communications, intercept them, process them, then return them; this will lead to significant overhead for each communication and therefore to a slowdown of the guest system.

[2]This is not really a technique in its own right, however, it remains an optimization that differs greatly from other techniques.

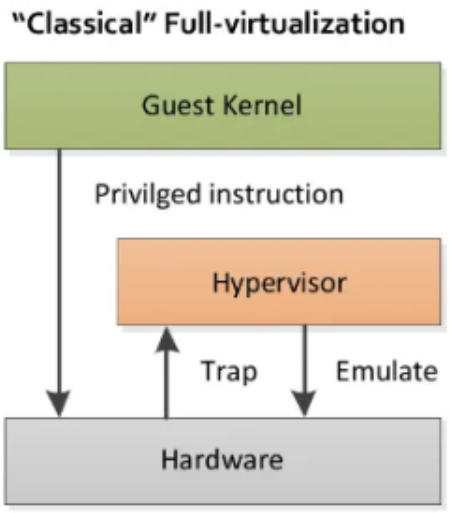

Figure I.1.3: Diagram representing full virtualization [2]

I.1.3.2 Paravirtualization

Paravirtualization is a virtualization mode in which the guest system is aware of its virtualization. In this case, the guest system communicates directly with the hypervisor (using *hypercalls*), which no longer requires communication translation; however, this requires modifications to the guest system.

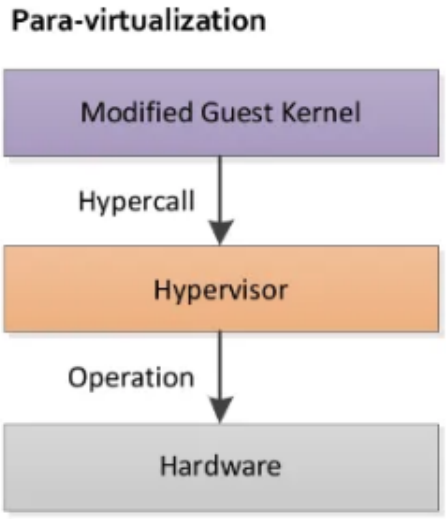

Figure I.1.4: Diagram representing paravirtualization [2]

I.1.3.3 Hardware-Assisted Virtualization

We will present the third type of virtualization, which is not really a separate type of virtualization (or emulation) but rather an enhancement. This technique solves one of the biggest problems of virtualization: execution speed.

This virtualization uses extensions integrated into processors, such as Intel VT-x or AMD-V. These extensions allow a processor, initially designed to run only one operating system, to directly assist with virtualization.

To do this, these extensions add an execution mode to the processor: guest mode. This mode will be used to execute guest code. The use of new instructions will allow mode switching when encountering a privileged instruction, which will require an action from kernel mode.

I.1.4 Emulation

Emulation consists of generating a set of hardware components (processor, bus, peripherals, etc.) to execute a guest system of a different architecture than the host. Unlike virtualization, emulation needs to translate instructions from the guest so the host can process them. This functionality allows multi-architecture compatibility at the cost of reduced performance.

In the next chapter, we will examine emulation operation in more detail by taking the example of the QEMU emulator.

Chapter I.2

QEMU

In this chapter, we explain in detail the operation of an emulator. To do this, we will take the example of QEMU [3].

I.2.1 Definition and Usage Modes

I.2.1.1 General Presentation

QEMU, Quick EMULator, is a system and binary emulator. It allows emulating (or virtualizing with a hypervisor) an architecture that may be different from the host's. This constitutes one of QEMU's greatest strengths along with being free software. QEMU's great versatility allows it to emulate a very large number of systems. It is also possible to use it with hypervisors such as *KVM* to switch from emulation to virtualization.

QEMU has two main usage modes: user mode, where it executes executables compiled for a different architecture than the host's, and system mode, where it emulates an entire system, including physical components such as the processor, RAM, various peripherals, etc.

In the context of reverse engineering and malware analysis, *PANDA* [4], based on QEMU, is often used. *PANDA* is dynamic analysis software that allows detailed exploration of the internal operation of the observed system. More specifically, *PANDA* gives the user access to all instructions executed by the target system. This allows recording absolutely everything that occurred in the system during a chosen time period.

As we will see later, it is also possible to retrieve QEMU's internal state (more specifically TCG's) at any time. In user mode, certain options allow, for example, retrieving assembly code in the guest architecture, once interpreted by TCG, then how TCG translates it for the host.

I.2.1.2 Usage Modes

We will detail the operation and usefulness of each QEMU mode, particularly how translation between two different architectures (guest and host) works, and what this implies for QEMU's internal architecture.

User Mode

As stated previously, this mode allows executing a binary compiled for a different architecture than the host machine's. It is currently possible to emulate a binary compiled for Linux (`qemu-linux-user`) and BSD (`qemu-bsd-user`)[1]. The diagram below represents the state of a machine when QEMU is used in emulation mode:

[1] For a particular architecture, it is possible to call `qemu-<arch>` with `arch` being an architecture supported by QEMU. For example, for ARM architecture [5], we would use `qemu-arm`.

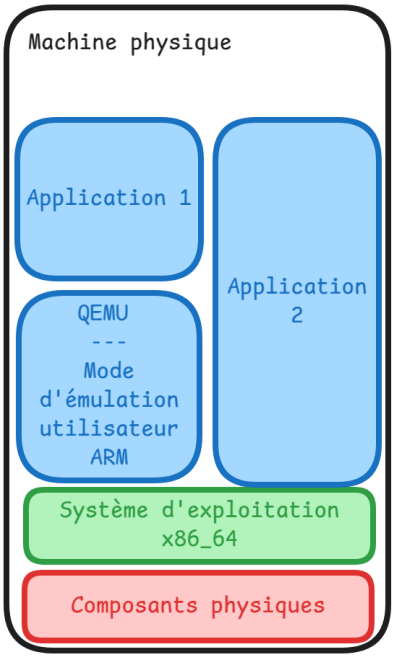

Figure I.2.1: Diagram of QEMU's user emulation mode

As indicated in QEMU documentation, user mode emulation has several characteristics:

- System call translation: Translation system for system calls that allows translating instruction parameters so that host and guest can communicate.
- POSIX signal management: Redirects signals from the host to the guest program.
- Thread management: Allows managing one host thread (virtual processor) for each guest thread.

System Mode

In system mode, QEMU will emulate an entire system, i.e., emulate all physical resources the system needs. As mentioned previously, QEMU can be coupled with a hypervisor from the following list [6]:

Accelerator	Host OS	Host Architectures
KVM	Linux	Arm (64 bit only), MIPS, PPC, RISC-V, s390x, x86
Xen	Linux (as dom0)	Arm, x86
Hypervisor Framework (hvf)	MacOS	x86 (64 bit only), Arm (64 bit only)
Windows Hypervisor Platform (whpx)	Windows	x86
NetBSD Virtual Machine Monitor (nvmm)	NetBSD	x86
Tiny Code Generator (tcg)	Linux, other POSIX, Windows, MacOS	Arm, x86, Loongarch64, MIPS, PPC, s390x, Sparc64

Figure I.2.2: List of hypervisors supported by QEMU and compatible hosts [6]

If, when using QEMU, no hypervisor is specified, QEMU uses a dynamic translation engine (`JIT`) called Tiny Code Generator (TCG). As defined in QEMU documentation related to TCG:

> *The Tiny Code Generator (TCG) exists to transform target insns (the processor being emulated) via the TCG frontend to TCG ops which are then transformed into host insns (the processor executing QEMU itself) via the TCG backend.*

Below is a diagram representing QEMU's operation in system mode:

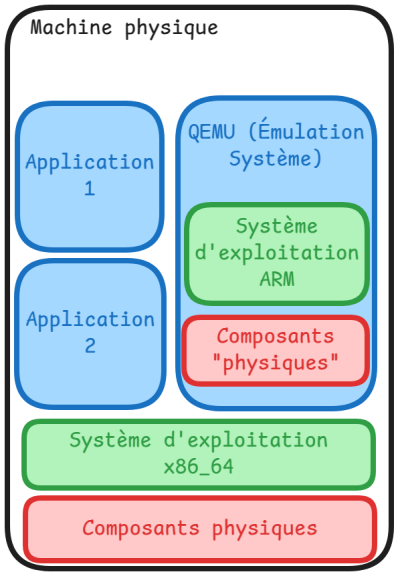

Figure I.2.3: Diagram of QEMU's system emulation mode

I.2.2 Translator Operation

Before examining QEMU's internal operation, we will first look at how a translator works, such as TCG [7] [8].

I.2.2.1 Instruction Translation

When emulating a system with a different architecture than the host, it is necessary to translate instructions. Indeed, two different architectures have two distinct instruction sets (ISA), which means two codes doing the same thing are not represented the same way in assembly. For example, if we take ARM assembly code and x64 code, and we want to perform a simple addition, we will have the following codes:

```
1  mov rax, 5
2  mov rbx, 3
3  add rax, rbx
```

```
1  mov r0, #5
2  mov r1, #3
3  add r0, r0, r1
```

As we can see, register names, the way to load a value into a register, and the syntax for addition are different between the x64 instruction set (left) and the ARM instruction set (right).

So, if we want to execute a binary compiled for ARM architecture on an x64 system, we must be able to make the host system (x64) understand what the guest code (ARM) does. Although we could simply translate from ARM to x64, it is also possible (and recommended) to use an instruction translator.

I.2.2.2 Usefulness of a Translator

Practically speaking, it is possible to emulate one system on another without using a generic translator[2]. The problem is that this approach only works on a case-by-case basis. For example, if we want to emulate guest system 1 on the host system, we can translate guest 1's language so the host understands it.

However, if we want to emulate other architectures (MIPS, x64, etc.) in addition to the one we already have (ARM), we cannot keep the previous solution which is only designed to translate one type of architecture. Indeed, in this case, the translator will try to translate instructions from other architectures without ever succeeding.

To address this, we must use a translator such as TCG which will receive instructions, translate them into its own instruction set (pseudo-architecture), then retranslate them for the target architecture. Thus, we have the following operation:

[2] https://www.usenix.org/legacy/publications/library/proceedings/usenix-nt97/full_papers/chernoff/chernoff.pdf

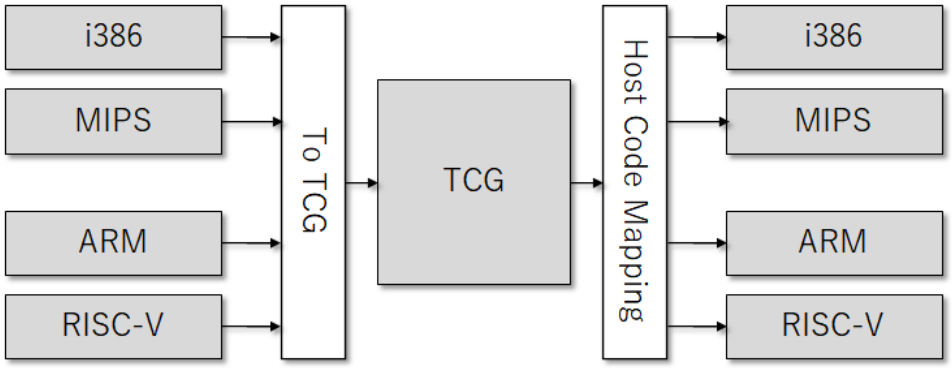

Figure I.2.4: Example of TCG translator, [9]

I.2.2.3 Translation Block

In assembly code, TCG separates pieces of this code into translation blocks. These blocks are separated either by jump instructions or by a size defined by TCG. TCG will translate the guest code, interpret it in its own language, then generate code for the host.

A translation block consists of the following elements:

- The address of the first instruction in the block.
- The machine code after translation.

These translation blocks can be reused by TCG; moreover, before translating any instruction, TCG checks if it has not already translated this block.

I.2.3 TCG, QEMU's Translator

We will now examine the exact operation of QEMU's internal architecture. We will see in more detail how translation is managed with TCG.

As explained previously, to emulate a different architecture than the host's, we use a translator - in QEMU's case, it's *Tiny Code Generator* (TCG).

Throughout this part, we will take as an example the following ARM code and execute it using QEMU on an x86-64 machine:

```
@ Compilation:
@ arm-linux-gnueabi-as -o add.o add.s
@ arm-linux-gnueabi-ld -o add add.o

.section .text
.global _start

_start:
    mov r0, #5
    mov r1, #7
    add r0, r0, r1
    mov r7, #1
    svc #0
```

We will emulate the execution of this binary using the following command:

```
qemu-arm -d in_asm,op,out_asm -D add.log -singlestep ./add
```

Here is an explanation of the command and its parameters [10]:

- `qemu-arm`: Runs an ARM program in user emulation mode.
- `-d`: Fills a log of binary execution with different options, defined as follows in the documentation:
  - `in_asm`: Shows guest assembly code (ARM).

  - `op`: Shows TCG's internal representation code.
  - `out_asm`: Shows assembly code generated for the host (x86-64).
- `-D`: Creates a file for the retrieved log.
- `-singlestep`: Runs the program stopping to fill the log for each instruction.

This command will retrieve the translations performed by TCG during binary execution, then write them to a file. We will therefore have the assembly code in ARM, the TCG internal representation code, and the code generated for the host.

We will then study the following log excerpt to understand how translation works:

```
IN:
0x00010054:
OBJD-T: 0500a0e3

OP:
 ld_i32 loc3,env,$0xfffffffffffffff0
 brcond_i32 loc3,$0x0,lt,$L0
 st8_i32 $0x1,env,$0xfffffffffffffff4

 ---- 0000000000010054 0000000000000000 0000000000000000
 mov_i32 loc6,$0x5
 mov_i32 r0,loc6
 mov_i32 pc,$0x10058
 call lookup_tb_ptr,$0x6,$1,tmp9,env
 goto_ptr tmp9
 set_label $L0
 exit_tb $0x736b64000043

OUT: [size=64]
  -- guest addr 0x0000000000010054 + tb prologue
0x736b64000100:
OBJD-H: 8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfd
OBJD-H: ff1512000000ffe0488d0514ffffffe9e4feffff
  -- tb slow paths + alignment
0x736b64000134:
OBJD-H: 90909090
  data: [size=8]
0x736b64000138:  .quad  0x000059c6dad02c20
```

There are three sections per code block:

- `IN`: Guest code section containing instruction address and instruction *opcode*
- `OP`: Section containing translation of guest code into TCG internal representation instructions.
- `OUT`: Section containing x86-64 code generated using TCG's internal representation.

Let's study these sections in detail, starting with the first section which is the guest code.

Guest Code

```
IN:
 0x00010054:
 OBJD-T: 0500a0e3
```

We find in the log, in the `IN` section, the ARM code executed by the guest which is at address `0x10054`. It is not represented in assembly in the log, but in *opcode*.

The *opcode* `0500a0e3` corresponds to the instruction:

```
mov         r0, #5
```

This instruction will be retrieved by TCG, then transformed to match TCG's internal representation syntax.

TCG Internal Representation Code

```
1  OP:
2   ld_i32 loc3,env,$0xfffffffffffffff0
3   brcond_i32 loc3,$0x0,lt,$L0
4   st8_i32 $0x1,env,$0xfffffffffffffff4
5
6   ---- 0000000000010054 0000000000000000 0000000000000000
7   mov_i32 loc6,$0x5
8   mov_i32 r0,loc6
9   mov_i32 pc,$0x10058
10  call lookup_tb_ptr,$0x6,$1,tmp9,env
11  goto_ptr tmp9
12  set_label $L0
13  exit_tb $0x736b64000043
```

From line 7 to 8, there is the translation into TCG's internal representation of the ARM instruction. We have the instruction `mov_i32 loc6,$0x5` which will place `5` in `loc6`, then the instruction `mov_i32 r0,loc6` will place the value contained in `loc6` into `r0`.

Finally, from line 9 to 13, the code serves to jump to the next translation block (*Translation Block*).

Code Generated for Host

```
1  OUT: [size=64]
2    -- guest addr 0x0000000000010054 + tb prologue
3  0x736b64000100:
4  OBJD-H: 8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfd
5  OBJD-H: ff1512000000ffe0488d0514ffffffe9e4feffff
6    -- tb slow paths + alignment
7  0x736b64000134:
8  OBJD-H: 90909090
9    data: [size=8]
10 0x736b64000138:  .quad  0x000059c6dad02c20
```

In this part, the generated code is given as a hexadecimal string shown on lines 4 and 5:

```
8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfdff1512000000ffe0488d0514ffffffe9e4feffff
```

We can verify this using the Python script in Annex 1 which writes the translated code to a file.

We will find in the `add_x86_64` file the assembly code that was generated by TCG for the host:

```
1  mov ebx,[rbp-0x10]
2  test ebx,ebx
3  jl near 0x28
4  mov byte [rbp-0xc],0x1
5  mov dword [rbp+0x0],0x5
6  mov dword [rbp+0x3c],0x10058
7  mov rdi,rbp
8  call [rel 0x38]
9  jmp rax
10 lea rax,[rel 0xffffffffffffff43]
11 jmp 0xffffffffffffff18
```

We can find at line 5 the x86_64 version of the instruction `mov r0, #5` that we had in ARM.

Below is a diagram summarizing the translation of the instruction `mov r0, #5`:

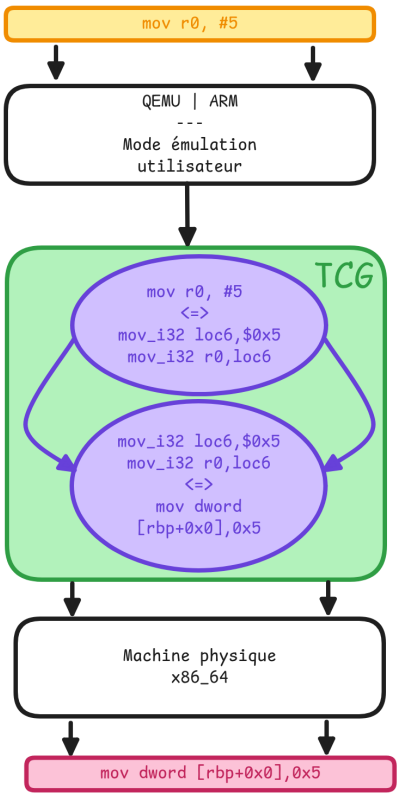

Figure I.2.5: Diagram summarizing the translation of an ARM instruction

I.2.4 Component Emulation

We have seen how QEMU translates instructions from one architecture to another. However, QEMU is also capable of emulating an entire system, which requires hardware components such as a processor, RAM, or hard drive. QEMU must therefore be able to emulate them. In this section, we will see the components that QEMU emulates and how it proceeds.

I.2.4.1 Memory Management Unit (*MMU*)

As indicated in QEMU documentation [11], the MMU, which is responsible for translating virtual addresses to physical addresses, is named *softmmu* in QEMU. *softmmu* is used by TCG to emulate virtual memory.

To achieve this, it implements a TLB (Translation Lookaside Buffer) which is a memory cache that will record the last memory pages the processor had to access. This will translate the address into a value which, combined with the untranslated address, will point to an address in QEMU's user space.

I.2.4.2 Processor, Memory Regions, and Hard Drive

We will now see how QEMU emulates three of a machine's most important components: the processor, RAM, and an NVME.

Processor

In emulation, the emulated processor is TCG which translates guest instructions into instructions the host understands. During virtualization, it is the hypervisor that acts as the processor from the guest's point of view.

Memory Regions

Memory regions are modeled by QEMU's memory API [12]. This API allows representing different types of memory regions; they are represented by a single C class `MemoryRegion`. The modeled memory regions are as follows:

- RAM: Memory range of the host reserved for the guest, `memory_region_init_ram()`.
- MMIO (*Memory-mapped I/O*): Guest memory range, managed by the host; each write or read calls the host, `memory_region_init_io()`
- ROM (*Read Only Memory*): On read, the API directly accesses a region of host memory. On write, it uses the same principle as for the MMIO region, `memory_region_init_rom_device()`
- IOMMU (*Input–Output Memory Management Unit*): Translates access addresses for this region to another region, `memory_region_init_iommu()`

Memory modeling is in the form of an acyclic graph of `MemoryRegion`; leaves correspond to RAM and MMIO while the rest of the nodes correspond to buses, memory controllers, and other memory regions.

Hard Drive

By default, QEMU uses the `qcow2` file format [13] for its disk images; `qcow` stands for *QEMU Copy On Write*. As its name indicates, this file type uses lazy allocation to only use storage when necessary. We can generate an image of this disk type using the `qemu-img` command.

To access storage, which is only a logical representation of the host's physical storage space, QEMU uses memory controllers which, in turn, use *Block Devices* [14] facilitating writing and reading data, as well as transferring data to permanent storage, among other things.

We have presented an overview of QEMU's internal operation and the possibilities offered by this software. In the next part, we will see a list of anti-emulation techniques that can be used to detect a virtual machine. Most of these techniques focus on a Windows machine that we generated using QEMU.

Part II

Anti-Virtualization/Anti-Emulation Techniques

Chapter II.1

Windows API

The Windows API [15] offers a large number of functions for efficient system interaction. Many of these functions can be used to detect if a system is virtualized. We can either examine the hardware that is emulated or the software running within the virtual machine.

Important Note

The techniques presented are not unique; there are many other ways to implement each technique. Additionally, the code excerpts presented are functions that return `true` if a virtual machine is detected.

II.1.1 Hardware Interaction

As explained in Chapter I.1, either the virtual machine emulates the host machine's components, or it provides new virtual components using the host's resources. For each technique presented, we will see why this technique works in an anti-virtualization context and how malware can use it to detect a virtual machine.

II.1.1.1 Screen Resolution

The first technique concerns screen resolution. It is based on the following principle: the majority of screens used today have a resolution that rarely falls below a certain threshold. However, when creating a virtual machine, or more precisely a test environment, it is generally not necessary to make the environment particularly complete. We often limit ourselves to a basic configuration: this includes a default screen resolution of 1024×768 in QEMU (or even lower).

We therefore seek to retrieve the resolution and compare it with a standard size. To do this, we will use, for Windows, the `GetSystemMetrics` function. This function will retrieve the requested measurement or configuration parameter; here is the syntax:

```
int GetSystemMetrics(
  [in] int nIndex
);
```

The `nIndex` parameter can take a large number of values; each of these integer values corresponds to a parameter. Among these, only two interest us [16]:

- `SM_CXSCREEN` (0): Screen width in pixels
- `SM_CYSCREEN` (1): Screen height in pixels

We just need to compare them with a standard resolution; in our case, we will take **1024×768**.

Below is a C function showing how to use this technique for Windows:

```
#include <windows.h>
#include <stdio.h>

bool check_RESOLUTION()
{
    int length = GetSystemMetrics(SM_CXSCREEN);
    int height = GetSystemMetrics(SM_CYSCREEN);

    return length <= 1024 || height <= 768;
}
```

II.1.1.2 Storage Size

This technique is identical in principle to the previous one, namely exploiting common hardware characteristics. However, instead of focusing on screen resolution, we will this time turn our attention to the machine's storage capacity.

Hard drive size has greatly increased since the first models. In the 1990s, hard drive size rarely exceeded 20 GB. Only fairly recently have we found hard drives with sizes exceeding 1 TB. The diagram below shows this evolution:

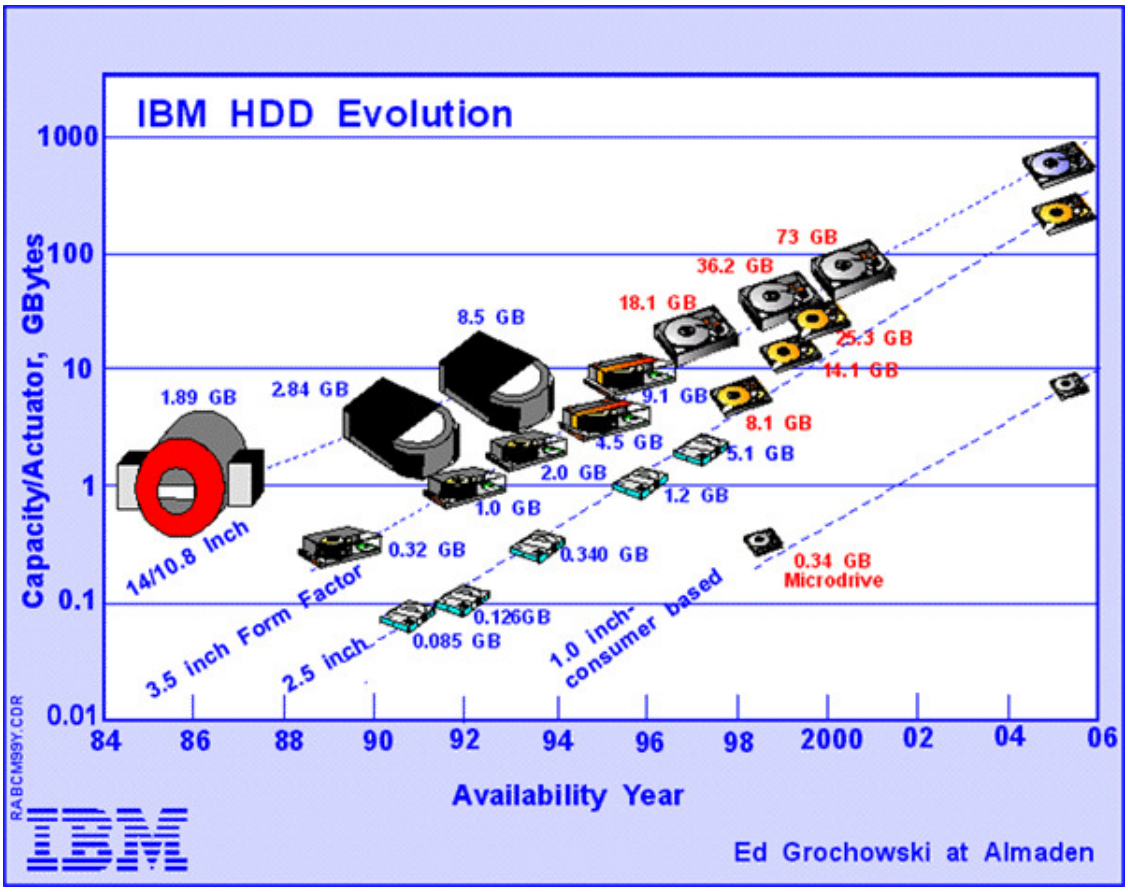

Figure II.1.1: Evolution of hard drive size [17]

Like screen resolution, nowadays it is rare to find a machine with "small" storage size. This property gives rise to two techniques:

- Test the total hard drive storage size
- Test the remaining hard drive storage size

Software can detect if it is being used in a virtual machine by relying on this. Indeed, when creating a virtual machine, its virtual disk size is limited by the hard drive we have[1]. This means that the total hard drive capacity as well as remaining storage space will have values lower than what can be found in physical machines.

[1]This can be countered using the *thin provisioning* technique, which allows allocating more memory than physically exists.

To use this, we will need the Windows API function `GetDiskFreeSpaceExA` [18]:

```
BOOL GetDiskFreeSpaceExA(
  [in, optional]  LPCSTR          lpDirectoryName,
  [out, optional] PULARGE_INTEGER lpFreeBytesAvailableToCaller,
  [out, optional] PULARGE_INTEGER lpTotalNumberOfBytes,
  [out, optional] PULARGE_INTEGER lpTotalNumberOfFreeBytes
);
```

This function will retrieve the total size (`lpTotalNumberOfBytes`) and remaining space (`lpTotalNumberOfFreeBytes`) of a hard drive designated by its name (`lpDirectoryName`). We just need to compare `lpTotalNumberOfBytes` and `lpTotalNumberOfFreeBytes` with size (and remaining space) values to determine if we are in a virtual machine. An example below in C code form:

```c
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>

bool check_HARD_DRIVE_SIZE()
{
    ULARGE_INTEGER freeBytesAvailable, totalBytes, totalFreeBytes;
    GetDiskFreeSpaceExA("C:\\", &freeBytesAvailable, &totalBytes, &totalFreeBytes);

    /* 214748364800 --> 200 GB || 32212254720 --> 30 GB */
    return totalBytes.QuadPart < 214748364800 || freeBytesAvailable.QuadPart <
    32212254720;
}
```

II.1.1.3 Memory Size

Identical in principle to the previous technique, this one will focus on available RAM. As explained previously, a virtual machine has its characteristics limited by those of the host machine; for example, RAM allocated to the guest cannot exceed what is available on the host.

Additionally, just like hard drive storage capacity, the amount of available memory has also increased considerably. Thus, if RAM size is too small (less than 4 GB), we can deduce either that we are on an old machine or in a virtual machine limited by its host. To retrieve RAM size, we will use the Windows API function `GlobalMemoryStatusEx` [19]:

```
BOOL GlobalMemoryStatusEx(
  [in, out] LPMEMORYSTATUSEX lpBuffer
);
```

This function will fill a `MEMORYSTATUSEX` structure which includes a `ullTotalPhys` field representing the total number of bytes in RAM. We can therefore implement the technique described above as follows:

```c
#include <windows.h>
#include <stdio.h>

bool check_MEMORY()
{
    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);

    return memInfo.ullAvailPhys / 1024 / 1024 / 1024 <= 4;
}
```

II.1.1.4 MAC Address

The network card is an essential component of every computer. Additionally, each of these cards has a unique identifier: the MAC address. This MAC address is a 6-byte address, represented in hexadecimal form (example: `12:34:56:78:9a:bc`). It consists of two parts[2]:

[2]Actually, we can say that the MAC address has 4 parts rather than 2; however, these two additional parts, which are the bit showing if the address is unique and the bit showing if the address is local, can be grouped with another part of the address (*OUI*).

- *Organizationally Unique Identifier (OUI)* (first three bytes): contains the identifier of a *NIC* manufacturer.
- *Network Interface Controller (NIC) Specific* (last three bytes): contains the card's unique identifier to distinguish it from other cards of the same manufacturer.

This address is a unique physical identifier for each card on each physical computer; however, this is also the case for virtual machines.

Indeed, when creating a virtual machine, the virtual machine manager (QEMU, VirtualBox, VMware, etc.) provides a network interface. The latter will therefore provide an emulated network card. To respect MAC address uniqueness, each virtual machine manager has its own manufacturer identifier (*NIC Specific*). For example:

- For VirtualBox: `08:00:27`
- For VMware: `00:0C:29`
- For QEMU/KVM: `52:54:00`

We can use this property to detect the presence of a virtual machine. To do this, we will need to access the machine's MAC address and compare it with different standard MAC addresses. Let's see how to implement this technique for a Windows machine.

We will use the `GetAdaptersInfo` function from the `iphlpapi.h` library; it is described as follows by **Microsoft** documentation:

> *The GetAdaptersInfo function retrieves adapter information for the local computer.*

And its syntax is as follows [20]:

```
IPHLPAPI_DLL_LINKAGE ULONG GetAdaptersInfo(
  [out]     PIP_ADAPTER_INFO AdapterInfo,
  [in, out] PULONG           SizePointer
);
```

This function will fill a `PIP_ADAPTER_INFO` structure with information about the network card. In this structure, we will focus on the `Address` field:

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO *Next;
  DWORD                   ComboIndex;
  char                    AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char                    Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT                    AddressLength;
  BYTE                    Address[MAX_ADAPTER_ADDRESS_LENGTH];
...
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

We will therefore implement the technique as follows:

- Retrieve the list of present network interfaces (`GetAdaptersInfo`)
- For each interface, format the found address into a character string of the form `XX:XX:XX:XX:XX:XX`
- And finally, test if the target machine's address contains a standard MAC address for a virtual machine.

Below is a possible implementation of this technique:

```
#include <windows.h>
#include <iphlpapi.h>
#include <stdio.h>
#include <string.h>

#define MAX_MAC_ADDRESSES 16
static char *recurrent_mac[6] = {"08:00:27", "00:0C:29", "00:1C:14", "00:50:56",
"00:05:69", "52:54:00"};

bool check_MAC()
```

```
{
    IP_ADAPTER_INFO adapter_info[MAX_MAC_ADDRESSES];
    DWORD dwBufLen = sizeof(adapter_info);

    DWORD dwStatus = GetAdaptersInfo(adapter_info, &dwBufLen);
    if (dwStatus != ERROR_SUCCESS)
    {
        return false;
    }

    PIP_ADAPTER_INFO pAdapter = adapter_info;

    while (pAdapter)
    {
        char mac_address[18];
        sprintf(mac_address, "%02X:%02X:%02X:%02X:%02X:%02X", pAdapter->Address[0],
        pAdapter->Address[1],
                pAdapter->Address[2], pAdapter->Address[3], pAdapter->Address[4],
                pAdapter->Address[5]);
        for (int i = 0; i < sizeof(recurrent_mac) / sizeof(recurrent_mac[0]); i++)
        {
            if (strstr(mac_address, recurrent_mac[i]) != NULL)
            {
                return true;
            }
        }
        pAdapter = pAdapter->Next;
    }
    return false;
}
```

II.1.1.5 Number of CPUs

A modern computer uses (in most cases) more than one logical processor, and quite often more than 4. However, when creating a virtual machine, it is possible to use fewer. The purpose of a virtual machine is not necessarily to be as performant as possible.

This technique is based on this principle. To apply it, we will retrieve the number of logical processors and compare it to an arbitrary value (four in our case). To do this, we will use the Windows API function `GetSystemInfo` [21]. This function will fill a `SYSTEM_INFO` structure defined as follows:

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

In this structure, we will use the `dwNumberOfProcessors` field which represents the number of logical processors in a system. We just need to compare this field with the chosen arbitrary value. Below is an example implementation:

```c
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>


bool check_NB_CPU()
{
    SYSTEM_INFO info;
    GetSystemInfo(&info);
    int nb_cpu = info.dwNumberOfProcessors;
    return nb_cpu < 4;
}
```

II.1.1.6 Power Capabilities

The Windows API allows a user to retrieve most system information. It is notably possible to obtain information about the system's power capabilities; these capabilities comply with the *ACPI* (*Advanced Configuration and Power Interface*) specification.

To retrieve this information, we will use the `GetPwrCapabilities` function [22] which will fill a `SYSTEM_POWER_CAPABI` structure containing information about the system's power capabilities:

```c
typedef struct {
... /* rest of the field */
  BOOLEAN                 SystemS1;
  BOOLEAN                 SystemS2;
  BOOLEAN                 SystemS3;
  BOOLEAN                 SystemS4;
  BOOLEAN                 SystemS5;
... /* rest of the field */
  BOOLEAN                 ThermalControl;
  BOOLEAN                 ProcessorThrottle;
  BYTE                    ProcessorMinThrottle;
  BYTE                    ProcessorThrottleScale;
... /* rest of the field */
} SYSTEM_POWER_CAPABILITIES, *PSYSTEM_POWER_CAPABILITIES;
```

We can use this structure to implement two anti-emulation techniques: component temperature control and sleep states.

Temperature Control

A Windows system will manage its component temperatures using thermal zones. A thermal zone is a logical partition that encompasses one or more components. These zones are represented in hardware by sensors that will capture heat and transmit it to the system, which will link them to thermal zones. When a component overheats, the sensor will transmit the temperature to the system which will match it to a thermal zone. This allows the system to act appropriately to cool the component in question.

This technique will use the `ThermalControl` field which will be used by the system to know if the system is capable of controlling component temperature. If so, then the field will be set to `True`; otherwise it will be set to `False`.

A physical machine is obliged to control its component temperatures; otherwise, it risks overheating and damaging itself. Conversely, a virtual machine does not have this obligation; indeed, even in emulation (or full virtualization), the system only simulates a "virtual" version of components. These "virtualized" (or emulated) components therefore cannot overheat and consequently cannot be linked to thermal zones.

Thus, for this technique, we just need to know if the `ThermalControl` field is present; in this case, we are in a physical machine or not. Here is an implementation:

```c
#include <windows.h>
#include <powrprof.h>
#include <stdbool.h>


bool check_THERMAL_CONTROL()
```

```
{
    SYSTEM_POWER_CAPABILITIES power_caps;

    if (GetPwrCapabilities(&power_caps) == true)
    {
        return !power_caps.ThermalControl;
    }
    return false;
}
```

Sleep States

In a system, there are several sleep states defined as ACPI states from S0 to S5 with certain variants:

- S0 (Working): System usable by the user (graphical display, interactions, etc.)
- S0 idle (Sleep): Equivalent to S0 state but inactive at low power; however, it can activate quickly and exit the idle state.
- S1 to S3 (Sleeping): System appears disabled and consumes less energy than S0; only one of these three states can be active in a system.
- S4 (Hibernation): System appears disabled; energy consumption is minimal. The system saves memory content to a file.
- S5 (*Soft off*): System appears disabled; it is a complete shutdown that will restart the user session at each reboot.

These sleep modes are standardized for physical machines. In the case of a virtual machine, certain sleep modes may be absent. The objective of this technique will therefore be to retrieve information about these sleep modes and check if they are present in the target system.

In the `SYSTEM_POWER_CAPABILITIES` structure, there are five fields, `SystemS1` to `SystemS5` which correspond to the five sleep modes described above. We will therefore test each of these sleep modes and if none is present, here is a possible implementation for this technique:

```
#include <windows.h>
#include <powrprof.h>
#include <stdbool.h>


bool check_SLEEP_MODES()
{
    SYSTEM_POWER_CAPABILITIES power_caps;

    if (GetPwrCapabilities(&power_caps) == true)
    {
        return !(power_caps.SystemS1 || power_caps.SystemS2 || power_caps.SystemS3
        || power_caps.SystemS4);
    }
    return false;
}
```

II.1.2 Artifact Interaction

To properly virtualize a system, the guest may use virtualization peripherals (for example *Virtiofs*). These peripherals may have services assigned to them and will therefore appear in Windows' active service list. Additionally, in Windows, there are also registry keys (`RegKeys`) assigned to virtualization extensions or virtual machine managers.

II.1.2.1 Services

As explained, in a virtual machine, there may be services that signal the presence of an emulated (virtualized) environment. For example, when creating a shared folder or choosing to modify the virtual machine's screen resolution, we add additional services. For example, for shared folders, we can use `Virtiofs` which will create an active service named `VirtioFsSvc`. We can also take the example of a machine created by VirtualBox which will therefore have services containing the character string `VBox`.

To implement this technique, we will need to build a string array containing possible names for the searched services. To list active services, we will use Windows API functions:

- `OpenSCManager`: allows connecting to the service control manager by providing a *handle*.
- `EnumServicesStatusA`: enumerates services in the service control manager; requires using a connection *handle* to the manager provided by `OpenSCManager`.

First, we will retrieve the *handle*; here is the syntax of `OpenSCManager` [23]:

```
SC_HANDLE OpenSCManagerA(
  [in, optional] LPCSTR lpMachineName,
  [in, optional] LPCSTR lpDatabaseName,
  [in]           DWORD  dwDesiredAccess
);
```

We will retrieve the *handle* without passing as argument the machine name (`lpMachineName`) and the service manager database name (`lpDatabaseName`).

However, we will specify the last argument which is necessary: the access right to the service manager. Since we wish to enumerate services, we will use the `SC_MANAGER_ENUMERATE_SERVICE` right, necessary for calling the `EnumServicesStatus` or `EnumServicesStatusEx` functions [24] which allow listing services.

The syntax of `EnumServicesStatus` [25] is as follows:

```
BOOL EnumServicesStatusA(
  [in]                SC_HANDLE              hSCManager,
  [in]                DWORD                  dwServiceType,
  [in]                DWORD                  dwServiceState,
  [out, optional]     LPENUM_SERVICE_STATUSA lpServices,
  [in]                DWORD                  cbBufSize,
  [out]               LPDWORD                pcbBytesNeeded,
  [out]               LPDWORD                lpServicesReturned,
  [in, out, optional] LPDWORD                lpResumeHandle
);
```

The `EnumServicesStatusA` function will fill a `LPENUM_SERVICE_STATUSA` structure which is an array of elements containing information about each service such as their name. Here is its syntax:

```
typedef struct _ENUM_SERVICE_STATUSA {
  LPSTR          lpServiceName;
  LPSTR          lpDisplayName;
  SERVICE_STATUS ServiceStatus;
} ENUM_SERVICE_STATUSA, *LPENUM_SERVICE_STATUSA;
```

Then we just need to compare the name of each found service with our predefined service name list. If we find a match, we can then conclude that we are in a virtual machine.

Here is a possible implementation:

```
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define NB_SERVICES 15

/* Definition of belong_to and SERVICES_NAME_LIST */
...


bool check_SERVICES()
{
    SC_HANDLE hSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ENUMERATE_SERVICE);
    if (hSCManager == NULL)
    {
```

```
        printf("OpenSCManager failed! Error: %lu\n", GetLastError());
        return false;
    }

    ENUM_SERVICE_STATUS serviceStatus[1024];
    DWORD bytesNeeded = 0, servicesReturned = 0, resumeHandle = 0;

    if (!EnumServicesStatus(hSCManager, SERVICE_WIN32, SERVICE_STATE_ALL,
    serviceStatus, sizeof(serviceStatus),
                            &bytesNeeded, &servicesReturned, &resumeHandle))
    {
        printf("EnumServicesStatus failed! Error: %lu\n", GetLastError());
        CloseServiceHandle(hSCManager);
        return false;
    }

    for (DWORD i = 0; i < servicesReturned; i++)
    {
        if (belong_to(serviceStatus[i].lpServiceName))
            return true;
    }

    CloseServiceHandle(hSCManager);

    return false;
}
```

The `belong_to` function and the `SERVICES_NAME_LIST` array can be found in Annex 2.

II.1.2.2 RegKey

Before explaining the anti-emulation technique, it is good to recall what a `RegKey`, short for *Registry Key*, is. A `RegKey` is a Windows registry entry that includes all sorts of system information. These keys have three fields: a name, a type, and data corresponding to the key. The data differs depending on the key. It can be a hexadecimal value or a character string representing a path, description, etc. This registry will therefore contain a bunch of keys concerning certain applications present in the system.

When QEMU generates a virtual machine, it will install in the guest a *daemon* called *QEMU Guest Agent*. This software is used to allow communication between the host and guest. Being part of the system, this software will have a `RegKey` assigned to it; we can find it at the following path in the registry editor:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\QEMU-GA
```

Each virtual machine manager has its own software that will allow these host-guest exchanges, so it is possible to detect them using registry keys. This technique is based on the possibility of retrieving a registry key. We will try to access a registry key. If it is possible, then this means that software related to virtualization (or emulation) exists.

To implement this technique, we will use the Windows API function `RegOpenKeyExW` [26]; this function has the following syntax:

```
LSTATUS RegOpenKeyExW(
  [in]           HKEY    hKey,
  [in, optional] LPCWSTR lpSubKey,
  [in]           DWORD   ulOptions,
  [in]           REGSAM  samDesired,
  [out]          PHKEY   phkResult
);
```

The `lpSubKey` parameter is the path of the registry key we wish to access. We will therefore create an array of registry key paths that indicate the presence of virtualization software and access each of these keys. If we succeed in accessing one of them, then we can conclude that we are in a virtual machine. Here is an implementation; the list of tested registry keys can be found in Annex 3:

```
#include <stdbool.h>
#include <stdio.h>
```

```
#include <windows.h>

...

bool check_REGS() {
  HKEY regkey;
  DWORD i;
  char value[1024];
  DWORD size;
  DWORD type;

  bool result = false;

  for (i = 0; i < sizeof(RegValuePath) / sizeof(RegValuePath[0]); i++) {
    if (RegOpenKeyExW(HKEY_LOCAL_MACHINE, RegValuePath[i], 0, KEY_READ,
                      &regkey) == ERROR_SUCCESS) {

                        result = true;
    }

  }
  RegCloseKey(regkey);
  return result;
}
```

II.1.3 Human Interaction

The most obvious way to detect a virtual machine is to detect any trace of unusual behavior for a physical machine. However, it is also possible to look at the machine's reactions based on its user's activity. Indeed, in most cases, a user performs expected actions, such as moving the mouse, interacting with windows, using the keyboard, etc.

In this section, we will present two techniques based on this principle.

II.1.3.1 Mouse Movement

In certain analysis environments, executable study can be done by recording the execution flow. During this recording, it is common for the graphical interface to freeze and no longer allow the user to interact with it. This allows eliminating superfluous instructions in the recording, such as those used to move the mouse.

However, a user will regularly move the mouse more or less voluntarily, regardless of context. This technique exploits this behavior to determine if the code is executing in a virtual environment. To do this, we will use a Windows API function: `GetCursorPos` [27]. This function has the following syntax:

```
BOOL GetCursorPos(
  [out] LPPOINT lpPoint
);
```

This function fills a `POINT` structure which has two fields:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT, *NPPOINT, *LPPOINT;
```

The retrieved `POINT` structure corresponds to the mouse cursor's position on the screen.

We will therefore need to compare the cursor position at two different time intervals. To do this, just call `GetCursorPos` once, let some time pass (with `Sleep` for example), then `GetCursorPos` again. The following code represents a possible implementation of this technique:

```
#include <stdbool.h>
#include <stdio.h>
#include <windows.h>

bool check_MOUSE() {
```

```
  POINT start, end;

  GetCursorPos(&start);
  Sleep(8000);
  GetCursorPos(&end);

  return start.x == end.x && start.y == end.y;
}
```

II.1.3.2 Window Change

As mentioned previously, in certain analysis environments, the graphical interface may freeze. This means that the currently foreground window will not change for a certain period of time. We can exploit this behavior by calling the `GetForegroundWindow` function [28] twice, spaced by a call to `Sleep`. If the two retrieved windows are identical, then we can deduce that the interface remained frozen, which is typical of certain analysis environments.

The `GetForegroundWindow` function is described as follows:

> *Retrieves a handle to the foreground window (the window with which the user is currently working).*

We will therefore retrieve a first handle using the `GetForegroundWindow` function, then wait a bit with the `Sleep` function. We retrieve a second handle and finally compare the two handles. If they are different, then it is unlikely to be an analysis environment. Below is a possible implementation of this technique:

```
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>


bool check_FOREGROUND()
{
    HWND foregroundHandle1 = GetForegroundWindow();
    Sleep(2500);
    HWND foregroundHandle2 = GetForegroundWindow();

    return foregroundHandle1 == foregroundHandle2;
}
```

Chapter II.2

Machine Instructions

We have seen certain anti-emulation techniques based on the Windows API, which will only work on a Windows system. We will now present some techniques based on assembly instructions. This allows (partially) making these techniques interoperable between different operating systems[1].

II.2.1 `cpuid`

The `cpuid` instruction is an assembly instruction introduced by Intel in 1993[2]. This instruction is used to retrieve processor information. To access this information, it is necessary to access a *leaf* of the instruction, i.e., a particular behavior of the instruction according to an input.

In the case of `cpuid`, the instruction takes input in the `eax` register[3] and returns requested information in `eax`, `ebx`, `ecx`, and `edx`. The Wikipedia page for `cpuid` precisely details for each input (value of `eax`) what information will be returned and in which registers. The diagram below summarizes the use of this instruction:

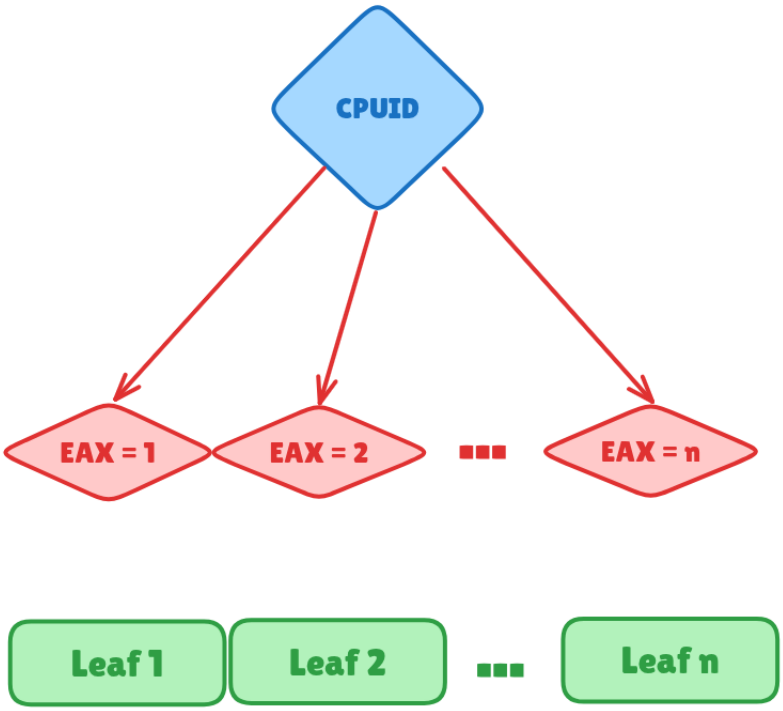

Figure II.2.1: Summary of `cpuid` instruction usage

This instruction can be used by all users, so it is an excellent way for malware to know if it is virtualized. There

[1] However, care must be taken to adapt each of these instructions according to the target system's architecture; calling conventions can change and some of these instructions are common to recent processors.
[2] Most modern processors, whether Intel or not, implement this instruction.
[3] The `cpuid` instruction only takes 32-bit registers, even in a 64-bit architecture.

are several techniques stemming from this instruction.

II.2.1.1 Hypervisor Bit

The first technique consists of detecting a hypervisor's presence. To do this, we will query the processor via the `cpuid` instruction, using leaf `0x1`. If we are in a virtual machine, the instruction will retrieve the value 1 from the hypervisor bit in `ecx`, more specifically bit 31 of the register. Conversely, if we are not in a virtual machine, then bit 31 of register `ecx` will equal 0; we just need to compare the retrieved value and return `true` if it equals "1" and `false` otherwise. Below is an example C and assembly implementation:

```
#include <stdio.h>
#include <stdbool.h>

bool check_HYPERVISOR_BIT()
{
    bool isVM = false;

    __asm__ volatile (
        "mov $1, %%eax\n\t"
        "cpuid\n\t"
        "bt $31, %%ecx\n\t"
        "setc %0\n\t"
        : "=r"(isVM)
        :
        : "eax", "ebx", "ecx", "edx", "cc"
    );

    return isVM;
}
```

II.2.1.2 Hypervisor ID String

The second technique consists of retrieving the hypervisor's identifier string. This element is a character string that allows authenticating the hypervisor used. When using the `cpuid` instruction in a virtual machine, the hypervisor will intercept the request to return its information and not the physical processor's information. However, this information contains clues about this same hypervisor's presence, notably the hypervisor's "brand."

This "brand," also called the hypervisor identifier string, is a 12-character string representing which virtual machine manager provides this hypervisor. For example, the hypervisor identifier string for:

- VirtualBox is `VBoxVBoxVBox`
- VMware is `VMwareVMware`
- QEMU is `TCGTCGTCGTCG`

These are only examples; there are many others.

In a physical machine, there is no hypervisor, so the hypervisor identifier string will be empty. To retrieve this string, use the `cpuid` instruction with `eax = 0x40000000`, which will return the first four characters in `ecx`, the next four in `ebx`, and the last four in `edx`. Then just test if the string is empty or if it corresponds to a known identifier. It can be implemented as follows in C:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define NB_BRAND_STRING 6

const char *known_hypervisor_id[] = {
    "KVMKVMKVM",    /* KVM */
    "Microsoft Hv", /* Hyper-V */
    "VMwareVMware", /* VMware */
    "XenVMMXenVMM", /* Xen */
    "VBoxVBoxVBox", /* VirtualBox */
    "TCGTCGTCGTCG"  /* QEMU */
};
```

```
static char hypervisor_id[13];

bool check_HYPERVISOR_ID() {
  unsigned int eax, ebx, ecx, edx;
  char hypervisor_id[13];

  __asm__ volatile("movl $0x40000000, %%eax;"
                   "cpuid;"
                   : "=b"(ebx), "=c"(ecx), "=d"(edx)
                   :
                   : "%eax"
    );

  *(unsigned int *)&hypervisor_id[0] = ebx;
  *(unsigned int *)&hypervisor_id[4] = ecx;
  *(unsigned int *)&hypervisor_id[8] = edx;
  hypervisor_id[12] = '\0';

  for (int i = 0; i < NB_BRAND_STRING; i++) {
    if (strstr((char *)hypervisor_id, known_hypervisor_id[i])) {
      return true;
    }
  }
  return false;
}
```

II.2.2 `rdtsc`

The `rdtsc` instruction [29] (meaning *read time stamp counter*) is used to retrieve the number of processor *cycles* since its last reset. This instruction will read the *Time Stamp Counter* register and place the content in `edx` and `eax`.

We can use `rdtsc` to count the number of *cycles* necessary for executing a serialized instruction. In a physical machine, each instruction takes a certain number of *cycles*; for example, on average, the `cpuid` instruction takes between 150 and 500 cycles. However, on a virtual machine, the `cpuid` instruction's cycle count is between 3,000 and 6,000.

We can observe such a difference for several reasons:

- When we retrieve processor information with `cpuid`, we cause a virtualization exit (**VMEXIT**, i.e., stop the code executing in the guest machine) and hand over to the hypervisor which will handle the interruption.
- If the virtual machine is emulated, then there is a strong chance that the number of cycles required for executing an instruction is higher than in a physical machine.
- Since the virtual machine depends on the physical machine's components, slowdowns can be observed within the machine if the host uses too many resources.

We will therefore implement detection as follows:

1. Repeat the operation a certain number of times (here 1000) to obtain a representative average.
2. At each iteration:
   a) Read and record the number of processor cycles elapsed since startup in a `start` variable using the `rdtsc` instruction.
   b) Execute the `cpuid` instruction, which acts as a serialized instruction.
   c) Read the cycle number again and record it in `end`.
   d) Calculate the difference `end - start`, representing the number of cycles taken by `cpuid`, and add this value to a total.
3. Calculate the average of measured cycles over all iterations (delta).
4. Return `true` if this average exceeds 500 cycles, which may indicate a virtualized environment.

We thus obtain the following C code:

```c
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

#if defined(_WIN32) || defined(_WIN64)
#include <intrin.h>
#pragma intrinsic(__rdtsc)
#pragma intrinsic(__cpuid)
#else
#include <cpuid.h>
#include <x86intrin.h>
#endif

static inline void cpuid__() {
  asm volatile("cpuid" : : "a"(0) : "ebx", "ecx", "edx");
}

bool check_RDTSC() {
  const int test_nb = 1000;
  unsigned int cycles = 0;
  unsigned int start = 0;
  unsigned int end = 0;

  for (int i = 0; i < test_nb; i++) {

    start = __rdtsc();
    cpuid__();
    end = __rdtsc();

    cycles += end - start;
  }

  unsigned int delta = cycles / test_nb;
  return delta > 500;
}
```

II.2.3 `sidt`

In this technique, we will see how to use a system's *Interrupt Descriptor Table* (in English, *IDT - Interrupt Descriptor Table*). Let's first look at what this structure looks like.

II.2.3.1 `IDT`

The IDT is a data structure (in table form) used by x86 architecture processors. It serves to list all system interrupt descriptors, so there are 256 entries, each of eight bytes (one entry per interrupt). This structure's location is recorded in a specific register called `IDTR` (*IDT Register*); in case of an interrupt, the processor only needs to find the *IDT* address by reading the *IDTR* and subsequently find the interruption in this table. However, this table is unique to each processor core and therefore to each processor. This means that when creating a virtual machine, the `IDT` must be at a different location than the physical processor's. This is the principle on which the technique we will present is based.

II.2.3.2 Technique Explanation

As explained above, when creating a virtual machine, the interrupt table (`IDT`) is often placed at a different address than the one used on a physical machine. Generally, this address is located higher in physical memory. Malware can thus detect if it is running in a virtual machine by retrieving the `IDT` address using the `SIDT [30]` instruction, then analyzing the bytes of the obtained address. If one of the high-order bytes is sufficiently high — typically close to `0xFF` — this may indicate execution in a virtualized environment. This technique, known as **Red Pill** [31], was presented by Joanna Rutkowska[4].

[4]Joanna Rutkowska's blog: The Invisible Things Blog

Several implementations of this technique exist. It depends mainly on the target operating system: indeed, not all IDTs are placed in the same memory location depending on the system. In the example below, the function aims to target a Windows x64 operating system's IDT. The IDT address in this case is often around `0x80ffffff` [32].

The presented implementation is only one of many possible variants of this technique:

```
#include <stdio.h>
#include <stdbool.h>

bool check_SIDT()
{
    struct
    {
        uint16_t limit;
        uint64_t base;
    } __attribute__((packed)) idtr;

    __asm__("sidt %0" : "=m"(idtr));

    uint8_t third_byte = (idtr.base >> 16) & 0xFF;
    return third_byte != 0xFF;
}
```

Chapter II.3

Use of These Techniques in Malware

As we saw previously, there are a large number of techniques; however, most malware only uses a small part of them. We find a very large number of uses of `cpuid` (II.2.1) and `rdtsc` (II.2.2); other techniques are quite popular (MAC address retrieval, storage available space analysis, etc.), although less so than the previous two. The article *Longitudinal Study of the Prevalence of Malware Evasive Techniques* [33] exposes this behavior in the following table:

Malware [2016, 2020]	**APT**
`ErasePEHeader` (8.9%)	`IsDebuggerPresentAPI` (7.5%)
`IsDebuggerPresentAPI` (7.1%)	`disk_getdiskfreespace` (5.2%)
`RDTSC` (5.9%)	`RDTSC` (4.1%)
`process_enum` (5.2%)	`CanOpenCsrss` (3.0%)
`vm_check_mac` (4.1%)	`process_enum` (2.7%)
`IsDebuggerPresentPEB` (4.1%)	`IsDebuggerPresentPEB` (2.5%)
`disk_getdiskfreespace` (3.8%)	`time_stalling` (2.3%)
`SizeOfImage` (3.8%)	`SizeOfImage` (2.2%)
`memory_space` (2.4%)	`vm_check_mac` (2.1%)
`CanOpenCsrss` (2.3%)	`NSIT_ThreadHideFromDebugger` (1.4%)

Figure II.3.1: List of presence rates of the most frequent evasion techniques (anti-VM, anti-analysis, etc.) in generic malware and APTs.

According to the article *Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware* [34], malware, on average, prefers to use anti-debug for more general uses and anti-VM when there is a particular target[1], as confirmed by the following table from their report:

[1] We observed this behavior in the first *QakBot* sample; most anti-VM techniques were made to detect *VMware*.

Family	% Anti-debug.	% Anti-VM	Family	% Anti-debug.	% Anti-VM
Sality	89.6%	76.2%	Ramnit	85.8%	71.6%
Zbot	72.9%	39.7%	Zeroaccess	41.6%	50.4%
Winwebsec	80.0%	52.9%	Reveton	74.8%	62.8%
Targeted (APT)	68.6%	84.2%			

Table 4: Percentage of samples using anti-debugging/anti-VM techniques in each malware family

This confirms the idea advanced by the first article [33]. However, this trend is evolving. Indeed, the graph below, from the article *Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware* [34], confirms it:

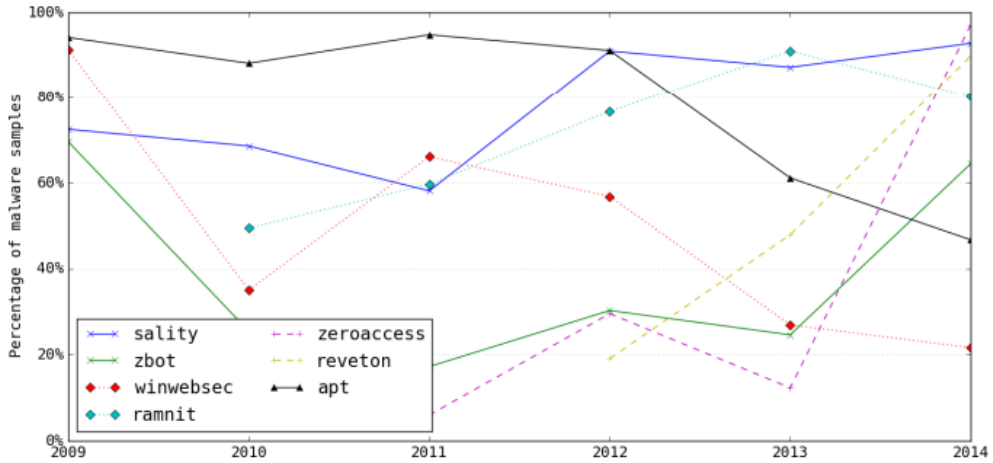

Figure 5. Evolution of the use of anti-VM techniques

Overall, anti-VM techniques have become democratized in malware. Despite a decrease in their presence in more recent samples, these techniques remain useful for malware developers and disruptive for analysts.

We will therefore see how to protect against these techniques.

Part III

Countermeasure Methods

Chapter III.1

Configuration Modification

When creating a virtual machine in a VM manager, it is possible to configure all the technical characteristics of the machine, such as disk size, amount of RAM, type of processor, etc.

In this section, we will present how to configure your virtual machine to counter the following techniques:

- Storage size
- Number of CPUs
- Memory size
- Screen resolution
- MAC address
- `cpuid`

Throughout this chapter, we will present the modifications to be made to counter certain methods. It is important to note that some modifications to counter these techniques are quite restrictive and impractical. This is why we will present in the next chapters (III.2 & III.3) other methods to counter them differently. We will use the `qemu-system-x86_64` command [35] to generate virtual machines.

Each of the values chosen in the following sections is not the only one that works; we base ourselves on a standard present among computer manufacturers. For example, the number of cores and the size of the memory have greatly evolved over the past 20 years, which allows us to establish a threshold of "normality" for these values.

Here is the command used to launch our virtual machine[1]:

```
qemu-system-x86_64 \
 -enable-kvm \
 -m 8G \
 -smp 6 \
 -cpu qemu64,-hypervisor,kvm=off \
 -drive file=dummy_windows.qcow2,format=qcow2 \
 -netdev user,id=net0 \
 -device e1000,netdev=net0,mac=52:55:44:33:22:11 \
 -vga virtio \
 -device usb-ehci,id=usb \
 -device usb-tablet,bus=usb.0
```

Important Note

The command above is for launching the machine once created (without an operating system). To add an operating system, simply add the following line to the command above:
`-drive file=<PATH_TO_ISO>,media=cdrom`
With `PATH_TO_ISO`, the path to the desired system image.

[1] We have virtualized it with KVM to obtain better performance.

III.1.1 Increase Hard Drive Size

The first detection technique we will counter is the analysis of the hard drive size (or SSD). Most hard drives have a size between 512 GB and 1 TB, and generally, only 100 to 300 GB remain on the hard drive of a computer. The size of the created hard drive must therefore have at least 200 GB (arbitrary value) free at all times. For example, for a virtual machine used to test the execution of a binary, 512 GB of storage will be sufficient. First, 512 GB is a reasonable size for the storage of a physical machine. Moreover, this machine will only contain the operating system and a few files, so at least 450 GB will remain.

To achieve this, we will create a sufficiently large disk image using the `qemu-img` command:

```
qemu-img create -f qcow2 dummy_windows.qcow2 512G
```

III.1.2 Modify the Number of CPUs and Amount of RAM

When creating the virtual machine, the first characteristics to configure are the number of processors and the amount of RAM. Our goal will therefore be to give our virtual machine a sufficient number of processors and memory to make it resemble a physical machine. However, this machine will be constrained by the resources of the host.

We have seen previously that if we use too small an amount of processors or RAM, it is possible for a code to detect that it is running in a virtual machine using techniques that use the amount of memory (II.1.1.3) or the number of processors (II.1.1.5). This is why we have chosen, in the configuration, six processors and 8 GB of RAM. We will use the `-smp 6` option which will generate a machine with 6 processors and `-m 8G` which will allocate 8 gigabytes for our machine.

III.1.3 Increase Screen Resolution

To be able to modify the resolution, once in the virtual machine, we need to add the `-vga` option which will allow us to emulate a VGA card. We will use the `virtio` drivers to be able to modify the resolution of the guest system as much as possible. Then, simply access the machine's settings and manually increase the resolution. Here is the option to add to the command:

```
-vga virtio
```

III.1.4 Modify the MAC Address

As we have seen, the MAC address (cf. II.1.1.4) provided by default by QEMU is recognizable by its first two bytes `52:54`. We must modify the address. For this, we will use the `-device` option which will allow us to create a new device. The syntax of this option is as follows:

```
device driver[,prop[=value][,...]]
```

In our case, we will create a network card with the `e1000` option. To modify the MAC address of this card, simply use the `mac=` property. Here is the complete option:

```
-device e1000,netdev=net0,mac=52:55:44:33:22:11
```

III.1.5 Modify the Information that `cpuid` Will Retrieve

We will now take care of the techniques using the `cpuid` instruction. For this, we will use the `-cpu` option with two different properties, `-hypervisor`[2] to hide the hypervisor bit and `kvm=off` to hide the hypervisor name. We therefore have the following option:

```
-cpu host,-hypervisor,kvm=off
```

[2]The `-hypervisor` option does not exist in the official QEMU documentation.

Chapter III.2

Hooks of Windows API Functions

In this section, we will see a way to bypass anti-emulation protections based on the Windows API using hooks. The presented approach is from a personal project available on GitHub, `https://github.com/HalfTimeOfLife/Panoptiv`, all the code snippets found below are from this repository.

III.2.1 DLL Injection

There are several ways to inject a DLL into a process, we can:

- Use a code specifically made to inject a DLL into a program as used in the project.
- Divert a DLL to replace it with a tampered version.
- Use a debugger extension (for example `!injectdll` from WinDbg[1]).

III.2.2 Installing a Hook

Several techniques exist to install a hook, we can use an existing library, for example Detours, or we can implement a more explicit solution: modify the bytes of the targeted function (Inline Hooking). That is, we will replace the prologue of the function with a `jmp` instruction to our hooked function. We will see below how to install a hook from a DLL.

III.2.2.1 Explanation of Inline Hooking

As explained above, to install a hook, we will need to replace the prologue of a function call. In Windows, the prologue of a function is often the same:

```
mov edi, edi
push ebp
mov ebp, esp
```

The `push ebp` and `mov ebp, esp` instructions are used to create a new call stack for the called function. The `mov edi, edi` instruction is a `nop` on two bytes, this feature is used to allow easier debugging of a function. This set of instructions corresponds to the following hexadecimal: `89FF5589E5`.

We will therefore seek to replace this prologue (five bytes) with a new set of five bytes that will correspond to a `jmp <FUNCTION_ADDRESS>`.

III.2.2.2 Example Code

First, here is the syntax of our function:

```
void InstallHook(char *moduleName, char *functionName, void *hookFunction, BYTE
*backupBytes, void **originalFunction);
```

The parameters are as follows:

[1] Author's article: !injectdll – a WinDbg extension for DLL injection

1. `moduleName`: Name of the module containing the function we want to hook.
2. `functionName`: Name of the function we want to hook.
3. `hookFunction`: Pointer to the function that will replace the function we are going to hook.
4. `backupBytes`: Pointer to a variable that will save the original bytes.
5. `originalFunction`: Pointer to a variable that will save the address of the original function we want to hook.

For the hook installation to be successful, we need several pieces of information. First, we need the address of the function we want to hook. For this, we will use the `GetProcAddress` function:

```
/* Get handle of module containing function to hook */
HMODULE hModule = GetModuleHandleA(moduleName);
if (hModule == NULL) {
    printf("[-] Failed to get module handle for %s\n", moduleName);
    return;
}

/* Get address of function to hook */
FARPROC pFunction = GetProcAddress(hModule, functionName);
if (pFunction == NULL) {
    printf("[-] Failed to get function address for %s in %s\n", functionName,
    moduleName);
    return;
}
```

Next, we can start installing the hook. This operation is done in several steps. We will start by saving the first 5 bytes of the original function and modifying the memory protections of the targeted function (`VirtualProtect`):

```
DWORD oldProtect;
memcpy(backupBytes, pFunction, 5);

DWORD relAddr = ((DWORD)hookFunction - (DWORD)pFunction) - 5;

VirtualProtect(pFunction, 5, PAGE_EXECUTE_READWRITE, &oldProtect);
```

Next, we will create a 5-byte array that will correspond to the new prologue of the function we have hooked, these 5 bytes will represent the `jmp <HOOK_ADDRESS>` instruction, then using `memcpy`, we will replace the old prologue with these new bytes that will divert the program's execution flow to our function:

```
BYTE patch[5] = { 0xE9 };
memcpy(patch + 1, &relAddr, 4);
memcpy(pFunction, patch, 5);

/* Set back the memory protection as it was */
VirtualProtect(pFunction, 5, oldProtect, &oldProtect);
```

Then, we save the pointer of the original function in a variable, to have the possibility to call it in the hook:

```
if (originalFunction)
    *originalFunction = (void*)pFunction;
```

III.2.3 Setting Up Substitutes

Once we are able to replace a Windows API function, all we have to do is implement substitutes for each function. For example, if we want to bypass the hard drive size detection technique (cf. II.1.1.2), we can create a new version of the `GetDiskFreeSpaceExA` function as follows:

```
BOOL WINAPI HookedGetDiskFreeSpaceExA(LPCSTR lpDirectoryName, PULARGE_INTEGER
lpFreeBytesAvailableToCaller, PULARGE_INTEGER lpTotalNumberOfBytes, PULARGE_INTEGER
lpTotalNumberOfFreeBytes) {

        if (!lpDirectoryName || !lpFreeBytesAvailableToCaller ||
        !lpTotalNumberOfBytes || !lpTotalNumberOfFreeBytes) return FALSE;
```

```
    lpTotalNumberOfBytes->QuadPart = 1000000000000;
    lpTotalNumberOfFreeBytes->QuadPart = 500000000000;

    return TRUE;
}
```

This technique is supposed to detect if the amount of storage is large enough to prove that it belongs to a physical machine. If this is not the case, then we can deduce that the program is running in a virtual machine.

The `HookedGetDiskFreeSpaceExA` function will always give the same total and remaining storage amount, a sufficiently large value has been chosen to bypass the technique described above. This can be done with any Windows API function[2].

We will now see how to counter anti-VM techniques using a debugger, which allows us to control the behavior of a malware in real time.

[2] All substitutes for Windows API functions can be found in the file `PanoptivDLL/dllmain.c` in the Panoptiv repository.

Chapter III.3

Apply a Patch

In this section, we will see how it is possible to bypass the anti-emulation protections of an executable using a debugger. Here, we will use `WinDbg` [36] in remote kernel debugging to try to enter the context of an executable called `AVMBinary_SIDT_no_stub.exe`, it is an executable that implements the anti-emulation protection `sidt` (cf. II.2.3).

III.3.1 Setting Up the Environment

To debug a kernel remotely, it is necessary to use a virtual machine in addition to the machine we want to debug. This machine must be on the same network as the target machine, as it must be able to connect to it using, for example, a gateway (software).

III.3.1.1 On the Target Machine

Before focusing on this machine, we must first properly configure the target machine to "allow" and enable debugging. To do this, use the following commands:

- `bcdedit -set TESTSIGNING ON`: Enables the ability to add unsigned drivers
- `bcdedit /debug on`: Enables debugging for the target machine
- `bcdedit /dbgsettings net hostip:<ip> port:<port>`: Specifies the port and IP address of the debugging machine, so that it can connect the debugger to the target machine[1]

III.3.1.2 On the Debugging Machine

On the debugging machine, simply activate WinDbg and establish a bridge between the debugging machine and the target machine.

There are different techniques to establish the bridge between the debugging machine and the target machine[2]. For example, we have the following possible solutions:

- Port redirection via a pipe: `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/attaching-to-a-virtual-machine--kernel-mode-`
- VirtualKD for VirtualBox and VMWare: `https://sysprogs.com/legacy/virtualkd/tutorials/install/`
- Ntoseye: `https://github.com/dmaivel/ntoseye`, different from WinDbg

III.3.2 Entering the Context of a Process

When debugging the kernel, the debugger (WinDbg) cannot interact directly with an executable (for this, we could debug an executable directly). Moreover, being in remote debugging, the debugger does not have access to the executable; however, it is possible to attach to any process from the kernel context.

To do this, we need to know when a process is created and find the process we are interested in from the list of ongoing processes. We will use the `NtCreateUserProcess` function from the `ntdll.dll` library. This function is

[1] You can also specify the key you want to use with `key:<key>`. If this option is not used, a random key is generated.
[2] Here, we used the `RvnKdBridge` tool which allows connecting to a virtual machine on the **esReverse** platform from *eShard*

the lowest-level function called by the `CreateProcess` function, which allows creating a new process. The diagram below shows this behavior:

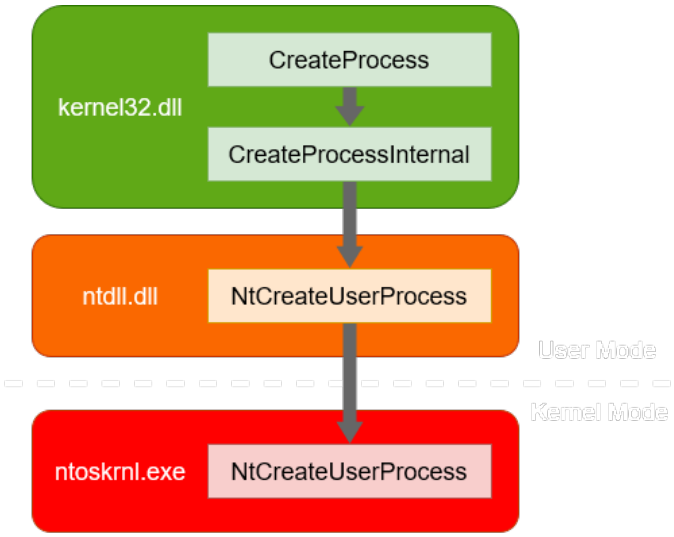

Figure III.3.1: Call chain of the `CreateProcess` function

We will therefore place a breakpoint on this function and launch our executable. However, at this point, our process is not yet present in memory, so we need to use another function to find the entry point of the targeted process.

The `NtMapViewOfSection` function is the lowest-level function called by `ZwMapViewOfSection` to map the executable of the process in memory. At the end of this function, we can find the list of processes present in memory using the command `!process 0 0`:

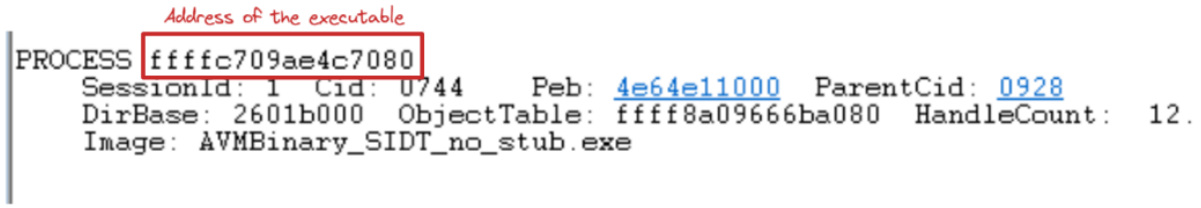

Figure III.3.2: Presence of the executable in the list of processes in memory

Once loaded into memory, it is possible to attach to this process to enter its context. We need to retrieve the `PROCESS` field of the executable and use it with the command `.process /p /r <PROCESS>`. The `/p` and `/r` options are defined as follows in the documentation:

> **/p** Translates all transition page table entries (PTE) of this process into physical addresses before access, if you use /p and Process is different from zero. This translation can cause slowdowns, as the debugger must find the physical addresses of all the memory used by this process. Additionally, the debugger may have to transfer a large amount of data through the debug cable. (This behavior is identical to that of `.cache forcedecodeuser`.)
>
> If you include the /p option and Process is equal to zero or you omit it, the translation is disabled. (This behavior is identical to that of `.cache noforcedecodeptes`.)
>
> **/r** Reloads user-mode symbols after the process context has been set, if you use the /r and /p options. (This behavior is identical to that of `.reload /user`.)

III.3.3 Finding the Entry Point of an Executable

When attaching to a process, we have the ability to see the list of loaded modules for that process. With this, we can retrieve the base address of the executable, which we will call `BASE_ADDRESS`. To do this, use the `lm` command:

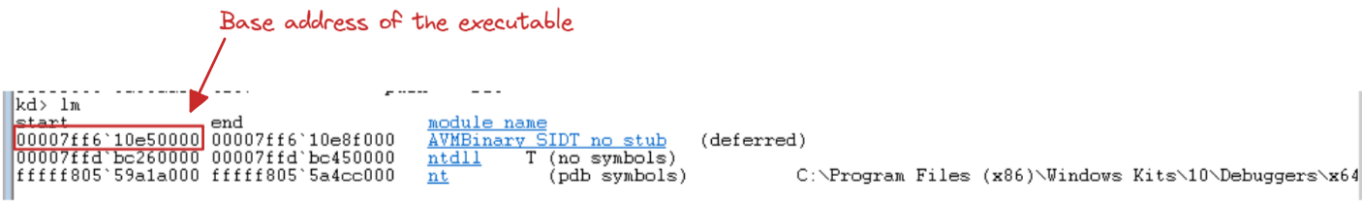

Figure III.3.3: Retrieving the base address of the executable

We can now retrieve the address of the entry point (`OFFSET_ENTRYPOINT`) using the `!dh`[3] command:

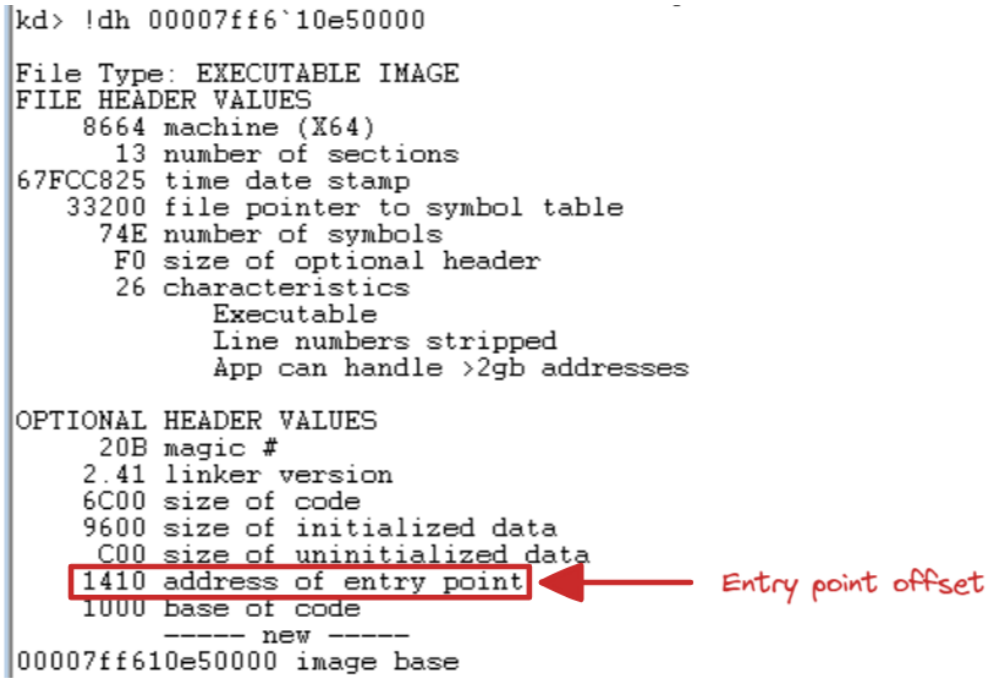

Figure III.3.4: Retrieving the entry point of the executable

Thus, we know that the entry point will be located at `BASE_ADDRESS` + `OFFSET_ENTRYPOINT`, so we just need to place a breakpoint at this location to start modifying our executable:

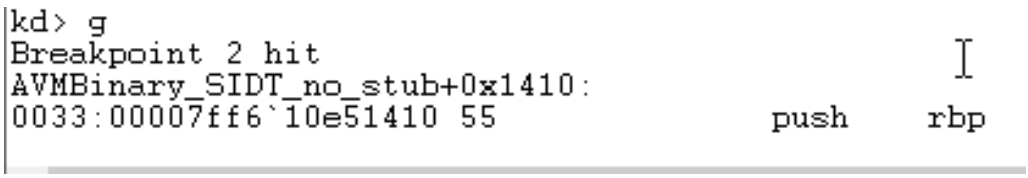

Figure III.3.5: Breakpoint at the entry point of the executable

We have shown how to access the context of an executable, now we need to be able to bypass the anti-VM protections and thus correct (modify) the behavior of the executable. To do this, we will see an example with the previously studied executable. That is, we will show how, with WinDbg, to ensure that the address of the system interrupt table (IDT) does not correspond to what the detection technique expects. We will show how to bypass the `sidt` technique (cf. II.2.3).

[3]Meaning *display headers*, this command displays the headers of a given image.

III.3.4 Example: Bypassing the SIDT Technique

The sidt technique (cf. II.2.3) involves retrieving the address of the system call table and determining if it is located at a high enough address to be considered part of a virtual system.

To bypass this technique, we need to retrieve the address of the instruction and place a breakpoint at this address. To find the address of the instruction, we will build a script that will scan our executable and find all the sidt instructions. However, we have two constraints: the first is that the bridge we use does not allow register modifications[4], in this case, it doesn't bother us too much, but we could have directly modified the IDT register to counter the technique. The second is much more restrictive: memory search via the s command does not work[5], so we cannot search in the process memory for calls to certain opcodes to find sidt.

We will therefore need to create a script capable of finding the instruction. However, it has a specificity that will complicate its exact search. The base opcode of the sidt instruction is 0f 01, but it is followed by a ModR/M byte whose value depends, among other things, on the register used. We will therefore need to base ourselves on the first two bytes to detect sidt, which will lead to a certain number of false positives.

The script will therefore:

- Iterate through all addresses from 4 to 4, between the base and end address of our process (use the lm command to obtain these addresses), we will use a while loop iterating through all addresses.
- Test if the current instruction is a sidt instruction with opcode 0f01. To do this, we will compare the first two bytes of the instruction with 010f[6].
- If the bytes match, then we display the instruction and place a breakpoint at this level.

When we retrieve the opcode of the current instruction, we will need to ensure that we are not accessing unallocated parts of memory. To do this, we will use the .catch command to capture errors. This command will encompass the execution of a set of commands; if one of them fails, the failed command is ignored (no variable is modified) and it skips the following ones. The next command executed will be the first one outside the .catch context.

We will use the .catch command to encompass the commands r @$t5 = poi(@$t3) and r @$t6 = @$t3. The poi command allows dereferencing a pointer to access the value at the targeted address; in our case, we will dereference the instruction pointer (saved in an intermediate variable @$t3). This is the command that may fail; indeed, it is possible that when trying to access memory at a certain address, the area in question is not allocated and thus an error occurs.

The second command will be used to record the state of the variable containing the pointer to the current instruction. This will allow us to compare it with the previous state that we will have recorded to know if the commands in the .catch were executed. We will therefore have the following command:

```
.catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 }
```

This command will, however, create a large number of memory errors and display them. To avoid this, we will use the .foreach command[7]. This command allows repeating a series of commands (OutCommands) for each element (Variable) of a list generated by a set of input commands (InCommands). Here is its syntax:

```
.foreach [Options] ( Variable { InCommands } ) { OutCommands }
```

We will do a .foreach with:

- Variable = ignored
- InCommands = .catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 } }
- OutCommands = {}

Then, we do a test to see if the input instructions were executed:

```
.if (@$t6 == @$t4) {
   r @$t3
   r @$t3 = @$t3 + 1
   .continue
}
```

[4] This is due to the RvnKdBridge tool we are using which does not support this functionality.
[5] In this case, it is possible that this is due to the way we attach to the process context.
[6] On Windows, addresses are little-endian (least significant byte first).
[7] https://stackoverflow.com/questions/52477397/windbg-possible-to-suppress-output-for-outmask-1-or-outmask-d

If this is not the case, we increment the instruction being examined and continue. All that remains is to test if the current instruction uses sidt. To do this, we will dereference the address with `poi` and compare the result with the opcode of `sidt`. Here is the complete code:

```
.printf "[+] Searching for sidt instruction between %p and %p\n", ${$arg1}, ${$arg2}
.printf "-------------------------------------------------------------------\n\n"

r @$t0 = ${$arg1}
r @$t1 = ${$arg2}
r @$t3 = @$t0
r @$t4 = 0
r @$t5 = 0
r @$t6 = @$t3

.while (@$t3 < @$t1) {

    .foreach (ignored { .catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 } }) {}

    .if (@$t6 == @$t4) {
       r @$t3
       r @$t3 = @$t3 + 1
       .continue
    }

    .if ((poi(@$t3) & 0xFFFF) == 0x010F) {
        .printf "[+] Found SIDT instruction at %p\n", @$t3
        bp @$t3
        .printf "[+] Breakpoint set at %p\n", @$t3
        u @$t3 L1
    }

    r @$t4 = @$t3
    r @$t3 = @$t3 + 1
}
```

Once all the breakpoints have been set, we need to modify the address retrieved by the instruction. To do this, using the command `db DEST L8`, we will look at the destination where `sidt` placed the instruction (we will call this data `DEST`) and modify the value at this location. We will rewrite this address using the `eb` command which writes a byte in memory at the chosen destination. We will do this for all the bytes of the address. For example, if the `sidt` instruction placed the IDT address at the address `rbp-0xb`:

```
.for (r @$t4 = @$t4; @$t4 >= 0x6; r @$t4 = @$t4 - 1) {
  eb @rbp - @$t4 ff;
};
```

Part IV

Practical Cases

Chapter IV.1

Al-Khaser

Important Note

In this section, we will use a virtual machine created with QEMU with the following characteristics:

- RAM size: 2 GB
- Hard drive size: 60 GB
- CPU: 1 core, 1 logical processor
- MAC address: `52-54-00-12-34-56`

Al-Khaser is a *PoC* malware, meaning an executable designed to test the stealthiness of a virtual system. It implements many anti-debugging, anti-VM, and anti-analysis techniques. The source code of this malware is available on the GitHub repository `al-khaser` [37].

The malware comes in the form of a Windows executable that has several options available to test different types of techniques. We will therefore focus on the implemented anti-VM techniques. Here are the ones we will use:

- `QEMU`: Detection of QEMU-based virtual machines
- `HYPERV`: Detection of virtual machines virtualized by Hyper-V
- `GEN_SANDBOX`: Generic anti-VM technique

The command used to run this malware will be the following:

```
.\al-khaser_x64.exe --check QEMU --check HYPERV --check GEN_SANDBOX
```

This set of options allows testing if a system is detected by the implemented anti-VM techniques. In this case, we have taken the techniques that detected the environment we tested.

Since this malware is only a *Proof of Concept*, there will be no detailed analysis of the functioning leading to certain behaviors. We will only focus on the implemented techniques.

IV.1.1 Anti-VM Techniques for QEMU

The `QEMU` option will use the following techniques:

```
VOID qemu_reg_key_value();
VOID qemu_reg_keys();
VOID qemu_processes();
VOID qemu_dir();
BOOL qemu_firmware_ACPI();
BOOL qemu_firmware_SMBIOS();
```

Among these, two techniques detect our virtual machine: `ACPI` and `SMBIOS`.

IV.1.1.1 ACPI

The first technique we will look at is called `qemu_firmware_ACPI`. It calls a function named `get_system_firmware` which will query the ACPI tables. *ACPI*, *Advanced Configuration and Power Interface*, is a standard for power management. Information about the system is recorded in several ACPI tables which the system can access via the BIOS.

The function will first call a Windows API function `EnumSystemFirmwareTables` [38] to obtain the list of ACPI tables. Here is the syntax of the function:

```
UINT EnumSystemFirmwareTables(
  [in]  DWORD FirmwareTableProviderSignature,
  [out] PVOID pFirmwareTableEnumBuffer,
  [in]  DWORD BufferSize
);
```

To retrieve the ACPI tables, we will need to set the signature, `FirmwareTableProviderSignature`, to `APCI`. The function will fill a list of ACPI tables, the parameter `pFirmwareTableEnumBuffer` points to this structure. Subsequently, the program will retrieve these tables one by one using the `get_system_firmware` function and search these tables for characteristic strings of virtual environments. These strings are as follows:

- `"FWCF"`
- `"QEMU0002"`
- `"BOCHS"`
- `"BXPC"`

The complete code of the `qemu_firmware_ACPI` function can be found in Annex 4.

IV.1.1.2 SMBIOS

The function that implements this technique is called `qemu_firmware_SMBIOS`. It calls the function named `get_system_fir` which will return a pointer to a buffer containing the SMBIOS table. SMBIOS, *System Management Basic Input/Output System*, is a standard that establishes a set of data structures allowing the firmware to communicate the characteristics of the machine (processor, RAM, motherboard, etc.) to the operating system.

As mentioned above, it is the `get_system_firmware` function that will retrieve the SMBIOS table and return a pointer (cf. Annex 5). This function will call the `GetSystemFirmwareTable` [39] function which has the following syntax:

```
UINT GetSystemFirmwareTable(
  [in]  DWORD FirmwareTableProviderSignature,
  [in]  DWORD FirmwareTableID,
  [out] PVOID pFirmwareTableBuffer,
  [in]  DWORD BufferSize
);
```

The first parameter corresponds to the firmware table we want to retrieve. In the case of `al-khaser`, to retrieve the SMBIOS table, `FirmwareTableProviderSignature` must be `RSMB`. Once the firmware is retrieved, the program checks if it contains one of the following two strings: *QEMU* or *qemu*. If so, the function will return `TRUE`. Here is the code for the function:

```
BOOL qemu_firmware_SMBIOS()
{
        BOOL result = FALSE;

        DWORD smbiosSize = 0;
        PBYTE smbios = get_system_firmware(static_cast<DWORD>('RSMB'), 0x0000,
        &smbiosSize);
        if (smbios != NULL)
        {
                PBYTE qemuString1 = (PBYTE)"qemu";
                size_t StringLen = 4;
                PBYTE qemuString2 = (PBYTE)"QEMU";

                if (find_str_in_data(qemuString1, StringLen, smbios, smbiosSize) ||
```

```
                find_str_in_data(qemuString2, StringLen, smbios, smbiosSize))
        {
                result = TRUE;
        }

        free(smbios);
    }

    return result;
}
```

IV.1.2 Anti-VM Techniques for Hyper-V

The `-check HYPERV` option will ensure that the malware uses the following functions:

```
BOOL check_hyperv_driver_objects();
BOOL check_hyperv_global_objects();
```

In our case, only one of these functions detects our virtual environment: `check_hyperv_global_objects`. This function first calls another function: `enumerate_object_directory`, the code of this function can be found in Annex 6. This function is used to retrieve the list of names of objects present in a directory of the Windows object manager. The function takes the name of the directory as a parameter and returns the list of objects contained in this directory.

These directories are represented by an `OBJECT_DIRECTORY` structure which is a hash table pointing to headers of existing objects. These objects are symbolic links that map device names accessible in user space (`C:`) to their equivalent in kernel space (`\Device\HarddiskVolume1`).

In our case, `enumerate_object_directory` is called to retrieve the objects in the `GLOBAL??` directory. This directory contains symbolic links for DOS devices of the system.

Once the names of these objects are retrieved, the technique will check if one of the following names is present:

- `VMBUS#`: Virtual bus that allows creating a communication channel between the host and the guest, used by Hyper-V for paravirtualization.
- `VDRVROOT`: Allows Windows to manage virtual disks.
- `VmGenerationCounter`: Allows detecting the creation of snapshots or the restoration of the virtual machine.
- `VmGid`: Generates and assigns a unique identifier to each virtual machine.

If this is the case, the program will deduce that it is running in a virtual machine.

Here is the code for the `check_hyperv_global_objects` function:

```
BOOL check_hyperv_global_objects()
{
    auto globalObjs = enumerate_object_directory(L"\\GLOBAL??");
    if (globalObjs == nullptr)
    {
        return FALSE;
    }
    for (wchar_t* globalObj : *globalObjs)
    {
        if (StrStrW(globalObj, L"VMBUS#") != NULL)
        {
            return TRUE;
        }
        if (StrCmpCW(globalObj, L"VDRVROOT") == 0)
        {
            return TRUE;
        }
        if (StrCmpCW(globalObj, L"VmGenerationCounter") == 0)
        {
            return TRUE;
        }
```

```
                if (StrCmpCW(globalObj, L"VmGid") == 0)
                {
                        return TRUE;
                }
        }
        return FALSE;
}
```

IV.1.3 General Anti-VM Techniques

al-khaser also implements a long list of more general anti-VM techniques. The list of all these techniques can be found in Annex 7. The complete code of these techniques can be found on the repository in the Generic.cpp[1] file. We will focus on those that detect our environment and have not already been explained.

IV.1.3.1 *Windows Management Instrumentation* (WMI)

WMI [40] is an interface that allows accessing data from a Windows system. It is possible to retrieve hardware information through this interface.

To execute WMI queries in C, the ExecQuery function is used. The al-khaser malware uses this function via its own ExecWMIQuery function (cf Annex 8). Here is the syntax of ExecQuery:

```
HRESULT ExecQuery(
  [in]  const BSTR           strQueryLanguage,
  [in]  const BSTR           strQuery,
  [in]  long                 lFlags,
  [in]  IWbemContext         *pCtx,
  [out] IEnumWbemClassObject **ppEnum
);
```

al-khaser implements a number of these WMI-based techniques. We will describe the general operation of the following two functions:

```
BOOL manufacturer_computer_system_wmi();
BOOL cpu_fan_wmi();
```

manufacturer_computer_system_wmi

This technique will use the WMI query SELECT * FROM Win32_ComputerSystem from which the program will extract the Manufacturer field corresponding to the name of the computer manufacturer. In a virtual machine, this field will be equal to the name of the virtual machine manager used (or emulator). For example, if we generate a virtual machine using QEMU, the field will be worth QEMU. Simply compare (using StrStr) the retrieved name to a predefined list of virtual machine manager names.

cpu_fan_wmi

All physical machines have a certain number of fans. This can be confirmed (on a Windows system) by looking at the fan instances present using the Win32_Fan class that can be retrieved using a WMI query. If this class is retrieved from a virtual machine, the query is not supposed to return anything since the fans of a machine are not virtualized/emulated[2].

IV.1.3.2 User Input

This technique is based on the same principle as mouse movement detection (cf. II.1.3.1). The idea is that on a physical machine, a user will constantly use their keyboard, or at least most of the time. It is therefore sufficient to look at the time of the last use of a keyboard key. If the date is too far from a defined threshold, then the program can deduce that it is probably running in a virtual machine.

Here is the implementation of this technique in al-khaser:

```
BOOL lack_user_input()
{
        int correct_idle_time_counter = 0;
```

[1] https://github.com/ayoubfaouzi/al-khaser/blob/master/al-khaser/AntiVM/Generic.cpp

[2] It is possible to counter this technique using the technique explained in this article:https://wbenny.github.io/2025/06/29/i-made-my-vm-think-it-has-a-cpu-fan.html

```
        DWORD current_tick_count = 0;
        LASTINPUTINFO last_input_info; // Contains the time of the last input
        last_input_info.cbSize = sizeof(LASTINPUTINFO);

        for (int i = 0; i < 128; ++i) {
                Sleep(0xb);
                // Retrieves the time of the last input event
                if (GetLastInputInfo(&last_input_info)) {
                        current_tick_count = GetTickCount();
                        if (current_tick_count < last_input_info.dwTime)
                                // impossible case unless GetTickCount is manipulated
                                return TRUE;
                        if (current_tick_count - last_input_info.dwTime < 100) {
                                correct_idle_time_counter++;
                                if (correct_idle_time_counter >= 10)
                                        return FALSE;
                        }
                } else // GetLastInputInfo must not fail
                        return TRUE;
        }
        return TRUE;
}
```

The `al-khaser` malware is a very good way to check if a virtual machine is properly configured to avoid these techniques. It implements a very large number of techniques. We have only seen a small part of the capabilities of this program here.

We will now take the example of a "real" malware named *QakBot* [3] which will allow us to see an example of the use of anti-VM protections implemented by malware.

[3] More precisely, *QakBot* is a banking Trojan that has greatly evolved since its first (assumed) appearance in 2008.

Chapter IV.2

QakBot

QakBot is a banking Trojan that has evolved to multiply its capabilities. However, its many variants seem to be moving towards a role as a malware loader.

IV.2.1 Sample

During this section, we will present the following version of *QakBot*:

Date	Version SHA-256
March 2023	`f5ff6dbf5206cc2db098b41f5af14303f6dc43e36c5ec02604a50d5cfecf4790`

Table IV.2.1: SHA-256 samples of the March 2023 version

The anti-VM protections implemented in this sample will use rudimentary techniques based on `cpuid`. However, *QakBot* also implements two detection methods that specifically concern VMWare.

IV.2.2 Analysis

At the entry point of the malware, we find the first piece of code that will allocate a heap for the program (`HeapCreate`) and import all the necessary DLLs (`setup_imports`):

```
00401a28    void _start() __noreturn
00401a28    {
00401a28        int32_t numArgs = 0;
00401a44        PWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &numArgs);
00401a4e        int32_t eax_2;
00401a4e
00401a4e        if (argv)
00401a4e        {
00401a4e            hHeap = HeapCreate(HEAP_NONE, 0x80000, 0);
00401a72            eax_2 = setup_imports(3);
00401a4e        }
00401a4e
...
```

For the rest, we used a *Time Travel Analysis* tool which allows us to keep the entire execution flow of a program. We can therefore look at the branches taken by a program according to the arguments given to the program, the instructions executed, etc. This technique combines the methodologies of static and dynamic analysis.

Thanks to this, we did not need to understand all the functions of the malware, only those we encounter in our analysis. After initialization, we see that the program tests whether the `argv` arguments are missing or whether the DLL import failed. If this is the case for both conditions, the program exits:

```
00401a7a        void* uExitCode;
00401a7a
```

```
00401a7a        if (!argv || eax_2 < 0)
00401a50            uExitCode = 1;
```

Otherwise, the program begins to detect the environment to install itself. It will first detect if a Windows window with the class name `snxhk_border_mywnd` exists:

```
00401a9a            if (DetectSnxhkWindow())
00401a9c                data_410514 = 0x27f3;
```

The complete code of this function can be found in Annex 9. This function calls the `EnumWindows` function to search among all windows if one of them has the searched class name. In our case, this function is used to know if one of the open windows has the class name `snxhk_border_mywnd`.

This technique can be qualified as anti-VM or at least allow preventing the malware from launching in a specific environment. Unfortunately, we have not found a reliable source concerning the environment targeted by this technique.

It continues to perform some tests on the number of arguments given to the executable and the values of the arguments. In our case, without passing any argument, we arrived at the following branch:

```
00401aed              else if (eax_6 == 0x43)
00401b25                       uExitCode = sub_00403ef7(2);
```

We then arrive at a function that interests us. The complete code of the `sub_403ef7` function can be found in Annex 10 and in renamed form in Annex 11. It implements anti-VM functions which are the following[1]:

- `sub_4033fc()` \ `avm_in_vmware_version()`
- `sub_40349a()` \ `avm_in_vmware_memory_size`
- `sub_4035b6()` \ `avm_devices_check()`
- `sub_40385e()` \ `avm_process_check()`
- `sub_40336e()` \ `call_cpuid()`

Each of these functions will return 0 if they do not detect a virtual environment and a higher value otherwise. We will subsequently study all these functions.

IV.2.2.1 `sub_4033fc()` \ `avm_in_vmware_version()`

The first function is an anti-VM function targeting the *VMWare* virtual machine manager.

To communicate with the hypervisor, *VMware* uses the input/output port `0x5658`. To perform this communication, the `in` instruction must be used, which requires the `eax` and `edx` registers. This instruction copies the value read from the specified port (contained in `edx`) to `eax`.

In the context of *VMware*, the manager will intercept any call to the `in` instruction with the input/output port `0x5658`. The program will therefore place this value in `edx`. `eax` will take the magic value `0x564D5868` which will allow `VMware` to authenticate the communication request. Before calling the `in` instruction, we must place the command we want to send to `VMware` in the `ecx` register.

In this case, the technique places the value `0xa` in `ecx`. As we can see using the backdoor header [41], it tries to retrieve information about the `VMware` version. Here is how this technique is implemented in assembly in the malware:

```
mov     dx, 0x5658
mov     ecx, 0x564d5868
mov     eax, ecx
mov     ecx, 0xa
in      eax, dx
```

If the virtual machine in which this instruction is executed is not generated by *VMware*, then the instruction will cause an error:

[1]The function also implements anti-analysis techniques that we will not discuss

```
0x403439    66 ba 58 56              mov  dx, 0x5658
0x40343d    b9 68 58 4d 56           mov  ecx, 0x564d5868
0x403442    8b c1                    mov  eax, ecx
0x403444    b9 0a 00 00 00           mov  ecx, 0xa
0x403449    ed                       general protection while executing in  eax, dx
```

Figure IV.2.1: Error executing the `in` instruction

Without error, the magic value `0x564d5868` which was in `eax` will be moved to `ebx`. The code of the presented technique will therefore compare the content of `ebx` with the magic value. If they are equal, then the instruction has worked and the program is in a *VMware* virtual machine, so it will return a value different from 0 (`0x6e`).

IV.2.2.2 `sub_40349a()` \ `avm_in_vmware_memory_size()`

This function takes up the principle of the previous technique. The code will call the `in` instruction but this time, it will retrieve the used memory (`ecx` = 0x14) [41]:

```
mov     dx, 0x5658
mov     ecx, 0x564d5868
mov     eax, ecx
mov     ecx, 0x14
in      eax, dx
```

If this instruction succeeds, we will retrieve the memory. It will then be compared with a lower bound (`0x10` = `16 KiB`) and an upper bound (`0x2000` = `8192 KiB`). If the retrieved size is between these bounds, then the function will return the retrieved size, otherwise 0. If this instruction fails, the function will return 0.

IV.2.2.3 `sub_4035b6()` \ `avm_devices_check()`

This function will fill two lists of bytes written in raw form in the code. These bytes will form a string that will be decrypted later in the code. We will see that the first list `var_8c` is a list of device descriptions that virtual machine managers can use (VMware, VirtualBox, etc.). The second `var_28` is a list of device class names used by these same virtual machine managers. The list of these descriptions and classes can be found in Annex 12.

Then, the function will use the `SetupDiGetClassDevsA` function from the Windows API which will return a handle to a set of device information. Here is its syntax:

```
WINSETUPAPI HDEVINFO SetupDiGetClassDevsA(
  const GUID *ClassGuid,
  PCSTR       Enumerator,
  HWND        hwndParent,
  DWORD       Flags
);
```

The first three parameters will be ignored in the code. The fourth, named `Flags`, will be set to `6`, which corresponds, according to the header of the `SetupAPI.h` library[2] to the following values:

```
#define DIGCF_PRESENT           0x00000002
#define DIGCF_ALLCLASSES        0x00000004
```

These values are defined as follows in the documentation:

> DIGCF_ALLCLASSES Return the list of installed devices for all device installation classes or all device interface classes.
>
> DIGCF_PRESENT Return only devices currently present in a system.

This function will therefore return a handle to a set of information about all devices without distinction of their class.

The `sub_4035b6()` function will then use another Windows API function, `SetupDiEnumDeviceInfo`, which allows retrieving information about all devices. This information will be placed in a `PSP_DEVINFO_DATA` structure. Here is the syntax of the latter:

[2] https://github.com/tpn/winsdk-10/blob/master/Include/10.0.16299.0/um/SetupAPI.h

```
WINSETUPAPI BOOL SetupDiEnumDeviceInfo(
  [in]  HDEVINFO         DeviceInfoSet,
  [in]  DWORD            MemberIndex,
  [out] PSP_DEVINFO_DATA DeviceInfoData
);
```

Once the `DeviceInfoData` structure is filled by the function, the program enters its final phase which is a loop performing two different actions:

- First, the program will retrieve the description property of the device using the `SetupDiGetDeviceRegistryPropertyA`[3] function with the property identifier 0. The syntax of this function can be found in Annex 13. Then, it will decrypt the strings of the device descriptions (`var_8c`) and test, using `StrStrIA`, if one of these strings is included in the description of one of the devices retrieved thanks to `SetupDiEnumDeviceInfo`. If so, return 1, otherwise 0.
- Second, the program will have the same behavior but this time, it will retrieve the class property of the device and decrypt the strings of the device classes (`var_28`). The comparison and return are identical to the previous action.

The complete code of this function can be found in Annex 14.

IV.2.2.4 `sub_40385e()` \ `avm_process_check()`

Like the previous function, this one decrypts a list of strings, in this case, process names available in Annex 15. Subsequently, it retrieves information about the current process. For this, it uses the `CreateToolhelp32Snapshot` function which will take a snapshot of the targeted process and its environment. Its syntax is as follows:

```
HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

The malware calls it as follows:

```
HANDLE hObject = CreateToolhelp32Snapshot(2, GetCurrentProcessId());
```

The first argument `dwFlags`, which allows establishing the portion of the system we want to "capture", is set to `2`, which corresponds to the constant `TH32CS_SNAPPROCESS` described in the documentation as follows:

> Includes all processes in the system in the snapshot. To enumerate the processes, see Process32First.

Once the snapshot is taken, the program will retrieve the first process encountered in the snapshot using the `Process32First` function:

```
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

This function will place the information about the process in the `LPPROCESSENTRY32 lppe` structure. Once the first process is obtained, the program enters a `do while` loop. This loop will retrieve the next process to study using the `Process32Next` function:

```
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

In the `do while` loop, we find two other `for` loops:

[3] This function will retrieve the properties of a device requested by the user.

- The first `for` loop goes through the list of previously decrypted process names (`var_14c`) and compares them with the name of the current process (`var_110`). The comparison is performed by the `__beep` function which is an alias for `lstrcmpiA`. This function allows comparing two strings and returns a negative value if the first string is less than the second, 0 if the strings are equal, and a positive value otherwise.
- The second loop follows the same principle but uses `StrStrIA` to determine if one string is included in the other.

At the end of the `do while` loop, the function will return 1 if one of the processes has been identified as being related to emulation. The complete code of the function can be found in Annex 16.

IV.2.2.5 `sub_40336e()` \ `call_cpuid()`

This technique is not part of the condition (`if`) to detect a virtual machine. It is called after the failure to detect an emulated (virtualized) environment. Moreover, it is not used with a comparison or a test, so we have not been able to explain the presence of the latter. Nevertheless, it implements a technique that can constitute an anti-VM protection using `cpuid`II.2.1:

- First, the program will detect the processor's identification string using `cpuid` with `eax = 0` and which will be compared later to the string `GenuineIntel` using the `lstrcmpiA` function. It seems that this is used to target only certain machines. Indeed, most emulators emulate a processor with the same processor identification string as the one present on the host machine.
- The anti-VM technique will also use `cpuid`, but with `eax = 0x1` (cf II.2.1.1) to determine if a hypervisor is present.

The code of this function is available in Annex 17.

IV.2.3 Analysis Result

As we have seen, the diversity of anti-VM protections in the *QakBot* malware is sufficient to detect a large part of virtual machine managers, although it seems to focus on *VMware*. However, these methods remain quite basic and target simple and poorly defended virtualization solutions. Sufficiently configured virtual machines can avoid most of these anti-VM protections.

In conclusion, *QakBot* uses a varied and effective arsenal of anti-VM protections against standard tools, particularly *VMware*, but it remains vulnerable to more sophisticated environments.

Conclusion

The objective of this thesis was to study the mechanisms implemented by malware to detect and bypass virtualized environments. As we have seen, there are many techniques. Most are based on system parameters that seem abnormal for a physical machine. Most of the methods presented are implemented using the Windows API, so they are only usable within a Windows environment. Equivalents exist for each of these techniques in other systems.

Despite the great diversity of techniques, the number of anti-VM protections remains quite low in most malware. These techniques are often rudimentary; most of the time, they are based on assembly instructions such as `cpuid` or `rdtsc`. The example of *QakBot* confirms this idea: this malware implements simple tests (use of the *VMware* I/O port, cpuid, etc.).

Although simple, QakBot's anti-VM protections allow it to effectively protect itself against specific virtualized environments. It is also possible that the simplicity of these techniques follows the logic of a particular campaign; *QakBot* will therefore not be intended to be reused in this form. There are also other methods that will delay the use of anti-VM protections as much as possible. For example, the phishing campaign carried out by APT29 [42] uses software called `WINELOADER` which notably allows the use of anti-VM techniques. This software is downloaded by another component called `GRAPELOADER` which will call a server to retrieve the `WINELOADER`. This method prevents any analysis of the software without prior verification of the environment.

We have presented effective methods to counter these anti-VM protections. However, these countermeasures do not take into account other factors often present in this software, such as anti-debugging techniques that will counter the use of a debugger. As shown during this thesis, malware contains on average more anti-debugging techniques than anti-VM protections.

Anti-VM protections can, however, be a double-edged sword. First, if they are poorly defined, these techniques can cause the software to ignore machines with unusual configurations. For example, although rare, it is still possible to find machines with less than 512 GB of storage. Additionally, these techniques will also ignore all virtual machines, which will greatly reduce the attack surface of the malware. Many companies use fleets of virtual machines to host workstations, servers, etc. These machines remain interesting to attack even though they are virtualized.

To conclude, anti-VM protections implemented by malware are essential elements for the sustainability of this software. They prevent analysis and help increase their lifespan. Although sometimes excessive, sometimes even counterproductive, these techniques are reliable when they work. They force analysts to develop increasingly efficient and secure tools and environments.

Appendices

Annex 1 — Script permettant de retrouver le code assembleur de la traduction du code ARM

```
import os

string = "8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfdff15120000
00ffe0488d0514ffffffe9e4feffff"

with open("add_x86_64_bytes", "wb") as f:
    f.write(bytes.fromhex(string))

os.system("ndisasm -b 64 add_x86_64_bytes > add_x86_64")
```

Annex 2 — Fonction `SERVICES_NAME_LIST` et `belong_to` — Détection des services liés à des machines virtuelles

```
static char *SERVICES_NAME_LIST[] = {
/* VM services used in QEMU */
"QUEM-GA", "VirtioFsSvc", "BalloonService", "spice-agent", "vmi", "vds",

/* VM services used in VBOX */
"VBoxSVC", "VBoxDrv", "VBoxUSBMon", "VBoxNetFlt", "VBoxNetAdp",

/* VM services used in VMWare */
"VGAuthService", "vmvss", "vm3dservice", "VMTools"};

bool belong_to(char *service_name)
{
    for (int i = 0; i < NB_SERVICES; i++)
    {
        if (strstr(service_name, SERVICES_NAME_LIST[i]))
        {
            return true;
        }
    }
    return false;
}
```

Annex 3 — Liste de clefs de registre standard pour les gestionnaires de machines virtuelles

```
/* Sample of regkey path */
LPCWSTR RegValuePath[] = {

  /* VMWare */
    L"SOFTWARE\\VMware, Inc.\\VMware Tools",

  /* VirtualBox */
    L"SOFTWARE\\Oracle\\VirtualBox Guest Additions",
    L"HARDWARE\\ACPI\\DSDT\\VBOX__",
    L"HARDWARE\\ACPI\\FADT\\VBOX__",
    L"HARDWARE\\ACPI\\RSDT\\VBOX__",
    L"SYSTEM\\ControlSet001\\Services\\VBoxGuest",
    L"SYSTEM\\ControlSet001\\Services\\VBoxMouse",
    L"SYSTEM\\ControlSet001\\Services\\VBoxService",
    L"SYSTEM\\ControlSet001\\Services\\VBoxSF",
    L"SYSTEM\\ControlSet001\\Services\\VBoxVideo",
    L"HKEY_LOCAL_MACHINE\\HARDWARE\\ACPI\\DSDT\\VBOX__",

    /* QEMU and other software */
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\QEMU Guest Agent VSS
    Provider",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\QEMU-GA",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\BALLOON",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Enum\\ROOT\\vdrvroot"
};
```

Annex 4 — Fonction `qemu_firmware_ACPI` du maliciel `al-khaser`

```
BOOL qemu_firmware_ACPI()
{
        BOOL result = FALSE;

        PDWORD tableNames = static_cast<PDWORD>(malloc(4096));

        if (tableNames) {
                SecureZeroMemory(tableNames, 4096);
                DWORD tableSize =
                enum_system_firmware_tables(static_cast<DWORD>('ACPI'), tableNames,
                4096);

                // API not available
                if (tableSize == -1)
                        return FALSE;

                DWORD tableCount = tableSize / 4;
                if (tableSize < 4 || tableCount == 0)
                {
                        result = TRUE;
                }
                else
                {
                        const char* strings[] = {
                                "FWCF",                 // fw_cfg name
                                "QEMU0002",         // fw_cfg HID/CID
                                "BOCHS",         // OEM ID
                                "BXPC"                  // OEM Table ID
                        };

                        for (DWORD i = 0; i < tableCount; i++)
                        {
                                DWORD tableSize = 0;
                                PBYTE table =
                                get_system_firmware(static_cast<DWORD>('ACPI'),
                                tableNames[i], &tableSize);

                                if (table) {

                                        for (DWORD j = 0; j < sizeof(strings) /
                                        sizeof(char*); j++)
                                        {
                                                if (!find_str_in_data((PBYTE)strings
                                                [j], strlen(strings[j]), table,
                                                tableSize))
                                                {
                                                        free(table);
                                                        result = TRUE;
                                                        goto out;
                                                }
                                        }

                                        free(table);
                                }
                        }

                        DWORD tableSize = 0;
                        PBYTE table =
                        get_system_firmware(static_cast<DWORD>('ACPI'),
                        static_cast<DWORD>('PCAF'), &tableSize);
```

```
                        if (table) {
                                if (tableSize < 45)
                                {
                                        return FALSE; // Corrupted table
                                }

                                // Format: [HexOffset DecimalOffset ByteLength]
                                FieldName  : FieldValue (in hex)
                                //                  [02Dh      0045          001h
                                ]         PM Profile : 00 [Unspecified] - hardcoded
                                in QEMU src
                                if ((BYTE) table[45] == (BYTE)0)
                                {
                                        result = TRUE;
                                }

                                free(table);
                        }
                }
        out:
                free(tableNames);
        }
        return result;
}
```

Annex 5 — Fonction `get_system_firmware` — récupération des tables micrologicielles

```
PBYTE get_system_firmware(_In_ DWORD signature, _In_ DWORD table,
                                                    _Out_ PDWORD pBufferSize)
{
        if (!API::IsAvailable(API_IDENTIFIER::API_GetSystemFirmwareTable)) {
                return NULL;
        }

        DWORD bufferSize = 4096;
        PBYTE firmwareTable = static_cast<PBYTE>(malloc(bufferSize));

        if (firmwareTable == NULL)
                return NULL;

        SecureZeroMemory(firmwareTable, bufferSize);

        auto GetSystemFirmwareTable = static_cast<pGetSystemFirmwareTable>(
                API::GetAPI(API_IDENTIFIER::API_GetSystemFirmwareTable));

        DWORD resultBufferSize =
                GetSystemFirmwareTable(signature, table, firmwareTable, bufferSize);
        if (resultBufferSize == 0) {
                printf("First call failed :(\n");
                free(firmwareTable);
                return NULL;
        }

        // if the buffer was too small, realloc and try again
        if (resultBufferSize > bufferSize) {
                PBYTE tmp;

                tmp = static_cast<BYTE *>(realloc(firmwareTable, resultBufferSize));
                if (tmp) {
                        firmwareTable = tmp;
                        SecureZeroMemory(firmwareTable, resultBufferSize);
                        if (GetSystemFirmwareTable(signature, table, firmwareTable,
                                                                        resultBuf⌋
                                                                        ferSize)
                                                                        == 0) {
                                printf("Second call failed :(\n");
                                free(firmwareTable);
                                return NULL;
                        }
                }
        }

        *pBufferSize = resultBufferSize;
        return firmwareTable;
}
```

Annex 6 — Fonction `enumerate_object_directory` — Permet de retrouver la liste des noms des objets présent dans un répertoire du gestionnaire d'objets.

```
std::vector<wchar_t *> *enumerate_object_directory(const wchar_t *path)
{
        if (!API::IsAvailable(API_NtOpenDirectoryObject) ||
                !API::IsAvailable(API_NtQueryDirectoryObject)) {
                return nullptr;
        }

        UNICODE_STRING usPath = { 0 };
        usPath.Buffer = const_cast<wchar_t *>(path);
        usPath.Length = static_cast<USHORT>(lstrlenW(path) * sizeof(wchar_t));
        usPath.MaximumLength = usPath.Length;

        OBJECT_ATTRIBUTES objAttr = { 0 };
        InitializeObjectAttributes(&objAttr, &usPath, OBJ_CASE_INSENSITIVE, NULL,
                                                                NULL);

        auto ntOpenDirectoryObject = static_cast<pNtOpenDirectoryObject>(
                API::GetAPI(API_NtOpenDirectoryObject));
        auto ntQueryDirectoryObject = static_cast<pNtQueryDirectoryObject>(
                API::GetAPI(API_NtQueryDirectoryObject));

        const int DIRECTORY_QUERY = 0x0001;
        HANDLE hDirectory = 0;
        NTSTATUS status =
                ntOpenDirectoryObject(&hDirectory, DIRECTORY_QUERY, &objAttr);
        if (status != 0) {
                // printf("\nNTODO failed: %x\n", status);
                return nullptr;
        }

        auto pObjDirInfo =
                static_cast<OBJECT_DIRECTORY_INFORMATION *>(calloc(0x800, 1));
        ULONG returnedLength = 0;
        ULONG context = 0;
        auto results = new std::vector<wchar_t *>();
        while (ntQueryDirectoryObject(hDirectory, pObjDirInfo, 0x800, TRUE, FALSE,
                                                                &context,
                                                                &returnedLength)
                                                                == 0 &&
                   returnedLength > 0) {
                // wprintf(L"\nobject: %s\n", pObjDirInfo->Name.Buffer);
                wchar_t *name = static_cast<wchar_t *>(
                        calloc(pObjDirInfo->Name.Length + 1, sizeof(wchar_t)));
                memcpy(name, pObjDirInfo->Name.Buffer,
                           pObjDirInfo->Name.Length * sizeof(wchar_t));
                results->push_back(name);
        }

        free(pObjDirInfo);

        return results;
}
```

Annex 7 — Fichier `Generic.h` de `al-khaser` — En-tête contenant toutes les fontions d'anti-VM génériques.

```
VOID loaded_dlls();
VOID known_file_names();
VOID known_usernames();
VOID known_hostnames();
VOID other_known_sandbox_environment_checks();
BOOL NumberOfProcessors();
BOOL idt_trick();
BOOL ldt_trick();
BOOL gdt_trick();
BOOL str_trick();
BOOL number_cores_wmi();
BOOL disk_size_wmi();
BOOL setupdi_diskdrive();
BOOL mouse_movement();
BOOL lack_user_input();
BOOL memory_space();
BOOL dizk_size_deviceiocontrol();
BOOL disk_size_getdiskfreespace();
BOOL accelerated_sleep();
BOOL cpuid_is_hypervisor();
BOOL cpuid_hypervisor_vendor();
BOOL serial_number_bios_wmi();
BOOL model_computer_system_wmi();
BOOL manufacturer_computer_system_wmi();
BOOL current_temperature_acpi_wmi();
BOOL process_id_processor_wmi();
BOOL power_capabilities();
BOOL hybridanalysismacdetect();
BOOL cpu_fan_wmi();
BOOL caption_video_controller_wmi();
BOOL query_license_value();
BOOL cachememory_wmi();
BOOL physicalmemory_wmi();
BOOL memorydevice_wmi();
BOOL memoryarray_wmi();
BOOL voltageprobe_wmi();
BOOL portconnector_wmi();
BOOL smbiosmemory_wmi();
BOOL perfctrs_thermalzoneinfo_wmi();
BOOL cim_memory_wmi();
BOOL cim_numericsensor_wmi();
BOOL cim_physicalconnector_wmi();
BOOL cim_sensor_wmi();
BOOL cim_slot_wmi();
BOOL cim_temperaturesensor_wmi();
BOOL cim_voltagesensor_wmi();
BOOL pirated_windows();
BOOL registry_services_disk_enum();
BOOL registry_disk_enum();
BOOL number_SMBIOS_tables();
BOOL firmware_ACPI();
BOOL hosting_check();
VOID looking_glass_vdd_processes();
```

Annex 8 — Fonction `ExecWMIQuery` — Permet d'exécuter une requête WMI.

```
BOOL ExecWMIQuery(IWbemServices **pSvc, IWbemLocator **pLoc,
                                IEnumWbemClassObject **pEnumerator, const TCHAR
                                *szQuery)
{
        // Execute WMI query
        BSTR strQueryLanguage = SysAllocString(OLESTR("WQL"));
        BSTR strQuery = SysAllocString(szQuery);

        BOOL bQueryResult = TRUE;

        if (strQueryLanguage && strQuery) {

                HRESULT hres = (*pSvc)->ExecQuery(strQueryLanguage, strQuery,
                      WBEM_FLAG_FORWARD_ONLY |
                          WBEM_FLAG_RETURN_IMMEDIATELY,
                      NULL, pEnumerator);

                if (FAILED(hres)) {
                        bQueryResult = FALSE;
                        print_last_error(_T("ExecQuery"));
                        (*pSvc)->Release();
                        (*pLoc)->Release();
                        CoUninitialize();
                }
        }

        if (strQueryLanguage)
                SysFreeString(strQueryLanguage);
        if (strQuery)
                SysFreeString(strQuery);

        return bQueryResult;
}
```

Annex 9 — Fonctions `DetectSnxhkWindow` et `EnumWindows_find_match_class` - Teste la présence d'une fenêtre windows ayant un certains nom de classe

```
int32_t DetectSnxhkWindow()
{
    void* decrypted_targeted_name = decrypt_ressource_into_buf(0x27ca);
    void* targeted_name = decrypted_targeted_name;
    int32_t match_found = 0;
    int32_t match_confirmed = 0;
    int32_t name_length = lstrlenA(decrypted_targeted_name);

    if (EnumWindows(EnumWindows_find_match_class, &targeted_name))
    {
        custom_free(&targeted_name);

        if (!match_found || match_confirmed > 0)
            return 1;
    }
    else
    {
        GetLastError();
        custom_free(&targeted_name);
    }

    return 0;
}

int32_t __stdcall EnumWindows_find_match_class(HWND hwnd, int32_t* user_data)
{
    uint8_t className[0x100] = {0};
    memset(className, 0, sizeof(className));

    GetClassNameA(hwnd, (LPSTR)className, 0x100);

    user_data[3] += 1;

    int32_t i = 0;
    int32_t length = user_data[1];  // Length of the string to compare
    uint8_t* target = (uint8_t*)*(void**)user_data;

    if (length > 0)
    {
        int32_t offset = data_410664;  // Likely a static XOR or transformation key

        do
        {
            uint8_t transformed_target =
                *(uint8_t*)(*(target + i) + offset);
            uint8_t transformed_classname =
                *(uint8_t*)(&className[i] + offset);

            if (transformed_target != transformed_classname)
                return 1;

            i += 1;
        } while (i < length);
    }

    user_data[2] = 1;  // Mark a match
    return 1;
}
```

Annex 10 — Fonction `sub_403ef7` — Implémente un certains nombre de protection anti-VM.

```
int32_t __fastcall sub_403ef7(void* arg1)
{
    void* var_8 = arg1;
    void* var_c = arg1;

    while (false)
        /* nop */

    if (sub_4033fc() <= 0 && sub_40349a() <= 0 && sub_4035b6() <= 0
        && sub_40385e() <= 0 && sub_403bdf() <= 0 && sub_403d22() <= 0)
    {
        sub_403deb();
        sub_403e6f();
        sub_40336e();

        while (false)
            /* nop */

        return 0;
    }

    int32_t result = 1;
    var_8 = decrypt_ressource_into_buf(0x1ac6);

    while (false)
        /* nop */

    if (var_8)
    {
        if (test_file_exists(var_8))
        {
            while (false)
                /* nop */

            result = 0;
            sub_4033fc();
            sub_40349a();
            sub_4035b6();
            sub_40385e();
            sub_403bdf();
            sub_403d22();
            sub_403deb();
            sub_403e6f();
            sub_40336e();
        }

        custom_free(&var_8);
    }

    if (sub_403aa0() > 0)
        result = 0;

    return result;
}
```

Annex 11 — Fonction `sub_403ef7` renommée — Implémente un certains nombre de protection anti-VM.

```
int32_t __fastcall anti_vm_and_analysis_checks(void* arg1)
{
    void* var_8 = arg1;
    void* var_c = arg1;

    while (false)
        /* nop */

    if (avm_in_vmware_version() <= 0 && avm_in_vmware_memory_size() <= 0
        && avm_devices_check() <= 0 && avm_process_check() <= 0
        && dll_injected_check() <= 0 && filename_check() <= 0)
    {
        sub_403deb();
        sub_403e6f();
        call_cpuid();

        while (false)
            /* nop */

        return 0;
    }

    int32_t result = 1;
    var_8 = decrypt_ressource_into_buf(0x1ac6);

    while (false)
        /* nop */

    if (var_8)
    {
        if (test_file_exists(var_8))
        {
            while (false)
                /* nop */

            result = 0;
            avm_in_vmware_version();
            avm_in_vmware_memory_size();
            avm_devices_check();
            avm_process_check();
            dll_injected_check();
            filename_check();
            sub_403deb();
            sub_403e6f();
            call_cpuid();
        }

        custom_free(&var_8);
    }

    if (sub_403aa0() > 0)
        result = 0;

    return result;
}
```

Annex 12 — Listes des descriptions et noms de classe testées par *QakBot*

```
Descriptions :

VMware Pointing
VMware Accelerated
VMware SCSI
VMware SVGA
VMware Replay
VMware server memory
CWSandbox
Virtual HD
QEMU
Red Hat VirtIO
srootkit
VMware VMaudio
VMware Vista
VBoxVideo
VBoxGuest


Noms de classe :

vmxnet
vmscsi
VMAUDIO
vmdebug
vm3dmp
vmrawdsk
vmx_svga
ansfltr
sbtisht
```

Annex 13 — Syntaxe de la fonction `SetupDiGetDeviceRegistryPropertyA`

```
WINSETUPAPI BOOL SetupDiGetDeviceRegistryPropertyA(
  [in]            HDEVINFO         DeviceInfoSet,
  [in]            PSP_DEVINFO_DATA DeviceInfoData,
  [in]            DWORD            Property,
  [out, optional] PDWORD           PropertyRegDataType,
  [out, optional] PBYTE            PropertyBuffer,
  [in]            DWORD            PropertyBufferSize,
  [out, optional] PDWORD           RequiredSize
);
```

Annex 14 — Fonction `avm_devices_check`

```
int32_t avm_devices_check()
{
    int32_t result = 0;
    int32_t descriptions;
    __builtin_memcpy(&descriptions,
        "\xd4\x05\x00\x00\x38\x02\x00\x00\xc4\x02\x00\x00\x5e\x1d\x00\x00\xee\x05\x0
        0\x00\x"
    "a1\x26\x00\x00\xc1\x32\x00\x00\xf5\x20\x00\x00\x17\x35\x00\x00\xdf\x22\x00\x00\
    x54"
    "28\x00\x00\x57\x25\x00\x00\xf2\x01\x00\x00\xe5\x1d\x00\x00\x62\x36\x00\x00",
        0x3c);
    int32_t classes;
    __builtin_memcpy(&classes,
        "\x9b\x0d\x00\x00\xfc\x26\x00\x00\x20\x02\x00\x00\x3a\x19\x00\x00\x68\x00\x0
        0\x00\x"
    "c5\x26\x00\x00\x1a\x00\x00\x00\x05\x0c\x00\x00\xaa\x2e\x00\x00",
        0x24);

    HDEVINFO DeviceInfoSet = SetupDiGetClassDevsA(nullptr, nullptr, nullptr, 6);

    if (DeviceInfoSet == 0xffffffff)
    {
        while (false)
            /* nop */

        return 0xffffffff;
    }

    struct SP_DEVINFO_DATA DeviceInfoData;
    DeviceInfoData.cbSize = 0x1c;
    uint32_t MemberIndex = 0;

    while (SetupDiEnumDeviceInfo(DeviceInfoSet, MemberIndex, &DeviceInfoData))
    {
        uint8_t* var_94 = nullptr;
        uint8_t* var_98 = nullptr;
        var_94 = retrieve_property_device(DeviceInfoSet, &DeviceInfoData, 0);

        if (var_94)
        {
            for (int32_t i = 0; i < 0xf; i += 1)
            {
                void* var_9c = decrypt_ressource_into_buf((&descriptions)[i]);

                if (var_9c)
                {
                    if (StrStr(var_94, var_9c))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        custom_free(&var_9c);
                        break;
                    }

                    custom_free(&var_9c);
                }
            }

            custom_free_(&var_94, 0);
```

```
        }

        var_98 = retrieve_property_device(DeviceInfoSet, &DeviceInfoData, 4);

        if (var_98)
        {
            for (int32_t i_1 = 0; i_1 < 9; i_1 += 1)
            {
                void* var_a0 = decrypt_ressource_into_buf((&classes)[i_1]);

                if (var_a0)
                {
                    if (StrStr(var_98, var_a0))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        custom_free(&var_a0);
                        break;
                    }

                    custom_free(&var_a0);
                }
            }

            custom_free_(&var_98, 0);
        }

        if (result > 0)
            break;

        if (result > 0)
            break;

        MemberIndex += 1;
    }

    SetupDiDestroyDeviceInfoList(DeviceInfoSet);

    return result;
}
```

Annex 15 — Listes des processus testés par *QakBot*

```
Processus :

Fiddler.exe
sample.exe
sample.exe
runsample.exe
lordpe.exe
regshot.exe
Autoruns.exe
dsniff.exe
VBoxTray.exe
HashMyFiles.exe
ProcessHacker.exe
Procmon.exe
Procmon64.exe
netmon.exe
vmtoolsd.exe
vm3dservice.exe
VGAuthService.exe
pr0c3xp.exe
CFF Explorer.exe
dumpcap.exe
Wireshark.exe
idaq.exe
idaq64.exe
TPAutoConnect.exe
ResourceHacker.exe
vmacthlp.exe
OLLYDBG.EXE
windbg.exe
bds-vision-agent-nai.exe
bds-vision-apis.exe
bds-vision-agent-app.exe
MultiAnalysis_v1.0.294.exe
x32dbg.exe
VBoxService.exe
Tcpview.exe
```

Annex 16 — Fonction `avm_process_check`

```
int32_t avm_process_check()
{
    int32_t result = 0;
    int32_t var_144 = 0xffffffff;
    int32_t* var_c = nullptr;
    int32_t* var_8 = nullptr;
    void* var_140 = nullptr;
    int32_t var_138 = 0;
    void* var_14c = nullptr;
    void* var_13c = nullptr;

    while (false)
        /* nop */

    var_140 = decrypt_ressource_into_buf(0x1ad8);
    void* eax_1 = decrypt_ressource_into_buf(0x1ad8);
    var_14c = parse(var_140, 0x3b, 0, &var_c);
    var_13c = parse(eax_1, 0x3b, 0, &var_8);
    custom_free(&var_140);
    custom_free(&var_140);

    if (!var_14c || !var_13c)
    {
        while (false)
            /* nop */

        return 0;
    }

    HANDLE hObject = CreateToolhelp32Snapshot(2, GetCurrentProcessId());

    if (hObject != 0xffffffff)
    {
        int32_t lppe;
        memset(&lppe, 0, 0x128);
        lppe = 0x128;

        if (Process32First(hObject, &lppe))
        {
            int32_t i;

            do
            {
                void var_110;

                for (int32_t j = 0; j < var_c; j += 1)
                {
                    if (!__beep(*(uint32_t*)((char*)var_14c + (j << 2)), &var_110))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        break;
                    }
                }

                if (!result)
                {
                    for (int32_t j_1 = 0; j_1 < var_8; j_1 += 1)
                    {
```

```
                        if (StrStr(&var_110, *(uint32_t*)((char*)var_14c + (j_1 <<
                        2))))
                        {
                            while (false)
                                /* nop */

                            result = 1;
                            break;
                        }
                    }
                }

                if (result)
                    break;

                i = Process32Next(hObject, &lppe);
            } while (i);

            CloseHandle(hObject);
        }
    }

    cleanup(&var_c, &var_14c);
    cleanup(&var_8, &var_13c);
    return result;
}
```

Annex 17 — Fonction `cpuid_call()`

```
int32_t call_cpuid()

{
    int32_t var_c = 0;
    int32_t var_8 = 0;
    char string1 = 0;
    void var_2f;
    int32_t ecx;
    int16_t* edi;
    edi = __builtin_memset(&var_2f, 0, 0x1c);
    *(uint16_t*)edi = 0;
    edi[1] = 0;
    void* lpString2 = nullptr;

    while (false)
        /* nop */

    char* var_3c = &string1;
    int32_t edx;
    cpu_brand_string(&string1, edx, ecx, var_3c);
    int32_t eax_1;
    int32_t ecx_2;
    int32_t edx_1;
    int32_t ebx;
    eax_1 = __cpuid(1, var_3c);
    lpString2 = decrypt_ressource_into_buf(0x42c);

    if (lpString2)
    {
        if (ecx_2 == 1 && !lstrcmpiA(&string1, lpString2))
        {
            while (false)
                /* nop */

            int32_t var_c_1 = 1;
        }

        custom_free(&lpString2);
    }

    return 0;
}

...

int32_t __convention("regparm") cpu_brand_string(int32_t arg1,
  int32_t arg2, int32_t arg3, void* arg4)

{
    while (false)
        /* nop */

    int32_t eax;
    int32_t ecx;
    int32_t edx;
    int32_t ebx;
    eax = __cpuid(0, arg3);
    int32_t var_c = ebx;
    int32_t var_8 = ecx;
    int32_t var_10 = edx;
    memcpy(arg4, &var_c, 4);
```

```
    memcpy((char*)arg4 + 4, &var_10, 4);
    memcpy((char*)arg4 + 8, &var_8, 4);
    *(uint8_t*)((char*)arg4 + 0xc) = 0;
    return 0;
}
```

Bibliography

[1] Red Hat. *Un hyperviseur, qu'est-ce que c'est ?* 2025. URL: https://www.redhat.com/fr/topics/virtualization/what-is-a-hypervisor.

[2] WikiBooks. *Les systèmes d'exploitation/Virtualisation et machines virtuelles*. URL: https://fr.wikibooks.org/wiki/Les_syst%C3%A8mes_d%27exploitation/Virtualisation_et_machines_virtuelles.

[3] QEMU. *QEMU*. URL: https://www.qemu.org/.

[4] panda-re. *Dépôt GitHub de panda*. URL: https://github.com/panda-re/panda.

[5] QEMU. *Arm System emulator*. URL: https://www.qemu.org/docs/master/system/target-arm.html.

[6] QEMU. *System Emulation : Introduction*. URL: https://www.qemu.org/docs/master/system/introduction.html.

[7] QEMU. *TCG Emulation*. URL: https://www.qemu.org/docs/master/devel/index-tcg.html.

[8] Alexander Graf. *Vortrag: QEMU's recompilation engine*. URL: https://chemnitzer.linux-tage.de/2012/vortraege/1062/.

[9] Formation DevOps. *QEMU*. URL: https://docker.uptime-formation.fr/KVM/Theorie-QEMU/.

[10] QEMU. *QEMU User space emulator*. URL: https://www.qemu.org/docs/master/user/main.html.

[11] QEMU. *Translator Internals*. URL: https://www.qemu.org/docs/master/devel/tcg.html.

[12] QEMU. *The memory API*. URL: https://www.qemu.org/docs/master/devel/memory.html.

[13] QEMU. *Qcow2 Image File Format*. URL: https://www.qemu.org/docs/master/interop/qcow2.html.

[14] Ceph Foundation. *QEMU and Block Devices*. URL: https://docs.ceph.com/en/reef/rbd/qemu-rbd/.

[15] Microsoft. *Windows API index*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list.

[16] Microsoft. *GetSystemMetrics function (winuser.h)*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getsystemmetrics.

[17] PC TROUBLESHOOTING and MAINTENANCE. *EVOLUTION OF HARD DISK*. 2008. URL: https://pctroubleshoo blogspot.com/2008/11/evolution-of-hard-disk.html.

[18] Microsoft. *GetDiskFreeSpaceExA function (fileapi.h)*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getdiskfreespaceexa.

[19] Microsoft. *GlobalMemoryStatusEx function (sysinfoapi.h)*. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-globalmemorystatusex.

[20] Microsoft. *GetAdaptersInfo function (iphlpapi.h)*. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-getadaptersinfo.

[21] Microsoft. *GetSystemInfo, fonction (sysinfoapi.h)*. 2021. URL: https://learn.microsoft.com/fr-fr/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsysteminfo.

[22] Microsoft. *GetPwrCapabilities function (powerbase.h)*. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/api/powerbase/nf-powerbase-getpwrcapabilities.

[23] Microsoft. *OpenSCManagerA function (winsvc.h)*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openscmanagera.

[24] Microsoft. *Sécurité du service et droits d'accès*. 2023. URL: https://learn.microsoft.com/fr-fr/windows/win32/services/service-security-and-access-rights.

[25] Microsoft. *EnumServicesStatusA function (winsvc.h)*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-enumservicesstatusa.

[26] Microsoft. *RegOpenKeyExW, fonction (winreg.h)*. 2024. URL: https://learn.microsoft.com/fr-fr/windows/win32/api/winreg/nf-winreg-regopenkeyexw.

[27] Microsoft. *GetCursorPos, fonction (winuser.h)*. 2022. URL: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getcursorpos.

[28] Microsoft. *GetForegroundWindow, fonction (winuser.h)*. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getforegroundwindow.

[29] Félix Cloutier. *RDTSC — Read Time-Stamp Counter*. 2023. URL: https://www.felixcloutier.com/x86/rdtsc.

[30] Félix Cloutier. *SIDT — Store Interrupt Descriptor Table Register*. 2023. URL: https://www.felixcloutier.com/x86/sidt.

[31] stack overflow. *Red Pill detect virtualization*. 2017. URL: https://stackoverflow.com/questions/46267618/red-pill-detect-virtualization.

[32] Tom Liston et Ed Skoudis. "On the Cutting Edge: Thwarting Virtual Machine Detection". In: (2006). URL: https://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf.

[33] Lorenzo Maffia et Dario Nisi† et Platon Kotzias et Giovanni Lagorio et Simone Aonzo et Davide Balzarotti†. "Longitudinal Study of the Prevalence of Malware Evasive Techniques". In: (2021). URL: https://arxiv.org/abs/2112.11289.

[34] Ping Chen et Christophe Huygens et Lieven Desmet et Wouter Joosen. "Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware". In: (2016). URL: https://inria.hal.science/hal-01369566/document.

[35] *qemu-system-x86_64(1) - QEMU emulator*. URL: https://man7.org/linux/man-pages/man1/qemu-system-x86_64.1.html.

[36] Microsoft. *WinDbg Preview*. 2025. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools.

[37] ayoubfaouzi. *Al-Khaser v0.81*. 2025. URL: https://github.com/ayoubfaouzi/al-khaser.

[38] Microsoft. *EnumSystemFirmwareTables function (sysinfoapi.h)*. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-enumsystemfirmwaretables.

[39] Microsoft. *GetSystemFirmwareTable function (sysinfoapi.h)*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemfirmwaretable.

[40] Microsoft. *Windows Management Instrumentation*. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page.

[41] Inc. VMware. *backdoor_def.h*. 2023. URL: https://github.com/vmware/open-vm-tools/blob/stable-13.0.0/open-vm-tools/lib/include/backdoor_def.h.

[42] Checkpoint research. *Renewed APT29 Phishing Campaign Against European Diplomats*. 2025. URL: https://github.com/kernelwernel/VMAware.

[43] Google. *Qu'est-ce qu'une machine virtuelle ?* URL: https://cloud.google.com/learn/what-is-a-virtual-machine?hl=fr.

[44] curiouslearnerblog. *Qemu-TCG instruction emulation*. 2016. URL: https://curiouslearnerblog.wordpress.com/2016/05/04/qemu-tcg-instruction-emulation/.

[45] Unprotect Project. *Sandbox Evasion*. URL: https://unprotect.it/category/sandbox-evasion/.

[46] Michael Sikorski et Andrew Honig. *Practical Malware Analysis*. San Francisco: no starch press, 2012. URL: https://www.kea.nu/files/textbooks/humblesec/practicalmalwareanalysis.pdf.

[47] a0rtega. *Dépôt GitHub de pafish*. URL: https://github.com/a0rtega/pafish.

[48] kernelwernel. *Dépôt GitHub de VMAware*. URL: https://github.com/kernelwernel/VMAware.