Detection of an HVM rootkit (aka BluePill-like)

Anthony Desnos, Éric Filiol, Ivanlef0u

ESIEA : Laboratoire de Sécurité de l'Information et des Systèmes (SI&S)
Laboratoire de Virologie et Cryptologie Opérationnelles (V&CO)

Abstract. Since the first systems and networks developed, virus and worms matched them to follow these advances. So after a few technical evolutions, rootkits could moved easily from userland to kernelland, attaining the Holy Grail: to gain full power on computers. Those last years also saw the emergence of virtualization techniques, allowing the deployment of software virtualization solutions and at the same time to reinforce computer security. Giving ways to a processor to manipulate virtualization have not only significantly increased software virtualization performance, but also has provide new techniques to virus writers.
These effects had the impact to create a tremendous polemic about this new kind of rootkits – HVM (Hardware-based Virtual Machine) – and especially the most (in)famous of them: *Bluepill*. Some people claim them to be invisible and consequently undetectable thus making antivirus software or HIDS definitively useless, while for others, HVM rootkits are nothing but fanciful. However, the recent release of the source code of the first HVM rootkit, *Bluepill*, allowed to form a clear picture of those different claims. HVM can indeed change the state of a whole operating system by toggling it into a virtual machine and thus taking the full control on the host and on the operating system itself.
In this paper, we have striven to demystify that new kind of rootkit. On first hand, we are providing clear and reliable technical data about the conception of such rootkit to explain what is possible and what is not. On a second hand, we provide an efficient, operational detection technique that makes possible to systematically detect *Bluepill*-like rootkits (aka HVM-rootkits).

1 Introduction

Hardware rootkits are for hackers the best way to obtain a full control on the victim. For now, they have been contained to external peripheral device on classical computer. But the apparition of virtualization features in modern processor and the possibility to install hypervisor on the top of operating systems, allowed the emergency of new threat of rootkits.

Intel and AMD provide in their last processors (dual core and amd64) mechanisms to use easily total virtualization or para-virtualization. Bringing a new ring, *Ring -1*, where a hypervisor boot firstly on the host and can manage several virtuals machines. This new class of rootkits, *HVM* rootkits, have hijacked this first purpose to move on the fly the state of an operating system to a virtual machine.

Furthermore, the announcement [31] of the first rootkit fully undetectable using virtualization, *BluePill*, has the effect to generate a general fury in the computer security world. This *security buzz* and the fact that the rootkit can control all timing resources, to monitor all inputs/outputs without installing any *hooks* in memory, results to make the conformist spotting methods ineffective. But this agitation and this no scientific thinking to ask a problem serenely, to stifle the creativity of researchers in the reuse of detecting ways.

Firstly, we present virtualization technologies, and particularly hardware virtualization (Intel and AMD). Thus we will introduce HVM rootkits, to explain their internal work and the controversy that has been emerged. Secondly, we will analyse technicals detecting suggested by the security community, to analyze the exact nature of an HVM rootkit (*BluePill*) and to extend technicals detecting, providing news one and to test them in real situations. At last, we will conclude and address some open-problems with respect to our work.

2 State of art

2.1 Virtualization

Virtualization is a set of technical material and/or software that can run on a single machine multiple operating systems separately from each other as if they were operating on distinct physical machines.

These techniques are not recent but issues for much of the work of IBM research center in Grenoble France in the 70s, which developed the experimental system CP/CMS, becoming the product (then called hypervisor) VM/CMS.

In the second half of the 80s and early 90s, embryos virtualization for personal computers have emerged. The Amiga computer could launch pc x386, Machintosh 68xxx, see solutions X11, and of course all in multitasking. In the second half of 1990, on x86 emulators of old machines of the 1980s were a huge success, including Atari computers, Amiga, Amstrad and consoles NES, SNES, Neo Geo.

But the popularity of virtual machines came with VMware in 2000, which gave rise to a suite of free and proprietary softwares offering virtualization.

Thus, several virtualization techniques can be considered :

- Emulation,
- Full Virtualization,
- Para-virtualization,
- Hardware Virtualization.

2.2 Hardware-assisted Virtualization

In this race of the best virtualization, manufacturers of processors arrived. They have equipped their processor with a new set of instructions, a new context, to optimize and facilitate the full virtualization or para virtualization (we called this type of virtualization, cooperative vitualization (figure 3)), thus obtaining the commutation of different virtual machines directly into the processor.

Examples :

- Xen
- Virtual PC

The two main manufacturers of mass market processors, Intel and AMD, respectively introduced the technology in the processor Vanderpool and Pacifica [16]. They are currently available by default in the Intel Dual Core, and AMD 64-bit.

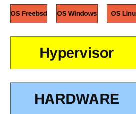

Fig. 1. Hardware-assisted Virtualization

AMD Virtualization As Intel, AMD cames with new features :

- Quickly switch from host to guest,
- Intercepting of instructions or guest's events,
- DMA access protection : EAP (External Access Protection),
- Tagged TLB between the hypervisor and the virtual machines.

SVM *: Secure Virtual Machine extensions* AMD gives a new set of instructions to take full advantage of virtualization : *SVM*. It allows you to run virtual machines and achieve the materially switching host/VM, ie that each virtual machine has a context that will automatically restored/saved by the processor at each context switching (Hypervisor $\Longleftrightarrow$ Virtual Machine). It can also handle exceptions caused by the virtual machine, intercept instructions or to inject interruptions.

We see the interest to activate this mode if it is available, because it allows you to have full control over the machine.

Invited *Mode* This new mode (real, not real, protected) is introduced by AMD to facilitate virtualization.

VMCB The *VMCB* (or control block of virtual machine), is a structure in memory to describe the state of a machine that will run, and several parts are to be considered :

- a list of instructions or events in the guest to intercept,
- bytes control specifying the execution environment of a guest or indicate special actions to do before executing the code of the guest,
- the state of the processor of the guest.

Activating of SVM Before activating the *SVM*, we must check that the processor has this feature. By executing the *cpuid* instruction with the address *8000_0001h*, the second byte of the *ecx* register must be set to 1.

To activate the *SVM*, we must set the *SVME* bit of *EFER* MSR to 1.

VMRUN This is the most important instruction. It makes possible to run a new virtual machine by providing a control block of virtual machine (*VMCB*), describing features expected and the status of this new machine.

VMSAVE/VMLOAD Both instructions complete the *VMRUN* instruction by saving and loading the control block.

VMMCALL This instruction calls the hypervisor, as in ring 3 or ring 0. The choice of the mode in which this instruction can be called is left to the hypervisor.

#VMEXIT When an interception is called, the processor makes a *#VMEXIT* thereby to switch the status of the virtual machine to hypervisor.

2.3 Rootkits

A rootkit is a program or a set of programs allowing a pirate to maintain an access to a computer system. Rootkits have existed since the beginning of hacking and are therefore constantly changing with new technology.
Features are various, but the main goal is the same, to hide all traces of a hacker :

- codes,
- process,
- networks,
- drivers,
- files,
- $\Longrightarrow$ everything a mind can imagine !

We can classify rootkits in two families :

- Ring 3 (user land),
- Ring 0 (kernel land).

The first family is the oldest, easy to use because it's in ring 3. It is simply an amalgamation of several binaries (ps, ls, netstat, etc.) that will be installed in place of the original, and that filters results to hide data. It is trivial to detect by hashes on the file system ([40]).

But recent years have seen the emergence of attacks all in memory, making progress rootkits in ring 3, and leaving the door open to new types of rootkits. Staying in memory for an attacker is interesting because no information will be written on a mass device (hard drive ...) and thus bypasses some tools of forensics [14].

Three types of userland infections are to be considered :

- Patch on the fly,
- Syscall Proxy,
- Userland Execve.

Path on the fly [9] is a technique [39] [15] to patch [3] dynamically a process, injecting codes, data, and to hijack functions.
Syscall Proxy is a technique which consists in executing a program entirely on the network by sending most of instructions to the exploited server. More precisely, when an usual program is running, it sends many system calls to the kernel in order for example to have access to the I/O peripherals. With Syscall Proxy, all system calls are sent by the attacker, treated by the kernel of the server, and their results are returned. However, even though this method appears original, it uses extensively the network resources. Its performance is thereby directly related because a huge amount of messages transits on the network (two per system call). But most of all, the capacity of detection by the administrators become pretty easy.

The last techniques consists to execute a program without the sys_exec syscall [23] (*sys_execve* on Linux). It replaces in memory the old process with a new code that we will run, or simply to insert a relocatable binary and to jump on it. By using the network,there is no writing on the hard drive [22]. Several automation tools (as *SELF* [29], pitbull [28], or more recently *Sanson The Headman* [35]) have emerged to use this technique easily.

Rootkits in ring 0 allow a stronger level of invisibility for the user. They are used to hide processes, connections, files or to bypass some mechanisms of protection. Three categories [32] are to be considered :

- Those installing hooks in the kernel code,
- Those installing hooks in fields of kernel structure,
- Those with no hooks.

The first category [36] [37] changes the system call table, the interrupt descriptor table, but also redirects some functions. It is therefore easily detectable with tools to make fingerprints of the kernel memory. The abstraction in the Linux kernel can bypass flows [37] by changing pointer functions in a structure. We can easily change the pointer function of the VFS structure, thus hiding all kinds of things. Again this kind of compromise can be detected with memory fingerprints.

The last category is this new generation of malware inherited from virtualization technology hardware.

2.4 Controversy

The problem which appears with this kind of rootkit because it doesn't install hooks in memory and use memory allocator of the system, and it can control various sources of time in a computer against a timing attack. All classic sources, as *RDTSC* instruction which allowed to know the number of ticks of the processor, or clocks in the mother board may be intercepted by the hypervisor, respectively directly on a call of an instruction or an input/output. As a consequence, hypervisor may altered the return value and to hijack the analyse of a detector.

Detecting a hypervisor may be the same problem as detecting an hvm rootkit ? Maybe not. But let us not be too positive [32] in our answer. An user, an administrator system knows if he owns a virtual machine monitor, as he would use himself a virtualization tools (Virtual PC, KVM, XEN). If the detecting system decides that a hypervisor is installed while the user didn't know, thus a rootkit is present. Of course we will show that a payload will enabled us to do without user's knowledge environment .

Several researchers have reacted quickly (maybe too) to this *security buzz* in suggesting sundries solutions :

Timing attack This attack is established on a simple rule. A rootkit alters results and append new instructions [33], we must have a safely database which can be compared to new measurements. However *BluePill* controls all timing resources, it can played with clocks and changed the return value of the time of the instruction.

Pattern matching Pattern matching consists of searching a signature of a rootkit, as for example loading or unloading functions. This method may be used in the *Bluepill* case (in the current release), but it can control I/O and harms the integrity of the reading memory (as a result, hiding itself).

TLB The attack though TLB to detect if a hypervisor is present, is based on the fact that a virtual machine monitor puts the TLB entries to 0 if it intercepts an instruction. It's easy for the detector to watch timing access of a page, to call an intercepted instruction, and read the new timing access to the same page and compared both results.

According to Joanna Rutkowska [32], she is capable to hijack this detecting kind, moreover in AMD processors, the TLB is tagged with an address space identifier (ASID) distinguishing host-space entries from guest-space entries.

DMA Access to the DMA though an external peripheral device [26] as firewire allows to recover physical memory without modification. It is therefore possible to detect an HVM rootkit by searching its signature.

In last processors of *AMD*, EAP (External Access Protection) [17] could be used by a hypervisor to fake the fingerprint.

Also, this solution would be no viable in the future, because IOMMU [2] will allow to solve this problem without access control to the memory by a device.

CPU Bugs This method is simple : crash the processor when virtualization is enabled. It is interesting in experimentation to detect a hypervisor, but no usable in production, and these bugs can be fixed in the next release of a processor.

3 *BluePill* rootkit

BluePill is the first (and only at the moment) public HVM rootkit, created by Joanna Rutkowska in 2006, it has been the subject of several publications, most of them about the subject that it is undetectable, without analysis of its working.

3.1 Installation

Bluepill must be loaded as a driver. But Windows Vista (and Windows Server 2008) integrates a security policy against no driver signing [41]. Thereby, either the driver is loaded by an exploitation [31], or we disable driver signing during the boot of the operating system (function key F8), in this case the field of action is confine (on Windows Vista).

For our experimentation, we have disabled driver signing, and loaded *BluePill* with the tool insdrv.

The first public release (0.11) of *BluePill* works only on *AMD* [16] and the output is on serial port, which is not very usefull (nevertheless it isn't required to have a null modem cable to recover the output). The last public release (0.32) available on web site adds *Intel* processor [24], and writing in the logs of the system.

3.2 Analysis

BluePill didn't install hooks, this is why it is impossible to reinstate during boot. There are several possibilities, either infecting operating system during operating system boot, or sooner during boot as *SubVirt* [30]. In other words, in both cases would make it a classic rootkit and more easily detectable.

BluePill moves the state of an operating system into a guest operating system. No more and no less. Now (except version 0.32, with an Intel keylogger), he hasn't classic rootkit mechanisms (hiding files process, networks, etc). As a result, we can ask us if it is not simply the result of a very good job of a kernel developer and not a designer rootkit? And why any payload is present when it's presented as the most frightening rootkits ?

This is its greatest weakness : to not contain viral payload. *BluePill* can be the same behaviour that a classical rootkit. There are two solutions. Either it hooks functions or structures

to realize these behaviors, what a classical rootkit and the detection mechanism is well known, or it monitors the input/output. It may choose the latest solution, but which is the price ? The time ....

Its great strength is to control everything remaining invisible, and could induce a big treatment and thus that causing its loss ?

Thereafter we will analyse summarily *BluePill* source code.

3.3 Working

The working of this new type of malware can be summarized into one sentence : "Switching the operating system into a virtual machine". It is clear that the switching of the state on the fly, allows an interesting stealth (no reboot as Subvirt [30]), and the state of a virtual machine allows a full control.

The algorithm [42] of the new kind of rootkit is in ten steps, but can be resumed in the following section.

Algorithm

1. Loading of the driver,
2. Verification/Activation of the hardware virtualization,
3. Memory allocation of different pages (control block, saved area of the host ...),
4. Initialization of different fields of the control block of the virtual machine (control area, virtual machine area),
5. Switch to the hypervisor execution code,
6. Call of the instruction which run the virtual machine,
7. Unloading of the driver.

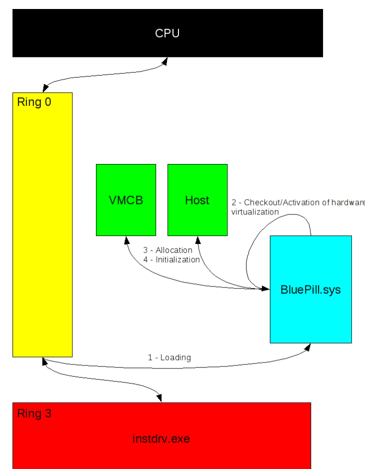

Fig. 2. BluePill during the infection

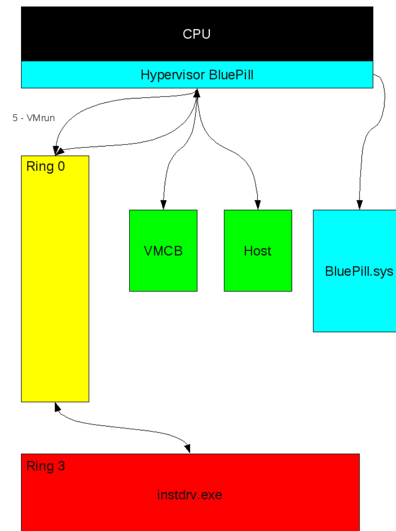

Fig. 3. Bluepill after the infection

Now, we will analyze each parts of this algorithm by associating it to the version 0.32-public [25] of *BluePill*.

Its code is splitted into different parts :

- amd64 : assembly code of the hypervisor, calling of the SVM/VMX instructions, reading/writing code of MSR ..
- common : common code of the rootkit (loading, unloading, etc),
- svm : code for the SVM instructions set,
- vmx : code for the VMX instructions set.

The common code allows via a structure *HVM_DEPENDENT* of function pointers to manage SVM or VMX :

```
/* common/common.h */

typedef struct
{
  UCHAR Architecture;

  ARCH_IS_HVM_IMPLEMENTED ArchIsHvmImplemented;
   [...]
} HVM_DEPENDENT,
```

Loading

Without doubt, the hardest part, it's to find an attack vector to load the rootkit. Typically, the attack requires getting a communication channel to the kernel to insert our code :

- Either by the interface of loading. Now, it is blocked in Windows Vista, because a driver must be signed to load, which has been bypassed [32] but quickly corrected by Microsoft,
- Either by memory devices (*/dev/kmem* on Linux, disabled on Windows Vista, but with a relocation of the code in memory (for example with *Kernsh* [38]).
- Either by the exploitation of a kernel security flaw.

The loading of *BleuPill* begins in the loading driver routine on Windows, the *DriverEntry* function :

```
/* common/newbp.c */

NTSTATUS DriverEntry(
  PDRIVER_OBJECT DriverObject,
  PUNICODE_STRING RegistryPath
)
{
[...]
[A] HvmInit();
[B] HvmSwallowBluepill();
[C] DriverObject->DriverUnload = DriverUnload;
[...]
}
```

Three main things are done : *HvmInit* will check the availability of virtualization hardware [A], and *HvmSwallowBluepill* will run the rootkit [B]. We must also setup [C] the field of the unloading routing of the driver of the structure *DriverObject* with the unloading function.

HvmSwallowBluepill :

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSwallowBluepill (
)
{
[...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL, &CallbackStatus);
}
[...]
}
```

The initialization of the rootkit must be done on each processor [A], that is why we associate to each processor the setup routine (*CmSubvert*).

```
/*  amd64/common-asm.asm */

CmSubvert PROC
[...]
        [A] call      HvmSubvertCpu
CmSubvert ENDP
```

This assembly routine calls the real installation routine [A] (*HvmSubvertCpu*).

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSubvertCpu (
  PVOID GuestRsp
)
{
[...]
[A] Hvm->ArchIsHvmImplemented();

[B] HostKernelStackBase = MmAllocatePages (HOST_STACK_SIZE_IN_PAGES, &HostStackPA);
[C] Cpu = (PCPU) ((PCHAR) HostKernelStackBase + HOST_STACK_SIZE_IN_PAGES
* PAGE_SIZE - 8 - sizeof (CPU));
[D] Cpu->ProcessorNumber = KeGetCurrentProcessorNumber ();
[E] Cpu->GdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_GDT_LIMIT), NULL);
[F] Cpu->IdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_IDT_LIMIT), NULL);
```

```
[G] Hvm->ArchRegisterTraps (Cpu);
[H] Hvm->ArchInitialize (Cpu, CmSlipIntoMatrix, GuestRsp);

[I] HvmSetupGdt (Cpu);
[J] HvmSetupIdt (Cpu);

[K] Hvm->ArchVirtualize (Cpu);
}
```

HvmSubvertCpu is the main installation routine, which is called on each processor. It will first check the availability of virtualization [A], then perform various allocations spaces and structures [B], [C], [E], [F]. At [B], the allocation of this saved host area allows the *vmrun* instruction to save information about the state of the processor. *KeGetCurrentProcessorNumber* gets the number of processor where is running this code [D].

Finally, event managements [G] by *ArchRegisterTraps*, and various initializations [H], [I], [J], launch the hypervisor [K] by *ArchVirtualize*.

Checking of hardware virtualization
Hvm→ArchIsHvmImplemented == SvmIsImplemented :

```
/* svm/svm.c */

static BOOLEAN NTAPI SvmIsImplemented (
)
{
[A] GetCpuIdInfo (0, &eax, &ebx, &ecx, &edx);
[B] if !(ebx == 0x68747541 && ecx == 0x444d4163 && edx == 0x69746e65)
                return FALSE;
[C] GetCpuIdInfo (0x80000000, &eax, &ebx, &ecx, &edx);
[D] GetCpuIdInfo (0x80000001, &eax, &ebx, &ecx, &edx);
[E] return CmIsBitSet (ecx, 2);
}
```

The *CPUID* assembly instruction gets information about features of the processor. The first function [A] checks if the processor has the extend *CPUID* instruction, and also check if we are on a AMD processor [B].

The function [C], [D], [E] check if the second byte of ecx register is setup.

Initialization of events management
Hvm→ArchRegisterTraps == SvmRegisterTraps :

```
/*  svm/svmtraps.c */

NTSTATUS NTAPI SvmRegisterTraps (
  PCPU Cpu
)
{
[...]

TrInitializeGeneralTrap (Cpu, VMEXIT_VMRUN, 3, SvmDispatchVmrun, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMLOAD, 3, SvmDispatchVmload, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMSAVE, 3, SvmDispatchVmsave, &Trap);
[...]
}
```

The initialization of the function that will handle interception are saved and associated with at the corresponding interception.

So, *BluePill* intercepts the following operations :

- instructions : vmrun, vmload, vmsave,

- registers msr efer, vm_hsave_pa, tsc,
- instructions : clgi, stgi,
- interrupts of SMM,
- debug exception,
- instructions : cpuid, rdtsc, rdtscp.

Allocation/Initialization
Hvm→ArchInitialize == SvmInitialize :

```
/*  svm/svm.c */

static NTSTATUS NTAPI SvmInitialize (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
[...]
Cpu->Svm.Hsa = MmAllocateContiguousPages (SVM_HSA_SIZE_IN_PAGES,
&Cpu->Svm.HsaPA);
[A] Cpu->Svm.OriginalVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
&Cpu->Svm.OriginalVmcbPA, MmCached);
[B] Cpu->Svm.GuestVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
NULL, MmCached);
[C] Cpu->Svm.NestedVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
&Cpu->Svm.NestedVmcbPA, MmCached);
[...]
[D] SvmSetupControlArea (Cpu);
[E] SvmEnable (&bAlreadyEnabled);
[...]
[F] SvmInitGuestState (Cpu, GuestRip, GuestRsp);
[...]
}
```

There are allocations [A], [B], [C] of all VMCB, then the initialization of the control area [D], allows the activation of the virtualization [E]. Then, the initialization of the VMCB of the state of the processor.

SvmEnable :

```
/*  svm/svm.c */

NTSTATUS NTAPI SvmEnable (
  PBOOLEAN pAlreadyEnabled
)
{
[...]
Efer = MsrRead (MSR_EFER);
[A] Efer |= EFER_SVME;
[B] MsrWrite (MSR_EFER, Efer);
[...]
}
```

To enable the *SVM*, the *SVME* byte of the *EFER* MSR must be set [A], [B] to 1.

SvmInitGuestState :

```
/*  svm/svm.c */

NTSTATUS SvmInitGuestState (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
```

```
  [...]
  Vmcb = Cpu->Svm.OriginalVmcb;

  Vmcb->idtr.base = GetIdtBase ();
  Vmcb->idtr.limit = GetIdtLimit ();
  GuestGdtBase = (PVOID) GetGdtBase ();
  Vmcb->gdtr.base = (ULONG64) GuestGdtBase;
  Vmcb->gdtr.limit = GetGdtLimit ();
  [...]
  Vmcb->cpl = 0;
  Vmcb->efer = MsrRead (MSR_EFER);
  Vmcb->cr0 = RegGetCr0 ();
  [...]

  Vmcb->rip = (ULONG64) GuestRip;
  Vmcb->rsp = (ULONG64) GuestRsp;
[...]
}
```

The initialization of state part of the VMCB is to setup fields needed by the processor needs, ie addresses of the idt and the gdt. But also information as cr* and dr* registers, and of course the current pointer and the stack pointer.

SvmSetHsa :

```
/*  svm/svm.c */

VOID NTAPI SvmSetHsa (
  PHYSICAL_ADDRESS HsaPA
)
{
}
```

Transfer
Hvm→ArchIsHvmVirtualize == SvmVirtualize :

```
/* svm/svm.c */

static NTSTATUS NTAPI SvmVirtualize (
  PCPU Cpu
)
{
[A] SvmVmrun (Cpu);
// never returns
}
```

The transfer to the cocde of the hypervisor which will launch the virtual machine and manage events, is located in the *SvmVmrun* [A] function.

Calling the virtual machine
SvmVmrun :

```
/* amd64/svm-asm.asm */

SvmVmrun PROC
            [...]
@loop:
            [...]
        [A] mov             rax, [rsp+16*8+5*8+8]    ; CPU.Svm.VmcbToContinuePA

        [B] svm_vmrun

        ; save guest state
            [...]
        call            HvmEventCallback
```

```
        ; restore guest state (HvmEventCallback migth have alternated the guest state)
            [...]
        jmp             @loop
```

The switching to the virtual machine is done by the vmrun [B] instruction which takes one argument, the address of the VMCB of the virtual machine, in the rax register [A].

Events management

The event management is significant for an HVM rootkit, since the viral code must be here.

HvmEventCallback :

```
/* common/hvm.c */

VOID NTAPI HvmEventCallback (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
[A] if (Hvm->ArchIsNestedEvent (Cpu, GuestRegs))
{
[B] Hvm->ArchDispatchNestedEvent (Cpu, GuestRegs);
return;
}

// it's an original event
[C] Hvm->ArchDispatchEvent (Cpu, GuestRegs);
}
```

According to the original source of the event [A], management will be treated differently. But, finally, the processing function well be either *SvmDispatchNestedEvent* [B] or *SvmDispatchEvent* [C].

Hvm→ArchDispatchEvent = SvmDispatchEvent :

```
/* svm/svm.c */

static VOID NTAPI SvmDispatchEvent (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
SvmHandleInterception (Cpu, GuestRegs, Cpu->Svm.OriginalVmcb, FALSE
[...]
}
```

SvmHandleInterception :

```
/* svm/svm.c */

static VOID SvmHandleInterception (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PVMCB Vmcb,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
  [...]
  [A] TrFindRegisteredTrap (Cpu, GuestRegs, Vmcb->exitcode, &Trap);

  switch (Vmcb->exitcode)
        {
        case VMEXIT_MSR:
        [...]
```

```
        case VMEXIT_IOIO:
    [...]
    default :
            [...]
                [B] TrExecuteGeneralTrapHandler (Cpu, GuestRegs, Trap,
WillBeAlsoHandledByGuestHv);
                [...]
        }
}
```

Depending on the type of the event, we seek [A] if an entry exists that supports this kind of events and runs it [B].

Unloading

The unloading of *BluePill* is done by the unloading routine filled in the structure of the driver while loading.

```
/* common/newbp.c */

NTSTATUS DriverUnload (
  PDRIVER_OBJECT DriverObject
)
{
[...]
[A] HvmSpitOutBluepill();
[...]
}
```

The function [A] will perform the unloading of the hypervisor is *HvmSpitOutBluepill* :

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSpitOutBluepill (
)
{
[...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, HvmLiberateCpu, NULL, &CallbackStatus);
}
[...]
}
```

As the loading, we attach an unloading routine [A], *HvmLiberateCpu*, on each process present.

HvmLiberateCpu :

```
/* common/hvm.c */

static NTSTATUS NTAPI HvmLiberateCpu (
  PVOID Param
)
{
[...]
[A] HcMakeHypercall (NBP_HYPERCALL_UNLOAD, 0, NULL);
[...]
}
```

An hypercall is the same as a system call but for a virtual machine. That's why this will create a communication from the virtual machine to the hypervisor [A]. So, the unloading routine of *BluePill* makes a hypercall to the hypervisor to unload itself.

HcMakeHypercall :

```
/* common/hypercalls.c */

NTSTATUS NTAPI HcMakeHypercall (
  ULONG32 HypercallNumber,
  ULONG32 HypercallParameter,
  PULONG32 pHypercallResult
)
{
[...]

// low part contains a hypercall number
[A] edx = HypercallNumber | (NBP_MAGIC & 0xffff0000);
[B] ecx = NBP_MAGIC + 1;

[C] CpuidWithEcxEdx (&ecx, &edx);
}
```

A little trick is present to unload the hypervisor, it makes an hypercall which call an instruction intercepted by the hypervisor with magic parameters. The *cpuid* instruction [C] is used with magic values [A] [B] in the edx and ecx registers, with the first register concatenates to the value of the desired hypercall (unloading).

SvmDispatchCpuid :

```
/* svm/svmtraps.c */

static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
[...]

[A] if (((GuestRegs->rdx & 0xffff0000) == (NBP_MAGIC & 0xffff0000))
[B]         && ((GuestRegs->rcx & 0xffffffff) == NBP_MAGIC + 1))
{
      [C] HcDispatchHypercall (Cpu, GuestRegs);
      return TRUE;
}

[...]
}
```

The function which intercepts the *cpuid* instruction is *SvmDispatchCpuid*, and will check if magic parameters [A], [B] are in registers. If it presents, the management function of hypercalls [C] is called.

HcDispatchHypercall :

```
/* common/hypercalls.c */

VOID NTAPI HcDispatchHypercall (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
switch (HypercallNumber)
{
  [A] case NBP_HYPERCALL_UNLOAD:
      [...]
     // disable virtualization, resume guest, don't setup time bomb
     [B] Hvm->ArchShutdown (Cpu, GuestRegs, FALSE);
          break;
}
```

```
}
```

If the number of the hypercall [A] is an unloading, the function of the right architecture is executed [B].

Hvm→ArchShutdown = SvmShutdown :

```
/* svm/svm.c */

static NTSTATUS NTAPI SvmShutdown (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  BOOLEAN bSetupTimeBomb
)
{
SvmGenerateTrampolineToLongModeCPL0 (Cpu, GuestRegs, Trampoline, bSetupTimeBomb);

CmStgi ();
CmSti ();

if (!Cpu->Svm.bGuestSVME)
    [A] SvmDisable ();

((VOID (*)()) & Trampoline) ();
  // never returns
}
```

The function [A] *SvmDisable* disables the virtualization, and shutdown the hypervisor.

As a conclusion about the summarized analysis of the code of *BluePill*, which finally does the work of a classical hypervisor but much more dynamic because it takes a host and switch into virtual machine. Also, it contains no viral payload (as hide files, processes, etc.), and doesn't hide itself in memory, it is quite empty against a real rootkit.

4 Detection Techniques for HVM Rootkits

We know that it's impossible to detect *Bluepill* with memory fingerprints, even if it is in memory as another driver. Pattern matching of signatures against *BluePill* will be possible, but only usable up till the next release (because *BluePill* can control the I/O).

We have based our research study on a simple fact : a hypervisor increases the time execution of some instructions, and an HVM rootkit will increase significantly this one, we must get the execution time of an instruction. A hypervisor will increase the execution time of an intercepted instruction since the commutation context from the virtual machine to hypervisor will be automatically added, and will be more increased if a viral payload is present. This is a timing attack but we have said that we didn't control sources of time [20]

An external source of time as a NTP server with an encrypted communication can be used, and it will increase time analysis of hypervisor to realize a mechanism for detection.

But a source of time may be relative and therefore do not use directly clocks of the system and can't intercepts by a hypervisor. At a much larger scale, the sun has been used to know the time. We can used on a computer a simple counter (our sun) as an increment of a variable (as shown Edgar Barbosa [12]) on one core, while the other core run an intercepted instruction. Joanna Rutkowska is agreed [32] that this mechanism is impossible to detect and she thinks that it's not possible to detect it now.

The Intel Dual Core processors have capabilities to change the frequency, which could make our results to be false. But with a database and if we set the frequency of a processor, we need few values...

Blue Chicken [32] is a technical which consists of an HVM rootkit withdraws away from memory when a large number of instructions are called and to reinstate after a given time (which is also contesting [12] because a hypervisor protection could then takes the control). We can use technicals of virus, for example a sequence of random calls to bypass it. But the best method is firstly to emerge a statistical model that will allow us to limit strongly instructions calls to detect an HVM rootkit.

A list of intercepted instructions is the list of all possible intercepted instructions by the hypervisor. One of them is interesting : vmmcall. It allows the virtual machine to call the hypervisor, so the instruction must be intercepted by the hypervisor.

But the best method is to find a suitable statistical model which will limit calls of an intercepted instruction to detect a rootkit.

4.1 Statistical model for detection

The goal is to model the behavior of a computer according to two different cases: with or without *BluePill*. If we manage to do it, then we will have a powerful detection technique based on decision statistical testing. In the case when a *BluePill*-like rootkit is active, we must be able to make the difference between simple *BluePill* module and *BluePill* module with an effective payload. From a technical point of view, everything lies on the choice of a suitable, powerful (statistical) estimator that will exhibit statistically significant different behaviors, according to the case concerned.

Let us first consider the different technical way on which we can build such efficient estimators.

NTP (encrypted) The first method is to take an external time quite accurate and reliable. Most servers use NTP, which lets to synchronize the clocks of computer across a network. We may use encryption on the protocol to have safe communications and to make the work of the rootkit more complex.

So, the core estimator will be the time difference of the value returned by the NTP server on a possible intercepted instruction.

Counter The second method is to a use a relative time as the real time, for example a counter. But it works only with a processor with several cores (at least 2) or on a multiprocessor machine (e.g. dual core processor).

We synchronize the two cores (or processors) to run our code in parallel. On the first core (or processor), a simple loop increments a variable, while on the other a potentially intercepted instruction (by the HVM rootkit) will be launched.

If a rookit exists, the value of the incremented variable will be increase will be superior to a case without rootkit.

In the first case (figure 4), instruction runs directly on the processor, in the next case (figure 5), the virtualization is enable this why we have extra instructions to do the commutation, with the extra code of the events management of *BluePill*, and therefore the code of the real event.

Working This code must be launched at Ring 0, because the implementation of threads on userland doesn't allowed us to choose the processor.

Version of Linux 2.6.X On Linux, the call of the function *kthread_create* creates a kernel thread, to choose the processor with *kthread_bind*, and run it with the function *wake_up_process*.

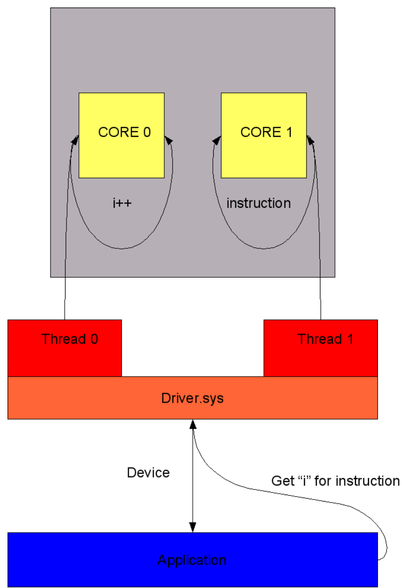

Fig. 4. Method of detection : Counter

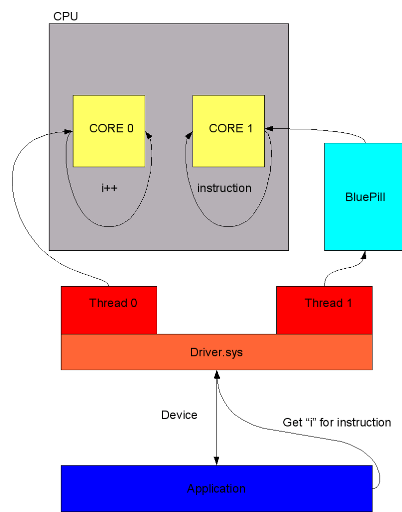

Fig. 5. Method of detection : Counter + BluePill

Example The thread that runs the counter (function *timepill_kthread_cpu0*) and the other runs the instruction (function *timepill_kthread_cpu1_noloop*) can be program as follows, with *ktimepill_counter_t*, a structure to get the loop counter (in the field *titmap*), and the call of the function (the field *inst*).

```
static void timepill_kthread_cpu0(void *data)
{
  int counter;
  atomic_t cc;
  unsigned long *p;
  ktimepill_counter_t *kct = (ktimepill_counter_t *)data;

  counter = 0;
  atomic_set(&cc, 0);

  if (kct == NULL)
    goto timepill_kthread_cpu0_out;

  down(&sem);
  up(&sem2);

  down(&semcount);
  counter = atomic_read(&stop_counter);
  up(&semcount);

  while (counter == 0)
    {
      atomic_inc(&cc);

      down(&semcount);
      counter = atomic_read(&stop_counter);
      up(&semcount);
    }

  p = (unsigned long *)kct->titmap;
  *p = atomic_read(&cc);

  kct->thread = NULL;
 timepill_kthread_cpu0_out:
  up(&thread0);
}
```

```
static void timepill_kthread_cpu1_noloop(void *data)
{
  ktimepill_counter_t *kct = (ktimepill_counter_t *)data;

  if (kct == NULL)
    goto timepill_kthread_cpu1_noloop_out;

  down(&semcount);
  atomic_set(&stop_counter, 0);
  up(&semcount);

  up(&sem);
  down(&sem2);

  kct->inst();

  down(&semcount);
  atomic_set(&stop_counter, 1);
  up(&semcount);

  kct->thread = NULL;
 timepill_kthread_cpu1_noloop_out:
  up(&thread1);
}
```

Version of Windows Vista On windows, the call of the function *PsCreateSystemThread* creates a kernel thread, and the function *KeSetSystemAffinityThread* chooses the processor.

This driver (because we are in kernelland) gets results of the number of loop and to send it to the main program by an `ioctl`.

Example As the Linux version, two threads (function *thread_counter* and *thread_inst*) get the counter and call the instruction, with the structure *timepill_kern_t* which has the field *map* to store values, and the field *inst* to the instruction.

```
static VOID NTAPI thread_counter(PVOID Param)
{
        int stop_counter;
        ULONG cc;
        unsigned long *p;
        timepill_kern_t *tkt;

        tkt = (timepill_kern_t *)Param;
        cc = 0;

        KeSetSystemAffinityThread((KAFFINITY)0x00000001);

        KeSetEvent(tkt->myevent,
                0,
                FALSE);

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
                NULL);
        stop_counter = tkt->counter;
        KeReleaseMutex(&mut, FALSE);

        while(stop_counter == 0)
        {
                cc++;
                KeWaitForSingleObject(&mut,
                        Executive,
                        KernelMode,
                        FALSE,
                        NULL);
                stop_counter = tkt->counter;
                KeReleaseMutex(&mut, FALSE);
        }

        p = (unsigned long *)tkt->map;
        *p = cc;

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

```
static VOID NTAPI thread_inst(PVOID Param)
{
        int i;
        ULONG eax, ebx, ecx, edx;
        timepill_kern_t *tkt;

        tkt = (timepill_kern_t *)Param;
        KeSetSystemAffinityThread((KAFFINITY)0x00000002);

        while(STATUS_TIMEOUT == KeWaitForSingleObject(tkt->myevent,
                Executive,
                KernelMode,
                FALSE,
                NULL));

                tkt->inst();

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
```

```
                NULL);
        tkt->counter = 1;
        KeReleaseMutex(&mut, FALSE);

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

5 Experimental results

5.1 Pillbox

To test our methods of detection, we have written a tool called *pillbox*, two parts are to be considered :

- the client picks up data (results), and sends to the server. The client is composed of a userland program which collects measures from the driver (for example, with the counter technique),
- the server receives results from the client and then analyse results.

The client has different methods to pick up results, depending on the privilege level:

In user land

- by the *RDTSC* instruction,
- by the
- gettimeofday function,
- by an external NTP server,
- by the counter method.

In kernel land

- by the counter method.

For us, we will focus on the counter method in kernel land, because it's the most efficient technique, and the most difficult to intercept by a rootkit.

For graphic representation, we used three types of format to more quickly analyze the results :

- axis of abscissas : the instruction, axis of ordinates : the relative time,
- axis of abscissas : identical relative times, axis of ordinates : the number of identical relative times,
- axis of abscissas : the relative time, axis of ordinates : the number of measures.

All tests have been done on an AMD 64 processor with virtualization, and with a frequency of 2 Ghz, and with Windows Vista.

As a first step, we will consider an instruction (*CPUID*) that *BluePill* can intercept, and observed cases with and without the rootkit.

Without BluePill In the first graph (figure 6), we observe that the average revolves around a relative value of 30 increments loop, to run the instruction. But also peaks are due to commutations of the system, but do not affect the analysis.

With the second (figure 7) and the last (figure 8), the *cpuid* instruction has an average of 33 incrementation loop.

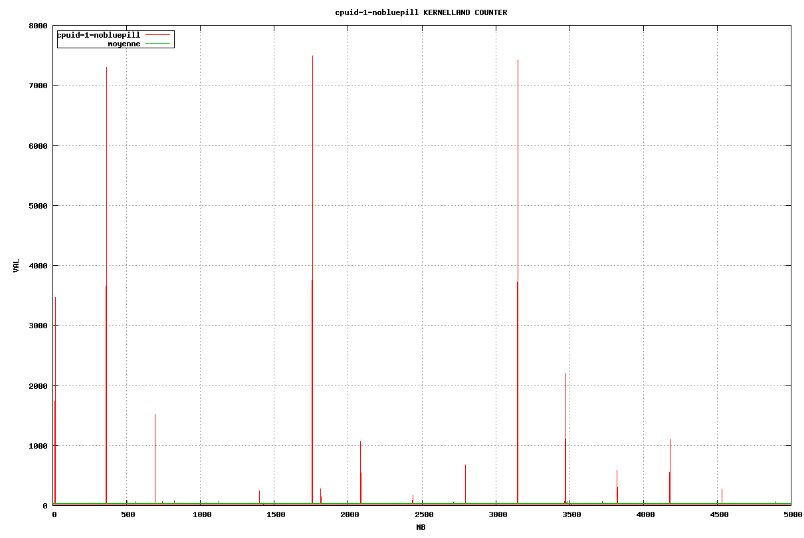

Fig. 6. Method : Counter (without BluePill)

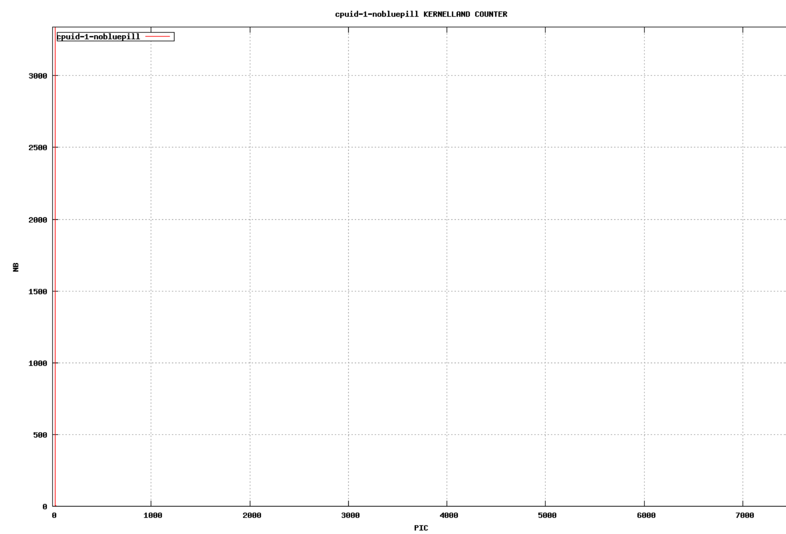

Fig. 7. Method : Counter (without BluePill), Picks

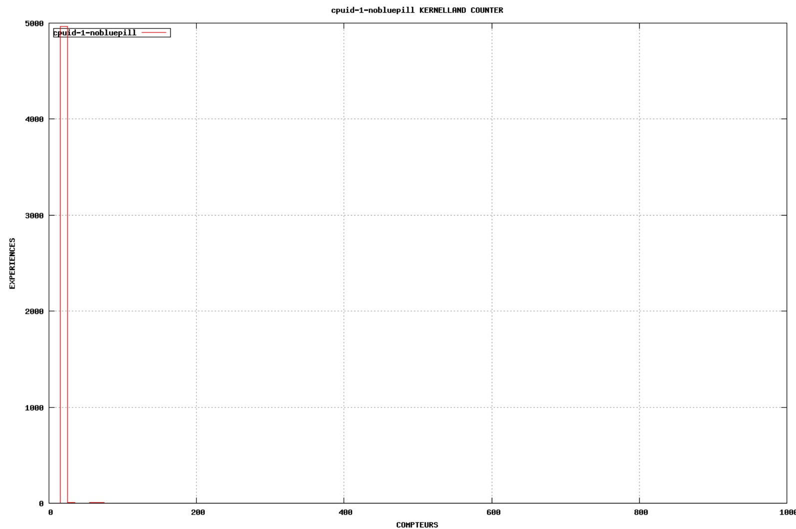

Fig. 8. Method : Counter (without BluePill), bar graph

With BluePill Now, if *BluePill* is present, this one intercepts the *cpuid* instruction, looks the state of registers to check a magic value, modity it if presents (we will not use magic values for our test), and call cpuid instruction:

```
static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
[...]

Vmcb = Cpu->Svm.OriginalVmcb;

if ((Vmcb->rax & 0xffffffff) == BP_KNOCK_EAX)
{
    _KdPrint (("Magic knock received: %p\n", BP_KNOCK_EAX));
    Vmcb->rax = BP_KNOCK_EAX_ANSWER;
}
else
{
         [...]
    fn = (ULONG32) Vmcb->rax;
    GetCpuIdInfo (fn, &(ULONG32) Vmcb->rax, &(ULONG32) GuestRegs->rbx,
                  &(ULONG32) GuestRegs->rcx, &(ULONG32) GuestRegs->rdx);
}
}
```

In the first graph (figure 9), the interception of the hypervisor increases the time of the instruction execution. In addition, other graphics (figures 10 11) check that the average is totally moved, since it is now 332. With an HVM rootkit with no viral payload, but playing the role of a hypervisor, we have a time which is ten times higher, which can easily detect the presence (or absence) of a hypervisor, which is for us an HVM rootkit as said in the previous sections because the user knows if he uses a hypervisor.

There is always the same behavior with the instructions that *BluePill* can intercept, and in particular with *vmmcall* that any hypervisor must manage.

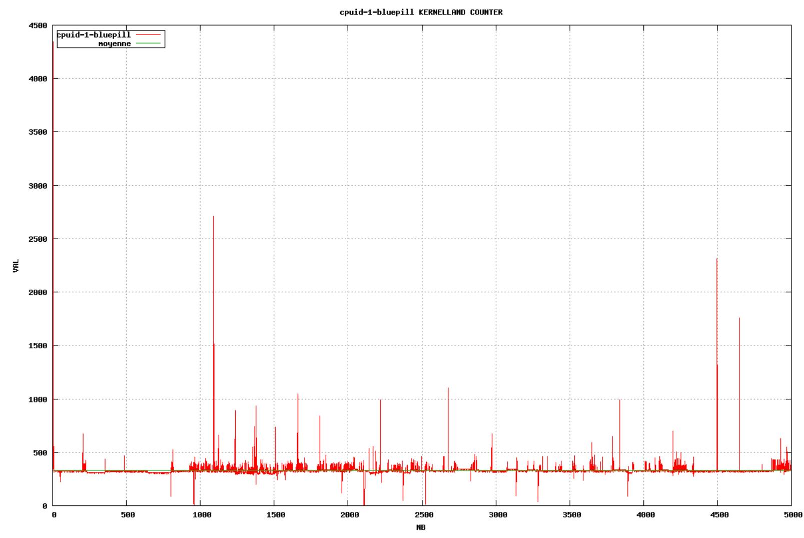

Fig. 9. Method : Counter (with BluePill)

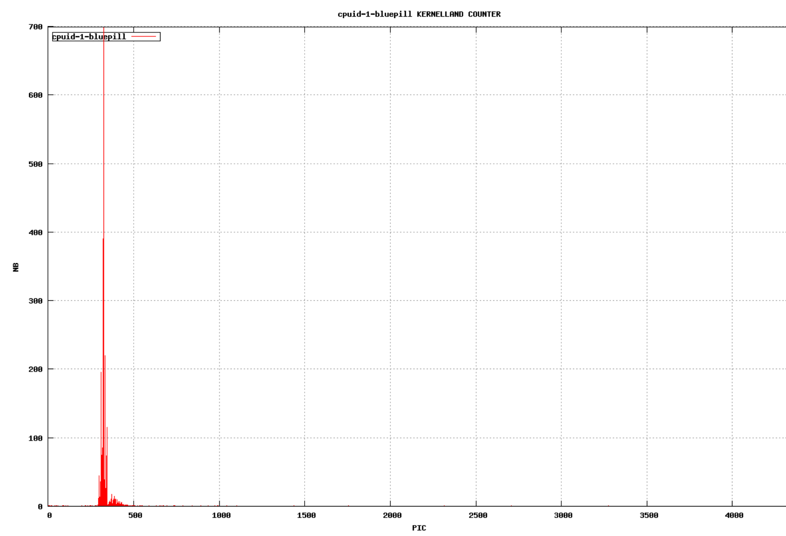

Fig. 10. Method : Counter (with BluePill), Picks

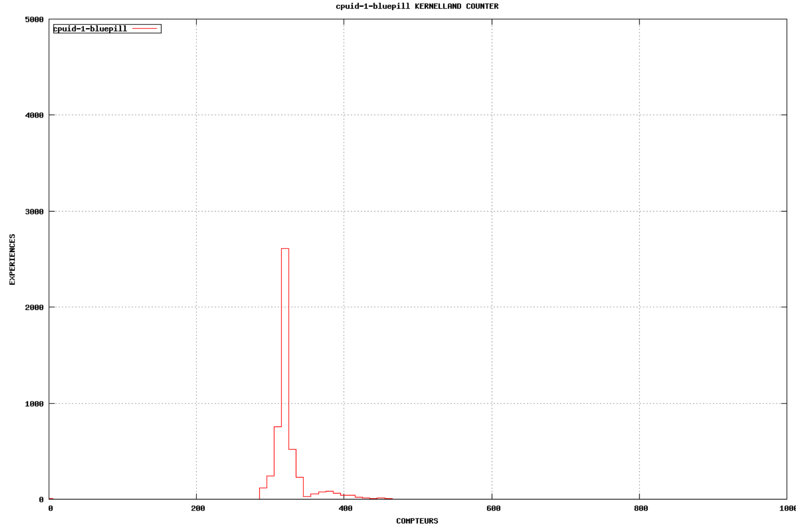

Fig. 11. Method : Counter (with BluePill), bar graph

5.2 Statistical Modelling of *BluePill*-like Rootkits

We are now considering the counter value, defined in Section 4.1, as a suitable estimator. Without loss of generality, that approach remains the same when considering the case of an estimator built from the NTP technique, which has been exposed in Section 4.1.

In a first step, a large number of experiments ($N = 10000$) have been performed in order to collect a statistically significant number of data. On every test sample, we have obtained, we have computed the mean μ and the corresponding standard deviation σ. Then in a second step, we have supposed that our estimator was distributed according a Gaussian distribution law. To verify this on a thorough way, we then performed a goodness-of-fit test (χ^2 test) to compare it to the normal distribution, with an error type I of $\alpha = 0.005$. Even if the χ^2 is not an optimal test (since it lacks of power and since the choice of the different test classes can be considered as subjective), it remains however a very efficient and convenient tool that is not to far from the reality in most cases. Future works will nonetheless consider more powerful tests (e.g. Shapiro-Wilk test). But without to much risk, we can claim that we should obtain the same result: our estimator is indeed normally distributed.

5.3 Without *BluePill*

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 26, 78$,
- Standard deviation $s = 13.34$,
- Normal distribution $\mathcal{N}(26; 13)$.

5.4 Avec *BluePill*

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 339, 25$,
- Standard deviation $s = 38, 26$,
- Normal distribution $\mathcal{N}(339; 38)$.

5.5 With *BluePill* and payload

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 1675,60$,
- Standard deviation $s = 77,91$,
- Normal distribution $\mathcal{N}(1675;77)$.

Now our statistical model is theoretically proved, we are going to consider how we can use it on a practical way to detect HVM-rootkits.

5.6 Statistical Detection

Our previous modelling results clearly demonstrate that our estimator significantly behaves different according to the presence (active) or absence of *BluePill*. We then are in the classical case depicted in Figure 12. To efficiently detect *BluePill*, it just suffice to build a simple hypothesis test. This approach has been thorughly defined in [43]. The two different hypotheses to be considered are the following:

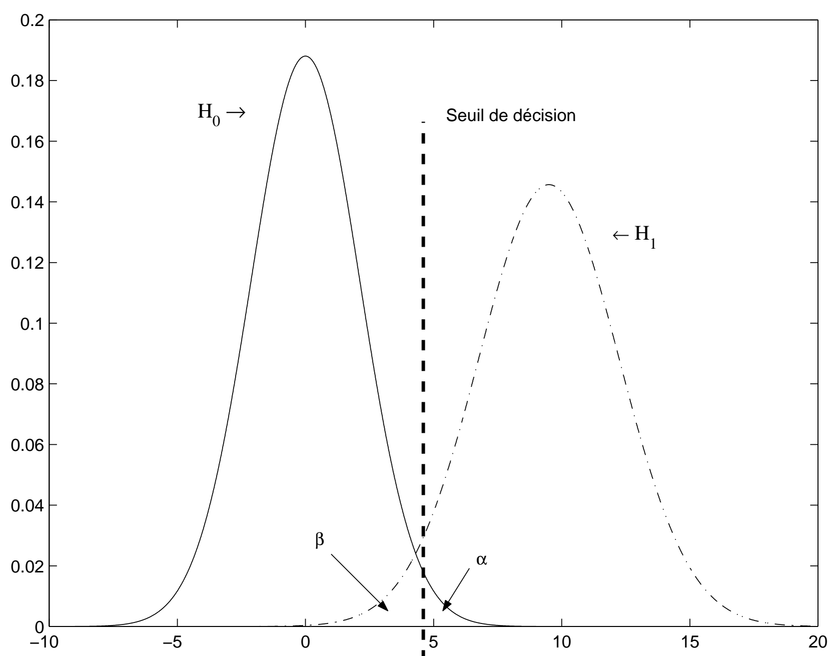

Fig. 12. Statistical Modelling of *BluePill* Detection

- The *Null hypothesis* $\mathcal{H}_0$: *BluePill* is not active (absent). Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_0(26;13)$.
- The *Alternative hypothesis* $\mathcal{H}_1$: *BluePill* is active. Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_1(339;38)$.

The type I error α (which consists to reject $\mathcal{H}_0$ while indeed it $\mathcal{H}_0$ is true) and the type II error (which consists in keeping $\mathcal{H}_0$ while it is a wrong hypothesis) are fixed according to the final detection efficiency we strive to achieve. Those error values then enables to fix a detection threshold and according to the relative value of our estimator with respect to this threshold, we can decide whether *BluePill* is active or not.

From a statistical point of view, this approach can very easily be generalized to the three hypotheses cases: *BluePill* is not active, *BluePill* active with no payload, *BluePill* active with a payload.

6 Future Work and Conclusion

The main conclusion of our work is that if no malware is really undetectable in practice [19,43], the contrary is also true: no antivirus can claim to detect every possible malware. This is in fact an endless issue. As any researcher in computer security should do, we must have a critical look on any such issue.

In fact, when considering the case of HVM rootkits, with time and reason, it was possible to determine the exact level of risk and to efficiently solve this critical issue. Taking profit of the rise of muti-core processors, the loop counter technique has been proved to be definitively efficient as detecting HVM-rootkit. In the same time we discovered that any HVM-rootkit is bound to add a significative execution time when active, and more critically when embedding a true payload.

It is obvioulsy possible to consider alternative time references to detect HVM-rootkits. In the case of single core processor, the GPU of any graphic card can play the role of the second core thus extending our approach. But it is also possible to easily prevent attacks with such rootkits. Security policy could ask for desactivating the virtualization capabilities at the BIOS level. Alternatively, we could install a prophylactic hypervisor to bar the subsequent installation of any malicious hypervisor.

We have shown that designing and writing HVM-rootkits requires a lot of dedicated, complex skills. The open information (documentation) is fortunately not very widely available. But what would happen if a *BluePill*-like code with a true, offensive payload was put in the wild? It is very likely that it would have a tremendous impact on the security of any virtualization-capable computer in the world. Indeed, at the present time, quite no efficient solution has been made available [27] by any AV company and/or processor manufacturers. It makes you wonder ...

References

1. Bochs : Highly portable open source ia-32 (x86) pc emulator. http://bochs.sourceforge.net/.
2. Input/output memory management unit. http://en.wikipedia.org/wiki/iommu.
3. ptrace(2) - linux man page. http://linux.die.net/man/2/ptrace.
4. Qemu : Open source processor emulator. http://bellard.org/qemu/.
5. Virtualpc. http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx.
6. Vmware. http://www.vmware.com/.
7. Vmware esx. http://www.vmware.com/fr/products/vi/esx/.
8. Xen. http://www.xen.org/.
9. Anonymous. Runtime process infection. phrack 59-0x08.
10. anonymous author. Building ptrace injecting shellcodes. *Phrack Magazine*, 12(59), Juillet 2002.
11. anonymous author. Runtime process infection. *Phrack Magazine*, 8(59), Juillet 2002.
12. Edgar Barbosa. Detecting bluepill. SyScan'07.
13. Nicolas Bareil. Playing with ptrace() for fun and profit. http://actes.sstic.org/SSTIC06/Playing_with_ptrace/SSTIC06-article-Bareil-Playing_with_ptrace.pdf.

14. Brian Carrier. Open source digital investigation tools. http://www.sleuthkit.org.
15. Casek. http://www.uberwall.org.
16. Advanced Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. 15 Secure Virtual Machine.
17. Advanced Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. 15.23 External Access Protection.
18. Samuel Dralet and François Gaspard. Corruption de la mémoire lors de l'exploitation. In *SSTIC 06*, 2006.
19. Eric Filiol et Sébastien Josse. A statitical model for undecidable viral detection. *In Eicar 2007 Special Issue, V. Broucek and P. Turner eds, Journal in Computer Virology, (3), 2, pp. 65 – 74*, 2007.
20. Eric Filiol. A formal model proposal for malware program stealth. *Virus Bulletin Conference Proceedings, Vienna*, 2007.
21. François Gaspard and Samuel Dralet. Technique anti-forensic sous linux : utilisation de la mémoire vive. *Misc*, (25), Mai 2005.
22. grugq. Remote exec. *Phrack Magazine*, 11(62), Juillet 2004.
23. The Grugq. The design and implementation of userland exec. Janvier 2004. http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-01/0004.html.
24. Intel. Intel 64 and ia-32 architectures software developer's manual. Chapter 19 introduction to virtual-machine extensions.
25. Joanna. Site web de bluepill. http://www.bluepillprojet.org.
26. Dornseif M. All your memory are belong to us. Cansecwest 2005.
27. Northsecuritylabs. Hypersight rootkit detector. http://www.northsecuritylabs.com.
28. pluf. Perverting unix processes. Mars 2006. http://7a69ezine.org/docs/7a69-PUP.txt.
29. Pluf and Ripe. Advanced antiforensics : Self. *Phrack Magazine*, 11(63), Aout 2005.
30. Samuel T.King Peter M.Chen Yi-Min Wang Chad Verbowski Helen J.Wang Jacob R.Lorch. Subvirt: Implementing malware with virtual machines.
31. Joanna Rutkowska. Subverting vista kernel for fun and profit. 2006. SyScan'06 & BlackHat Briefings 2006.
32. Joanna Rutkowska and Alexander Tereshkin. Isgameover() anyone ? 2007. BlackHat Briefings 2007.
33. Jan K. Rutkowski. Execution path analysis: finding kernel based rootkits. *Phrack Magazine*, 13(59), Juillet 2002.
34. Desnos Guihéry Salaün. Sanson the headman. *Rapport Interne Ifsic*, 2007.
35. Desnos Guihéry Salaün. Sanson the headman. Mars 2008. http://sanson.kernsh.org.
36. sk devik. Rootkit linux kernel /dev/kmem. http://packetstormsecurity.org/UNIX/penetration/rootkits/suckit2priv.tar
37. Stealth. Rootkit linux kernel lkm. http://packetstormsecurity.org/groups/teso/adore-ng-0.41.tgz.
38. The ERESI team. The eresi reverse engineering software interface. http://www.eresi-project.org.
39. Core Security Technologies. Coreimpact outil de test d'intrusion. http://www.coresecurity.com/content/core-impact-overview.
40. Tripwire. Configuration audit and control solutions. http://www.tripwire.com.
41. Microsoft Windows. Driver signing requirements for windows. http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx.
42. Michael Myers Stephen Youndt. An introduction to hardware-assisted virtual machine (hvm) rootkits. http://crucialsecurity.com.
43. Éric Filiol. Techniques virales avancées. *collection IRIS, Springer Verlag, France*, 2008.