Master research Internship

Internship report

TANSIVTx: Time-Accurate Network Simulation Interconnecting VMs with Hardware Virtualization Towards Stealth Analysis

Domain: Operating Systems - Cryptography and Security

Author:
Léo Cosseron

Supervisor:
Martin Quinson
Louis Rilling
Matthieu Simonin
Myriads - Inria/Irisa

Abstract: Security researchers confine malware samples in isolated and controlled environments called sandboxes, where they can be safely analyzed. However some malware analyze their environment to detect the presence of a sandbox and evade the analysis. In this work we focus on malware that are using timing analysis on the network to detect sandbox environments. The goal of the internship is to study an implementation of an analysis environment resistant to network fingerprinting. One key aspect of the approach is the use of hardware virtualization (Intel VT-x) to improve performance and robustness towards evasion techniques. This implementation will indeed extend the current prototype, TANSIV, which relies on QEMU's emulation mode. In this internship report, we present two different approaches to add hardware virtualization support to TANSIV.

Contents

1 Introduction

In order to study the behavior of malware samples, security specialists execute them in safe, isolated environments called sandboxes. In such environments, malware can be executed without any risk of harming the system. These sandboxes often rely on virtualization, which is a technology to create an execution environment to run a guest operating system or application in isolation from the host system.

Thanks to these sandboxes, malware can be safely executed and analyzed with many specialized tools. These tools are integrated with the sandbox and provide information about the malware interaction with its environment, like which files are read or written to by the malware. Moreover sandboxes and malware tools often work automatically, which enables to analyze malware at scale.

As malware authors want to hide their program's behavior, they develop many *evasion techniques* to detect the presence of a sandbox, and hide the malicious behavior accordingly. There are many classes of evasion techniques, such as detecting if there is any human-like behavior on the system, based on mouse movements or keyboard inputs. Our focus is on timing analysis techniques, that compare several time references to check for inconsistencies. These techniques rely on local time references, like the execution time of a loop. Sandboxes work around this issue by having consistent local time references, and thus staying stealth from this class of evasion techniques.

The goal of the internship is to design stealth sandboxes that are resistant to timing analysis evasion techniques. Such techniques can use the network as part of their time references. They can then compare the network performance of network message with a fingerprint of the target victim environment (network speed, location, …). For example, a malware can send a request to a known server, such as Google's DNS server, and time the delay before getting a response. It can then compare the results with a known latency distribution, and check for any inconsistency. As these techniques are easy to implement and the methods used to simulate the network in VMs are usually flawed, it is a serious concern.

Instead of writing a sandbox from scratch, we wish to complete other existing sandboxes in order to use their existing set of malware anlysis tools. Thus we want to develop an approach that will be portable and compatible with some existing sandboxes, that may work with different hypervisors and on different OSes (Microsoft Windows, Linux) …

Our current approach, TANSIV, works by connecting the network end-points of a sandbox with an accurate network simulator. TANSIV monitors the progression of time in the VM and the network simulator, which is different from the wall-clock execution time. This fined-grained control allows the simulator to accurately deliver the message on network end-points. This first approach is based on QEMU's emulation mode as it provides tools to accurately control the flow of time in the VMs. However the execution is slow due to emulation and can be easily detected by evasive malware, defeating the goal of stealthiness. Moreover sandboxes often choose hardware virtualization over software emulation as it provides better performance and stealthiness.

These issues with the current approach lead us to envision a new architecture for TANSIV. How can we design a run-time efficient sandbox with a realistic network, in order to stay stealth from malware?

In this internship, I will work on extending the TANSIV prototype by searching for a new approach to control the flow of time with hardware virtualization. Then I will evaluate this new approach and compare it to the existing emulation-based prototype.

In this report, we begin in section 2 by a bibliographic review focusing on the different techniques for virtualization, sandboxing techniques for malware analysis, and finally evasion techniques found

in malware, in particular timing-analysis based ones that TANSIV aims to tackle. Then in section 3 we go through the technical context of the internship. We continue by presenting the solutions developed during the internship in section 4. Next in section 5 we go through some technical details of the implementation. Lastly, in section 6 we present the evaluation of the internship work, before concluding in section 7 to discuss upcoming work.

2 Bibliography

In this bibliographic review, we begin by describing the different approaches for virtualization, in an effort to justify our choice for hardware virtualization. Then we go through sandboxing techniques for network analysis, including malware analysis tools. This section also focuses on network interconnection with the sandbox, which is the key problem of the internship. The last section discusses several evasion techniques that are present in malware, and in particular timing-analysis based ones that our project aims to stop. Finally in the conclusion we discuss the link with the internship work .

2.1 Virtualization techniques and tools

In this section, we define virtualization and associated concepts. Then we focus on a first virtualization technique called emulation. Finally we present another approach, hardware virtualization.

2.1.1 Virtualization concepts

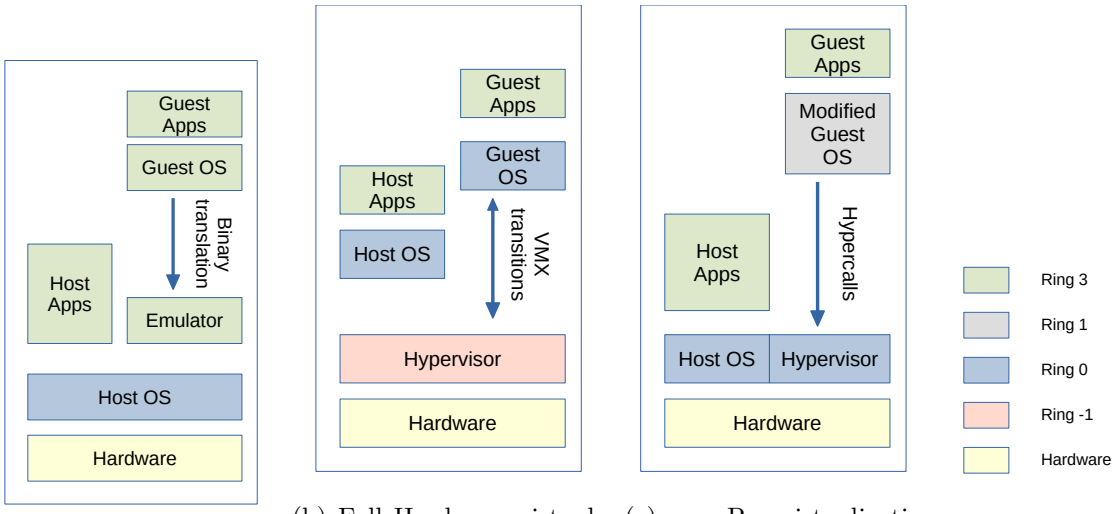
Virtualization is a technology to execute on a **host** system, in isolation, another operating system or application, called a **guest**. There are two distinct strategies to implement virtualization : Emulation and Hardware virtualization. Both approaches are detailed in the next sections. Emulation consists in running the guest system through a software, which translates the guest instructions into instructions for the host. With hardware virtualization, the guest system runs mostly natively on the host, under the supervision of a dedicated virtualization software.

To introduce the virtualization concepts we mostly use Popek and Goldberg definitions in [1]. A **Virtual machine monitor** (VMM) is a software that creates an execution environment to run other programs. This program must have three characteristics. First, a program executed on the VMM must behave as it if was run natively on a real machine. The only differences have to be caused by hardware limitations from the VMM, or timing issues. For instance, it is acceptable to see lower performance in a virtual machine if less memory has been allocated to it than what is available on the host, or if multiple virtual machines are running concurrently on the same system.

Secondly, a VMM has to be efficient, meaning that most of the program's instructions should be executed on the host processor.

Finally, the VMM has to fully control the guest resources, such as memory, the CPU or peripherals. It means that the guest can only use resources allocated to it, and can not allocate resources for itself.

According to Popek and Goldberg [1], *A **Virtual Machine** (VM) is the environment created by the virtual machine monitor.* Although emulators are not VMMs according to Popek and Goldberg, because they break the second condition, we add them to the scope of the definition. A virtual machine simulates the resources of a real computer, like the processor, the disk, or the network,



(a) Emulation (b) Full Hardware virtualization (Intel x86) (c) Paravirtualization (Xen)

Figure 1: Virtualization approaches

with the aim of executing guest operating systems or applications. Virtual machines are isolated from each other.

An **hypervisor** is a software component that is used to control the execution of VMs. The hypervisor intercepts hardware requests of the guest systems and redirects them to the adequate hardware resources. It manages the sharing of hardware resources between the host and guest systems. There are two main types of hypervisor. A type 1 (Native hypervisor) runs directly on the hardware, and thus control all hardware resources. Example of type 1 hypervisors are Hyper-V and KVM. A type 2 (Hosted hypervisor) runs as a process inside an existing OS. Example of type 2 hypervisors are VMware Workstation [2], VirtualBox [3] and QEMU [4].

As shown in figure 1, there are many approaches for virtualization, where the different components may run at different ring levels. Some privileged instructions can only be executed on lower ring levels. Rings are presented more in detail in section 2.1.3. In the next sections we detail both emulation (Figure 1a) and hardware virtualization (Figures 1b and 1c), and highlight their advantages and disadvantages regarding our project.

2.1.2 Emulation

A first approach for virtualization is emulation. Emulation consists in a piece of software, called an emulator, which provides an execution environment that acts like another system. Thus with emulation it is possible to execute an application, or even an operating system, on a host system with a different hardware architecture.

Unlike a hypervisor, which abstracts some components but execute nearly all the code of a guest operating system on the host processor natively, an emulator provide a complete abstraction of hardware to the guest operating system. An emulator replicates all the behavior of the hardware it emulates, even when the physical hardware is compatible with the guest operating system. With emulation, software replaces hardware.

Emulators are separated into several modules. The main part is the CPU module, which trans-

lates instructions from the guest OS into instructions for the host architecture. Moreover the CPU modules also save information about the current state of the CPU, such as register contents, in appropriate data structures. Many optimizations are done at this critical translation step, to achieve acceptable performance. For example, the guest and host registers may be directly mapped, and translated code sections are usually cached. The memory module handles all memory related operations, including the emulation of the MMU to translate logical addresses into physical addresses. Finally there are several modules which are dedicated to the emulation of I/O peripherals, such as mouse and keyboard, disk, sound card, network controller, …Instead of emulating the bus, which is not useful in a software only approach, as there are no physics constraints, each emulated device is independent. Usually they only share code for common operations such as interrupt handling.

BOCHS [5] and QEMU [4] are two examples of emulators. According to [6], emulators have many advantages that makes emulation an appealing approach for sandboxing malware samples. Indeed it allows the sandboxes to obtain a lot of precise information about the instructions and memory accesses done by a malware. As long as the host machine is powerful enough, it is possible to emulate any architecture with acceptable performance, which is interesting for malware written for less common architectures.

Another advantage is QEMU's icount mode which correlates the number of instructions with the time perceived inside the VM. This mode is useful in order to have deterministic executions and use record and replay, which is a technique used by many some sandboxes [7] explained in section 3.1. It can also be used to make comparable executions times of an application from the VM perspective on different machines. In TANSIV's emulation implementation, the icount mode is used as a virtual timer, by considering that the time corresponds to the number of instructions. Assuming a 1GHz single-core processor with one instruction/cycle, one instruction equals 1 ns. Instead of synchronizing QEMU's clock with the host system, the number of instructions is used as a high-precision timer which allows to stop the VM and send network messages between VMs at the best moment in an effort to have a realistic network.

The advantages of emulation make it appealing for sandboxes, such as [8] which is based on QEMU. The current prototype of TANSIV works with QEMU emulation mode, which enables the possibility to control the flow of time in VM with an high precision. Nevertheless it has many drawbacks, including the very slow execution. By using a timer linked to an external source, any malware can notice the execution of instructions is much slower than it should be, and thus deduce it is running in an emulator. Moreover emulation can be very resource intensive for the host system, and it is not always possible to accelerate some operations by using some of the host system hardware. Finally most sandboxes use hardware virtualization, for better speed and performance.

Due to these flaws, the current TANSIV implementation is not satisfying and encourages us to discard emulation and look upon other options for virtualization.

2.1.3 Hardware virtualization

A more efficient approach for virtualization is Hardware Virtualization. With this approach, the guest OS runs natively on the host machine, and its execution is managed by a hypervisor. Thus unlike emulation, the guest OS architecture must use an instruction set compatible with the host processor. For instance an x86_32 guest can be virtualized on x86_32 and x86_64 processors. Next we detail two types of hardware virtualization: Full Hardware Virtualization and Paravirtualization.

Full Hardware virtualization Full hardware virtualization refers to the possibility to run an unmodified guest system by presenting to it an abstract system, to hide the real hardware.

It is supported by many virtualization software, such as QEMU, Xen or KVM. The main advantage of hardware virtualization is the near native speed performance, thus it is widely used among sandboxes [9], [10].

On x86, the hardware privileges of the different OS component are determined by a hierarchy of protection rings (From 0 to 3). The most privileged is ring 0 where there the kernel runs, while applications run on ring 3. Ring 1-2 are sometimes used for some drivers. With the growth of virtualization, Intel and AMD introduced dedicated hardware support for virtualization by providing many new instructions. These instructions enable a special execution mode where it is possible to launch several guest kernels, which believe they have ring 0 rights on the system. These instructions only work on a new ring level, dubbed Ring -1, where the hypervisor runs.

Next we focus on how hardware virtualization is implemented on Intel x86, which is the architecture we work on. AMD uses similar instructions. Processors that support hardware virtualization have a special execution mode called VMX operation. The VMM runs in root VMX operation while the VMs run on VMX non-root operation.

In VMX root mode operation, the processor behaves as if it was outside VMX operation, but has access to several instructions to interact with VMs.

In VMX non-root operation, the processor has some limitations, and some instructions or events cause *VM-Exits* events where the VMM takes back control.

Each VM is referenced by a VM control structure (VMCS) which can be manipulated by the VMM with dedicated instructions (VMWRITE/VMREAD). There is a VMCS for each core the VM runs on. This structure contains the characteristics of the VM it is associated to, and is used to switch between VMX root and VMX non-root operation. For instance the state of the guest and host registers are stored in this data structure.

We depict in figure 2 how the VMM interacts with two different VMs. One core is allocated to the first VM, resulting in one VMCS structure in memory, while the second VM has one VMCS structure for each of its two allocated cores.

With this structure, the VMM runs on ring -1 while the VM OS runs on ring 0, and their applications can run on ring 3.

However, other architectures often work very differently to offer support for hardware acceleration. For instance ARM processors have a special execution, called Hyp mode, that is more privileged than Kernel mode and has its own set of features [11]. An ARM hypervisor, such as KVM/ARM, must at least run partially in this mode. It is very different from Intel's VT-x virtualization, because while VMX root mode and VMX non root mode can use the same processor's features, Hyp mode has a different set of available features than Kernel and User mode. Finally ARM supports virtualization for timers, a feature absent on x86.

Hardware virtualization offers much better performance than emulation as the guest operating system runs directly on the hardware. Moreover it is stealthier because there is no way of knowing the current VMX mode fo the guest OS, and execution times are more coherent than in emulation mode. For malware analysis tools it is also a better option. For instance DRAKVUF [10] opted for hardware virtualization because it helps hiding the monitoring and it can be used to trigger VM-Exit events in order to trap of the execution of malware code in the VM. We want to use hardware virtualization in TANSIV for all these reasons.

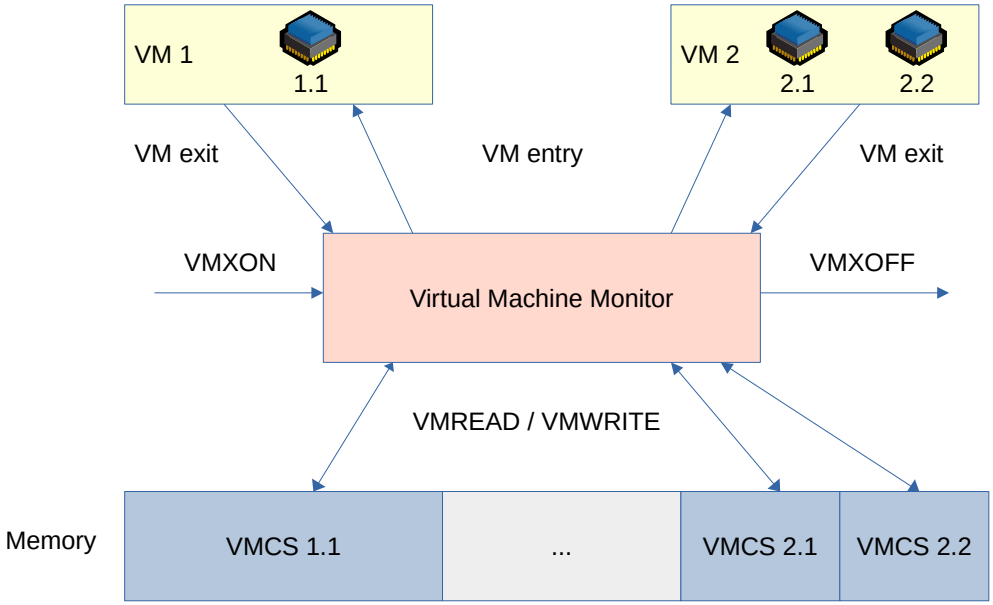


Figure 2: VMM and VMs interaction

Paravirtualization Paravirtualization is another approach for hardware virtualization that consists in providing a software interface to the guest OS that is different from the underlying hardware surface. With this approach the guest operating system must be modified so that it is able to interact with the hypervisor. The guest is fully aware that it is virtualized and issues commands to the hypervisor (*hypercalls*) instead of privileged instructions.

An example of a paravirtualized system is Xen [12]. It introduces a paravirtualized interface to handle memory management, the CPU and I/O devices.

Depending on the architecture support for a software-managed Translation Lookaside Buffer (TLB), a cache to accelerate the translation of virtual addresses into physical addresses, the implementation of memory virtualization is different. In x86, the TLB is hardware-managed, thus for ensuring performance page translation of the current address space must be in the hardware TLB. Hence Xen opted to let the guest OS manage the hardware page table, while monitoring and validating them to guarantee isolation. Xen also reserves some space in the TLB to avoid a complete TLB flush during switches between the VM and the VMM.

Regarding the CPU, the VMM must be more privileged than the VM. In 2003, Intel processors designers opted to run the hypervisor on ring 0 and the guest on ring 1. Thus some privileged instructions could not be handled by the guest in ring 1 and were handled by the VMM. However nowadays Intel's x86_64 CPUs use dedicated hardware virtualization instructions making the concept of rings less relevant in the context of virtualization. Privileged instructions are paravirtualized and go through Xen. Exceptions are dealt with a table containing an handler for each kind of exception. As system calls are really common, Xen implements a fast handler to avoid intercepting each system call.

I/O peripherals are not emulated, instead Xen implements an abstract interface to the guest OS for each device. Data is transferred by using dedicated data structures called asynchronous I/O rings. Moreover hardware interrupts are replaced by a more lightweight event system. This

approach makes it possible for the guest OS to effectively interact with I/O peripherals, while Xen is able to monitor the data transferred.

QEMU also supports paravirtualization, but only to offer efficient virtual peripheral devices, without the need to replace privileged instructions by hypercalls.

The main downside of paravirtualization is the necessity to modify the guest OS, which can be easily detected by evasive malware. Moreover nowadays nearly every processor supports Intel/AMD VT-x/AMD-V extensions, which improved the performance of full hardware virtualization. This is why we discarded this virtualization technique in our approach.

2.2 Sandboxing for malware analysis

2.2.1 Overview of sandboxes

Sandboxes are restricted, isolated and controlled environments which are used to executed another program or operating system safely. In this section we focus on sandboxes that are using virtualization to isolate applications.

In a sandbox, a malicious software has no access to any resource outside the sandbox and thus can not harm anything. Sandboxing is a very common approach to study malware samples. These sandboxes are enhanced with specific malware analysis tools, with the intention of understanding the behavior of malware pieces. As researchers may have to analyze thousands of samples every day, this process is automated.

Sandboxes have an allocated amount of system resources (CPU, disk, RAM), no access by default to the various peripherals available on the system, and network access is often restricted or highly monitored. Nevertheless as detailed below in 4.1.1, malware samples contain code that may hide their malicious behavior if a sandbox environment is detected. Thus sandboxes must be stealth enough so that it would be possible to study malware code.

Virtualization is used to implement sandboxes as it provides everything needed. Hypervisors can be modified to include analysis tools which can run from outside the VM, as we discuss next in 2.2.2.

Figure 3 shows the structure of a sandboxing environment. The hypervisor used to perform virtualization is completed with a sandboxing toolbox that interacts with the VMs. The possibilities for sandboxing tools are detailed in section 3.2, but for instance they include process monitoring. Moreover an human operator can interact with the sandbox by intending to collect execution traces (like system calls), or to perform debugging in order to better understand the behavior of the studied malware.

Most sandboxes, like [8], [9], [13] rely on virtual machines to work. In the remaining sections, we discuss about malware analysis tools which can be used in combination with sandboxes, and how network access must be implemented in sandboxes to ensure stealth.

2.2.2 Malware analysis tools

Malware analysis tools are used to understand the behavior of malware samples by executing them in a sandbox. Their goal is to understand what are the actions of malware pieces on their environment, like files read or created and interactions with the network. Their execution is monitored as well to understand how the malware work. This is done by analyzing the instructions and system calls. Everything of interest is logged and can be further analyzed to classify the malware and understanding what it does. Thus malware analysis tools must monitor all these information to

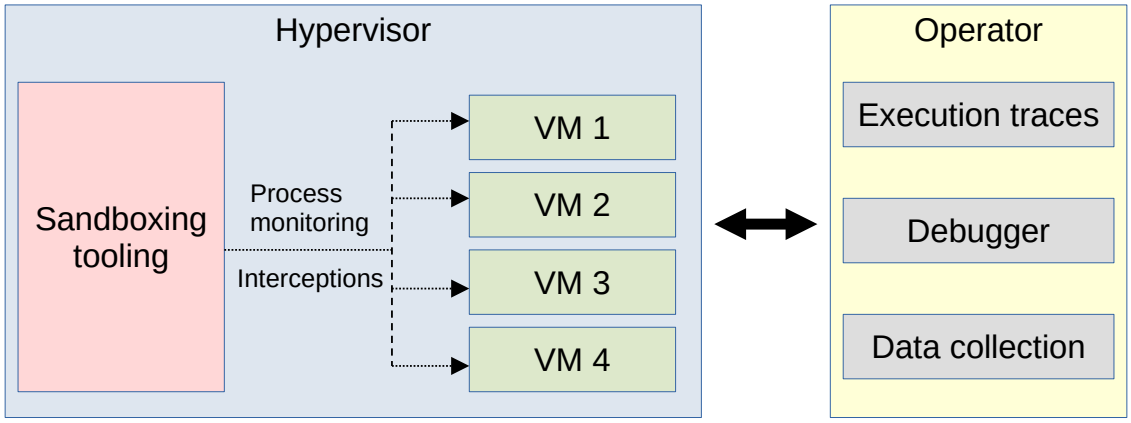


Figure 3: Sandbox structure

be relevant, while staying stealth to avoid evasive malware. In this section we review the different features supported by these tools

The most common feature is execution tracing, which consists in monitoring system calls. For example DRAKVUF [10], PANDA [7], and PyRebox [8] all support this feature. In particular, the DRAKVUF paper mentions that as a means to trace the execution of rootkits, which are malware running at the kernel level, regular system call monitoring is not enough because it only logs the interaction between the kernel and user-mode programs. To solve this problem they use another technique called breakpoint injection to trap kernel functions and monitor them as well.

Breakpoint injection, supported by DRAKVUF and PyRebox, is used to quietly add a breakpoint instruction anywhere in the memory during the VM's execution. Although it is usually used for debugging purposes, DRAKVUF go further by combining breakpoint injection with other techniques for better monitoring purposes and more stealth. For instance it uses this technique to hijack any process and start a malware sample, without leaving any code or trace on the system, which greatly improve stealthiness.

Another technique is tainting analysis, notably used by PANDA. A tainted data (which can be a register, RAM, a network packet, ...) is tracked during execution, and the sandboxing tool can link any tainted data with its label. In PANDA, which is based on QEMU's emulation mode, it is implemented during the translation step of emulation. During this step, QEMU translates guest code into its own internal representation, TCG, before translating it into host-compatible code. However QEMU can be configured to translate this TCG intermediate representation into LLVM code, which can then be executed by a LLVM JIT compiler. PANDA works by modifying this LLVM intermediate representation, making tainting analysis architecture independent.

Monitoring the file system is also a crucial feature of virtual machine introspection. Knowing which files are read, modified, created and deleted by a malware can improve the understanding of a malware's behavior. Most dynamic analysis tools implement it by modifying the disk emulator, but DRAKVUF proposes another approach thanks to its presence in the kernel space. Indeed in

Windows operating systems a file access allocates a specific object in the kernel heap, with many interesting information in headers. This allows DRAKVUF to monitor the filesystem, including information like file paths, which are not easily obtained with a conventional approach.

Snapshots are an interesting feature of VMs regarding dynamic malware analysis. Indeed most virtualization solutions offer a way to take a snapshot of a VM, including current CPU and memory state, and restore it later. Most sandboxes like DRAKVUF use it to efficiently clone VMs, but PANDA takes advantage of it to offer a record and replay feature. After taking a snapshot, PANDA records all inputs that can not be deduced (Data entering the CPU, hardware interruptions and memory operations from external devices) in a "Non-deterministic log". The information in this log are enough to replay the execution from the last snapshot in a deterministic manner, which is very useful for dynamic malware analysis, as regular debuggers can not backtrack the code execution.

Another set of software to perform malware analysis are instrumentation tools such as Pin [14] and DynamoRIO [15]. These tools work by inserting new code during the execution of a program, and observe its behavior. For instance, Pin takes an executable and some code that describe the instrumentation work as its parameters. At the beginning of a code sequence, Pin can instrument the code if required, and then generate executable code until the next branch thanks to a JIT compiler. This generated code and the result of the instrumentation are saved in a cache to improve performances. Instrumentation tools like Pin can be used to check the contents of memory, registers, or the control flow of a malware sample. Thus these tools are very useful in order to better understand the behavior of some malware samples.

Finally some sandboxes such as PyreBox and PANDA can be scripted by an operator for the purpose of performing a more sophisticated analysis. It is one of PANDA's core features, as it uses a plugin architecture to function. The user is able to call the plugins thanks to an API, and use them as wished in the script. It is more powerful than a debugger as features made for dynamic malware analysis, such as record and replay, can be used as well.

One of our goal is to make TANSIV compatible with a dynamic malware analysis tool that supports hardware virtualization, such as DRAKVUF.

2.2.3 Interconnecting sandboxes with network

Network and virtual machines In order to achieve network fingerprint resilience, the way the network is simulated is critical. Indeed if network performance are not realistic due to a bad simulation, malware can easily detect it by checking the latency of its communication with a server, for instance with a DNS request. Our objective is to simulate a network with realistic performance.

In most sandboxes, the network implementation is overlooked and leaves many evasion possibilities for malware samples. [16] reveals that many sandboxes have a predictable IP address with the default configuration of virtualization software. This is also true for MAC addresses, which always begin with the same bytes for each VM based on the same virtualization solution. This survey also mentions VMs that do not provide any internet access at all, or VMs with an abnormally high speed internet connection.

Turret [17] introduces an approach where they simulate the network using ns-3 [18], aiming to control all network communications, with the goal of finding performance attacks on distributed systems. ns-3 is a discrete-event network simulator, which provides many network models and a simulation engine. It abstracts the different components of a network system, and simulates the network events in chronological order. ns-3 also has support to emulate network devices, and interact with a real network. In this mode, the clock of the simulator and the clock of the host are

synchronized, which enables the possibility to execute real-time applications.

ns-3 has support to let unmodified applications use a simulated network, thanks to a feature called Direct Code Execution (DCE) [19], [20]. This is especially useful to perform reproducible experiments. It works with a first layer to virtualize stacks, heaps and memory to manage allocated ressources, because it has a single-process design. A second layer has the different network protocols found in the Linux Kernel and is used to execute the kernel network stack within the simulator. Finally a third layer reimplements sockets to be compatible with the previous layer, in order to be POSIX compliant and usable with most applications. Unlike emulation mode, time can be slowed down by DCE in order to give more time to the simulator to perform its computations. It is particularly useful because thanks to this feature it is possible to simulate very fast networks, even if the simulator is not fast enough to do it synchronized with the host clock.

Despite all these features available in ns-3, we decided to use the Simgrid network simulator [21] instead, notably because it works much better than ns-3 on large-scale simulated networks. As ns-3, Simgrid is a discrete-event network simulator.

Although our goals are different, the approach we consider for TANSIV is similar. Turret uses KVM VMs for its experiments, and each VM is mapped to an end node on ns-3, which emulates the network. ns-3 has been modified to add new instructions, such as `save/load` to backup or restore the state of the simulation, and `freeze/resume` to stop or resume the execution of the simulation. Indeed these instructions are needed because the objective is to perform actions on network messages. These actions require to precisely control the flow of time as the VM execution might be frozen and network performance should be realistic. The VMs and the network simulator must have the same perception of elapsed time. Unfortunately how it is achieved is not detailed, as the authors only claim that the host clock or an external time source can be used to resynchronize the VMs and ns-3.

TANSIV uses an approach simular to Turret. As in [22], we interconnect QEMU with the SimGrid simulator, but currently time control only works in emulation mode. The main difference between Turret and TANSIV architecture is that Turret aims to synchronize the simulator clock with the host clock, thanks to the feature explained above, while TANSIV chooses to synchronize the clocks of the VMs with the simulator clock (which is independent from the host clock). The advantage of Turret's approach is that it is not needed to modify the clocks of the VMs (with exceptions, for instance when loading a VM snapshot). TANSIV has the advantage of being able to simulate networks faster than what would have been possible by taking the host clock as its reference. As shown in the next subsection, one of the main goals of the internship is to find solutions to accurately control the flow of time in the VMs and the simulator.

Controlling time Precise time manipulation in the VMs and the network simulator is essential to achieve realistic network performance, and stay stealth from malware which are timing their communications on the network.

Indeed time can be manipulated by sandboxes for various reasons : speeding up the execution when no code of interest is running, stopping execution to extract some data or take a snapshot, ... Despite that we wish to fully control how the time runs regarding the network. Thus during the execution of a VM in a sandboxing environment, there may several clocks which are very likely to be independent.

First the **virtual clock**, which serves as the reference clock for the time perceived within the VM. It can be manipulated by sandboxes for the purpose of analyzing the studied malware more

efficiently. Each VM has its own virtual clock. Then the **simulator clock**, which is the reference clock for the time perceived within the network simulator we wish to add on top of the hypervisor. This abstract clock is used to deliver network messages, including between different VM that are monitored at the same time. Finally there is the **host clock**, which is the real world time as perceived by an operator.

The goal of TANSIV is to precisely control the flow of time between the different network end points, in order to accurately deliver network messages and simulate a real network. This requires accurate timers for the VMs, as network messages are delivered to the simulator with their exact send date, which is used to compute the moment to deliver the message on the other end.

During the bibliographic review we have not found any article that tackles that particular issue. The TANSIV project aims to solve this problem by accurately controlling the time in the monitored VMs.

2.3 Evasion techniques

In this section, we go through the various techniques malware samples can use to evade analysis by a sandbox or a debugger.

2.3.1 Evasive code

There are two main categories of evasion techniques. **Automated dynamic analysis evasion** are all the techniques to detect if the malware is executed in a sandbox. A sandbox environment is very likely a sign that the malware is being studied either automatically or by a security researcher. The second category is **Manual dynamic analysis evasion**, which refers to all the techniques to evade manual analysis software such as debuggers.

Sandbox evasion Security researchers use sandbox environments geared towards the study of malware samples. With this approach, malware can not perform any damage and researchers have full control of the execution environment. In anticipation of reverse engineering attempts, malware authors implement many techniques and heuristics to detect the presence of a sandbox environment, and hide their malicious behavior accordingly. As security researchers rely on automatic analysis with sandboxes to study thousands of malware pieces, understanding how sandbox evasion techniques work is essential to implement countermeasures.

A first class of techniques detailed in [6], [23] is stalling code. Most sandboxes timeout after a few minutes if no progress has been done in the malware analysis, and assume the sample does not work or is not dangerous. To avoid delaying the execution time of malware on real victim systems, some samples use stalling loops with systems calls that are monitored by sandboxes. These loops execute in milliseconds on a real system, but might take minutes if an analysis tool monitors and logs each call to this system call. A first way to implement stalling code is to use a delay function such as `Sleep`. However most sandboxes have sleep skipping methods to accelerate execution. But if implemented incorrectly sleep skipping can be detected by malware too, by checking the coherency of stalling loops execution time. Section 2.3.2 details evasion techniques based on time.

[24] presents an interesting way to detect stalling code. Their tool uses various heuristics to detect the presence of stalling code, then it builds a control flow graph of the suspicious code section. Next an algorithm is used to detect stalling loops in the graph. Finally the code is patched to skip the stalling code sections.

Environment checks are also very common, because by default a virtualization software leaves many traces [16], [23]. Malware can verify if any suspicious program is installed or running. A system with a very low number of installed programs is a sign of a fake system as well. A blacklist with keywords like "virtual" is used to check many names that might have been modified by virtualization: peripheral names (disk, network card, sound card, ...), hostname, or even the malware's own filename. Checks are also done at the network level, as for instance virtualization tools use by default MAC addresses with the same first few bytes. A GeoIP service can be used as well to check the current location and the IP address of the VM. The IP address may reveal to which organization the network belongs to.

The hardware configuration can be reviewed by malware as well [23]. Often VMs are configured with only 1 core to save resources on the host machine, but malware can fingerprint the number of cores, and 1 is very likely to correspond to a VM. Other CPU characteristics can be checked as well, such as the core's temperatures, which may not work in a VM. A low capacity for the disk can also hint towards the presence of a sandbox. No internet access is also suspicious. But a network card with an unusual speed can be the sign of a virtualized peripheral.

Finally user-interaction checks are common in sandboxes [16]. The absence of a windowing system, no mouse movement or keystrokes are characteristic of automated sandboxes. Sandboxes must emulate the behavior of an human user.

It is essential for sandboxing environment to detect most of these techniques, as most malware always embed a few of them.

Anti-debugger For some interesting malware, such as the ones that evade automated sandboxing analysis, security specialists can perform a malware analysis by using debuggers. To evade these analysis as well, malware use various approaches to complicate the manual analysis, such as debugger evasion.

The first set of techniques, presented in [16] attempt to detect the presence of a debugger, and execute the malicious code accordingly. Fingerprinting the execution environment is the most common tactic. In Windows operating systems, each process has a Process Environment Block (PEB) which can contain sections which are a sign of a debugger's presence, such as a *BeingDebugged* field. Malware can also check for the presence of breakpoints. On the first hand, software breakpoints work with the `0xCC` opcode, which can be searched for by evasive programs. On the other hand, hardware breakpoints work with a dedicated CPU register, for instance "DR" registers on x86. These registers can be checked to verify if any hardware breakpoint is present. The malware can also look for system artifacts to look for traces left by the debugger, like registry entries, some file names, or the parent process, which should be `explorer.exe` if the malware is executed as intended.

Evasive code can also use traps, which are specific instructions with a different behavior when executed by a debugger. For instance, the `int 2h` instruction launches an exception without a debugger, that a malware handles with a try-catch. But if a debugger is present it handles the exception itself, and that can be detected by malware. These traps are easy to circumvent by ignoring these instructions.

Some malware are configured to target a specific victim system, and do not work on any other, including a sandbox. Usually these malware have their payload encrypted, and the key is derived from attributes specific to the target system. Some malware even use more sophisticated approaches to complicate the reverse engineering of the key, for instance with the use of AI-powered keying.

This techniques consists in encrypting the malware with an encryption key. This key is generated by the neural network only if it determines that the execution environment is not a sandbox. As neural networks are too complex for reverse tools and humans, this is a very effective way to craft an evasive malware.

Control flow manipulation is another way to evade debuggers, by using Windows' control flow mechanisms. Thread hiding consists in preventing the events from reaching the debugger. Multi-threading can be considered as well, by executing the malicious code on only one thread which may not be monitored by the debugger. Another trick is self-debugging, as a process can only be attached to one debugger.

Finally, other techniques include exploiting known debuggers vulnerabilities to crash them for instance, lockout evasion which consists in blocking all mouse and keyboard inputs while the malware is running, or fileless malware which are loaded through the network and can not be analyzed by regular debuggers.

To conclude this section there is a big variety of evasion techniques. [16] claim over 70% of malware use at least one of them, thus debugger and execution environments must be configured with that in mind to avoid evasion.

2.3.2 Timing analysis evasion techniques

For the TANSIV project, our focus is on evasion techniques that rely on timing analysis. Indeed our goal is to make sandboxes stealth towards timing-based techniques that use the network as a time reference.

[25] mentions timing-based measures that can detect inconsistencies, which hint towards the presence of a sandbox. Indeed malware samples check time to detect fast-forwarding, which is often employed by sandboxes to speed up the execution and bypass stalling code.

There are many time sources a malware can use as a reference. The Windows API offers several functions that can be used as a timer, such as `GetTickCount`. External time sources can be used as well, such as requests to an NTP server or with a custom protocol with a Command and Control server. Another timers that can be very accurate are loop counters. It consists in running a thread with a loop that increments a counter permanently. Then this counter can be used as a timestamp by the malware.

Moreover these timing sources can be used by malware to check for any timing discrepancies, using one of the techniques detailed in sections 4.1 and 4.2. Thus access to timers is at the core of most evasion techniques.

The authors of [25] propose a solution to fast-forward the execution of time. To do so they use a dedicated data structure that accumulates the time quantity elapsed during the execution. The execution is paused during this step to ensure precision. Each time a delay function is called by the studied program, like `Sleep`, the corresponding time quantity is added to the accumulator, and the execution is fast-forwarded. Then when the malware queries one of Windows functions that can be used as a timer, the sandbox fakes the result by returning a value based on the accumulator.

They also encountered the case of malware using two different time sources used to check their consistency. This is countered by modifying the result of the second query with the first one added to a value proportional to the elapsed time in the VM.

However there are still many unmonitored time sources, like loop counters and network-based timers. As for evasion techniques, there will always be a way to get a timer that the sandbox will not be able to monitor.

2.4 Conclusion

In this bibliographic review, we have first seen the differences between full emulation, hardware virtualization and paravirtualization. Then we have seen an overview of sandboxes and associated analysis tools for malware analysis. Finally we reviewed many evasion techniques used by malware to evade both dynamic and manual analysis.

In the first section we reviewed different approaches for virtualization that can be used as a sandbox. Emulation-based sandboxes are powerful because we control everything in the execution environment, including the CPU state, memory and executed instructions. Moreover QEMU offers the possibility to use a virtual timer that corresponds to the number of executed instructions, which allows to control the moment a VM can be interrupted at the instruction level. The current TANSIV prototype is based on QEMU emulation mode to make use of this timer. However emulation performance is significantly worse than on a real system, which is not convenient if an human analyst works on the sandbox and is also easily noticeable from the VM by exploiting timing inconsistencies.

Hardware virtualization is thus a viable alternative, because it offers near-native speed performance, solving both problems. In return as the VM runs directly on the hardware level instead of the software level, it can be trickier for analysis tools to have good monitoring options, and controlling the flow of the time in the guest system becomes a challenge. Paravirtualization, although it offers even better performance, is not an option as it requires to modify the guest OS, which defeats the discretion of virtualization. As the TANSIV project aims to accurately simulating the network it requires a precise control of the time flow in the VM to accurately deliver network messages. Addressing this challenge is the main goal of the internship.

Sandboxes for malware analysis are virtualization based. Virtualization offers many benefits such as the possibility to easily clone VMs and great scaling capacities. Malware analysis tools are able to monitor the actions of the analyzed malware pieces, and understand their behavior. For example knowing which files a malware manipulates during its execution is something security analysts are looking for. However the network aspect of the sandboxes is often overlooked, usually just contenting to emulate the network and a few services, without paying attention to the realism of message delivery timing.

Evasion strategies are evolving very fast, to which sandboxes respond by adding more countermeasures, which can be evaded as well. In particular, techniques that fingerprint the network performance are a serious concern the TANSIV project seeks to mitigate. The malware just have to use timers to check if the network performance are following a known network latency distribution. Our goal is to design a sandbox which is resistant to these attacks by accurately simulating the network.

In the next sections, I will assess the practical feasibility of extending TANSIV to TANSIVTx by using hardware virtualization, by presenting and evaluating solutions to have a precise control of time in Virtual Machines.

3 Technical context

3.1 From TANSIV …

3.1.1 Presentation

The goal of the TANSIV project is to execute malware samples to study in virtual machines, while ensuring these samples do not evade the analysis by checking the environment. TANSIV focuses on malware that use data from network interactions in their evasion techniques. Indeed, it is conceivable that a malware could compare the performance of a communication with a remote server with a known latency distribution to take its evasion decision.

To achieve that goal, TANSIV works by interconnecting the network end-points of virtual machines with an accurate network simulator. The simulator coordinates the flow of time in the virtual machines and delivers the network messages. Before my internship, the current approach used the QEMU emulator for virtualization and Simgrid as the network simulator.

TANSIV itself is divided into two main components:

- TANSIV-coord: It coordinates the execution of VMs and relies on Simgrid to simulate the network.
- TANSIV-client: A library that is loaded from the QEMU process and allows for interacting with the coordinator. Its main tasks consist in interrupting the network traffic (in and out), making requests to the coordinator, and accurately interrupt the VM by arming a timer.

3.1.2 Deadlines

TANSIV works with the concept of deadlines:

1. TANSIV-coord schedules the next deadline by choosing the minimum between the next network message delivery date, if there are any, and the minimum latency on the network.
2. All virtual machines can run until they are interrupted by a TANSIV-client's specific timer when the deadline is reached.
3. During a deadline, TANSIV-coord delivers all messages that should be delivered at the deadline timestamp. Meanwhile, messages intercepted by TANSIV-client during the last execution window are delivered to Simgrid through TANSIV-coord which accurately computes their delivery dates according to the network configuration.

Between deadlines, the virtual machines operate as usual. However all networks messages issued by a VM are intercepted at the network backend level and stored by TANSIV-client, where they will be timestamped and stored, before being sent to Simgrid at the next deadline.

Simgrid computes the next delivery date according to the message metadata (sender, receiver, size, …), the network configuration (topology, bandwidth, latency, …), and the current network state.

3.1.3 Synchronization of virtual and simulation clocks

One of the challenges TANSIV is trying to address is the synchronization of the virtual machine clocks with the simulation clock.

On the first hand, Simgrid is a discrete-event simulator, which means that its internal simulation clock progresses by jumping to the time of the next event.

On the other hand the clock of a virtual machine progresses overtime regardless of what is being executed.

With TANSIV's deadline mechanism, the deadlines timestamps correspond to Simgrid's events. The goal is to synchronize Simgrid with the virtual machines at these events, as there might be network messages to send.

While the virtual machines must be synchronized with the simulator, thanks to deadlines, they do not have to be synchronized between each other outside of deadlines. A virtual machine does not rely on others during an execution time slot as a message can only be received at a deadline. Moreover the next deadline can be scheduled only when all the virtual machines reach the current deadline, to ensure all messages that were sent during the last time slot have been received by the simulator.

For example, in a setup with two virtual machines, if the first virtual machine is stopped by a remote `gdb` to perform debugging at the middle of a time slot, the second virtual machine can still be executed until the next deadline is reached. At this deadline, the second virtual machine and TANSIV will wait for the first one. When the use of `gdb` is over, the first VM can resume its execution until it reaches the deadline as well. Thus no network messages are lost during this process, and they will all be delivered accurately, regardless of the time (from the point of view of the host clock) a VM takes to reach a deadline.

TANSIV uses a special execution mode from QEMU, icount, that only works with emulation and is single-threaded only. In icount mode, QEMU counts the number of instructions executed. This counter works as a virtual clock which can be used by QEMU's timers subsystem. TANSIV takes advantage of this feature which enables the possibility to fire a timer that will expire after a precise amount of instructions have been executed.

During the handling of a deadline, once the date of the next event/deadline has been computed by Simgrid, each TANSIV-client will schedule a timer to the next deadline. The conversion between the simulator clock and the virtual clock in icount mode is straightforward because one instruction equals to 1 ns.

Another advantage of this icount mode is that the flow of time is uncorrelated to the real time. Thus if the VM workload is low, QEMU will be able to execute instructions faster, and the virtual time may progress faster than real time.

3.1.4 Example

Figure 4 shows all the components of a TANSIV setup, and how they interact between them:

To better understand how TANSIV works, let's assume that the network has been configured with a latency of 20ms between the VMs, and that the TANSIV simulation starts at D1=0ms.

1. As there are no messages to deliver, the next deadline D2 will be equal to 20ms. If VM1 sends a message for VM2 at T1=10ms, it will intercepted by the TANSIV-client, timestamped, and added to the queue of messages.

2. Once D2=20ms is reached, the message will be transferred to TANSIV-coord, which then send the data of the message to Simgrid. Simgrid will answer that the message must be delivered at T2=30ms to VM2. As there are no other messages, the next deadline D3 will be scheduled at T2=30ms.

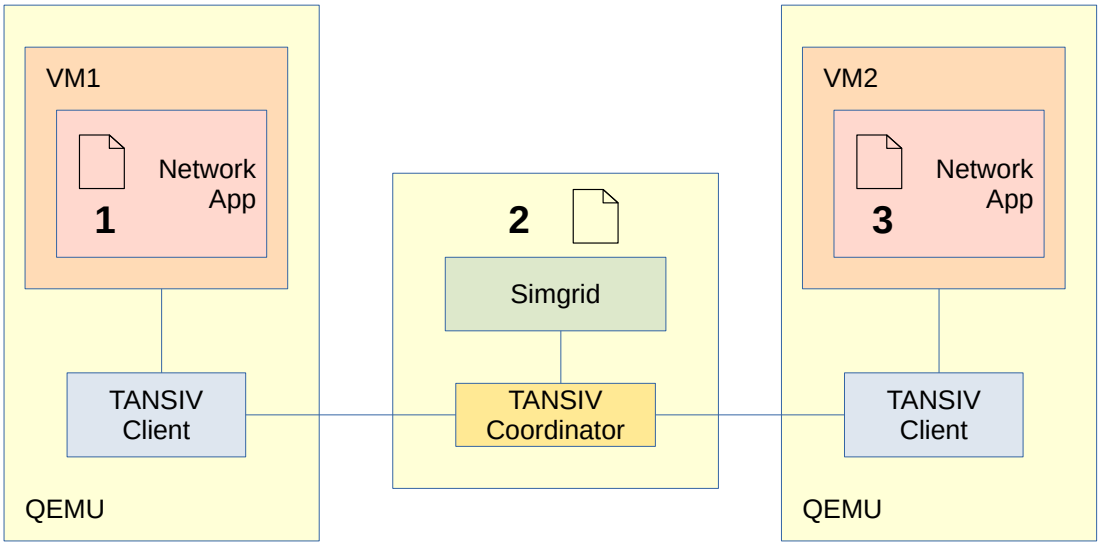


Figure 4: Transfer of a network packet between two VMs with TANSIV

3. The execution will resume until D3. When D3 is reached, the message will be delivered to VM2 by TANSIV-client.

3.2 ...To TANSIVTx

The goal of the internship is to make TANSIV work with hardware virtualization. However many mechanisms involved here do not work anymore when QEMU is using KVM to work in hardware virtualization.

First there is no equivalent to the `icount` mode in hardware virtualization, and thus instructions can not be used anymore to measure time. Moreover QEMU timers work differently, by using regular Linux APIs such as `clockgettime` as their time source, which has a significant overhead. Moreover the status of timers are only checked in QEMU's main loop, which is executed when there is time to, causing another overhead. Thus one of the problem to tackle during this internship was to find another solution to accurately interrupt virtual machines running with hardware virtualization.

However TANSIVTx will not change many core elements of TANSIV, such as the TANSIV's protocol or the concept of deadlines.

3.2.1 KVM

To move from TANSIV to TANSIVTx, we will use the KVM [26] hypervisor with QEMU. On Linux, QEMU uses KVM as its backend for hardware virtualization. The purpose of this section is to present how KVM works as it is important in order to understand the modifications that will be required in order to add hardware virtualization support to TANSIV.

KVM works by opening a device node, which has its own memory, isolated from userspace. Userspace can interact with this device node with a series of `ioctls`. An `ioctl` is a system call to communicate with a device, such a KVM interface. They use C structs as their arguments, and the kernel may return a struct as well.

In the case of KVM, `ioctls` are used to create a virtual machine, create a new VCPU, allocate memory for a virtual machine, and so on.

One of the most important `ioctls` is `KVM_RUN`. This `ioctl` is used to run a guest CPU, in our case by using Intel's hardware virtualization system, VT-x. Indeed with KVM, userspace (QEMU) is expected to create a dedicated thread for each VCPU that will execute `KVM_RUN`.

Moreover `KVM_RUN` works with a data structure that is shared between QEMU and KVM, that contains information about the current state of the VM. For instance it contains the exit reason if `KVM_RUN` terminates with an error, to let QEMU handle it and resume the VCPU execution.

Indeed KVM has two ways to handle VM-exits:

- "Light VM-Exits", are handled exclusively in Kernel space. Most VM-Exits are handled this way because they require minor implication from the hypervisor. These VM-Exits are the fastest to proceed.

- "Heavy VM-Exits", are handled by kernelspace and userspace. In the case of a QEMU/KVM setup, when this kind of VM-Exit happens, the `KVM_RUN` ioctl that QEMU used to run the VCPU thread will return with a special exit status. Then QEMU can check a value in a data structure shared with KVM to check the VM-Exit reason, and call the appropriate handler, before resuming the VCPU with another `KVM_RUN`.

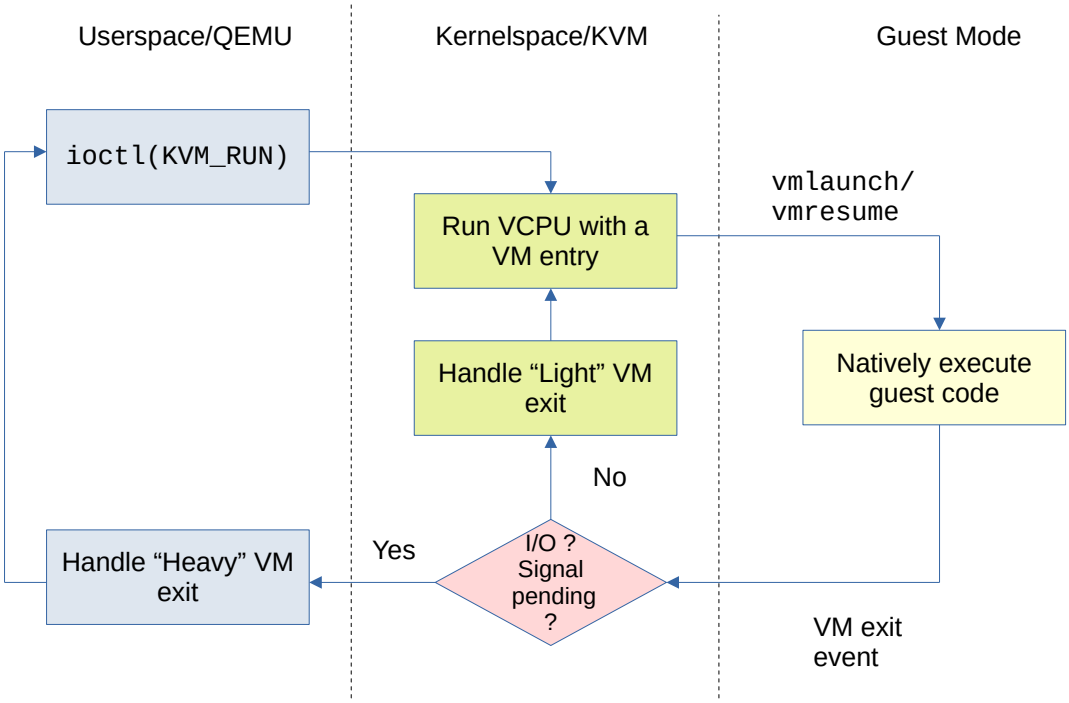


Figure 5: KVM guest execution loop, from [26]

Figure 5 shows the execution loop of a guest CPU with KVM. After a `KVM_RUN`, KVM will start/resume the execution of the VCPU with a `vmlaunch/vmresume` instruction. Next the hardware will natively run code for this VCPU, until a VM-Exit event happens, as explained back in section 2.1.3. Then by checking the VM-Exit reason written by the processor in the VMCS of the VCPU, KVM will either handle it entirely in kernelspace ("Light" VM-Exit), or exit to userspace if it is required, such as when the VM-Exit is related to I/O operations, or if a signal is pending for the VCPU ("Heavy" VM-Exit). In this case QEMU handles the VM-Exit itself, and resumes the loop by calling `KVM_RUN`.

To summarize, the most important concepts with KVM are:

1. Each VCPU is running in a dedicated thread
2. The execution loop of a guest CPU consists in calling `KVM_RUN` from QEMU, then enter VMX non-root mode from KVM, and handle the VM-Exit either in KVM or in QEMU

Thus in the rest of this report, interrupting a virtual machine running with KVM means interrupting all the VCPUs of a specific virtual machine.

3.3 Time management on Linux (Intel x86_64)

3.3.1 TSC

The TSC (Time-stamp counter) is an hardware counter/register available on x86_64 CPUs. Although it was introduced in the Pentium Intel processor, it became a reliable clock source only during the past few years due to many issues. First in order to be used as a clock source, a processor must support a "Constant TSC" mode. When it is present, the TSC will increase at the rate of the processor's nominal frequency, which is constant. Before that feature was added, the TSC was increasing at the same rate as the processor's frequency, which was variable over time. There were also issues to synchronize the TSC value between each core, as the TSC is not a shared register. This is still an issue on AMD processors. Finally some processors are known to have a buggy implementation of TSC, which cannot be considered reliable.

On recent Intel processors most of these issues are fixed and the TSC can be used as a clock source. As it is the most accurate time source, I will use timers based on the TSC to replace QEMU's timers.

In a KVM guest, the TSC works differently. It is based on the host TSC, but adjusted with an offset and a scaling ratio. By default, the TSC frequency of a KVM guest is the same than the host frequency, however it can be adjusted, within the limits of the hardware. Thus when the guest attempts to read its TSC with the `rdtsc` or the `rdmsr` instructions, a VM-Exit will be issued by the hardware, as the hypervisor must convert the host TSC to the guest TSC.

3.3.2 Timers

Along with the TSC, Intel provides a mode for LAPIC (Local Advanced Programmable Interrupt Controller) timers called TSC-Deadline. LAPIC timers can be used to send interrupts between the different cores when they expire. This timer works by writing a value in a register. When the TSC value becomes greater or equal than the value stored in this register, a timer IRQ is issued. This is the highest-resolution timer available on x86_64 architecture.

All timers available on a system, including TSC-Deadline, are registered in a list, along with their various features and their accuracy, determined by a hard-coded score value. Then these timers are used by the `hrtimer` Linux API, which chooses the best timer available. When available the TSC-deadline timer is the most likely to be chosen, and thus is the timer with the highest precision available on Intel systems.

Another interesting timer is the VMX preemption timer, available on Intel x86_64 processors. The VMX timer is fired by loading a value in the adequate VMCS field of a VCPU. When the processor is in VMX non-root mode operation, the timer counts down at a rate proportional to the TSC. When it reaches 0, a VM-Exit is triggered, with a reason to indicate that the VM-Exit was caused by the preemption timer.

3.3.3 Other time sources

By default, KVM clock do not use the TSC as their time source, but instead rely on a paravirtualized clock called "KVM-clock". This clock allows the guest to access the host time, which is often something wished in virtual machines. This clock is based on the host TSC and has the advantage of being easily modifiable by the host, which is why as explained later I will use this clock for the virtual machines. However as any paravirtualized component, the virtual machines are fully aware that it comes from a hypervisor, which means that eventually a switch to another time source will be required to be more stealth.

The RTC is an hardware clock that works on battery and uses a crystal oscillator to keep track of time, even when the system is not powered on. On Linux the RTC is not used as a time source/clock, but only has a mean to initialize other time sources during boot, which are more accurate.

The ACPI Power Management timer is a clock source integrated in ACPI boards, which all modern computes are equipped with. However due to bad hardware implementations and the relatively low precision, this time source is not an option for TANSIV.

The PIT (Programmable Interrupt Timer) was an hardware timer that released a signal after a specific value has been reached. It used a 1.193182 MHz clock signal and was ultimately replaced by LAPIC, which uses more accurate time sources, such as the TSC.

The HPET is another programmable timer introduced in 2005 that used a clock with a frequency larger than 10 MHz, providing much better precision than the PIT. However it was gradually replaced by the TSC, which offers better precision and a lower overhead. Moreover on recent Intel processors the Linux kernel disables HPET due to instability issues.

Finally NTP (Network Time Protocol) is a protocol to synchronize a computer's clock with a remote, reliable time source. As it is only used for synchronization, it is not usable for TANSIV. Moreover we do not want the guest to use NTP, as we want to control the flow of time in the virtual machines.

4 Approach

In this section we present the different approaches developed during the internship to add hardware virtualization support to TANSIV. To begin, we go through an overview of the problems introduced by the switch from emulation to hardware virtualization, then we present a first approach based on signals, and a second approach that uses the VMX preemption timer.

4.1 Overview

To begin, we present the challenges brought by hardware virtualization on TANSIV, that our solutions will need to address. First there is the problem of accurate interruption of virtual machines, then the issue of the flow of time in virtual machines, and finally the handling of inaccuracies.

4.1.1 Accurate interruption of virtual machines

I worked on finding a solution to be able to interrupt a virtual machine at an exact moment. It is crucial for TANSIV, because if the VMs are stopped too late, some messages will have to be delivered late, which ruins TANSIV's purpose to be accurate, and can even change the causality, because a message could have been different or not sent at all if the first message would have been received on time. To achieve this goal, I used the Linux kernel's timers, which have better precision than QEMU's timers. The objective is to use these timers to wait until the next deadline and interrupt Virtual Machines when this deadline is reached, as explained in section 3.1.2.

To get the most precise timers available, I wrote a Kernel module which uses Linux's `hrtimer` API, which themselves are based on the TSC-Deadline. The technical details of this kernel module are presented in section 5. This kernel module communicates with QEMU in order to configure the timer and be notify it when a timer expires. There are two things that must be done when the timer is expired:

1. Stop all the VCPUs of the VM associated with the timer
2. Notify TANSIV that the deadline has been reached

Note that ideally VCPUs must be stopped immediately after the expiration of the timer, and the VCPUs should resume immediately after the timer has been started.

To be precise, accurately interrupting a virtual machine means triggering a VM-Exit event (See section 2.1.3 and 2 for an explanation of VM-Exits) from outside the VM, exactly when the VM virtual clock reaches the deadline.

This cannot be done from inside the VM (With a `vmcall` instruction for instance) because it would be easily noticeable by an evasive malware. The two solutions presented later in this section use different means to trigger this VM-Exit at the right moment.

According to the Appendix C of Intel System programming guide [27], there are 70 different reasons that may trigger a VM-Exit, but only a few of them can happen from the outside of the VM. Most of them are related to interrupts.

Moreover in order to stop a VCPU thread, the VM-Exit must be handled correctly and not immediately followed by a VM-Enter.

I implemented two main approaches to stop the virtual machine when the timer expires, a first one based on signals, and a second one based on the VMX preemption timer.

4.1.2 Flow of time in virtual machines

Another challenge to address is the flow of time in virtual machines. Indeed unlike in emulation where the clock stops after a timer interrupt, as the whole virtualization process is handled by QEMU, the virtual clocks keeps ticking even outside of VMX guest mode. This is not an issue if we take care of adjusting the virtual clock before resuming the virtual machine, to ensure the clock does not take into account the time spent in the deadline handler.

Also, as there is a delay between the moment the clock is adjusted and the moment the VM execution resumes, this time is essentially lost for the virtual machine, which can result in a loss of performance, and also be noticeable by the VM for example by the vanishing of some periodic events during this lost time slot.

4.1.3 Handling of inaccuracies

Finally, unlike in emulation where we have instruction level accuracy, there might be inaccuracies introduced by the use of hardware virtualization and thus less precise methods of interrupting a virtual machine.

There are two cases of inaccuracies the may break the deadline algorithm:

1. A message happens to be timestamped before the previous deadline
2. A message happens to be timestamped after the current planned deadline

The first case can happen if a message is timestamped before the previous deadline , but inserted in the message queue after. This can occur because sending a packet is not an event that happen at a precise moment, but rather during a short time period, during which the packet is processed by the network card, before being sent on the network cables. Ideally we would like to timestamp a message just before it is sent on a network cable, because Simgrid simulates the propagation of a message on a network. As we do not have this precision, it is possible to see a deadline issued on the middle of this process. In this case, we decided to fix the the message timestamp to be equal to the previous deadline, to stay in the precision margin of a packet emission.

The second case happens if a message is timestamped after the current deadline, which is an issue as the VM should have been stopped when this deadline is reached. With the different solutions detailed below, this case can happen due to inaccuracies and delay of the timers based solution to interrupt the virtual machines. The virtual machines may be stopped slightly after the requested deadline. If this case happens, we log it along with the message timestamp and the deadline to check that the difference is still very low, and we fix the message timestamp to be equal to the current deadline in order to keep the algorithm coherent.

4.2 Signals

Now we focus on the first solution I designed during the internship that attempts to satisfy all the problems introduced in the last sections. This approach relies on signals, a choice that we justify in the next subsection. Then we present the architecture of this solution, before listing its advantages and disadvantages.

4.2.1 Justification

QEMU offers an API to stop a VCPU thread in KVM mode. However the kernel module cannot call a QEMU function directly, because it is located in userspace. To use QEMU's API to stop VCPUs, I used signals, which are one of the way to effectively notify a user process from the kernel.

Signals are notifications that can be send to a specific process or thread. On the reception of a signal, the process execution flow is interrupted and a signal handler, either registered by the process or a default one, is executed.

Sending a signal on a VCPU thread is one of the few ways to trigger a VM-Exit outside of the VMs. Indeed sending a signal will cause the kernel to send an IPI (Inter Processor Interrupt) to the CPU core that is running the targeted VCPU thread. This hardware-level IPI causes a VM-Exit to happen immediately. This approach also does not require modification in the kernel/KVM hypervisor, as everything can be handled from an external kernel module.

4.2.2 Outline of the approach

The idea is to use a signal to notify QEMU about the expiration of the timer, and stop/resume the VCPUs from QEMU by using the dedicated functions. This functions work by sending another signal to the VCPUs thread. This special signal will be handled by a signal handler defined in QEMU, that will set the `immediate_exit` bit to 1 in the `kvm_run` data structure. When this bit is set, KVM will return to the userspace calling process (In our case QEMU). From there, the VCPU thread can stay idle until the appropriate function is used to resume the VCPUs.

To implement this idea, I added my own signal handler in QEMU. Instead of implementing a real signal handler, I used `signalfd`, which creates a file descriptor which changes the way signals are handled by a targeted process. Instead of interrupting the execution of the process and call a specific function on a signal arrival, the signal is put on hold. Then the program can poll the file descriptor provided by `signalfd` to check for any pending signals. QEMU offers a way to register a poll notifier in a dedicate thread, called an IOthread.

QEMU is an event-driven program that works with event loops. By default there is one event loop, the main loop. This main loop executes periodically a set functions to make QEMU work. More event loops can be created and are called IOthreads.

Thus I added my own notifier in an IOthread to execute some TANSIV related code when a signal is blocked by the `signalfd`. This code does the following:

```
foreach VM do
    while Not all VCPUs are stopped do
        call qemu_kick_vcpu on each VCPU
    end
    Call TANSIV-client's deadline handler;
    Set up the timer with an ioctl;
    Adjust the VM Virtual Clock to compensate the time spent in the deadline handler;
    foreach VCPU do
        Resume the VCPU with cpu_resume
    end
end
```

Algorithm 1: Deadline handling modifications

The figure 6 shows how it works. To begin, the TANSIV client issues an ioctl to register a deadline in the `tansiv-timer` kernel module. This ioctl handler will fire the `hrtimer` associated with the right VM with the deadline provided. When the `hrtimer` expires, it will send a signal, here I have chosen `SIGUSR1`, to the IOthread, one of QEMU's main loops. The signal will be registered by the `signalfd` we created, and after it has been polled, the algorithm written above begins.

All the VCPUs of each VM are stopped. This QEMU function guarantees that the VCPUs are stopped when it returns. Then TANSIV-client's deadline handler is called. This deadline handler works like in emulation mode, except that it will communicate with the Kernel to set the timers after the message delivery computations have been done. Then we adjust the VM virtual clock to

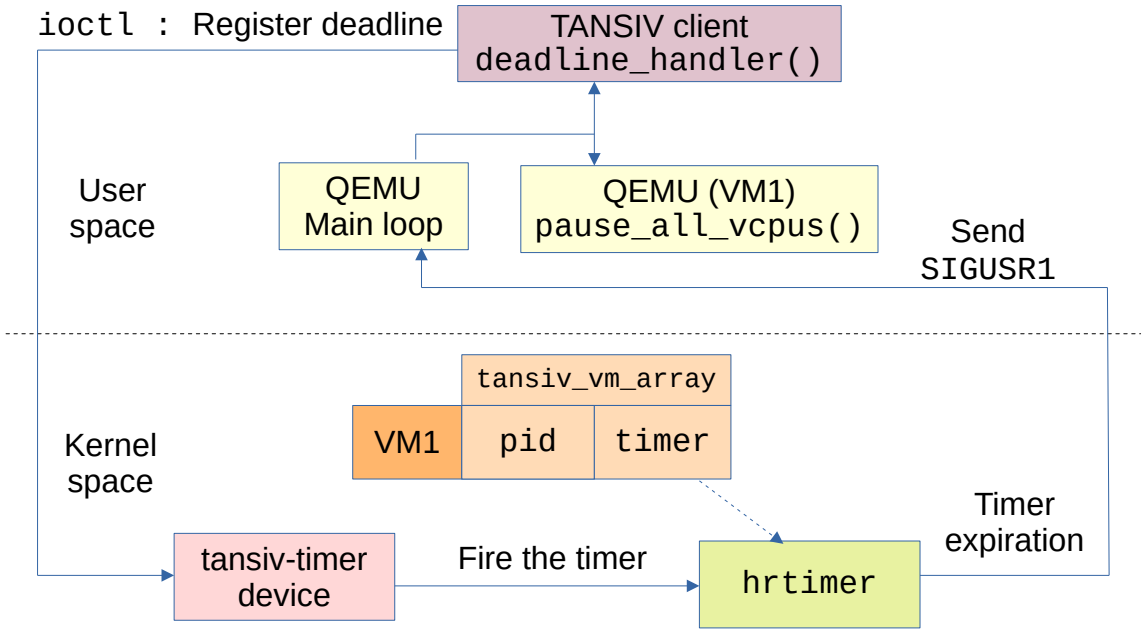


Figure 6: VM interruption approach with signals

not take into account the time spent in QEMU and TANSIV deadline handler. As explained in the subsection 4.4, the clocks keeps ticking even when the VCPUs are not executing code. Finally we resume the execution of all VCPUs by calling the appropriate QEMU function.

4.2.3 Advantages

This approach has several advantages:

- It does not rely on QEMU and can work with any userspace interface for KVM, as it is possible to add a signal handler to any program.
- No modifications are needed in Kernel/KVM code.

4.2.4 Issues

Although this solution worked, the delay caused by signals was the big drawback. Indeed there is a delay between the moment the signal request is made in the timer callback and the moment the VCPUs are stopped. During this delay the Virtual Machines can still execute some instructions, and if a network message is sent, the algorithm will fail because the message was sent after the

current deadline. Moreover when the different involved threads (Kernel timer, VCPUs, QEMU's IOThread) are running on different cores, the delays of signals are even longer.

To improve some of the delays explained here, we started with single VCPU for each VM and pinned all threads related to a VM (QEMU, the VCPU, Kernel timers) to a single core, different for each VM. This avoids the extra latency of an Inter-Processor Interrupt (IPI). This process is explained in section 5.3.

These delays are problematic because a message could potentially be sent slightly after the actual deadline.

This solution requires some tweaks to work with a VM with multiple VCPUs. Indeed in this case we should pin each virtual core to a physical core, and use one `hrtimer` per VCPU instead of a global `hrtimer` for the VM. These timers must also be synchronized to interrupt all virtual cores at the same time, which is another challenge as well.

Finally another drawback is that this approach is specific to Unix/POSIX compliant operating systems, as some OSes, such as Microsoft Windows, do not implement signals at all.

4.3 VMX preemption timer

In this section we introduce another approach for TANSIVTx, that aims to solve some issues present in the previous approach with signals, notably the precision of the interruption of VMs. This time we will use the VMX preemption timer introduced in section 3.3.2.

We begin with a justification of this new approach, then we present how it works. Finally we review the advantages and issues of this solution.

4.3.1 Justification

Besides interrupts and guest software actions, one of the few remaining ways to trigger a VM-Exit is the VMX preemption timer. As explained in section 3.2.2, the VMX timer is an Intel VT-x specific feature, that can be configured with a value and trigger a VM-Exit when the timer expires.

The goal of this approach is to use this feature instead of signals, which introduce delays, to trigger a VM-Exit and stop all the VCPUs threads when a deadline is reached.

4.3.2 Overview of the approach

An interesting behavior of the VMX preemption timer is that if it is configured with the value "0", it will immediately cause a VM-Exit, before any guest instruction get executed. Thus this behavior can be exploited to trigger a VM-Exit immediately from the callbacks of the kernel module timers.

The Linux Kernel has code to support the VMX preemption timer, but it does not work exactly like what we need here. Indeed if a VM-Exit is triggered by the VMX preemption timer, the kernel handles it internally and does not return to userspace to handle this VM-Exit. That is because KVM uses the VMX timer to emulate the guest TSC-Deadline timer, introduced in section 3.3.2. When a guest arms its TSC-Deadline timer, a VM-Exit will be issued, and KVM will arm the VMX preemption timer. When the VMX preemption timer expires, a VM-Exit will be issued and KVM will do the emulation work to notify the guest that its TSC-Deadline timer expired. All of that work is done from the kernel.

I had to change how this VM-Exit is handled by forcing KVM to return to userspace (QEMU). From there, the VCPU will already be stopped, and KVM will wait for the userspace handling of

the VM-Exit to be completed before resuming the VCPU. These changes are detailed in section 5.4.

I added an API for Kernel modules to load the value 0 in the VMX timer of all VCPUs from a given VM, and added a new VM-Exit handler to QEMU to support a VMX timer userspace exit.

Figure 7 explains how it works. It is similar to the design of the first solution with signals, detailed in section 4.2.2. The main difference happens when the `hrtimer` expires. It will call the functions I mentioned above, which will load 0 in the VMX preemption timers of its associated VM. This is done by writting 0 in the correct field of all the VMCS a VM has, because with KVM there is one VMCS for each VCPU. Then when the VM will resume its execution, a VM-Exit will be issued immediately, before any instructions are executed. To begin we decided to start with a Single-VCPU VM, with the `hrtimer` pinned on the same core as the unique VCPU. This avoids the delay of an IPI interrupt. Thus we will already be outside VMX guest mode when the timer handler is executed, and no VM instructions will be executed before the VMX preemption timer VM-Exit is used. Altough not done yet at this stage of the internship, this approach can be expended on VMs with multiple VCPUs, by having a dedicated timer for each VCPU and pinning each VCPU on a dedicated host core.

This VM-exit will be handled as described earlier, by returning to userspace (QEMU). Then from QEMU we can run the same algorithm as for the approach with signals, to compute the next deadline and resume the execution of all VMs.

4.3.3 Advantages

This approach offers several advantages:

- Assuming the kernel timer and the VCPUs are running on the same core, the VCPU will already be stopped because of an hardware interrupt. Then as the timer callback will set the VMX preemption timer to 0, when the VCPU thread will be rescheduled by the kernel, it will immediately cause a VM-Exit and returns to userspace. Thus this approach completely removes the delay between the timer expiration and the moment the VCPUs are really stopped.
- No need to separately stop the VCPUs threads and notify QEMU, as both are done at the same time.
- No signals are needed, thus this idea could work on operating systems such as Microsoft Windows.
- Not KVM specific. Any Intel VT-x compatible hypervisor could be used with this approach.

4.3.4 Issues

Nevertheless there are still some issues with this idea:

- Modifications of the Linux Kernel/KVM are required.
- This approach comes into conflict with other uses of the VMX preemption timer KVM/QEMU may have. As mentioned earlier KVM uses the VMX preemption timer to emulate the TSC-Deadline timer for the guests. Currently it is not a big issue because the OS we use for our VMs (Debian 11) use other timers (Lapic without the TSC-deadline mode/HPET) with our configuration, but it could become one if we start to use other OSes that use this timer.

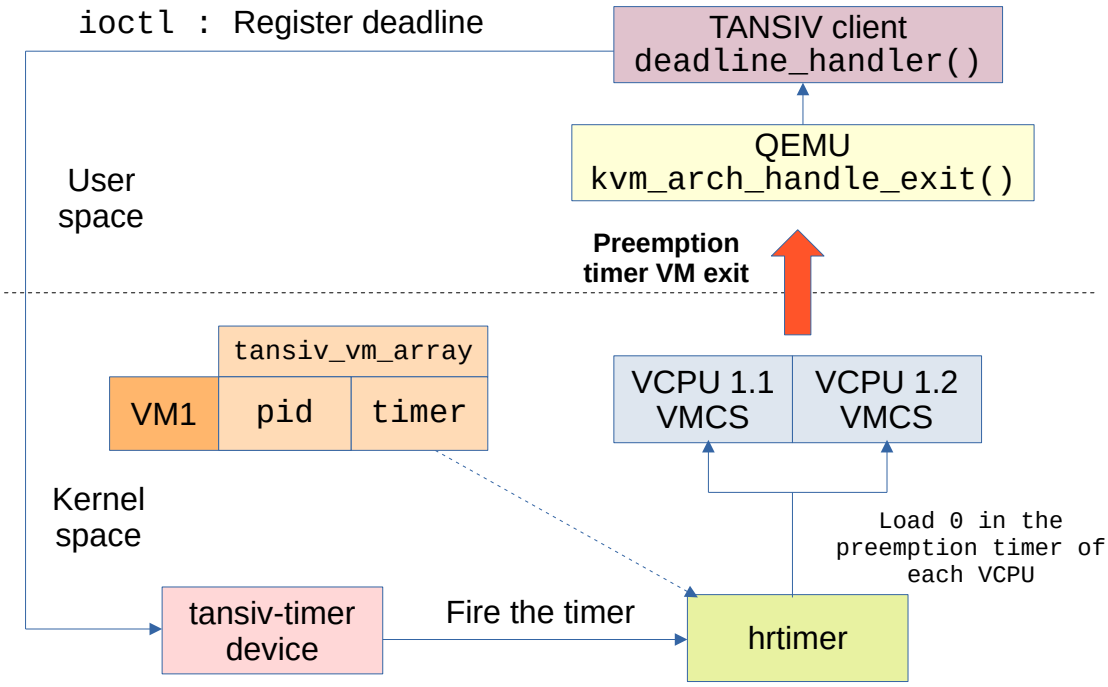


Figure 7: VM interruption with VMX preemption timer

- It does not eliminate all delays, in particular the delay between the timer expiration date and the moment the hardware interruption is effective and the timer callback is executed. However as explained in Chapter 6 of Intel System Programming guide [27], interrupts are handled by hardware and thus the delay is likely to be negligible if we stay in the setup with the timer and the VCPU running on the same physical core.

4.4 Time adjustment

One of the research problems was to find a way to synchronize the virtual time (as perceived by the guest OS) with the simulation time. Virtualization software such as KVM work by synchronizing the virtual time with the host time instead.

Thus while the VMs are not running (while handling a VM-Exit, not scheduled by the OS, ...), their virtual clocks are still running. This behavior is not desirable for TANSIV, as we wish to stop the clocks during the deadline handling.

As it is not possible to stop the virtual clocks used by the virtual machines, as they usually rely on the host TSC, instead we have to adjust the virtual clock before resuming the execution of VCPUs to compensate the time spent inside the deadline handler.

Ideally, in order to stay stealth, we wished to use the TSC as the main time source in the virtual machine, and adjust this virtual TSC after each deadline. However this was more complex than expected because this TSC is deeply tied to the host TSC in Linux's code, and small adjustments like the ones we have to do (A few milliseconds at most) are considered as TSC synchronization attempts by the kernel, which is not what we want. Indeed the desciption of the function `__kvm_synchronize_tsc` in the file `/arch/x86/kvm/x86.c` [28] explains that it is used to synchronize the guest's TSCs from the write attempts by the host.

That is why we instead opted for KVM-clock as the main time source for our virtual machines, because it offers an API to change the time. However it is not a stealth approach, as the guest will be aware this is a paravirtualized clock.

The figure 8 summarizes the problem of time adjustment and how it is handled in both approaches (Signals and VMX preemption timer). After a first deadline D1 has been issued by the simulator, the execution of the VMs will start slightly later, at a timestamped called T1. Then the VMs are supposed to be stopped at the next deadline D2. In real time, assuming the VM was never interrupted/Rescheduled on another core, it corresponds to T1_END. However as explained earlier, the processing of a deadline is not instantaneous. There is a first delay in blue, between the moment the deadline has been reached and the moment the VM must be stopped. This delay is different depending on the approach:

- With signals, the blue delay corresponds to the delay of signals.
- With the VMX preemption timer, this delay is almost non existent as it is only the timer interruption delay, assuming the timer and the VCPU are running on the same physical core.

Then there is a second processing delay, in pink, common to both approaches, that corresponds to the deadline handling algorithm, notably with calls to Simgrid to compute the next deadline. During the blue and orange delays, the VM's virtual clocks keep ticking, as it is based on one of the host clocks. Thus before resuming the execution of the VM at T2, we have to subtract to the VM's virtual clock the time spent in the blue and pink delays.

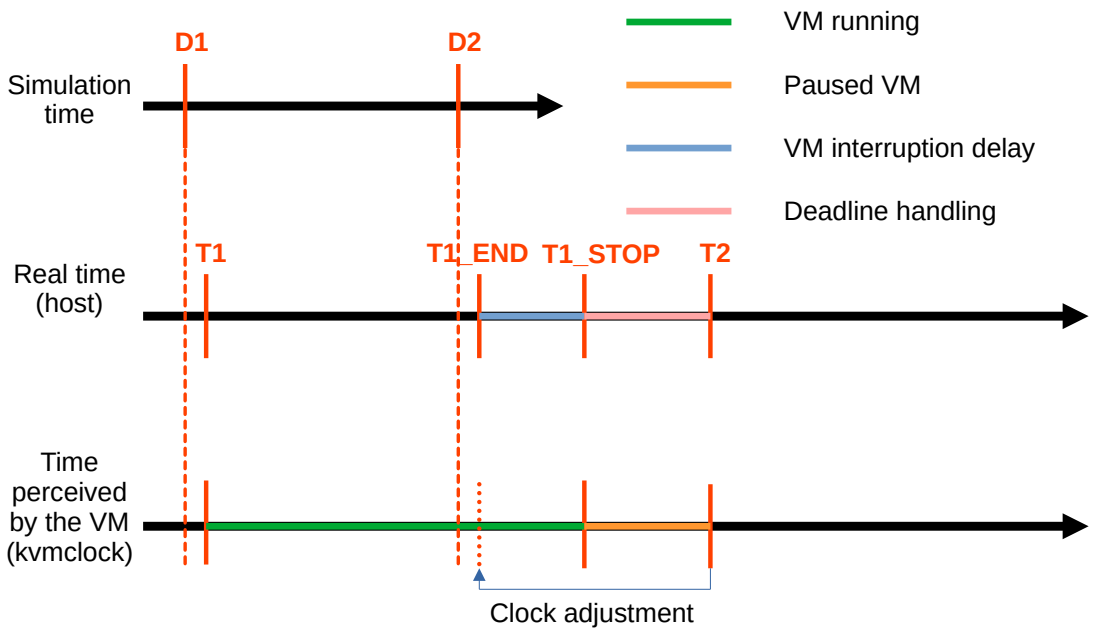


Figure 8: Time adjustment issues

5 Implementation

The goal of this section is to present more in detail some technical aspects of the implementation of both approaches.

We begin by a short overview of the implementation work done during the internship. We continue with an explanation of the kernel module I wrote to support TANSIVTx. Then I explain how I used Cgroups to pin the VM's VCPUS on the host cores. Finally I conclude this section with a description of the modifications required in the Linux Kernel for the VMX preemption timer approach.

5.1 Overview

My work works with a fork of QEMU 6.1.0, modified to add the TANTAP network backend and support to interact with the tansiv-timer kernel module.

Otherwise it with the 3.31 version of Simgrid, and version 5.17.0 of the Linux Kernel.

For the different approaches presented in section 4, I had to write from scratch a kernel module, and modify several components.

I modified TANSIV's interface to schedule timers to use my kernel module instead of QEMU's timers, and how the virtual machines are created in order to enable hardware virtualization and create the appropriate Cgroups (See section 5.3).

I modified several components of QEMU as well. For the signals approach I added my signal handler in QEMU, while for the VMX preemption timer approach I added a new VM-Exit handler. Moreover I added a way to modify the KVM-clock value from other QEMU components. Finally I added code in QEMU's initialization process to add support for my kernel module.

Finally the VMX preemption timer required to modify some code in the Linux Kernel, mostly KVM related. These changes are detailed in section 5.4.

To summarize, I created a kernel module with around 600 lines of codes, and modified 200+ lines in TANSIV, 200+ lines in QEMU, and about 50 lines in Linux.

5.2 Tansiv-timer kernel module

In order to use `hrtimers`, I wrote a kernel module called tansiv-timer, as an `hrtimer` can only be used from kernelspace. Moreover it is also convenient in order to use virtualization-related functions offered by the Kernel's API.

The kernel module uses `hrtimers` configured with relative time (That means the deadline provided will be considered as as the time to run the timer for rather than a timestamp at which the timer must expire), pinned to a core, in order to interrupt the VCPU when the timer expires, and a last option to set the interrupt context in which the timer will be expired to soft IRQ.

When a timer expires, a callback function is executed and will interrupt the VM and notify the userspace about it, depending on the solution chosen (Signals or VMX preemption timer)

To communicate between the userspace and the kernel module, I created several ioctls. `ioctls` are a way to add syscalls specific to a device. The notion of device encompasses many things, such as external peripherals or a kernel module. In our case the kernel module initializes a device and creates several `ioctls` to communicate with it from the userspace. Here everything is handled by software. A custom `struct` can be used as an argument and modified by the device, acting as a return object.

The `TANSIV_REGISTER_VM / TANSIV_REGISTER_VCPU` ioctls are used to register VMs and VCPUs when they are initialized by QEMU, by providing information such as their PID. As VCPUs are threads in the QEMU process, they share the same PID, so in order to identify them we use their thread identifier (TID) instead.

The `TANSIV_REGISTER_DEADLINE` ioctl is used to start an hrimer to a specific VM with the provided deadline.

5.3 Cgroups

As explained earlier, in order to reduce the latency of the interruption caused by the expiration of a timer, I pinned a timer and the VCPU on the same core. Pinning the timer is done by providing a special option in the timer configuration, while pinning the VCPU has been done by using Cgroups.

Cgroups are used to control the resource usage (CPU, memory, network, ...) of a set of processes.

In order to pin each VM to a core, I created a Cgroup for each VM created, and added the pid of the QEMU process when it is first started. As all child processes/threads are automatically added to their parent Cgroup, each VM will have all its associated threads running in the same Cgroup.

Cgroup configuration is done with controllers, which can be configured and affect all the processes in the Cgroup. The interesting one is the `cpuset` controller, which determines which CPU cores the VM processes can run on. I have set this controller to use only one CPU core, a different one for each VM.

With this technique, as the hrtimer will be executed on the same core as the VM, we avoid the Inter-processor interrupt latency after the timer expiration.

5.4 VMX preemption timer

In order to implement the solution using the VMX preemption timer, adding a function to be able to setup the VMX preemption timer from a kernel module was not enough due to the way VM-Exits are handled. Indeed, as written in section 3.2.1, there are two ways a VM-Exit can be handled, and a VMX preemption timer expiration is a "Light VM-Exit", handled exclusively in kernelspace.

In order to stop the execution of a VCPU thread after the expiration of VMX preemption timer, it was required to modify how this VM-Exit is handled by the Kernel to turn it into an "Heavy VM-Exit". I did the following modifications to do so:

- The Kernel VM-Exits handlers return 0 if it is required to switch to userspace, and 1 otherwise. I modified the kernel VMX preemption timer VM-Exit handler to return 0 instead of 1.
- I introduced a new exit reason, called `KVM_EXIT_DEADLINE_REACHED`. This reason is set a memory region shared between QEMU and KVM.

Finally I modified QEMU's VM-Exit handler to call the appropriate functions, as described in section 4.3.

6 Evaluation

In this section we present the evaluation of the approaches introduced in section 4. First we focus on measuring the accuracy of the proposed solutions when it comes to interrupting virtual

machines. Then we observe the performance of TANSIVTx and how it competes against TANSIV with emulation and KVM without TANSIV.

6.1 Interruption of Virtual Machines

6.1.1 Experiment overview

The objective of this first experiment is to check the accuracy of these solutions to accurately interrupt virtual machines. There are many experiments possible to measure the different parts of TANSIV that require a great accuracy, some of these are presented in section 6.3 because they could not be completed at this moment of the internship. Here we focus on a small experiment to measure the precision of `hrtimers`.

Ideally we want the delay between the moment the `hrtimer` is armed by the kernel and the moment the VM is interrupted to be exactly equal to the delay between both deadlines.

This delay can be measured thanks to two values. First when an `hrtimer` is armed, it stores in a field the absolute first date from which the timer can expire and the callback be executed. This is the first value to consider, as ideally the timer should expire at this date.

The second value is the moment the `hrtimer` callback is executed. It can be obtained by calling the function `ktime_get()` at the beginning of the timer callback.

By logging both values and computing their difference, we can measure the precision of the timers that are used to interrupt the virtual machines. We want this difference to be as close to 0 as possible.

For the approach with the VMX preemption timer, this delay is indeed the precision of the interruptions of VMs, because they will not execute any code once the `hrtimer` is expired. However for signals there is an additional delay caused by the propagation of signals.

This experiment was run on an Intel i7-1065G7 CPU, with a minimum latency of 10 ms. Thus there will be deadline at least every 20 ms, as explained at the beginning of section 3. The setup includes two VMs, without any network traffic, thus deadlines will only occur every 20ms as there will be no messages to deliver. The execution lasted for about a minute to gather enough data.

It was run with the VMX preemption timer approach, but for this delay there are no differences with the approach for signals.

6.1.2 Results

VM	Mean	Median	Standard deviation	Min	Max
VM1	8950	5319	15085	2224	347745
VM2	9298	5018	19611	2100	591068

Figure 9: Delay between the `hrtimer` actual and programmed expiration date. All values are in nanoseconds.

The table 9 and the distribution of the delay 10 summarize the results. First we can see that the results are similar for both VMs, which was expected.

The mean and the median show that for most interruptions, the delay is of a few microseconds, which is still acceptable for the accuracy we wish with TANSIVTx. This is confirmed by the

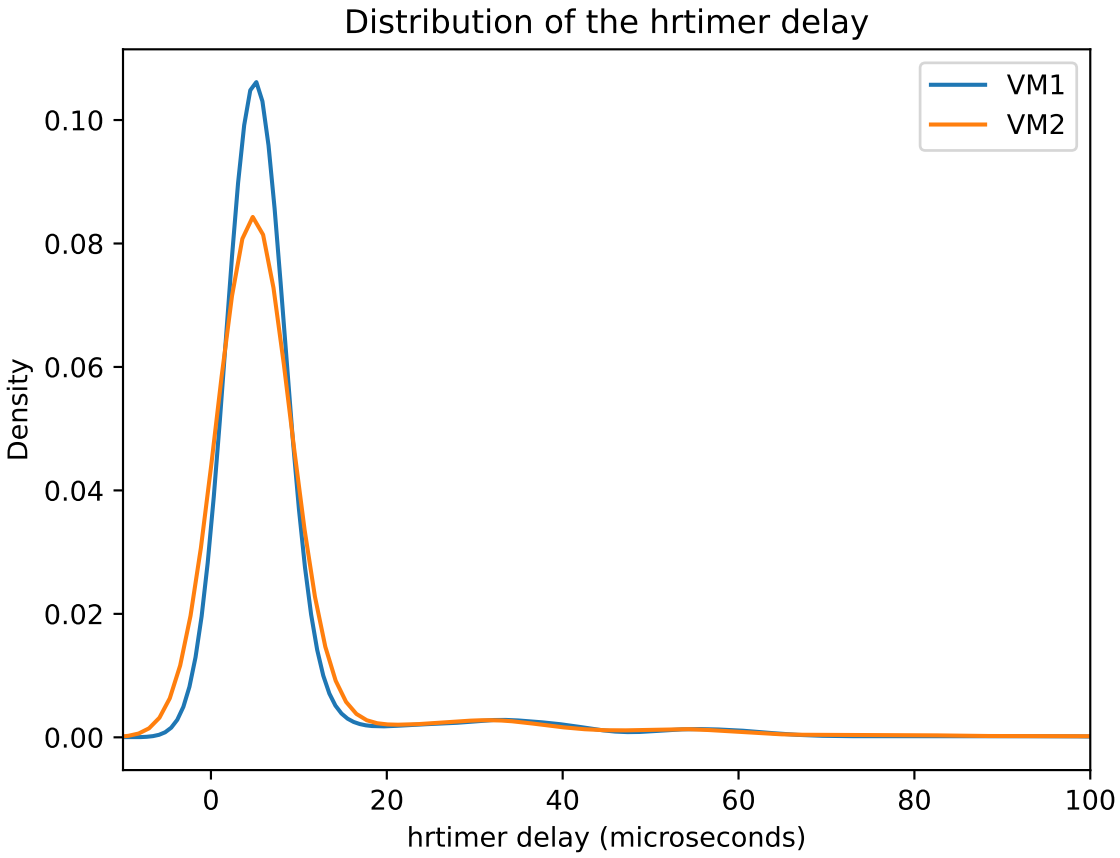


Figure 10: Distribution of the `hrtimer` delay between the programmed and the actual expiration date

distribution of the delay, where we can see that almost all the delays are under 20 microseconds. However the large standard deviation and the maximum values show that there are some outliers, where the delay between the programmed expiration date and the execution of the timer's callback can be of a few hundred microseconds. This delay is too important for the precision we are aiming for.

There are many potential explanations for this phenomenon. As this experiment was run on a busy laptop, it is very likely that others programs interfered with the execution of VMs. For example the CPU could have decided to schedule another program on the cores allocated to the VCPUs, which may cause an additional delay before the hrtimer interrupt happens. This experiment can be improved by excluding the cores allocated to the VCPUs from those usable by the OS to schedule other applications.

6.2 Performance of TANSIVTx

6.2.1 Experiment overview

The objective of this second experiment is to measure the performance of TANSIVTx. Indeed one of the major drawbacks of TANSIV with emulation mode was the poor performance, partially caused by the use of an emulator instead of virtualization. The goal of this experiment is to evaluate the impact on performances of the switch from emulation to Hardware virtualization. We will also compare to KVM without the TANSIV protocol, to have an indication of the overhead caused by the TANSIV protocol.

The experiment is the following : we will measure how long a VM takes to fully boot. We begin the timer when we invoke the QEMU process, and ends it when we can connect to a VM with SSH by checking port 22, as it roughly corresponds to the time a VM is booted and operational. This is an interesting choice because the boot process requires a lot of computations and is usually very slow in a fully emulated Virtual Machine.

As the boot process does not really use the network, it is not useful to modify the parameters related to the network configuration. Regarding the VMs configuration, an interesting parameter would be the number of VCPUs of the VM. Indeed as TANSIV with emulation is single-threaded only (due to icount mode), it can be interesting to compare it with TANSIVTx / unmodified KVM which run each VCPU in its own thread. Other VMs parameters (RAM/Disk space/...) are not relevant for this experiment and are left unchanged.

Here is the setup of the experiment: This experiment was ran on a computer equipped with an Intel i7-1065G7 CPU and a Samsung 980 SSD. This CPU model has 4 cores available, thus in total there are 8 logical cores available because hyperthreading is enabled. The fact that this machine is equipped with an SSD is important because it has a major impact on boot time. The VMs are running a Debian 11 OS.

For TANSIV and TANSIVTx, we have to start two Virtual Machines at the same time, because TANSIV can not work without at least two Virtual Machines.

Note that for TANSIVTx we have not pinned each VCPU on a specific core in this experiment, thus this setup is not the most accurate, and should be completed by a future experiment with one core per VCPU.

The results are presented in the next section.

6.2.2 Results

Figure 11, shows the result of the boot experiment on TANSIV, TANSIVTx and KVM.

In this histogram, we can clearly see that there is an improvement in boot time from the switch to TANSIV to TANSIVTx. Indeed, for a VM with 1 VCPU, the boot time takes around 150s while it is around 25s for TANSIVTx, which is 6 times faster. KVM without TANSIV is also faster than TANSIVTx, taking around 5 seconds less.

When we increase the number of VCPUs, we can see that TANSIVTx and KVM still take the same amount of time for booting, while TANSIV takes significantly more time, with over 250 seconds for 8 VCPUs instead of 155 seconds with 1 VCPU. That was to be expected because while TANSIVTx and KVM have 1 thread per VCPUs and can use all the available cores of the host CPU, TANSIV is single-threaded and can only use one physical core. It can still emulate multiple VCPUs thanks to a round-robin algorithm to schedule the different VCPUs.

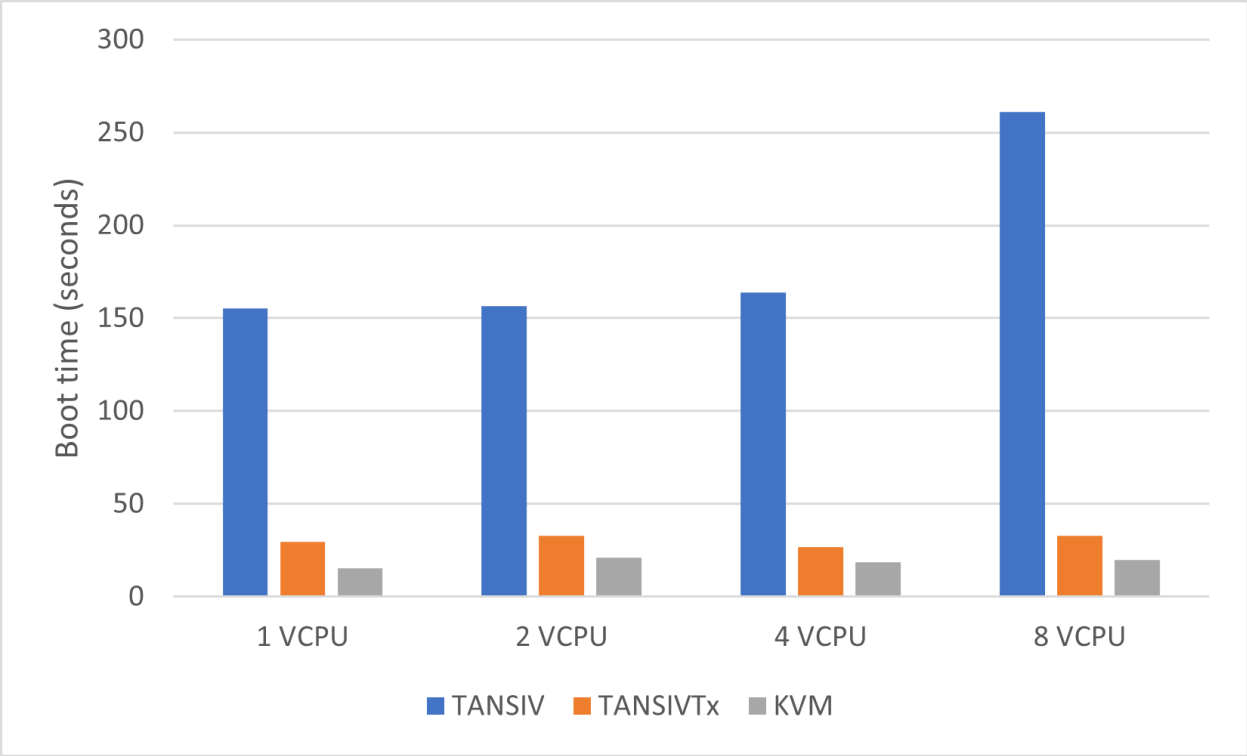


Figure 11: Boot time for the different virtualization options, depending on the number of VCPUs

This first experiment highlights the fact that TANSIVTx will likely perform significantly better than TANSIV, notably for applications that exploit multithreading. Nevertheless the overhead of TANSIVTx is still noticeable, and thus can be the object of future work in order to improve the performance even more.

6.3 Future experiments

Due to lack of time, we had not the time to run all the experiments we wished at this stage of the internship. The purpose of this section is to present some of these experiments.

Regarding performances, the boot time experiment of section 6.2.1 only serves to measure the impact on performances of the switch from emulation to hardware virtualization, and the overhead of TANSIVTx compared to unmodified KVM. However in the context of the boot process, there are almost no network messages sent between each VM. Thus another experiment is needed to measure the impact of TANSIV in a context where many Virtual Machines are sending packets on the simulated network.

A way to do it could be to run a Redis benchmark. This benchmark can be configured to run commands (like network requests) on multiple clients (virtual machines).

This experiment could also be used to check the impact of network parameters on performances (For instance latency has a direct impact on the minimum delay between deadlines, and thus on performances), and how well TANSIV scales when many VMs are present.

For the accuracy, we only presented here some measurements of the precision of `hrtimers` which are part of the methods developed in the internship to interrupt Virtual machines. But there are

other things related to the accuracy of TANSIV to quantify, such additional delay of signals for the first approach.

In section 4.1.2 we mentioned the "lost time" for virtual machines, which corresponds to the delay between the adjustment of the virtual clock of the VM and the moment the VM VCPUs can resume their execution. It would be very interesting to measure this lost time, to see if it negligible or not compared with the time spent in TANSIV-client/Simgrid, and what proportion of its time slot the VM can actually use.

Moreover we have not measured in the context of TANSIVTx the accuracy of the delivery of network messages and how well it compares to TANSIV. This can easily be done, for example by sending many ping requests with the command `ping -f remote_vm`, and comparing the results with the ones from TANSIV and what was expected according to the configuration of the network.

6.4 Limits and Discussion

As highlighted in this section and the previous ones, there are still some limits to our approach. Here we will go through them and discuss potential solutions.

First our solution use KVM-clock as the time source for the guests. It is a paravirtualized component, thus the virtual machine is fully aware it is a clock source provided by the KVM hypervisor. Its use is justified because it is easily modifiable by the host. A better approach would be to use the TSC as the guest clock source, and modify its value by writing to the guest's TSC registers. As the TSC is not a paravirtualized clock, this is a stealthier solution. But this would require some modifications in the kernel to be able to make small adjustments to it as required by TANSIV's algorithm.

On Linux, QEMU only supports the use of KVM for Hardware virtualization. Our current implementation relies on these two components (KVM/QEMU) to work. Thus it would not work on OSes different from Linux, because they do not support KVM. For instance on Mac OS, QEMU uses Apple's Hypervisor framework for hardware acceleration. Nevertheless the concept of the solution, in particular the VMX preemption timer one, are not tied to a specific hypervisor/OS, and should be portable on another hypervisor, as long as we stick to the x86 architecture.

Moreover currently TANSIV is not yet compatible with other sandboxing tools. A concern about others sandboxes is that most of them use various time manipulation techniques to perform malware analysis. For instance as presented in section 2.3.1, sandboxes accelerate the time to counter stalling code. We will need to ensure that TANSIV is compatible with these anti-evasion techniques that manipulate the time.

Furthermore our approaches do not work yet with enough accuracy on VMs with multiple VCPUs. Indeed having multiple VCPUs introduce new problems, such as the precision required to restart all the VCPUs after a deadline, to avoid stealing execution time from some VCPUs.

Finally there is still a problem of precision, in particular to the timestamping of messages sent by the Virtual Machines, as first mentioned in section 4.1.3. There is a delay between the moment the message is sent by the VM and the moment it is received and timestamped by the TANSIV-client. If a deadline happens in the middle of this delay, this message will not be received in time, which is a major issue for the reasons explained in section 4.1.3. Measuring this delay precisely is crucial to better understand this problem, and see if a fix more sophisticated than adjusting the messages timestamps is required.

7 Conclusion

In this internship report, we first reviewed in section 2 the state of the art of virtualization techniques, sandboxing tools and evasive malware in the bibliographic review. Then in section 3 we presented the technical context of the internship work, including the TANSIV architecture. In the section 4 we presented the different approaches to add hardware virtualization support to TANSIV, before presenting some implementation details in section 5. Finally we present the evaluation results and limits of our approach in section 6, before concluding.

TANSIV's goal is to provide the illusion of an unmodified execution to applications that are using the network, by using a network simulator, Simgrid. In particular we want this illusion of a realistic network to be robust enough so that malware can't use evasion techniques based on network timing.

Moving TANSIV in emulation mode to TANSIVTx with hardware virtualization is important because it improves the performances, in particular thanks to multiple VCPUs support, the stealthiness, as emulation is easier to notice than hardware virtualization for en evasive malware, and compatibility with other sandboxing tools, as most use hardware virtualization instead of emulation.

To move from TANSIV to TANSIVTx, I presented two approaches, that both have advantages and disadvantages in regards to portability, stealthiness and accuracy of TANSIV. The first approach with signals is the simplest one and does not require any modifications to the Linux Kernel. However the precision is not satisfying due to the delay of signals propagation.

The second option with the VMX preemption timer better meets our expectations in terms of accuracy because the delay of VM interruption is much shorter than with signals.

TANSIVTx can be improved to better support VM with multiple VCPUs, because it will most likely vastly improve performances over TANSIV in emulation mode which is single-thread only due to the use of icount mode, even if multiple VCPUs are emulated. Moreover more evaluations should be done to thoroughly evaluate TANSIVTx.

Other upcoming work for TANSIVTx would be to improve the VMX preemption timer approach by fully using it as a timer to interrupt VMs. Other long terms goals include to add support for hypervisors others than KVM and add compatibility with sandboxing tools dedicated to malware analysis such as DRAKVUF [10].

References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," in *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*, H. Schorr, A. J. Perlis, P. Weiner, and W. D. Frazer, Eds., ACM, 1973, p. 121. DOI: `10.1145/800009.808061`. [Online]. Available: `https://doi.org/10.1145/800009.808061`.

[2] "Vmware workstation." (1999), [Online]. Available: `https://www.vmware.com/products/workstation-pro.html`.

[3] "Virtualbox." (2007), [Online]. Available: `https://www.virtualbox.org/`.

[4] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, USENIX, 2005, pp. 41–46. [Online]. Available: `http://www.usenix.org/events/usenix05/tech/freenix/bellard.html`.

[5] "Bochs." (1994), [Online]. Available: `https://bochs.sourceforge.io/`.

[6] C. Kruegel, "Full system emulation: Achieving successful automated dynamic analysis of evasive malware," in *Proc. BlackHat USA Security Conference*, 2014, pp. 1–7.

[7] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, J. T. McDonald, M. D. Preda, and N. Stakhanova, Eds., ACM, 2015, 4:1–4:11. DOI: `10.1145/2843859.2843867`. [Online]. Available: `https://doi.org/10.1145/2843859.2843867`.

[8] "Pyrebox." (2017), [Online]. Available: `https://github.com/Cisco-Talos/pyrebox`.

[9] "Icebox." (2019), [Online]. Available: `https://github.com/thalium/icebox`.

[10] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, Eds., ACM, 2014, pp. 386–395. DOI: `10.1145/2664243.2664252`. [Online]. Available: `https://doi.org/10.1145/2664243.2664252`.

[11] C. Dall and J. Nieh, "KVM/ARM: the design and implementation of the linux ARM hypervisor," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, 2014, pp. 333–348. DOI: `10.1145/2541940.2541946`. [Online]. Available: `https://doi.org/10.1145/2541940.2541946`.

[12] P. Barham, B. Dragovic, K. Fraser, *et al.*, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, M. L. Scott and L. L. Peterson, Eds., ACM, 2003, pp. 164–177. DOI: `10.1145/945445.945462`. [Online]. Available: `https://doi.org/10.1145/945445.945462`.

[13] "Cuckoo sandbox." (2010), [Online]. Available: `https://cuckoosandbox.org/`.

[14] C. Luk, R. S. Cohn, R. Muth, *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds., ACM, 2005, pp. 190–200. DOI: `10.1145/1065010.1065034`. [Online]. Available: `https://doi.org/10.1145/1065010.1065034`.

[15] "Dynamorio." (2001), [Online]. Available: `https://dynamorio.org/`.

[16] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Comput. Surv.*, vol. 52, no. 6, 126:1–126:28, 2020. DOI: `10.1145/3365001`. [Online]. Available: `https://doi.org/10.1145/3365001`.

[17] H. Lee, J. Seibert, M. E. Hoque, C. E. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, IEEE Computer Society, 2014, pp. 660–669. DOI: 10.1109/ICDCS.2014.73. [Online]. Available: https://doi.org/10.1109/ICDCS.2014.73.

[18] G. F. Riley and T. R. Henderson, "The *ns-3* network simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds., Springer, 2010, pp. 15–34. DOI: 10.1007/978-3-642-12331-3_2. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3%5C_2.

[19] H. Tazaki, F. Urbani, E. Mancini, *et al.*, "Direct code execution: Revisiting library OS architecture for reproducible network experiments," in *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, K. C. Almeroth, L. Mathy, K. Papagiannaki, and V. Misra, Eds., ACM, 2013, pp. 217–228. DOI: 10.1145/2535372.2535374. [Online]. Available: https://doi.org/10.1145/2535372.2535374.

[20] J. S. Ivey, B. P. Swenson, and G. F. Riley, "Simulating networks with NS-3 and enhancing realism with DCE," in *2017 Winter Simulation Conference, WSC 2017, Las Vegas, NV, USA, December 3-6, 2017*, IEEE, 2017, pp. 690–704. DOI: 10.1109/WSC.2017.8247825. [Online]. Available: https://doi.org/10.1109/WSC.2017.8247825.

[21] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *J. Parallel Distributed Comput.*, vol. 74, no. 10, pp. 2899–2917, 2014. DOI: 10.1016/j.jpdc.2014.06.008. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.06.008.

[22] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego, "Qbox: An industrial solution for virtual platform simulation using qemu and systemc tlm-2.0," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01292317.

[23] T. Roccia, M. Rivero Lopez, and C. Shah, "Evolution of malware sandbox evasion tactics-a retrospective study," vol. 9, 2019. [Online]. Available: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/.

[24] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM, 2011, pp. 285–296. DOI: 10.1145/2046707.2046740. [Online]. Available: https://doi.org/10.1145/2046707.2046740.

[25] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2750–2765, 2020. DOI: 10.1109/TIFS.2020.2976559. [Online]. Available: https://doi.org/10.1109/TIFS.2020.2976559.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The linux virtual machine monitor," in *Proceedings of the Linux symposium*, Ottawa, Ontorio, Canada, vol. 1, 2007, pp. 225–230.

[27] Intel, *Intel® 64 and IA-32 architectures software developer's manual volume 3 (3A, 3B, 3C & 3D): System programming guide*, Dec. 2021. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[28] [Online]. Available: `https://elixir.bootlin.com/linux/v5.17/source/arch/x86/kvm/x86.c#L2527`.