Ramooflax

übervisor

Stéphane Duverger
stephane.duverger@eads.net

EADS Innovation Works
Suresnes, FRANCE

Abstract. Ramooflax is a free and open source[1] virtualization tool, delivered under the form of a minimalistic kernel acting as an hypervisor and a remote client allowing high level access to the features implemented into the hypervisor.
This tool' architecture seamlessly simplifies the deployment of a flexible, powerfull, isolated and relieved of any software dependencies system analysis environment.
Its main target being up and ready operating systems, installed on physical x86 32 and 64 bits machines equipped with hardware virtualization extensions.

1 Introduction

This tool tries to highlight the use of hardware virtualization extensions as a means to help remote analysis of operating systems.
The *on-the-fly*[2] approach wasn't convincing from a seamlessly point of view. The main disadvantage being that the hypervisor is living inside the virtualized system memory and under this situation depends on features and concepts implemented by the virtualized system: memory model, scheduling, interruptibility and so on. If in the simplest situation, this dependency is only related to the virtualized system initialization, it can in the worst situation tamper the hypervisor continuity and integrity.
Although hardware virtualization enables efficient protections implementation for that kind of hypervisors, some complex mechanisms still need to be implemented as well.
Our approach seems to be original in the sens that it is absolutely independent from the targeted system. We are, on the other side, constrained to start our software stack before the targeted system[3]. At first being more restrictive, this approach has the ability to allow a complete control over devices visibility, such as the amount of physical memory that will be given to the virtualized system.
This tool focuses on:

[1] GPLv2
[2] the hypervisor is loaded while system is running as a driver.
[3] Several scenarii are possible. The simplest one could be using a bootable USB key.

– being lightweight, simple and fast
– taking benefit of existing stuffs (i.e. BIOS)
– being solely hardware dependent
– delegating analysis complexity to a remote client

The remote analysis is operated with a python framework providing high level access to the hypervisor features, in order to easily implement numerous plugins:

– remote debugger
– process memory mappings graph
– behavioral analysis of a boot loader
– ...

One could also think about a plugin illustrating modern CPU mechanisms, from a pedagogical perspective, as Nate Robins[**?**] did with OpenGL API.
This article presents the conception and implementation of the hypervisor and the python framework giving the opportunities to develop analysis tools.

2 The hypervisor

2.1 Targeted architecture

Our hypervisor supports x86 family processors that come with *recent* hardware virtualization extensions[4]. By *recent*, we mean using mainly the last advances regarding MMU virtualization, more specifically EPT[5] and RVI[6].
These features interest resides in the fact that they really simplify the hypervisor implementation while drastically increasing performances as opposed to an SPT[7] implementation. The hypervisor attack surface is also reduced once relieved of the SPTs complexity.
All processors featured with hardware virtualization extensions (CoreXX, Phenom, Athlon, ...) do not necessarely provide *recent* extensions such as the MMU one. We can find them in really cutting edge processors that are usually delivered into servers ou high performances workstations.
At the term of Intel[**?**] roadmap, we should more likely encouter that features in processors delivered with standard workstations.

2.2 The Ramooflax concept

The objective is to virtualize already installed operating systems on physical dedicated machine. Virtualization is enabled at boot time in order to start the already installed operating system in a virtualized environment.

[4] Intel VT-x and AMD-V
[5] Intel's Extended Page Tables
[6] AMD's Rapid Virtualization Indexing, formerly Nested Page Tables
[7] Shadow Page Tables

This allows virtualization, and so analysis, of operating systems running in their *native* environment more specifically regarding devices which are hardly emulated by common existing virtualization solutions.
The idea is to boot the hypervisor from an external storage media (USB key), and once the hypervisor has been initialized, to tell the BIOS (now virtualized) to boot the already installed operating system.

2.3 Architecture

Ramooflax is compound of 3 minimalistic kernels: Loader, Setup and VMM.

Loader Following the multiboot standard, being able to be loaded from GRUB, this kernel only setups *longmode* on the CPU. Mainly because the multiboot standard does not allow to directly start a 64 bits longmode kernel. Our hypervisor runs into longmode to be able to manage 32 and 64 bits VMs. The loader finally loads the setup.

Setup This 64 bits kernel is responsible for CPU and virtualization features initialization. Init code once used is not needed anymore and can be dropped out from memory. That's why putting it into a kernel was a simple idea to get rid of this unneeded code later.
The setup then retrieves physical memory size (RAM) and relocate the VMM at its end. By providing the VM a slightly lower RAM size, we can be sure that the VM won't try to allocate physical memory pages used by the VMM (except targeted attacks of course).
Once the VMM initialized, the setup installs in conventional memory[8] the `int 0x19` instruction and starts VMM execution. The VMM will starts its single VM on the previously installed instruction. The BIOS will then load the boot sector from its first bootable device (ie HDD in most situation).
The interest behind this being to reuse existing software pieces that are already able to access SATA or SCSI devices, whatever they can be.

VMM The final hypervisor is a PIE[9] ELF 64 bits executable. As previously mentioned, the 64 bits code allows us to virtualize 32 and 64 bits VMs.
The PIE was needed to be able to relocate the VMM without being dependent of the amount of available RAM. Every machine can have a different RAM size, this seemed us to be an elegant solution.
The hypervisor role is to control the VM behavior on the CPU and devices. The section **??** will detail the hypervisor mechanisms.
The figure **??** summarizes the VMM and VM boot sequence.

[8] The 640KB of lower memory ... nostalgia.
[9] Position Independent Executable

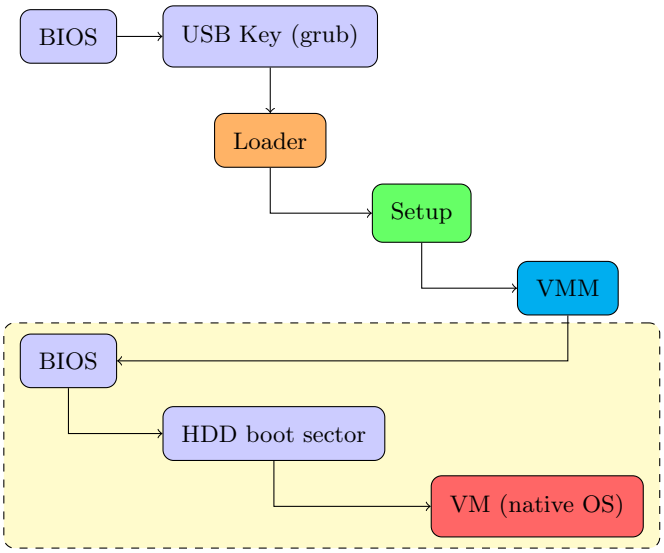

Fig. 1. Virtualized operating system boot chain.

2.4 Interest

We wanted to minimize the impact on the VM execution environment (being as close as its *real world*) and promote the ease of deployment. Looking at existing hypervisors, we found:

- inadequate solutions
  - Xen, VirtualBox, KVM
  - too complex to deploy (dom0, linux+userland, ...)
  - emulated environment (dumb bios, devices)
- intrusive solutions
  - bluepill[**?**], vitriol[**?**], virtdbg[**?**], hyperdbg[**?**], abyss[**?**]
  - *in vivo* virtualization
  - OS dependency

It was more interesting to restart from scratch, on one hand because existing architectures did not met our requirements, on the other hand because the author is not ashamed to confess that he indulges keep on reinventing the wheel although this kind of hypervisor has never been met before[10].
Notice that existing hypervisors only virtualize minimalistic and featureless BIOS. Our solution allows the analysis of *real life* BIOS. Ramooflax ambitious !

[10] As far as the author knows.

2.5 Limitations

To date, Ramooflax has been tested with success under Windows XP/7 Pro 32 bits and Debian GNU/Linux 5.0 32 bits. Simpler operating systems should also work. Linux 64 bits kernels have also been virtualized. However we prefer focus on hypervisor and python framework features rather than providing a featureless hypervisor able to run any VM in any CPU mode. The hypervisor only runs on AMD processors, for the moment. The Intel port needs to be rewritten.
The hypervisor only virtualizes a single Core, which does not prevent VMs to make use of the unvirtualized ones (discouraged). Cores virtualization is relatively complex to setup in our architecture, mainly because the Application Processors or Cores initialization must be done by the hypervisor to enable virtualization on each of them, but must also intercept initialization done by the VM (standard SMP kernel boot code).
It is still possible to hide the remaining Cores, either by giving specific parameters to the VM (`/numproc`, `maxcpus`, ...) or using only uniprocessor kernels, either by directly acting on `cpuid, rdmsr` instructions. The later solution may not be sufficient due to ACPI tables configuration or other unknown pieces of information stored elsewhere telling there are many Cores installed.
Finally, the hypervisor does not implement nested virtualization (virtualization of hardware virtualization extensions). As it does make use of them, they are hidden to the VM.

3 A brief overview of hardware virtualization

Hardware virtualization extensions, provided by means of a reduced instructions set and interceptions mechanisms, greatly simplify hypervisor development. Although initially compatible, Intel and AMD have developped on their own side these hardware virtualization extensions leading to incompatible implementations.
We present in this section, non exhaustively, their approach illustrating their common points, differences and over all their limitations.

3.1 Common elements

Whether it is for Intel or AMD, VM execution on the CPU relies upon configuring a huge data structure (VMCS[11], VMCB[12]) which is responsible for the setup of system registers, sensitive instructions execution interception, but also injection and interception of events (interrupts, exceptions).
Under AMD this structure is directly accessible from main memory, while under Intel it is accessed using specific instructions (`vmread` et `vmwrite`) each VMCS field having its own encoding value.

[11] Intel Virtual Machine Control Structure
[12] AMD Virtual Machine Control Block

The hypervisor and the VM both have their own data structure, to be able to save/restore each of them.
In example, the VMCB structure allows interception and configuration of the following elements:

- interception:
  - read/write accesses on cr, dr, idtr, gdtr, ...
  - pushf, popf, cpuid, iret, int, hlt, ...
  - vmrun, vmmcall, vmload, ...
  - exceptions, hardware and software interrupts, smi, ...
- setup values for:
  - cs, ds, ..., gs (base, limit, attributs)
  - efer, cpl, rflags, cr[0-4], dr6 et dr7
  - ...

The execution/interruption of a VM are respectively called `vm-entry` and `vm-exit`. Notice that the CPU operates automatic restrictive checks upon each `vm-entry` and `vm-exit` with regard to the values stored into the VMCS/VMCB.
Notice that Intel and AMD offer hardware virtualization instructions intercept. This allows an hypervisor to properly emulate them in case a VM needs to run an hardware based hypervisor.
Each intercept provides details on the context it happened, for instance:

- exception vector
- targeted cr register
- `out` desination port and operation size
- ...

Of course, Intel and AMD do not give the same detail level on each `vm-exit`. Moreover, the hypervisor state might not be fully restored after a `vm-exit` (GDT limit under Intel, LDT under AMD).
Despite all of these features, an hypervisor must be able to manage CPU mode transitions (real, protected, long) on its own, as well as physical memory mappings as seen by the VM, events injection consistency (ie exception injection while injecting exception can lead to double fault exception) ...
Embedding a disassembly and emulation engine is often needed to assist and get further details on the `vm-exit` context.

3.2 Intel-VT (vmx)

Intel approach perfectly applies to the *virtual cpu* notion. The hypervisor runs in a privileged `vmx-root` mode, whereas the VM runs in a `vmx-nonroot` mode, once virtualization enabled.
Instead of providing standard intercept mechanisms for control registers, Intel offers a register *shadowing* system. The hypervisor setups a bitmap of register bits which will be alterable, read to a fixed value or will generate a `vm-exit`.
In other words:

- some bits are owned by the VMM and their access lead to `vm-exit`
- some bits are read in a *read shadow* copy of control register
- some bits are owned by the VM

This limits `vm-exit`s only to filtered bits (ie enabling protected mode, paging, ...) and seamlessly give a fake version of the control register to the VM.
This *shadowing* mechanism is not limited to control registers. It also applies for instance to `rflags`.
Last note on control registers filtering, Intel provides the general purpose register used in the instruction to access the control register. This prevents the hypervisor from disassembling.
In return, this *shadowing* setup is sensitive and depends upon many things amongst them the processors revision or specific virtualization features. For instance, Intel does not allow to disable paging under the VM and so the *shadowing* must be configured to hide enabled paging bit depending on the VM execution mode.
The MSRs accesses are pretty straightforward to manage. The hypervisor chooses which of them will reflect reality and which of them will be set to specific values for the VM. These MSRs being automatically saved/restored upon `vm-entry` and `vm-exit`.

3.3 AMD-V (svm)

AMD's hardware virtualization is far more simpler and sadly far more subtle than Intel one. It can be seen as a new CPU mode. There is no register *shadowing*. The hypervisor logic is based on intercepts only.
Thus for control registers, one can only setup bitmaps for enabling/disabling read/write accesses. The system registers loaded from the VMCB will be the real ones used by the CPU at the VM runtime, except some of them which are directly accessed through the VMCB. There is no bit control granularity as we could find under Intel.
Even if it seems simpler, it isn't convenient for performances because many more `vm-exit`s will be raised for unnecessary bits. Moreover, it can be complex for the hypervisor to hide only some bits (ie `rflags.tf` used for single-stepping). AMD rather provides interceptions of RFLAGS related instructions (`pushf`, `popf`) and the hypervisor is thus constrained to emulate them. Instruction emulation is sensitive and complex under x86 because of the wide variety of executions modes and protection features.

3.4 MMU virtualization

Former CPUs did not provided MMU virtualization features. Hypervisors were constrained to implement the so called Shadow Page Tables (SPTs), complex mechanism responsible for virtualization of the physical memory of the VM (translatation of physical VM addresses to physical VMM addresses, also called system addresses).
This approach was massively based on `#PF` and thus considerably lowered performances. A basic SPT implementation could have been:

- VM cr3 is owned by the VMM
- VMM provides any page tables used by the CPU while the VM is running

– initially (and upon each cr3 write) every page tables entries are cleared
– on each `#PF`, the VMM walks through the VM page tables and fills in the SPTs accordingly
– the VMM modifies the final physical address to target the physical memory space dedicated to the VM leaving in the whole system memory space (system)

The figure **??** summarizes this mechanism.

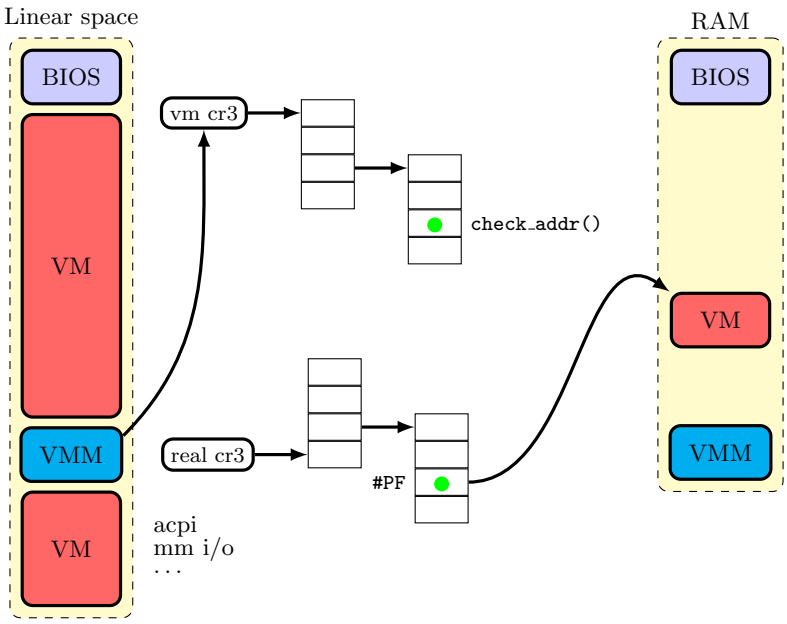

Fig. 2. Shadow Page Tables.

Recent CPUs provide MMU hardware assisted virtualization. The CPU is able, once having translated VM virtual addresses to VM physical addresses, to translate VM physical addresses to VMM physical addresses. This second level of page tables are called the Nested Page Tables (NPT). Since it is done by the CPU and does not rely on fault management, performances are really increased and the hypervisor complexity reduced (no complex TLBs, dirty/accessed bits management, ...).

These NPTs are generally configured once, except when mode transitions are operated (real to protected) under Ramooflax.

The figure **??** illustrates NPTs.

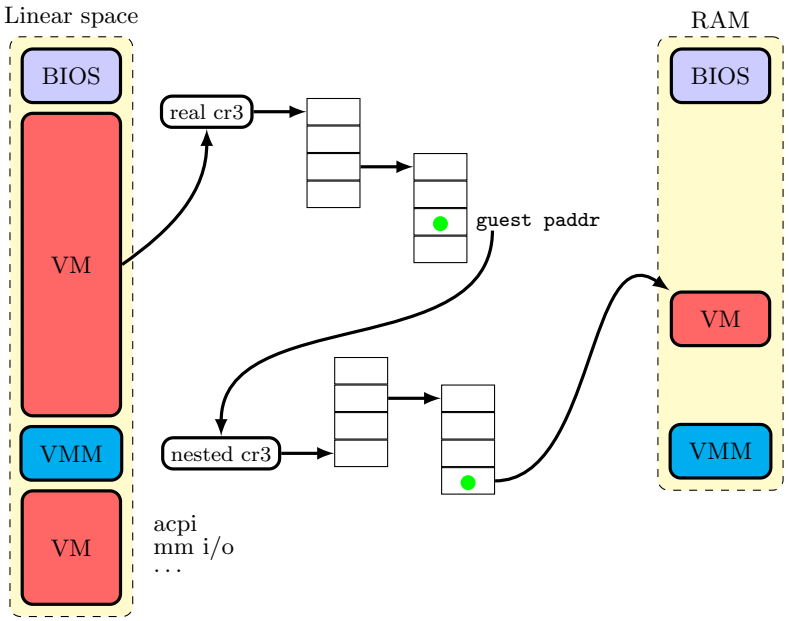

Fig. 3. Nested Page Tables.

3.5 (Un)-real mode management

Actually, hardware virtualization exists into x86 since the 80386. When protected mode first appeared, Intel wanted to offer conveniences to operating system developers in order to allow for real mode applications to still be able to run under protected mode kernels. A new mode appeared, the v8086.
This mode emulates the real mode mechanisms such as the ones involved into interrupts or far jumps, for tasks running in protected mode. It is feature-rich and especially allows interrupts and I/O redirection using bitmaps.
This can really be seen as hardware real mode virtualization. So that's why Intel recommends[13] the use of v8086 mode to manage VM running in real mode. This recommendation comes from the fact that it is not possible to disable paging and so protected mode under a VM as previously mentioned.
Under AMD, hardware virtualization comes with a brand new CPU mode called *paged real mode*, where it is possible to enable paging without enabling protected mode, which is under normal circumstances prohibited. The VM run easily in real mode and the hypervisor still benefits from memory accesses indirection thanks to paging.

[13] Intel Volume 3B Section 27.2

So under Intel, an hypervisor which wants to run VMs in real mode has to be able to manage v8086 tasks which is not a simple job. This is far more complex than AMD special *paged real mode*.
Last point, using v8086 mode on `vm-entry` is subject to numerous checks operated by the CPU, especially regarding segmentation registers setting.

Segmentation reminder Segment registers consist of a visible and an hidden part. The visible part, accessible to the developer, is called the selector. It has a length of 16 bits and its interpretation depends upon the CPU execution mode. The hidden part consist of a set of fields (base address, limit, attributs) defining properties when accessing memory using a given segment register.
In protected mode, a selector can be seen as an index into segment descriptor table (GDT, IDT, LDT). Each descriptor defining the fields which are to be loaded into the hidden part of the register when it is written.

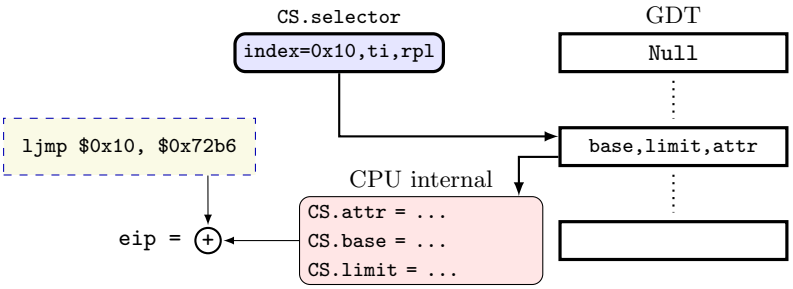

Fig. 4. Protected mode *far jump*.

In example, the `mov 0x1234, %ax` instruction retrieves the 2 bytes located at linear address `ds.base_address + 0x1234`. The segment base address coming from the descriptor whose index is stored into the `ds` selector.
In real mode, there is no such descriptor tables. The selector value is multiplied by 16 before being stored into the base address field of the hidden part. That's why one can only[14] access ∼1MB into real mode using 4 bits of segment selector and 16 bits of offsets. The limit (64KB) and attributs have default values.
The previous instruction will thus retrieve in real mode, the 2 bytes located at linear address `ds.selector<<4 + 0x1234`.
For obvious performance reasons, each time a selector is written, the CPU automatically fills in its hidden part.

[14] except when using A20 Gate

Unreal mode The hidden part of segment registers is normally ignored by real mode developers, because no *official* means allow to configure it. Thereby, the default base and limit values for segment hidden parts at CPU startup are respectively fixed to 0 and 64KB. As general purpose registers are 16 bits wide, segment offsets can't exceed 2^{16} bytes.
Nevertheless, real mode allows address prefix usage and it's then possible to access memory for instance using 32 bits general purpose register, generating 32 bits offsets into segments. However, since limit is fixed to 64KB any access beyond would raise `#GP`.
As it is possible to setup hidden segment register parts in protected mode, what happens when entering protected mode then returning to real mode ? Intel recommends reloading 16 bits segments with 64KB limit. However if it is not done and segment selectors are not rewritten once returned to real mode, the hidden part will remain the one configured while in protected mode.
Some real mode developers were fed up with memory addressing limits. As they didn't wanted to rewrite their code for protected mode they made use of this *internal cache* feature. While entering protected mode they set up base to 0 and limit to 4GB then return to real mode without reloading segment registers (at least data related ones `ds,es`). Using an address prefix on memory access they were able to access 4GB of memory while in real mode.
This *unreal mode* is still intensively used by BIOS developers.

Intel fail When a hypervisor resumes a VM in v8086 mode, the CPU will verify[15] that the base address of the segment is equal to the selector value multiplied by 16.
As a consequence, it is really tricky and some times impossible to resume an unreal mode VM using v8086. The following BIOS code excerpt illustrates the problem:

```
seg000:F7284          mov     bx, 20h
seg000:F7287          cli
seg000:F7288          mov     ax, cs
seg000:F728A          cmp     ax, 0F000h
seg000:F728D          jnz     short near ptr unk_7297

seg000:F728F          lgdt    fword ptr cs:byte_8163     (1)
seg000:F7295          jmp     short near ptr unk_729D
seg000:F7297          lgdt    fword ptr cs:byte_8169

seg000:F729D          mov     eax, cr0
seg000:F72A0          or      al, 1
seg000:F72A2          mov     cr0, eax                   (2)
seg000:F72A5          mov     ax, cs
seg000:F72A7          cmp     ax, 0F000h
```

At (1) and (2), the BIOS enters protected mode after loading a new GDT.

[15] Intel Volume 3B Section 23.3.1.2

```
seg000:F72AA          jnz     short near ptr unk_72B1
seg000:F72AC          jmp     far ptr 10h:72B6h          (3)
seg000:F72B1          jmp     far ptr 28h:72B6h

seg000:F72B6          mov     ds, bx                     (4)
seg000:F72B8          mov     es, bx
```

Once in protected mode, the BIOS initalize hidden parts of the `cs` segment register doing a *far jump* at (3). It also reloads data segment registers at (4).

```
seg000:F72BA          mov     eax, cr0
seg000:F72BD          and     al, 0FEh
seg000:F72BF          mov     cr0, eax                   (5)

seg000:F72C2          mov     ax, cs
seg000:F72C4          cmp     ax, 10h                    (6)
seg000:F72C7          jnz     short near ptr unk_72CE
seg000:F72C9          jmp     far ptr 0F000h:72D3h
seg000:F72CE          jmp     far ptr 0E000h:72D3h
```

It finally goes back to real mode at (5) and checks the value of `cs` at (6). In a non virtualized situation, `cs` should be equal to 0x10 whatever could be the base address of `cs`. However, using v8086 mode if the base address loaded into `cs` at (3) is not 0x100 then the hypervisor won't be able to resume this BIOS code. This is the same situation for reloaded data segment registers.
A solution could be to emulate real mode mechanisms in protected mode and do not make use of v8086 mode. This is a pretty complex task. Every segment register access should be trapped in order to emulate real mode *far call/jump*, *mov/pop seg*, *iret*. Interrupts mechanism should also be emulated as it is not the same as in protected mode. Moreover, Intel hardware virtualization does not provide any means to intercept segment registers accesses. One may imagine a way to trap them using a trick. A solution could be to force GDT and IDT limit to 0, thus leading to `#GP` on any segment register access and emulate the desired behavior.
Recently, Intel provided an *unrestricted guest* mode allowing an hypervisor to run VMs in real mode. Paging and protected mode being disabled, it is thus needed to have hardware MMU virtualization feature such as EPT to be able to protect hypervisor address space.

3.6 Event intercepts

Hardware virtualization extensions allow events intercepts and injection. This is valid for exceptions, hardware and software interrupts.
If the injection mechanism offers the finest granularity (vector number), the intercept mechanism is far less fine grain. Excepted for exceptions, interrupts intercepts are of *enable/disable* type.
With this approach, the hypervisor is constrained to intercept all of the raised interrupts, introducing an inacceptable latency, especially for the timer irq. One can imagine another tricky mechanism using for instance

a *shadow IDT* to filter only on desired vectors, but it's too bad that the CPU is able to do it for exceptions and not interrupts.
Notice that under AMD, hardware interrupts are kept *pending*. This means that the hypervisor is constrained to enable interrupts in its own code, then handle the interrupt in its IDT to detect which vector has been raised (providing the fact the interrupt controllers have the same mappings that the VM ones, which is the case in Ramooflax).
Intel is a bit more forgiving, doing or not (configurable) the *acknowledgement* cycle between the CPU and the interrupt controller, thus allowing the hypervisor to directly receive the IDT vector number on interrupt intercepts.
About software interrupts (`int xx`), it is not possible to intercept them under Intel while it is possible under AMD but still as an *on/off* mechanism.
It's funny to remember that 20 years ago, the v8086 mode was really feature-rich against modern hardware virtualization, allowing for interrupts intercepts using per vector bitmaps.

3.7 SMIs special case

Hardware virtualization allows for SMIs[16] intercepts whether under Intel or AMD.
Under AMD, intercepted SMIs are kept *pending* until the hypervisor enables interrupts in its context which leads to CPU entering SMM.
AMD recommends to not intercept SMIs in order SMM code to be able to handle them in any situation. Despite this, *Erratas*[17] in AMD CPUs force hypervisors to intercept SMIs.
SMIs can come from different sources such as interrupts or I/O operations. In this later case, AMD recommends[18] to *containerise* SMM before handling the *pending* SMI. In other words, to create a dedicated VM for the SMM code. But to be able to *containerise* SMM, you must have access to some BIOS locked MSRs.

4 Ramooflax organization

Ramooflax is developed in C and assembly language. The remote client has been written in python.
Ramooflax provides a simple configuration menu for the few present options, CPU manufacturer, control and debug device, proxy mode.
The proxy mode is used to intercept, log and emulate MSRs accesses for instance. The `cpuid` instruction is managed this way by default because the hypervisor needs to hide some features to the VM.
Below are some screenshots of the configuration menu:

[16] System Management Interrupts
[17] Errata 342, non intercepted SMIs lead to interrupts disabling in the VM[**?**]
[18] AMD Manual Vol.2 section 15.22.3

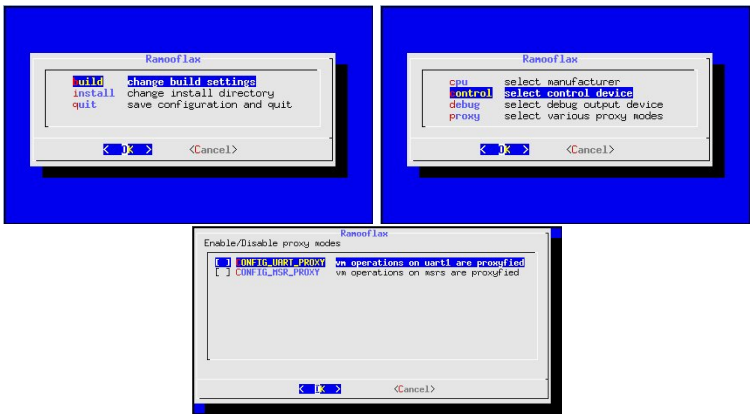

Fig. 5. The Ramooflax configuration menu.

Ramooflax code size is not that huge and is shown below:

```
riot(~) sloccount ramooflax
[...]
SLOC    Directory       SLOC-by-Language (Sorted)
11356   vmm             ansic=11093,asm=263
6767    include         ansic=6767
1898    common          ansic=1898
1443    setup           ansic=1362,asm=81
1391    client          python=1391
177     tools           sh=150,ansic=27
102     loader          ansic=92,asm=10

Totals grouped by language (dominant language first):
ansic:        21239 (91.81%)
python:        1391 (6.01%)
asm:            354 (1.53%)
sh:             150 (0.65%)
[...]
```

We can find the 3 kernels, the client and some tools. Each module is located in its own directory, excepted for common code pieces (drivers, libc). Each virtualization specific implementation is located under *src/svm* or *src/vmx*.

The final kernel (VMM) is organized in sub-systems, responsible for a specific feature:

```
riot(vmm) ls -l src/
total 32K
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 control
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 core
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 devices
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 disasm
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 drivers
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 libc
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 svm
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 vmx
```

5 Ramooflax initialization

As previously (section **??**) explained, Ramooflax is made of 3 minimalistic kernels but only one of them will be resident in memory, the vmm.

5.1 Ramooflax loader

This is the first kernel to run and it is loaded by a bootloader (ie GRUB). Multiboot[**?**] compliant, this 32 bits kernel is loaded at physical address 2MB. Its role is to enable longmode then load the setup.
It sets up a temporary GDT mixing 32 and 64 bits segments, an *indentity mapping* paging model (virtual equals physical address) using 1GB or 2MB pages depending upon CPU features.

5.2 Ramooflax setup

This is the configuration kernel. It is responsible for device initialization, especially debugging and control ones, as well as virtualization structures. It also loads the final vmm kernel by choosing its physical location. The idea behind Ramooflax was to have the minimum impact on VM native environment and avoid complex protection mechanism implementation. The vmm module being a PIE one can thus be relocated anywhere in physical memory and especially at the end of the RAM.
More precisely, the vmm is loaded at `size(RAM) - size(VMM)`. The RAM size is retrivied from GRUB. If the physical machine has more than 4GB installed, the hypervisor will always be relocated below the 4GB limit.
In order to reduce the amount of RAM, the setup prepares fake SMAPs, which are a kind of physical memory organization map provided by the BIOS, specifically crafted for the VM. The hypervisor will be in charge of VM SMAPs access interception to give it the fake ones.
Each SMAPs entry is composed of a base address, a size and a type defining the kind of memory area (ACPI, reserved, usable).
The setup only patches one entry from the original SMAPs, the one describing the first high memory chunk above 1MB and below 4GB. It is usually found right before ACPI entries. The patch consists in substracting the vmm area size from the original entry size.
The following logs, from a Linux kernel, detail BIOS SMAPs. The entry to be patched could have been the third one:

```
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009bc00 (usable)
 BIOS-e820: 000000000009bc00 - 00000000000a0000 (reserved)
 BIOS-e820: 0000000000100000 - 00000000dd04d400 (usable)
 BIOS-e820: 00000000dd04d400 - 00000000dd04f400 (ACPI NVS)
 BIOS-e820: 00000000dd04f400 - 00000000e0000000 (reserved)
 BIOS-e820: 00000000f8000000 - 00000000fc000000 (reserved)
 BIOS-e820: 00000000fec00000 - 00000000fec10000 (reserved)
 BIOS-e820: 00000000fed18000 - 00000000fed1c000 (reserved)
 BIOS-e820: 00000000fed20000 - 00000000fed90000 (reserved)
 BIOS-e820: 00000000feda0000 - 00000000feda6000 (reserved)
 BIOS-e820: 00000000fee00000 - 00000000fee10000 (reserved)
 BIOS-e820: 00000000ffe60000 - 0000000100000000 (reserved)
 BIOS-e820: 0000000100000000 - 0000000120000000 (usable)
```

We will detail the implemented mechanisms used to intercept VM SMAPs access.

The setup also prepares a simple physical memory pages allocator for future dynamic memory needs. Our hypervisor can't use virtualized OS services.

The GDT, IDT and paging structures are relocated at the end of the physical memory where the hypervisor is living. Notice that setup prepares a set of page tables for real and protected modes. Mode transitions are really common during VM boot process. For performance reasons we preferred to fix only one PML4 entry and provide specific mode page tables.

The VM Nested Page Tables are set up to exclude physical memory area of the hypervisor. Related entries are set as non present. Other entries are configured as *identity mapping*, VM physical addresses are the same as system ones.

Finally, the setup configures virtualization related data structures. The I/O and MSRs interception bitmaps are prepared for keyboard, ps2 system controllers and EFER. The control registers, sensitive instructions like `cpuid, htl, intn` and all virtualization related instructions are also intercepted.

By default, exceptions and hardware interrupts are not intercepted.

The setup finishes its execution by installing the first VM instructions in conventional memory: `int 0x16` and `int 0x19`. The first one is a BIOS service which allows to wait for a keystroke. The second one tells the BIOS to load the bootsector of its first bootable device which uses to be an hard drive where the native operating system is already installed.

By doing this, we take benefit of existing BIOS features (devices access like USB, SATA, ...). The hypervisor seamlessly virtualizes real mode code whether it is BIOS or not.

6 Ramooflax execution model

The hypervisor waits for `vm-exit` that will give it the opportunity to execute its treatments. The figure **??** details the execution paths architecture of the hypervisor.

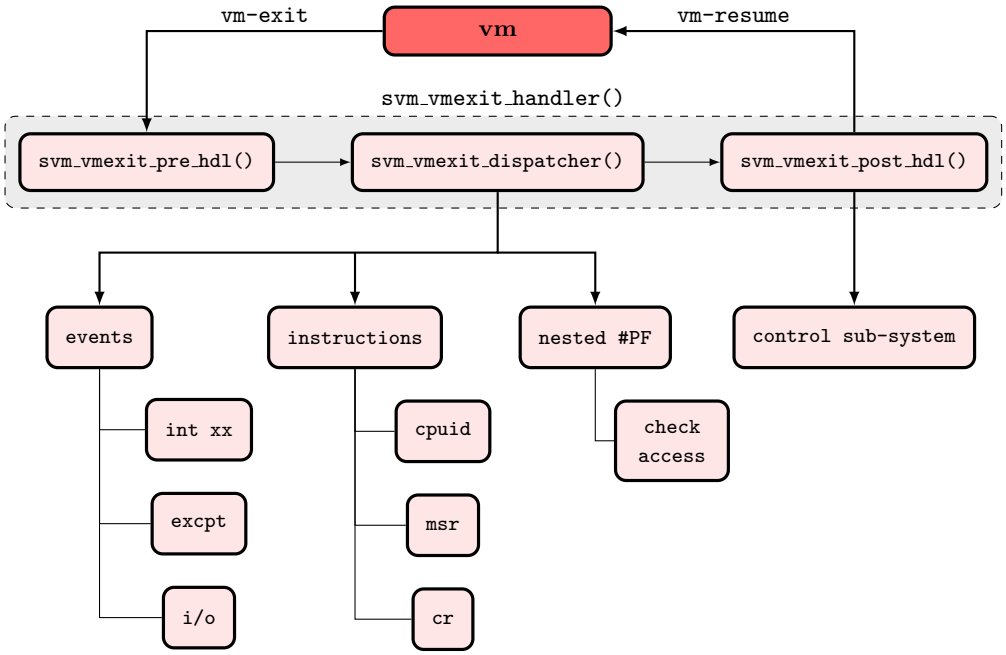

Fig. 6. Hypervisor execution paths.

```
void svm_vmexit_handler(raw64_t tsc)
{
  svm_vmexit_pre_hdl();
  svm_vmexit_dispatcher())
  svm_vmexit_post_hdl(tsc);
}

void svm_vmexit_pre_hdl()
{
  vmcb_ctrls_area_t *ctrls = &info->vm.cpu.vmc->vm_vmcb.ctrls_area;
  vmcb_state_area_t *state = &info->vm.cpu.vmc->vm_vmcb.state_area;

  svm_vmsave(&info->vm.cpu.vmc->vm_vmcb);
  info->vm.cpu.gpr->rax.raw = state->rax.raw;
  info->vm.cpu.gpr->rsp.raw = state->rsp.raw;

  if(ctrls->tlb_ctrl.tlb_control != VMCB_TLB_CTL_NONE)
     ctrls->tlb_ctrl.tlb_control = VMCB_TLB_CTL_NONE;
}

void svm_vmexit_post_hdl(raw64_t tsc)
{
  vmcb_state_area_t *state = &info->vm.cpu.vmc->vm_vmcb.state_area;

  vmm_ctrl();
  db_post_hdl();

  state->rax.raw = info->vm.cpu.gpr->rax.raw;
  state->rsp.raw = info->vm.cpu.gpr->rsp.raw;
  info->vm.cpu.gpr->rax.raw = (offset_t)&info->vm.cpu.vmc->vm_vmcb;

  info->vmm.ctrl.vmexit_cnt.raw++;
  svm_vmexit_tsc_rebase(tsc);
}
```

We can see a pre-processing which is architecture specific, then a call to the subsystem responsible for the raised `vm-exit`, followed by a post-processing which checks if the remote client wants to interact with the hypervisor before resuming the VM.

7 System registers filtering

7.1 Control Registers

Accesses to cr0, cr3 and cr4 registers are intercepted because they are responsible for sensitive settings such as mode transitions, TLBs control and different options related to paging and cache consistency.
About TLBs management, the hypervisor must keep the standard CPU behavior which is:

- a write to cr3 leads to non-global TLBs flush[19]
- a change to cr4 PAE, PSE or PGE bits leads to full TLBs flush

Since Nested Page Tables, TLBs are tagged with an ASID[20], which allows the hypervisor to flush only those related to the VM. Once again, these features are not found into every CPUs shipped with hardware virtualization extensions.
Another interesting point is the cache consistency via CD bit from cr0. The Linux kernel for instance sets it (cache disable) while booting and disables MTRRs in order to not reference caches anymore. Our hypervisor running as seamlessly as possible, undergoes MTRRs disabling (rather emulating it). To be consistent, the hypervisor must follow the VM settings of cr0 related cache settings.
The following hypervisor log illustrates a classical cache disabling scheme under Linux:

```
<0x67aaf:0xc1013d68:124>rdmsr 0x2ff | 0x0 0xc00
<0x67ab0:0xc1013d68:124>rdmsr 0xc0010010 | 0x0 0x160600
<0x67ab1:0xc1013d32:16>cache disable
<0x67ab2:0xc1013d54:137>wbinvd
<0x67ab4:0xc1013d68:124>rdmsr 0x2ff | 0x0 0xc00
<0x67ab5:0xc1013d79:124>wrmsr 0x2ff | 0x0 0x0
<0x67ab5:0xc1013d79:124>disabling mtrr
```

The logs are generated using the hypervisor *proxy msr* mode. Linux disables caches using cr0, flush CPU cache lines and finally disables MTRRs. All the memory is now *uncacheable* and every line is invalidated. If the hypervisor, while intercepting write to cr0, doesn't disable its own cr0 cache setting, it may still fill cache lines after the VM does the writeback. Thus when MTRRs will be disabled, information will be lost and the hypervisor will enter an inconsistent state.
The use of Nested Page Tables implies a punctilious PAT and MTRRs management. Rather emulating it, the hypervisor follows VM settings.

[19] Those whose page table entries don't have `global` bit set.
[20] Address Space IDentifier

7.2 CPUID and MSRs

The hypervisor has a *passthrough* mode where it lets full access to a ressource to the VM. It can also run in *proxified* mode where it must ensure execution of ressource access.
A `wrmsr` implies:

- msr filtering depending on architecture
- emulate access if the msr is stored into the VMCS/VMCB
- native execution on the contrary

The interception of `cpuid` or `rdmsr` is handled as:

- native execution or read into VMCS/VMCB
- post-processing to filter out information

```
static int __resolve_cpuid()
{
  uint32_t idx = info->vm.cpu.gpr->rax.low;

  __resolve_cpuid_native();  /* native execution */
  __resolve_cpuid_arch(idx); /* amd/intel centric post-processing */

  /* generic post-processing */
  switch(idx)
  {
    case CPUID_FEATURE_INFO:
      __resolve_cpuid_feature();
      break;
    default:
      break;
  }

  return CPUID_SUCCESS;
}
```

8 Events filtering

8.1 Software interrupts

Software interrupts intercept is only performed when the VM is in real mode. It allows specific filtering of BIOS services requests mainly related to SMAPs and GateA20 via `int 0x15`. The interception leads to the emulation of the service if needed, or redirect real mode code execution to the correct IVT handler.

8.2 Hardware interrupts

Hardware interrupts are not intercepted. But it is possible to do so. The hypervisor can be interruptible, under AMD, in order to detect which IDT vector has been raised and need to be injected to the VM.

8.3 Exceptions

Under AMD, where sofware interrupts intercept is available, exceptions management is rather simple. The hypervisor only checks that the raised exception is not related to the control subsystem (`#DB` and `#BP` for breakpoints and single-stepping). If it is a leggit VM exception, it is injected. Under Intel, which does not allow software interrupts intercept, a specific exception handler must be implemented into the hypervisor to deal with software interrupts intercept emulation trick based on `#GP` exceptions.

8.4 Input/Output operations

The I/O intercepts is operated via a bitmap (one bit per port). On a `vm-exit`, the CPU provides information related to the I/O operation, direction, size, whether it was a string operation or not, the port. The hypervisor can thus emulate or proxify the operation requested by the VM.
Notice that some I/O operations emulation can be very complex and dangerous. Especially the *string* ones. They are not fully supported into Ramooflax.

9 Emulation

9.1 Instructions

For the time being, the hypervisor embeds a disassembly engine (udsi86[**?**]) giving precious help to the emulation subsystem. Under AMD, the need for a disassembler is far less consequent than under Intel. We could have skip it because emulated instructions have a simple encoding: `mov to/from cr, intn, clts`.
Instruction emulation is sensitive under x86 because of the numerous protected mode mechanisms. As for sensitive instructions intercepted and then natively executed, the emulation engine must take into account the `vm-exit` context, especially the TF bit from rflags. If it is set, the hypervisor should inject, right after emulating the instruction, a `#DB` exception. By the way it is a simple detection mechanism over hypervisors, as mentioned into [**?**].
If stealthness is not the main objective of Ramooflax, it has to be consistent with regard to the virtualized system.

9.2 Devices

The hypervisor offers some devices emulation: UART, PIC, KBD and PS2 system controllers.
The UART emulation allows to simply retrieve kernel logs written to a serial port and redirect it into the debugging interface used by the hypervisor. A *passthrough* mode is not sufficient due to the fact that the UART settings can differ from VM to hypervisor (parity, speed, ...). Emulation allows fine grain control.

The KBD and PS2 system controllers emulation was needed, partially, to prevent the VM from rebooting the system. Historically, some bits lurk into these controllers that allow a hard reset. They also control GateA20 enabling/disabling.

10 Remote communication

The hypervisor lets the VM directly access devices, except those which are used by the hypervisor, as explained in figure **??**.

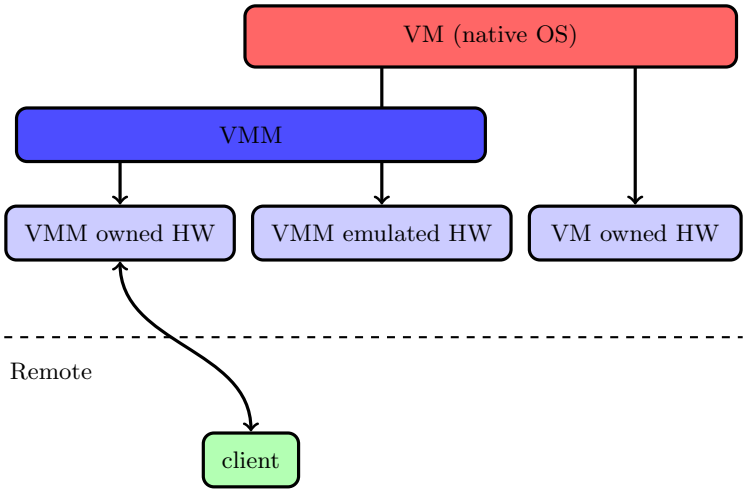

Fig. 7. VMM/VM/Client devices access.

The hypervisor distinguishes its debug from its control interface. The debug interface is used for writting debug logs, while control one is used to remotly interact with the client.

10.1 UART

The serial port is used as a debug interface. It is slow, unreliable and it can hardly be found nowadays in modern workstations.

10.2 EHCI Debug port

The USB EHCI specification tells that a physical USB port can be used as an EHCI Debug port if the controller allows it. Most of the EHCI controllers have the feature implemented.

The Debug port interest comes from the fact that it is standardized, easy to control (as opposed to classical USB) and faster than serial port (480 Mbits/s).

On hypervisor side The hypervisor implements a Debug port driver on the EHCI controller side. The VM loses a physical USB port. The hypervisor ensures the VM will never get this port back under its control.

On remote client side The main problem comes from the USB specification. It does not allow two host controllers to directly exchange data between each other. To exchange data with a Debug port, you need a Debug Device. One can find it on the market which can expose a serial USB device on the remote client.
One can also take benefit of embedded development boards or smartphones providing USB OTG[21] controllers which can run as a host or device controller. That's how many smartphones are able to act as mass storage devices when connected to a host controller on a PC, simply because the device controller is able to emulate a mass storage device.
Under Linux, an API is available to developpers in order to tell device controllers to emulate any usb device: the Gadget API.
We thus developped a USB Gadget emulating a Debug Device while exposing a serial interface to the userland. This gadget is now part of the official Linux kernel branch since 2.6.36 release.

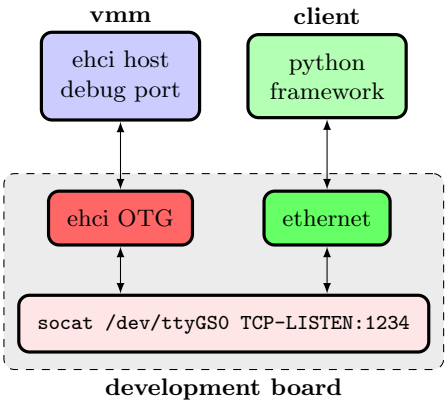

Fig. 8. VM control via EHCI Debug port.

The hypervisor is connected through USB Debug Port to a development board with a device controller running the Debug Device Gadget. The

[21] USB On-the-Go

exposed serial interface (`ttyGS0`) is forwarded into a TCP connection on the port 1234 using `socat`. A remote client can thus access the hypervisor using a network connection.

11 Remote interaction

As previously mentioned, the hypervisor waits for events coming from the VM whether they are leggit or related to remote client interaction.

11.1 Taking control

The critical point for remote control is to ensure that the hypervisor will be able to take over the VM when the remote client decides it.
Under Intel, a specific timer called *vmx_preemption_timer* has been recently implemented. It allows `vm-exit` to be raised after a number of CPU cycles. Under AMD, there is no such feature.
For the time being, the hypervisor computes a ratio at each `vm-exit` to determine whether it should check its controlling interface or not. The idea is to raise as many `vm-exit` as possible in order to guarantee the take over as instantly as possible. On the other hand, the more `vm-exit` you have, the less reactive the VM becomes.
The hypervisor uses an heuristic related to modern OS principles: the use of cr3. Modern operating systems such as Linux and Windows schedule processes on a regular basis, which implies a write to cr3. Since the hypervisor intercepts writes to cr3, it can check its controlling interface regularly.
Notice that no interrupt is raised for the Debug port, which prevents us from intercepting interrupts on remote client requests. However, since interrupts are a global setting (all of them are intercepted), it should have introduced inacceptable latency to the VM.

11.2 GDB stub

The control subsystem is implemented under the form of a GDB stub. This allows traditional gdb clients to connect to the hypervisor.
The hypervisor implements the basic GDB protocol commands:

- read/write general purpose registers
- read/write memory
- add/remove software/hardware breakpoints
- single-stepping

The GDB protocol is well designed for userland processes. However, when dealing with a kernel or a VM, it shows its limits. As an example, connecting to the VMware GDB stub usually brings the analyst into kernel code and attaching to a specific process is a really inconvenient task using classical GDB protocol.
That's why we decided to implement extensions to the GDB protocol in order to be more convenient.

11.3 GDB specific extensions

System registers access The first limitation was that we can't access system registers. We offer the possibility to read/write:

- cr0, cr2, cr3, cr4
- dr0-dr3, dr6, dr7
- efer, dbgctl MSRs
- cs, ss, ds, es, fs, gs (base address only)
- gdtr, idtr, ldtr, tr

Memory access The read/write memory feature from the classical GDB protocol does not really make sense under a VM. From an application point of view, the GDB stub has only access to the virtual space of the currently debugged process. Under an hypervisor, the remote client may want to access physical or virtual addresses depending on the execution mode of the VM (real, protected, ...). When accessing virtual addresses for a specific process, the hypervisor must access physical memory using the process page tables.
Our extension allows to read/write physical and virtual memory. When using virtual addresses, the hypervisor does make use of the current cr3. We also provide a special cr3 tracking feature. The client is able to tell the hypervisor to work with a specific cr3 when accessing memory. This is usefull when installing software breakpoints into a process userspace. We also offer a translation service (virtual to physical).

Last Branch Record Amongst the numerous features provided by the x86 architecture, the LBRs one is really interesting. It allows branch recording into the following MSRs:

- `from_eip`, eip value before branch
- `to_eip`, branch target
- `last_excp_from`, eip value before exception is raised
- `last_excp_to`, exception branch target

We allow enabling/disabling of the feature into Ramooflax. Notice that AMD provides LBRs virtualization by means of VMCB backed MSRs. This is really appreciable when dealing with bugged BIOS that does not restore LBRs enabled setting upon SMM resuming.

Virtualization control We also provide some features to control over virtualization settings from VMCS/VMCB. They are not complete but should ultimately allow full control over virtualization extensions. To date, we only allow interception bitmaps modification related to exceptions and control registers.

11.4 The art of single-stepping

The different single-stepping scenarii while debugging a virtual machine can be really complex to handle, especially on privilege level transitions. We can summarize them as follows:

– global single-step
– ring3 only, specific process single-step
– rin0-ring3, specific process single-step
– kernel thread single-step

The Ramooflax single-stepping implementation is merly based upon TF bit from rflags and the interception of `#DB`. We have only implemented global and ring3 specific process single-stepping.
Single-stepping a process is pretty easy to handle independently from the operating system running under the hypervisor, mainly because we can identify a process from its cr3 register or its kernel stack pointer (tss.esp0). However, it is far more complex to identify a kernel thread. From an hardware point of view, only the kernel stack of the kernel thread is changed upon scheduling. At the same privilege level (scheduling from and to a kernel thread), only the general purpose registers will be updated (esp/ebp) and this is not feasible to intercept write access on them.
Notice that Ramooflax can not ensure a consistent single-stepping state at any time. When single-stepping a process, the hypervisor does not single-step into the process kernel control path (scheduling, interrupt handlers, syscalls, ...), and thus disable single-stepping. The hypervisor will wait for a re-scheduling of the correct cr3 and thus its rflags register which will lead to `#DB`. The kernel can kill the process and never re-schedule it. The hypervisor will never know about it.
About its stealthness, but also its consistency, the hypervisor must intercept TF bit related instructions:

– `pushf` in order to hide TF to the VM
– `popf, iret` in order to prevent the VM from modifying TF
– `intN`, exceptions and hardware interrupts, in order to preserve TF setting

The hypervisor also uses single-stepping internally to restore software breakpoints. This never leads to remote client interaction.

12 Remote client

We have chosen to develop a python API to remotely control the hypervisor. We detail the different elements of the API as well as some usage examples.
The purpose of this section is to illustrate hypervisor features and the simplicity of its services access. Already existing frameworks, like Metasm[**?**], could have been used with Ramooflax. The one we developped is only illustrative.

12.1 Python framework elements

The API provides some easy to use classes:

– VM, providing high level features
– CPU, allowing registers and filters access
– Breakpoints, ... self explained
– GDB, a GDB client providing hypervisor specific extensions
– Memory, controlling memory access
– Event, allowing developers to implement their own `vm-exit` handlers

12.2 VM

At the highest level, it provides the following services: run, interact, singlestep, resume, stop, attach, detach.
We thought it could be interesting to be interactive and scriptable. The interactive mode allows a scapy-like[**?**] interactive python shell, while the scriptable mode is good for automating analysis tasks.
The VM class is instancied as:

```
vm = VM(CPUFamily.AMD, 32, "192.168.254.254:1234")
```

We have the CPU model, the target address and port of the development board. Controlling the hypervisor via serial interface has not been implemented on the client side. But the hypervisor part should work.
The interactive mode is used as:

```
vm.run(dict(globals(), **locals()))
```

It is entered/leaved with `ctrl+d, ctrl+c`. The scriptable mode needs some additional steps:

```
vm.attach()  # remote connection
vm.stop()    # stop the vm

# xxxx (breakpoints, filters, ...)

vm.resume()  # resume the vm and wait for next vm-exit
vm.detach()  # detach so that the vm gets control back
```

12.3 CPU, Memory and Breakpoints

These classes give access to system and general purpose registers, exceptions management, breakpoints. Notice that registers modification is lazily operated (upon vm resume).
Following are some examples installing breakpoints and reading memory:

```
# data write breakpoint
vm.cpu.breakpoints.add_data_w(vm.cpu.sr.tr+4, 4, filter, "esp0")

# physical memory read
xx = vm.mem.pread(0xa0000, 12)

# enabling a specific cr3 for translations
# then reading a virtual memory page
vm.cpu.set_active_cr3(my_cr3)
pg = vm.mem.vread(0x8048000, 4096)
```

The breakpoint is related to the `esp0` field from the TSS pointed to by the TR register. We will explain later the use of *filter*. We can name breakpoints (here *esp0*). It is also easy to list installed breakpoints:

```
>>> vm.cpu.breakpoints
esp0 0xc1331f14 Write (4)
kernel_f1 0xc0001234 eXecute (1)
```

As well as system registers (and general purpose ones):

```
>>> vm.cpu.sr
cr0     = 0x000000008005003b
cr2     = 0x00000000b7681ed0
cr3     = 0x00000000371f9000
cr4     = 0x0000000000000690
dr0     = 0x0000000000000000
dr1     = 0x0000000000000000
dr2     = 0x0000000000000000
dr3     = 0x0000000000000000
dr6     = 0x00000000ffff0ff0
dr7     = 0x0000000000000400
dbgctl  = 0x0000000000000000
efer    = 0x0000000000001000
cs      = 0x0000000000000000
ss      = 0x0000000000000000
ds      = 0x0000000000000000
es      = 0x0000000000000000
fs      = 0x0000000000000000
gs      = 0x00000000c1367c00
gdtr    = 0x00000000c132e000
idtr    = 0x00000000c132d000
ldtr    = 0x0000000000000000
tr      = 0x00000000c1331f10
```

12.4 Event

Rather implemeting the counter-intuitive conditional breakpoints syntax from the GDB protocol, we decided to implement a *callback* mechanism based on filters that can be linked to any `vm-exit`.

This approach allows for implementing anything interesting in python, from conditional breakpoints to complex memory analysis functions.

These filters dissociate elements which are hardware dependent (and provided by the framework) from those which are software dependent (specific to the virtualized operating system).

Thus, most of the previous classes services give the opportunity to define filters associated to a python function.

The `vm.resume()` method gives control back to the VM and once a `vm-exit` is raised, directly calls the corresponding filter and forwards its return value.

A usage example could be to return `True` when the filter wants to enter interactive. The following code illustrates it. It enters interactive mode when a `#PF` is raised by the instruction located at `0x1234`:

```
def handle_excp(vm):
   if vm.cpu.gpr.eip == 0x1234:
     return True
   return False

vm.cpu.filter_exception(CPUException.page_fault, handle_excp)

while not vm.resume():
   continue

vm.interact(dict(globals(), **locals()))
```

The annexe **??** shows a script that retrieves a specific process *page directory* given its name, under Linux 2.6.

13 Conclusion

Although hardware virtualization extensions ease hypervisor implementation, its development still remains complex and sensitive. If Ramooflax is far from being a finished product, it provides to date sufficient features to try complex operating systems analysis in native environment.
About the actual limitations, especially regarding SMM, open-source BIOS could in the long term allow for a better virtualization of native systems. The exploration of ACPI tables and all of their subtleties could also be an ideal application field for Ramooflax.

14 Annexe: `process_finder`

The principle is to install a filter on writes to cr3 . On each write, the hypervisor takes control over the VM and contacts the remote client. The API will call the installed filter.
The filter is responsible for inspecting the kernel stack of the last scheduled process (`tss.esp0`), retrieving its `thread_info` then its `task_struct` and its `mm_struct` to reach the `pgd`. If the `comm` field is the desired one, the filter returns `True`.
Notice that we prefer to walk only the process list on the first write to cr3, because nothing can ensure that all of the processes will be scheduled each time our filter is called. This comes from the fact that the latency introduced by the analysis can bring the virtualized kernel to strategically re-schedule highest priority processes leading to a small subset of the process list being scheduled.

```
#!/usr/bin/env python

import sys
from vm import *

if len(sys.argv) < 2:
    print "need prog name"
    sys.exit(-1)

process_name = sys.argv[1]
process_cr3 = 0

# Some offsets for debian 2.6.32-5-486 kernel
com_off = 540
next_off = 240
mm_off  = 268
pgd_off = 36

def next_task(vm, task):
    next = vm.mem.read_dword(task+next_off)
    next -= next_off
    return next

def walk_process(vm, task):
    global process_cr3
    head = task
    while True:
        mm = vm.mem.read_dword(task+mm_off)
        if mm != 0:
            comm = task+com_off
            name = vm.mem.vread(comm, 15)
            pgd  = vm.mem.read_dword(mm+pgd_off)
            print "task",name
            if process_name in name:
                process_cr3 = pgd - 0xc0000000
                print "===> task cr3",hex(process_cr3)
                return True

        task = next_task(vm, task)
        if task == head:
            return False

def find_process(vm):
    esp0 = vm.mem.read_dword(vm.cpu.sr.tr+4)
    thread_info = esp0 & 0xffffe000
    task = vm.mem.read_dword(thread_info)
    if task == 0:
        return False
    return walk_process(vm, task)

# Main (architecture dependent only)
vm = VM(CPUFamily.AMD, 32, "192.168.254.254:1234")

vm.attach()
vm.stop()
vm.cpu.filter_write_cr(3, find_process)

while not vm.resume():
    continue

vm.cpu.release_write_cr(3)

print "success"
vm.detach()
```

References

1. Intel virtualization roadmap
   http://software.intel.com/file/1024
   http://www.xen.org/files/xensummit_4/VT_roadmap_d_Nakajima.pdf
2. OpenGL tutors (N. Robins)
   http://www.xmission.com/~nate/tutors.html
3. bluepill (J. Rutkowska)
   http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html
4. vitriol (DDZ matasano)
   http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf
5. virtdbg (D. Aumaitre, C. Devine)
   http://code.google.com/p/virtdbg/
6. hyperdbg (A. Fattori)
   http://code.google.com/p/hyperdbg/
7. abyss (Ivanlef0u)
   http://www.ivanlef0u.tuxfamily.org/?p=120
8. multiboot specification
   http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
9. udis86, disassembler library for x86 and x86-64 (V. Thampi)
   http://udis86.sourceforge.net
10. Detecting simple hypervisors (N. Falliere)
    http://0x5a4d.blogspot.com/2009/11/detecting-simple-hypervisors.html
11. Revision Guide for AMD Family 10h Processors (AMD)
    http://support.amd.com/us/Processor_TechDocs/41322.pdf
12. The METASM assembly manipulation suite (Y. Guillot)
    http://metasm.cr0.org/
13. Metasm HowTo: bintrace (A. Gazet)
    http://esec-lab.sogeti.com/dotclear/index.php?post/2010/07/19/90-metasm-howto-bintrace
14. Scapy: a powerful interactive packet manipulation program (P. Biondi)
    http://http://www.secdev.org/projects/scapy