Ramooflax

das pre-boot übervisor

Stéphane Duverger

EADS Innovation Works
Suresnes, FRANCE

SSTIC Juin 2011

Introduction

On souhaitait disposer d'un outil

- permettant analyse/contrôle d'un système complet (bios, kernel, ...)
- pour machines **physiques** de type PC (x86 32 et 64 bits)
- avec un minimum de dépendances logicielles

On souhaitait disposer d'un outil

- permettant analyse/contrôle d'un système complet (bios, kernel, ...)
- pour machines **physiques** de type PC (x86 32 et 64 bits)
- avec un minimum de dépendances logicielles

Solution envisagée

- un hyperviseur à VM unique
- pilotable à distance
- type 1 (*bare metal*)
  - isolation plus simple
  - rendre vision contrôlée du matériel
  - indépendance logicielle
  - nécessité de démarrer avant la VM

Paysage des hyperviseurs disponibles

Solutions courantes

- VirtualBox, KVM: inadaptées, type 2 (*hosted*)
- Xen: trop complexe à déployer/adapter

Paysage des hyperviseurs disponibles

Solutions courantes

- VirtualBox, KVM: inadaptées, type 2 (*hosted*)
- Xen: trop complexe à déployer/adapter

Solutions minimalistes

- type bluepill, vitriol, virtdbg, hyperdbg ...
- trop intrusives, virtualisation *in vivo* non acceptable
- dépendance trop forte à l'OS virtualisé

Paysage des hyperviseurs disponibles

Solutions courantes

- VirtualBox, KVM: inadaptées, type 2 (*hosted*)
- Xen: trop complexe à déployer/adapter

Solutions minimalistes

- type bluepill, vitriol, virtdbg, hyperdbg ...
- trop intrusives, virtualisation *in vivo* non acceptable
- dépendance trop forte à l'OS virtualisé

repartir de zéro !

Un hyperviseur *standalone* minimaliste

Caractéristiques souhaitées

- simple, léger, peu d'impacte sur les performances natives
- basé sur Intel-VT (vmx) et AMD-V (svm)
- tirer profit de l'existant (BIOS)
- mécanismes simples dans une brique complexe (vmm)
- déléguer complexité fonctionnelle de l'analyse (client distant)

Un hyperviseur *standalone* minimaliste

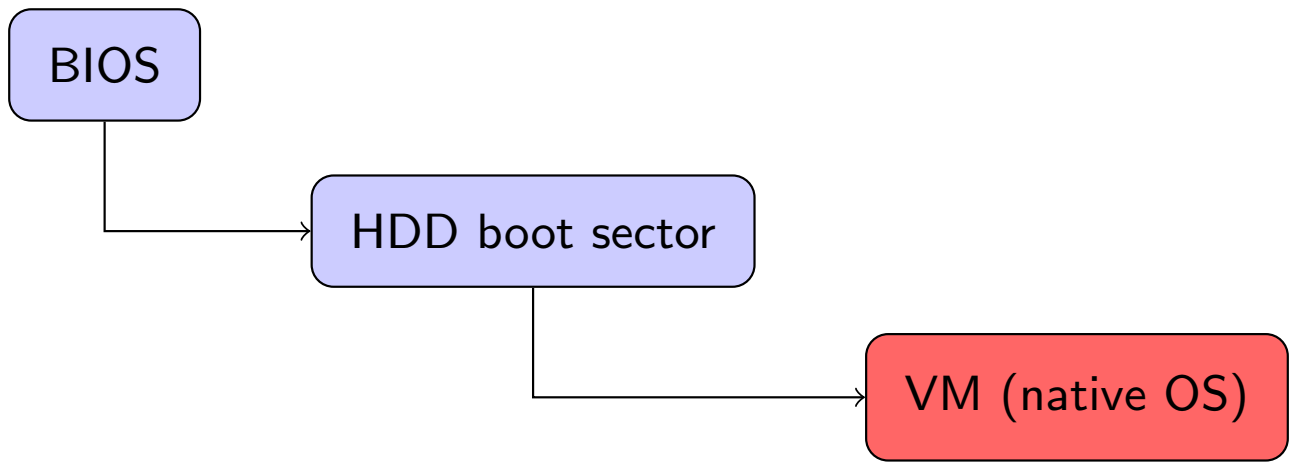
Caractéristiques souhaitées

- simple, léger, peu d'impacte sur les performances natives
- basé sur Intel-VT (vmx) et AMD-V (svm)
- tirer profit de l'existant (BIOS)
- mécanismes simples dans une brique complexe (vmm)
- déléguer complexité fonctionnelle de l'analyse (client distant)

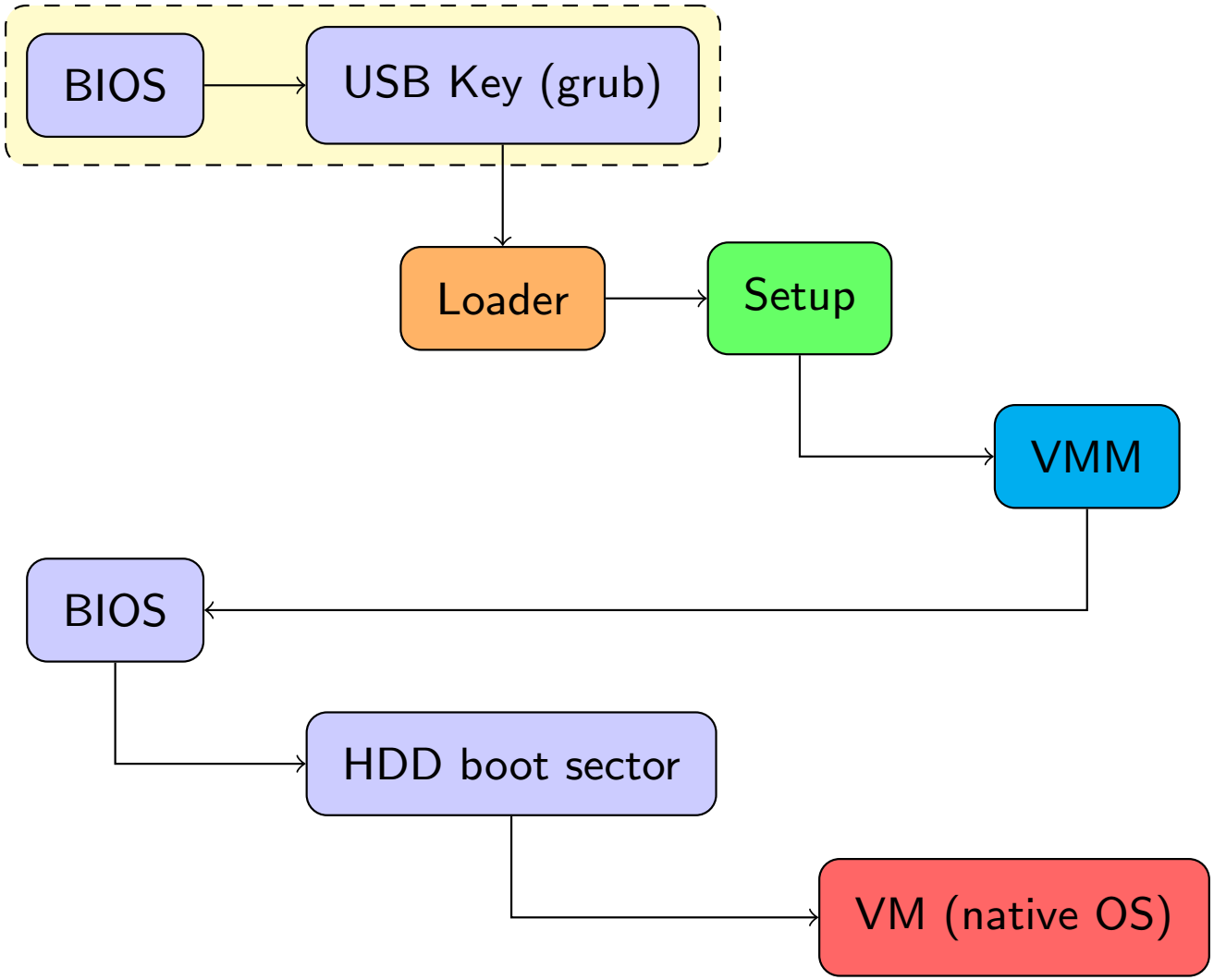
Cible des processeurs *stuffés*

- dépendance aux extensions *récentes* de virtualisation matérielle
- plus particulièrement Intel EPT et AMD RVI
  - code plus simple
  - performances accrues
  - surface d'attaque réduite

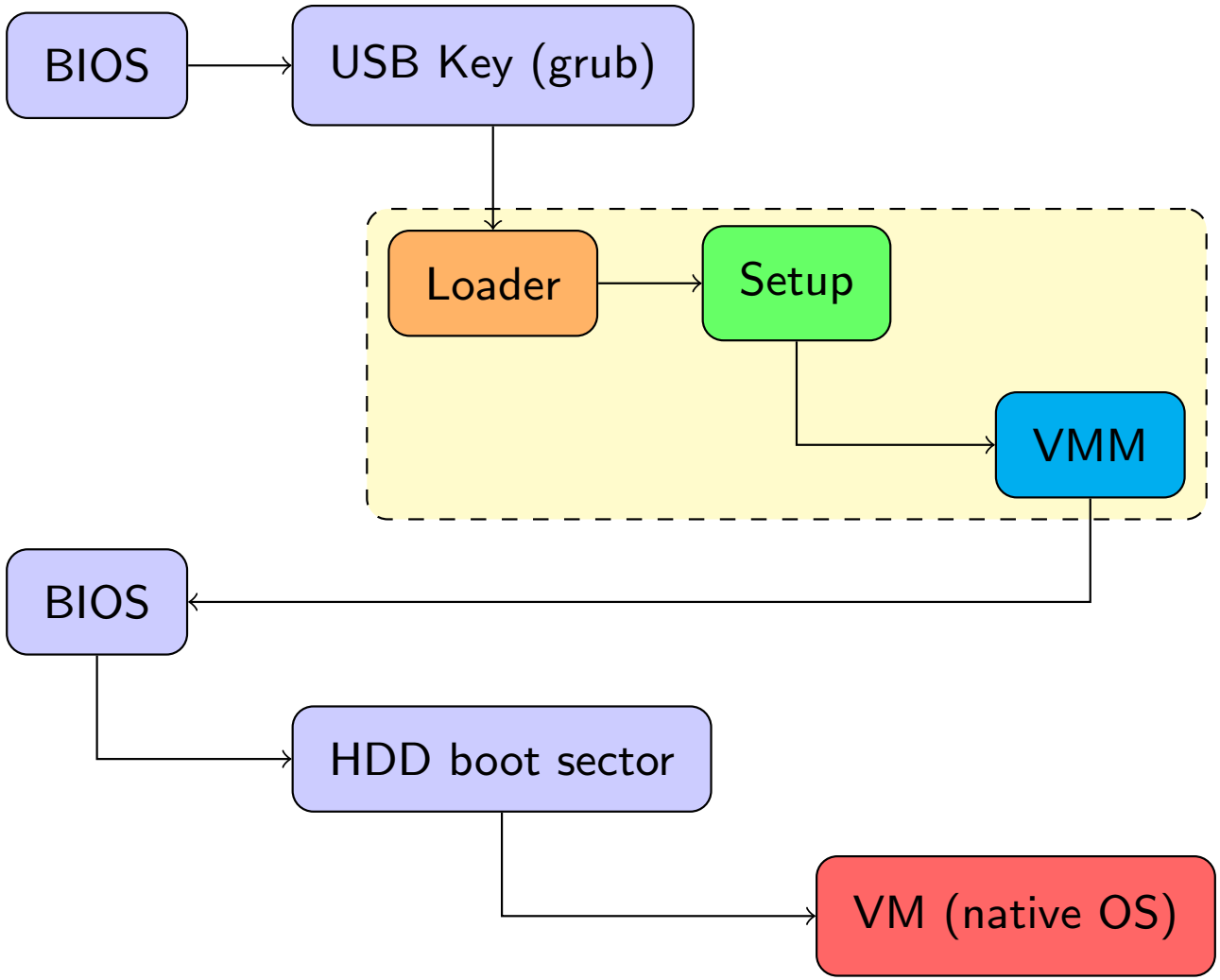
Chaîne de démarrage classique



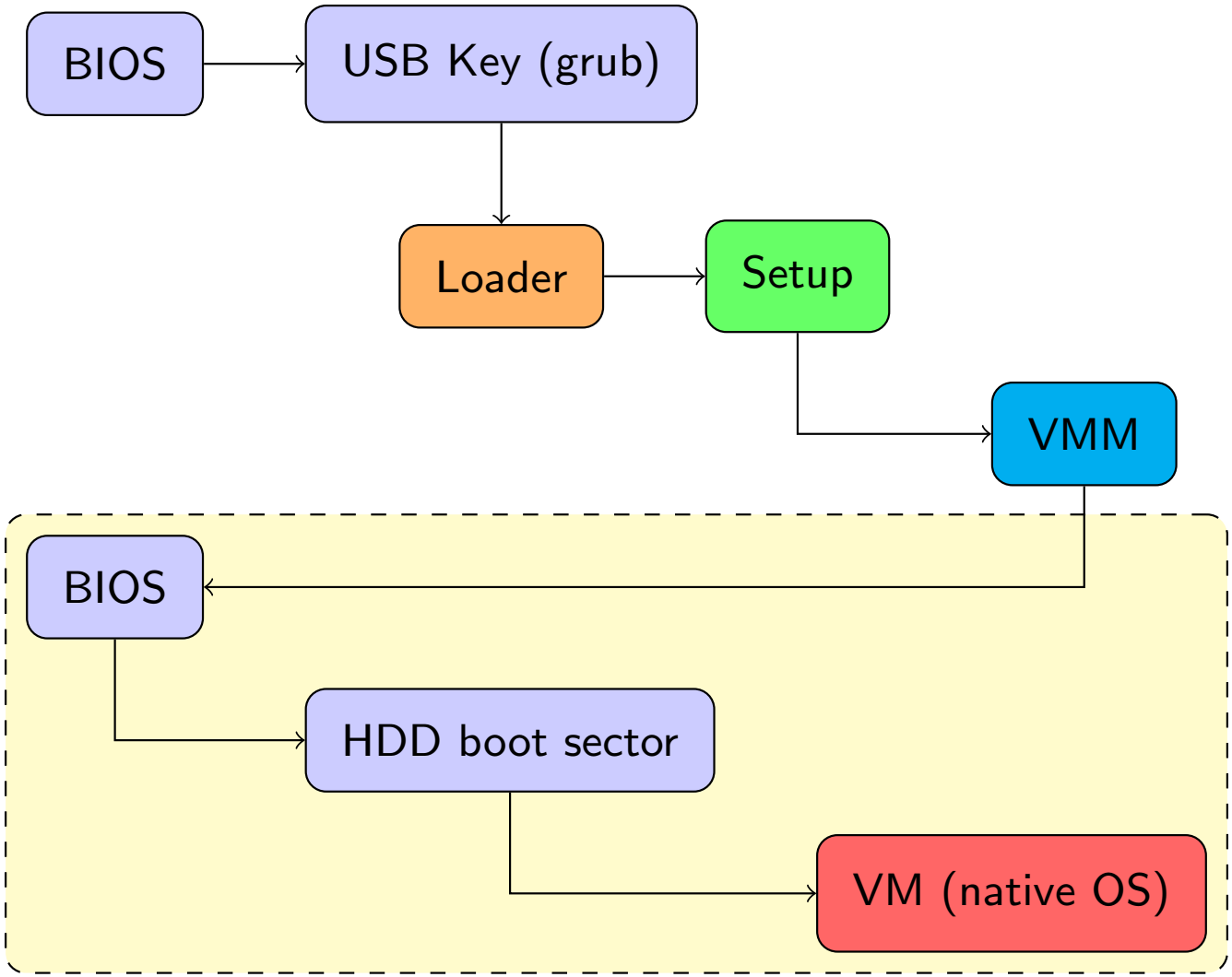
Chaîne de démarrage de Ramooflax



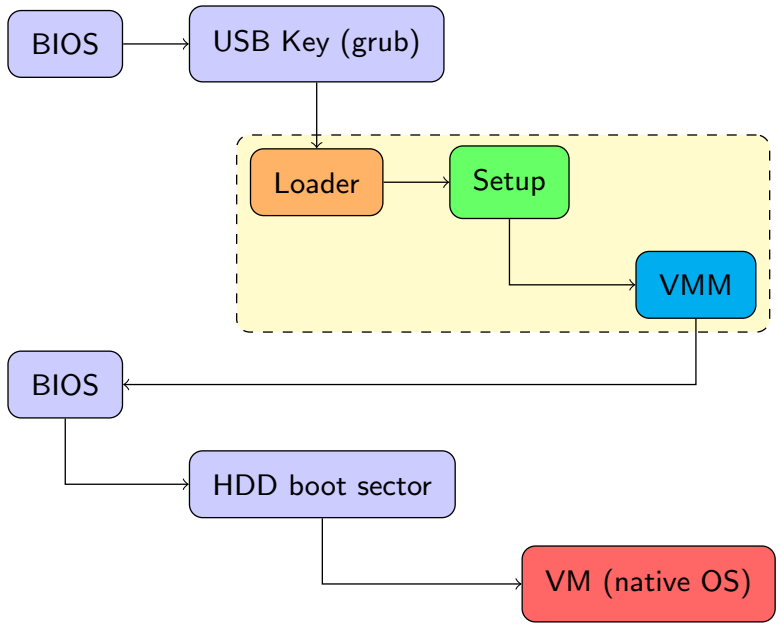
Chaîne de démarrage de Ramooflax



Chaîne de démarrage de Ramooflax



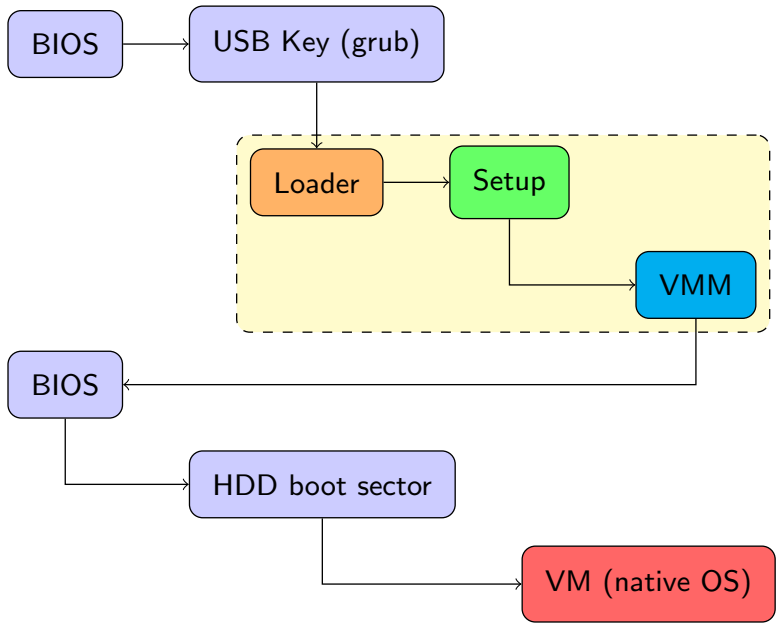
Composants de Ramooflax



Loader

- boot en mode protégé 32 bits (standard multiboot)
- bascule le cpu en *longmode* (64 bits) puis charge Setup

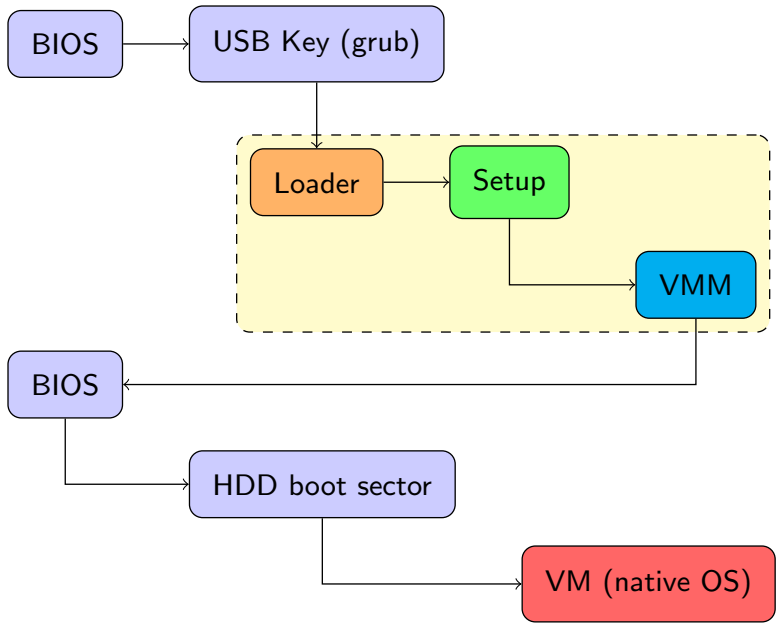
Composants de Ramooflax



Setup

- initialise les structures de virtualisation, les drivers, la mémoire
- récupère la taille de la RAM et précalcule l'espace nécessaire à VMM
- reloge vmm à $taille(RAM) - taille(VMM)$
- réduit la taille de la RAM (prépare des SMAPs spécifiques pour la VM)
- installe `int 0x19` en mémoire conventionnelle
- invoque VMM

Composants de Ramooflax



VMM *résidant*

- binaire PIE (taille de RAM variable)
- démarre son unique VM en mode réel sur `int 0x19`
- demande au BIOS (virtualisé) de démarrer l'OS natif

Éléments communs entre Intel-VT (vmx) et AMD-V (svm)

Intérêt

- simplifier le développement d'hyperviseurs
- jeu d'instructions réduit ($\sim$ 10)
- notion de vm-entry/vm-exit
  - vm-entry charge la VM, sauve le VMM
  - vm-exit charge le VMM et sauve la VM

Configuration d'une structure de données

- AMD VMCB, Intel VMCS (asynchrone accès via `vmread, vmwrite`)
- préparation des registres systèmes (`cr, dr, gdtr, idtr, ...`)
- injection d'évènements (interruptions, exceptions)
- mise en place de bitmaps d'interceptions
  - évènements
  - instructions sensibles
  - accès aux I/O, MSRs ...

De nombreuses limitations

- Intel/AMD non compatibles
- fonctionnalités différentes selon modèle de CPU
- difficile de savoir ce que le CPU propose avant de l'acheter ! `http://cpuid.intel.com ?`

De nombreuses limitations

- Intel/AMD non compatibles
- fonctionnalités différentes selon modèle de CPU
- difficile de savoir ce que le CPU propose avant de l'acheter ! `http://cpuid.intel.com ?`

- informations insuffisantes lors des vm-exit
- nécessaire d'embarquer un désassembleur/émulateur
- interception des interruptions matérielles et logicielles de type on/off
- pas d'interception des interruptions logicielles sous Intel
- interruptions matérielles *pending* sous AMD
- cas des SMIs (bugs CPU, bugs BIOS, virtualisation SMM nécessaire, . . .)

Gestion du mode réel catastrophique sous Intel
problématique pour le BIOS !

Virtualisation du BIOS

BIOS et mode réel

- mode par défaut du CPU
- 16 bits, adressage mémoire 20 bits (1Mo), pas de protection
- utilisé intensivement par le BIOS

Virtualiser le mode réel *historiquement*

- virtualisation matérielle existe depuis 80386: mode v8086
- cpu émule mécanismes du mode réel (interruptions, far call, . . .)
- redirection/interception des I/O, interruptions

Virtualisation du BIOS

BIOS et mode réel

- mode par défaut du CPU
- 16 bits, adressage mémoire 20 bits (1Mo), pas de protection
- utilisé intensivement par le BIOS

Virtualiser le mode réel *historiquement*

- virtualisation matérielle existe depuis 80386: mode v8086
- cpu émule mécanismes du mode réel (interruptions, far call, . . .)
- redirection/interception des I/O, interruptions

Virtualiser le mode réel avec `vmx/svm`

- AMD propose un *paged real mode* (`CR0.PE=0 && CR0.PG=1`)

Virtualisation du BIOS

BIOS et mode réel

- mode par défaut du CPU
- 16 bits, adressage mémoire 20 bits (1Mo), pas de protection
- utilisé intensivement par le BIOS

Virtualiser le mode réel *historiquement*

- virtualisation matérielle existe depuis 80386: mode v8086
- cpu émule mécanismes du mode réel (interruptions, far call, . . .)
- redirection/interception des I/O, interruptions

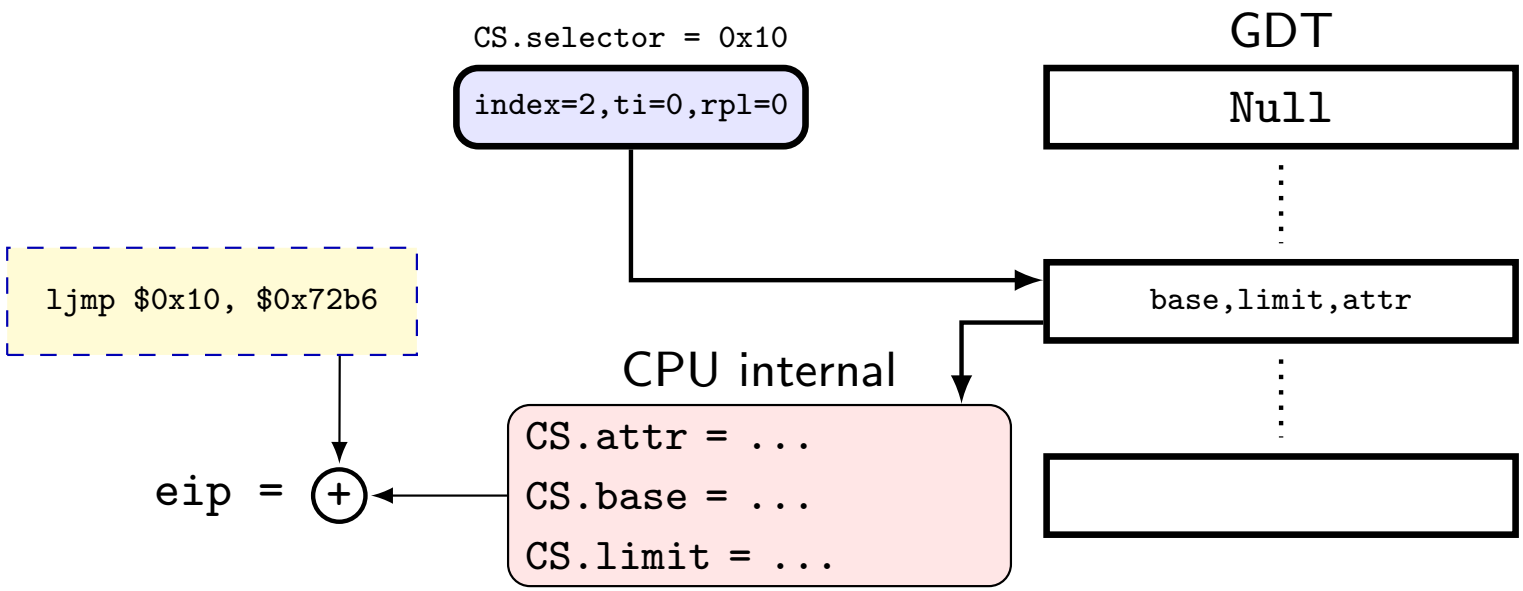
Virtualiser le mode réel avec `vmx/svm`

- AMD propose un *paged real mode* (`CR0.PE=0 && CR0.PG=1`)
- Intel interdit `CR0.PG=0` et donc `CR0.PE=0`
  - préconise l'utilisation du mode v8086
  - vm-entry en v8086 sont très restrictifs
  - surtout concernant la segmentation

Petit rappel sur la segmentation

Gestion des registres de segment

- partie visible accessible au programmeur (sélecteur)
- partie cachée gérée par le cpu (base, limite, attributs)
- mode réel: $base = selector * 16$, $limit = 64K$
- mode protégé: descripteurs de segments



Virtualisation du BIOS

Le mode irréel (*unreal, flat real, big real mode*)

- permet d'accéder à plus d'1Mo en mode réel
- transition mode protégé vers mode réel en laissant base=0 et limite=4Go
- utilisé par les BIOS pour accéder aux *memory mapped devices* ...

```
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
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
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
```

Virtualisation du BIOS

L'échec d'Intel

- vm-entry en mode v8086 vérifie[1] $base = selector * 16$
- impossible de virtualiser mode irréel avec v8086

[1] Intel Volume 3B Section 23.3.1.2

Virtualisation du BIOS

L'échec d'Intel

- vm-entry en mode v8086 vérifie[1] $base = selector * 16$
- impossible de virtualiser mode irréel avec v8086

Sans les extensions de virtualisation matérielle récentes

- émulation du mode réel en mode protégé
- intercepter accès aux registres de segment: *far call/jump*, *mov/pop seg*, *iret*
- double fail: Intel ne permet pas d'intercepter l'écriture de registres de segments
- solution: forcer limite GDT et IDT à 0 et intercepter les `#GP`

[1] Intel Volume 3B Section 23.3.1.2

Virtualisation du BIOS

L'échec d'Intel

- vm-entry en mode v8086 vérifie[1] $base = selector * 16$
- impossible de virtualiser mode irréel avec v8086

Sans les extensions de virtualisation matérielle récentes

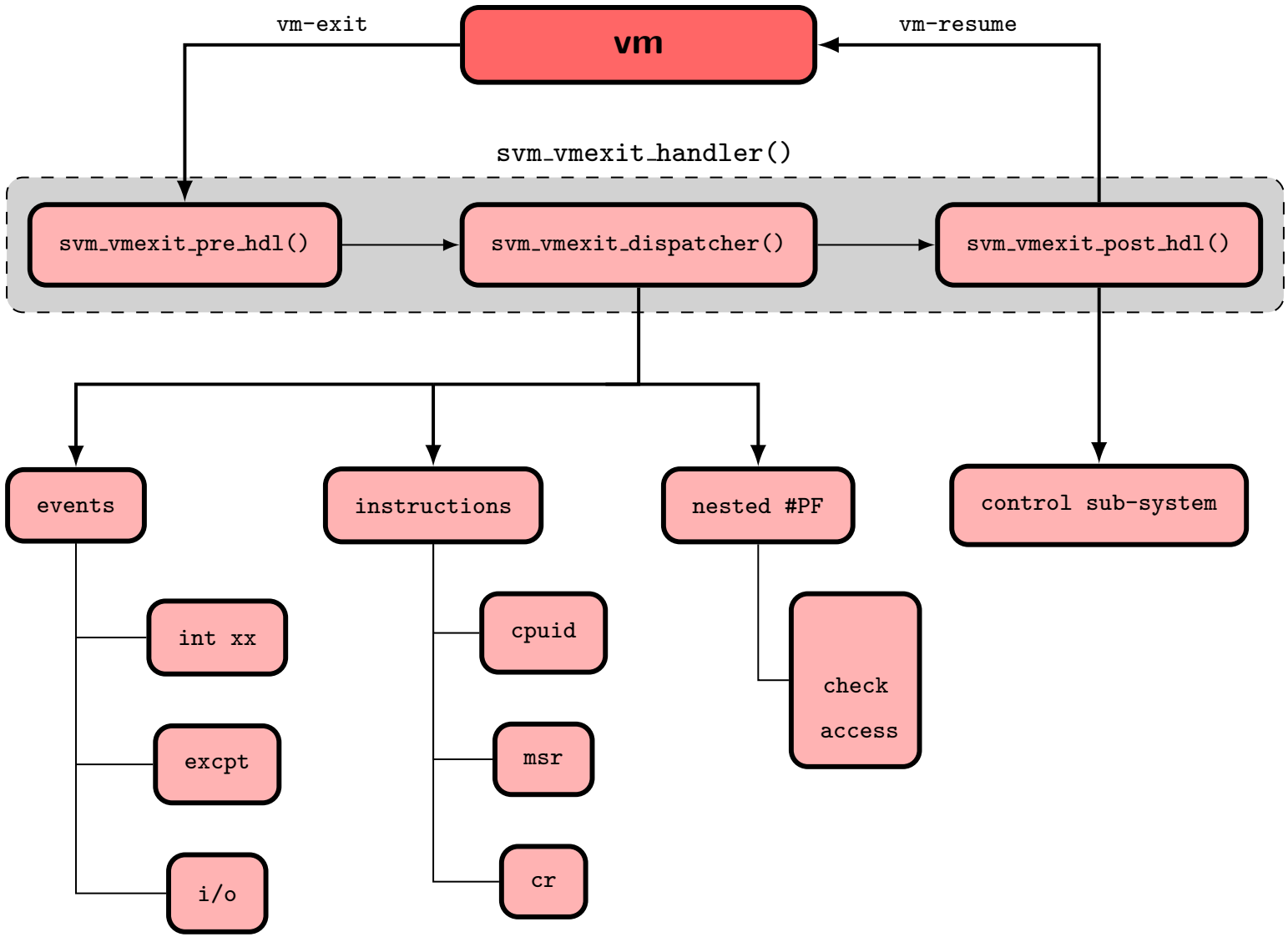
- émulation du mode réel en mode protégé
- intercepter accès aux registres de segment: *far call/jump*, *mov/pop seg*, *iret*
- double fail: Intel ne permet pas d'intercepter l'écriture de registres de segments
- solution: forcer limite GDT et IDT à 0 et intercepter les `#GP`

Dans les CPUs récents et suffisament haut de gamme

- nouveau mode *Unrestricted Guest* (autorise `CR0.PE=0 && CR0.PG=0`)
- nécessite d'avoir Intel EPT pour protéger la mémoire du VMM

[1] Intel Volume 3B Section 23.3.1.2

Modèle d'exécution



Filtrage des registres systèmes

Control Registers

- cr0 pour les transitions de mode, cohérence du cache et implications sur mapping mémoire
- cr3 pour l'interaction distante
- en tant que *feature* du client

Lecture d'un MSR et CPUID

- exécution native ou lecture dans la VMCS/VMCB
- post-traitement pour filtrer les fonctionnalités à masquer

Écriture d'un MSR

- émule `wrmsr` s'il existe dans la VMCS/VMCB
- sinon exécution native

Filtrage d'évènements

Exceptions

- interception fine de `#DB` et `#BP` par le sous-système de contrôle
- sous Intel, filtrage de `#GP` pour la gestion des interruptions logicielles

Interruptions logicielles

- uniquement en mode réel
- filtrer les accès aux SMAPs (`int 0x15`)

Interruptions matérielles

- non interceptées par défaut
- on laisse la possibilité de le faire

Émulation

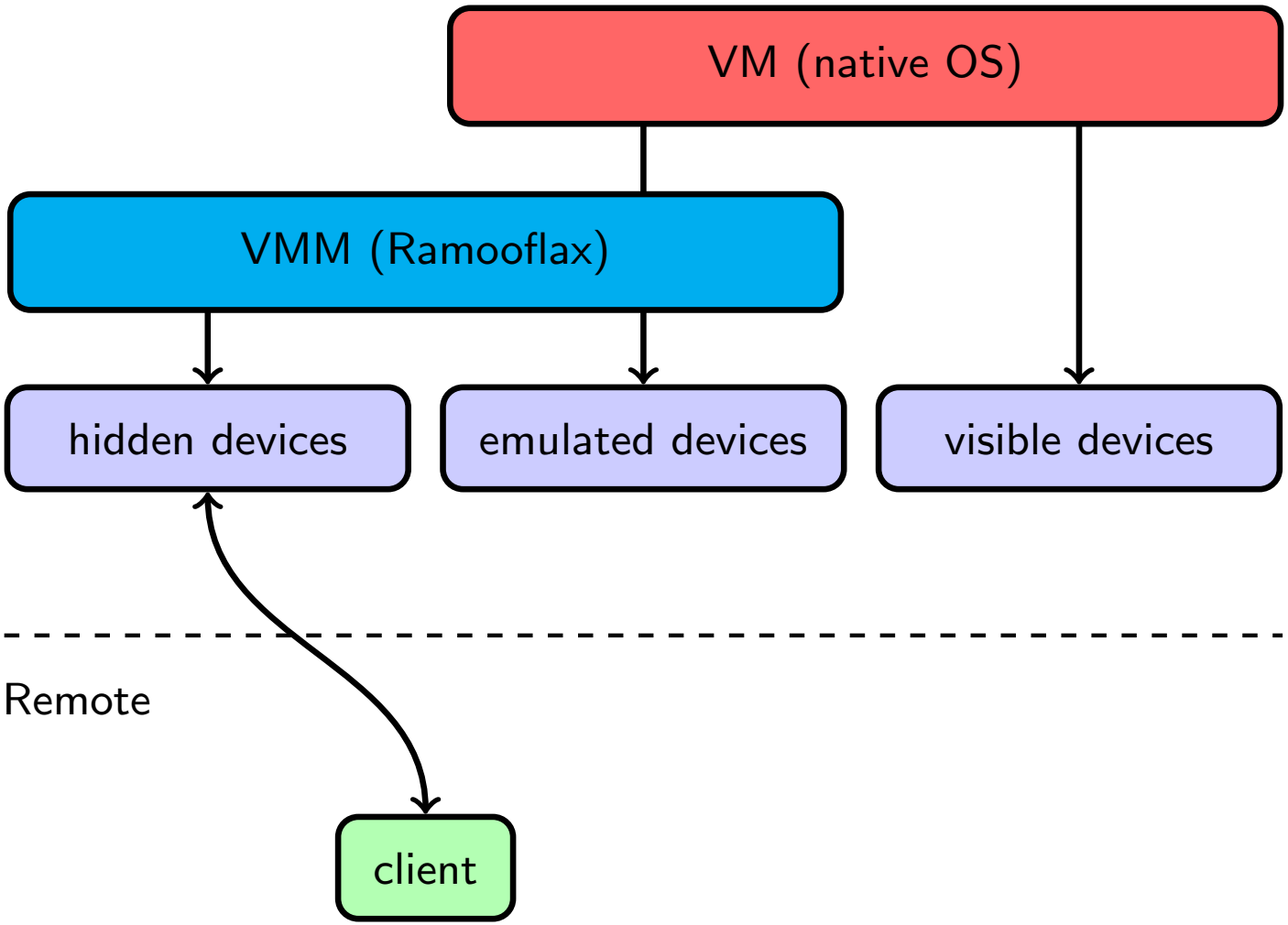
Instructions

- désassemblage/émulation nécessaire pour la gestion de certains vm-exit
- Ramooflax embarque udis86 .... un peu exagéré
- instructions émulées sont assez simples
- prise en compte du contexte d'exécution (assez sensible)

Périphériques

- émulation/interception (partielle) d'UART, PIC, KBD et PS2 System Controller
- principalement pour éviter les tentatives de *reboot*

Interaction VMM, VM, client



Communication distante

UART

- lent, peu fiable
- principalement utilisée pour le debug

EHCI Debug Port

- selon la norme USB 2.0, un port USB physique peut servir de Debug Port
- présent sur la plupart des contrôleurs hôtes actuels
- fiable, standardisé et rapide
- quasiment aussi simple à programmer qu'un port série

Implémentation côté Ramooflax

- driver de Debug Port
- contrôleur EHCI piloté par la VM

Communication distante

EHCI Debug Port: côté client

- norme USB: échange de données entre contrôleurs hôtes impossible
- nécessité de disposer d'un Debug Device
  - acheter un périphérique spécifique (ie Net20DC)
  - profiter des contrôleurs USB On-The-Go (*smartphones* . . .)

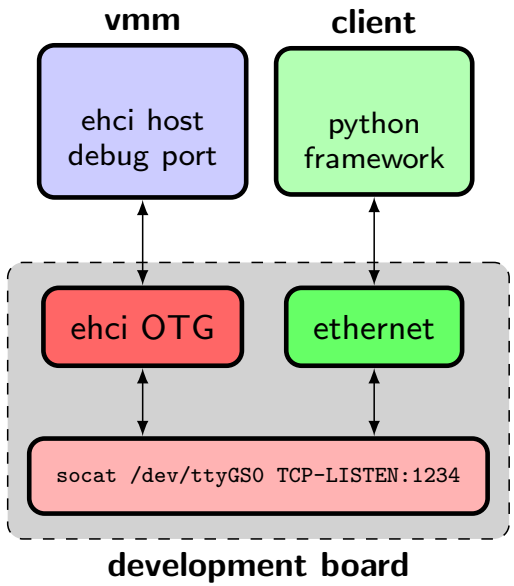
Communication distante

EHCI Debug Port: côté client

- norme USB: échange de données entre contrôleurs hôtes impossible
- nécessité de disposer d'un Debug Device
  - acheter un périphérique spécifique (ie Net20DC)
  - profiter des contrôleurs USB On-The-Go (*smartphones* . . .)

Émuler un Debug Device sous Linux

- la Gadget API permet l'émulation de périphériques USB (*mass storage* . . .)
- développement d'un Gadget Debug Device exposant une interface série (`ttyGS0`)



Interaction avec le client

Prise de contrôle

- le VMM attend des vm-exit
- trouver un compromis entre réactivité client et performances de la VM
- garantir que l'hyperviseur peut prendre la main sur la VM à la demande du client
- depuis peu Intel propose un *vmx_preemption_timer*, mais pas AMD

Via les interruptions ?

- pas d'irq générées pour le Debug Port
- complexité, latence, ...

Context switch

- OS modernes ordonnent régulièrement des processus
- intercepter les écritures de cr3

Interaction avec le client

Implémentation d'un stub GDB de base

- lecture/écriture des registres généraux
- lecture/écriture mémoire
- ajout/suppression de breakpoints softwares et hardwares
- single stepping

Limitations du protocole

- conçu pour debugguer des applications en espace utilisateur
- pas d'informations ring 0 (segmentation, paging, ...)
- pas de distinction mémoire virtuelle/physique

Interaction avec le client

Extensions spécifiques à Ramooflax

- accès aux registres systèmes
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6 et dr7, dbgctl
  - cs, ss, ds, es, fs, gs pour leur adresse de base
  - gdtr, idtr, ldtr et tr

Interaction avec le client

Extensions spécifiques à Ramooflax

- accès aux registres systèmes
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6 et dr7, dbgctl
  - cs, ss, ds, es, fs, gs pour leur adresse de base
  - gdtr, idtr, ldtr et tr
- accès mémoire
  - distinction adresses virtuelles/physiques
  - mécanisme de traduction
  - possibilité de fixer un cr3 avec lequel effectuer toutes les opérations

Interaction avec le client

Extensions spécifiques à Ramooflax

- accès aux registres systèmes
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6 et dr7, dbgctl
  - cs, ss, ds, es, fs, gs pour leur adresse de base
  - gdtr, idtr, ldtr et tr
- accès mémoire
  - distinction adresses virtuelles/physiques
  - mécanisme de traduction
  - possibilité de fixer un cr3 avec lequel effectuer toutes les opérations
- contrôle de la virtualisation
  - interception des accès aux registres de contrôle
  - interception des exceptions
  - idéalement . . . paramétrage complet des structures VMCS/VMCB

Interaction avec le client

Gestion du single-step

- basé sur TF et l'interception d'exceptions
- nombreux modes dans une VM
  - single-step global (*implémenté*)
  - single-step d'un thread noyau
  - single-step d'un processus en ring3 uniquement (*implémenté*)
  - single-step d'un processus en ring3 et en ring0
- pas de notions liées aux concepts de l'OS virtualisé (terminaison d'un processus)
- furtivité/cohérence à prendre en compte (interception `pushf`,`popf`,`intN`,`iret`)

Interaction avec le client

Gestion du single-step

- basé sur TF et l'interception d'exceptions
- nombreux modes dans une VM
  - single-step global (*implémenté*)
  - single-step d'un thread noyau
  - single-step d'un processus en ring3 uniquement (*implémenté*)
  - single-step d'un processus en ring3 et en ring0
- pas de notions liées aux concepts de l'OS virtualisé (terminaison d'un processus)
- furtivité/cohérence à prendre en compte (interception `pushf`,`popf`,`intN`,`iret`)

Cas particulier: `sysenter/sysexit`

- non interceptables sous AMD et Intel (!!!)
- ne masquent pas TF lors des transitions ring3/ring0
- implémentation d'un mécanisme détourné basé sur la génération de fautes

Une interface python à l'hyperviseur

Composants du framework

- VM, fonctionnalités haut niveau
- CPU, accès aux registres, filtres d'exceptions
- Breakpoints, soft/hard
- GDB, client GDB classique muni des extensions propres à l'hyperviseur
- Memory, contrôlant tous les accès à la mémoire
- Event, permettant au développeur de fournir son propre gestionnaire de `vm-exit`

Composants du framework: VM

- run, stop, resume, singlestep, attach, detach

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

- mode interactif

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

- mode scriptable

```
vm.attach()  # connexion distante
vm.stop()    # arrêt de la vm

# xxxx (breakpoints, filtres, ...)

vm.resume()  # resume la vm et attend le prochain vm-exit
vm.detach()  # déconnexion, la vm reprend la main
```

Composants du framework: CPU, Memory et Breakpoints

- nommage des breakpoints

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

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

- mise en place du tracking de cr3

```
# lecture d'une page de mémoire virtuelle
vm.cpu.set_active_cr3(my_cr3)
pg = vm.mem.vread(0x8048000, 4096)
```

Composants du framework: Event

- syntaxe non-intuitive de GDB pour les breakpoints conditionnels
- permettre au développeur d'exécuter des fonctions après un vm-exit
- dissociation architecture/mécanismes propres à l'OS
- filtrer une exception, un accès à cr3, un breakpoint, ...

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

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

while not vm.resume():
   continue

vm.interact()
```

Démo !

Conclusion

Support

- AMD fonctionnel
- testé avec succès sous
  - Windows XP/7 Pro 32 bits
  - Debian GNU/Linux 5.0 32 bits
- OS plus simples devraient fonctionner (DOS, OpenBSD, . . .)

Conclusion

Support

- AMD fonctionnel
- testé avec succès sous
  - Windows XP/7 Pro 32 bits
  - Debian GNU/Linux 5.0 32 bits
- OS plus simples devraient fonctionner (DOS, OpenBSD, . . .)

Limitations

- Intel à réécrire
- pas de SMP, multi-cores
  - délicat à mettre en œuvre
    - initialiser tous les Cores et activer la virtualisation
    - intercepter l'initialisation des Cores faite par la VM
  - contournement via `/numproc`, `maxcpus`, . . .
- OS 64 bits marchotent
  - peu testé . . .
  - ioAPIC et SMP caractériels
- pas de *Nested Virtualization*

Merci !

https://github.com/sduverger/ramooflax