



UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

**CORSO DI LAUREA MAGISTRALE IN TECNOLOGIE DELL'INFORMAZIONE
E DELLA COMUNICAZIONE**

**ANALISI DI CODICE KERNEL TRAMITE
VIRTUALIZZAZIONE HARDWARE-ASSISTED**

Relatore: Dott. Mattia MONGA

Correlatore: Dott. Roberto PALEARI

Tesi di Laurea di:
Aristide FATTORI
Matricola 735939

Anno Accademico 2009–10

Ai miei genitori e a Annalisa

Ringraziamenti

Innanzitutto, il primo ringraziamento va al mio relatore, il Dott. Mattia Monga, per avermi dato la possibilità di svolgere questo lavoro di tesi e di andare a presentarlo a ASE 2010. Il secondo ringraziamento va al mio correlatore Roberto Paleari, senza il quale non sarei mai riuscito a arrivare alla conclusione di questa tesi.

Un ringraziamento molto speciale va ai miei genitori e a mia nonna per tutto il supporto che mi hanno saputo dare in ogni situazione e in ogni mia scelta durante questi sei lunghi anni di studi. È banale e forse inutile dirlo, ma senza di voi non sarei mai arrivato dove sono arrivato. Grazie anche a mia sorella che è sempre paziente con me, anche quando sono più intrattabile ;-).

Grazie a Annalisa che mi è vicina da sempre, non so come avrei fatto in tutti questi anni senza il tuo supporto nei momenti più difficili. Ti amo.

Un ringraziamento anche per tutti i componenti del LaSER, con cui ho passato gli ultimi tre anni, sempre più formativi e divertenti di anno in anno, rigorosamente in ordine alfabetico per cognome: Sullivan, DarkAngel, Gianz, Jack, Brown, Lorenzo, Srdjan, Jeppojejs, Ema, AdrenalineSoars e Robi. A tutti, grazie per tutto quello che mi avete insegnato e per tutti i CTF. A Jack, però, va un ringraziamento speciale: inseparabile collega e carissimo amico per questi **sei** anni di università, senza il quale, ne sono sicuro, non sarei mai riuscito a andare avanti e a affrontare tutte le difficoltà incontrate lungo il cammino. Le nostre carriere si sono separate ora, ma spero sinceramente che questa sia solo una breve parentesi e che torneremo a lavorare insieme il prima possibile ;-)

Ringrazio anche tutti gli altri miei amici che hanno sempre saputo fornirmi quel poco di distacco dalla vita universitaria, essenziale per impedirmi di venire “inghiottito” dall’informatica: Ale, Cucco, Luca e Daniele.

Infine, un ringraziamento a tutti i miei colleghi che ho conosciuto durante la mia carriera universitaria.

Un grazie anche a pimpa e naima che, seppure indirettamente, hanno enormemente contribuito a questo lavoro.

Indice

1	Introduzione	1
1.1	Motivazioni del lavoro	1
1.2	Idea	3
1.3	Organizzazione della tesi	4
2	Concetti preliminari	6
2.1	Virtualizzazione	6
2.1.1	Virtualizzazione Software	8
2.1.2	Soluzioni software ai problemi della virtualizzazione	12
2.1.3	Introduzione al supporto hardware alla virtualizzazione	13
2.1.4	Soluzioni ai problemi della virtualizzazione tramite VT-x	19
2.1.5	Uso della virtualizzazione hardware-assisted	21
2.2	Concetti di debugging	21
2.2.1	Breakpoint	22
2.2.2	Watchpoint	23
2.2.3	Single stepping	24
3	Scenario d'utilizzo	26
3.1	Kernel debugging	26
3.1.1	Analisi dinamica basata sul kernel	28
3.1.2	Analisi dinamica basata su Virtual Machine Monitor	29

3.2	Hypervisor come soluzione	30
3.3	Vantaggi dell'approccio con hypervisor	31
4	HYPERDBG	33
4.1	HYPERDBG: Hypervisor Debugger	33
4.2	Architettura	35
4.3	Dettagli	37
4.3.1	Installazione dell'hypervisor	37
4.3.2	Gestione delle VM Exit	40
4.3.3	Gestione della memoria	47
4.3.4	Funzionalità di debug	60
4.3.5	Analisi dipendenti dal sistema operativo	64
4.3.6	HYPERDBG GUI	65
5	Implementazione e limitazioni	71
5.1	Dettagli d'implementazione	71
5.2	Implementazione delle tecniche di introspezione	72
5.3	Limitazioni	74
5.3.1	Shadow page table e watchpoint	74
5.3.2	Driver video	75
5.3.3	Driver della tastiera	77
5.4	Ulteriori sviluppi futuri	81
6	Lavori correlati	83
6.1	Instrumentazione dinamica del kernel	83
6.1.1	DTrace	84
6.1.2	KernInst	85
6.2	Debugger a livello kernel	86
6.3	Strumenti di analisi basati su macchine virtuali	86
6.4	Analisi di malware tramite virtualizzazione hardware assisted	88
6.5	Programmazione aspect-oriented	89

7 Conclusioni	91
Bibliografia	93

Introduzione

Un sistema operativo e specialmente il suo nucleo principale, cioè il *kernel*, è spesso la componente più complessa di un sistema informatico e il suo corretto funzionamento è vitale. Infatti, un difetto, o un collo di bottiglia, in una qualsiasi delle sue parti, possono portare a conseguenze disastrose. È necessario, di conseguenza, che tali sistemi siano implementati con estrema cura da parte degli sviluppatori che, inoltre, devono poter fare affidamento su strumenti e tecniche adatte a supportarli nelle verifiche di correttezza dei loro prodotti. Esistono diverse tipologie di strumenti e tecniche di analisi di codice kernel che possono essere utilizzate a tale scopo, alcune delle quali specifiche per studiare le proprietà statiche di un sistema e altre più adatte a verificarne le proprietà dinamiche. In questa tesi, l'attenzione sarà rivolta alle analisi *dinamiche* che permettono agli sviluppatori di raccogliere informazioni sul sistema analizzato durante l'esecuzione: per questo motivo sono spesso preferite, in quanto evitano le complicazioni delle analisi rivolte a tutti i possibili comportamenti.

1.1 Motivazioni del lavoro

Le tecniche di analisi dinamica di sistemi operativi attualmente esistenti possono essere divise in due categorie principali: basate sul kernel e basate sull'uso di macchine virtuali. Il primo approccio consiste nell'installare alcuni componenti aggiuntivi nel kernel che si vuole analizzare, in modo da poter intercettare tutti gli eventi interessanti

dal punto di vista dell'analisi (come la creazione di nuovi processi, l'invocazione di particolari system call o l'esecuzione di determinate funzioni del kernel) e di intraprendere delle particolari azioni in risposta a questi eventi [3, 42]. Questa soluzione richiede l'inserimento a priori di specifici "punti di aggancio" (hook) nel kernel che si vuole analizzare, in modo da intercettare dinamicamente gli eventi di interesse. Questo prerequisito può risultare molto difficile da soddisfare, soprattutto nel caso si intenda analizzare sistemi che non offrono nativamente alcun supporto per l'analisi dinamica o che non rendono accessibile liberamente il loro codice sorgente. L'approccio utilizzato dal secondo gruppo, invece, consiste nell'eseguire il sistema operativo che si intende analizzare all'interno di una macchina virtuale e nell'intercettare e analizzare gli eventi di interesse dal virtual machine monitor (VMM), il componente software che si occupa di controllare l'esecuzione della macchina virtuale [14]. Nonostante questa seconda tecnica offra una maggiore trasparenza al sistema analizzato e, allo stesso tempo, non dipenda strettamente dai meccanismi interni di tale sistema, soffre ugualmente di alcune limitazioni per quanto riguarda la sua applicabilità. Infatti, non è possibile utilizzarla per analizzare le proprietà di sistemi che non sono stati avviati all'interno di una macchina virtuale. Inoltre, dal momento che tipicamente un VMM virtualizza l'accesso alle risorse hardware per permettere a più sistemi operativi di accedervi contemporaneamente, questo secondo approccio risulta particolarmente inadatto a analizzare sistemi che richiedono particolari configurazioni hardware. Infatti, è possibile che il VMM impedisca a un sistema operativo guest di accedere direttamente a alcuni device hardware. Per esempio, si supponga che il componente che si vuole analizzare sia un driver creato appositamente per un particolare device. Tale driver viene eseguito all'interno di un sistema virtualizzato per poter essere analizzato. Qualora, però, il VMM impedisca al sistema virtualizzato di interagire direttamente con quel particolare device, il driver non potrebbe funzionare correttamente, rendendone vana l'analisi.

1.2 Idea

L'obiettivo che questo lavoro di tesi si propone è quello di apportare un contributo al campo dell'analisi dinamica di codice kernel, presentando un approccio che abbia le stesse potenzialità di entrambi quelli finora utilizzati senza, però, essere affetto dalle medesime limitazioni.

In particolare, la tecnica proposta permette di effettuare analisi di codice kernel con le seguenti proprietà: (I) dinamicità completa, (II) trasparenza al sistema analizzato, (III) non invasività e (IV) indipendenza dal sistema che si intende analizzare. La dinamicità completa consente di analizzare sistemi in esecuzione nativamente su una macchina fisica e che non mettono a disposizione nessun particolare supporto per le analisi. La trasparenza, anche se potrebbe sembrare poco importante, è una proprietà essenziale per alcune tipologie di analisi o di programmi da analizzare. Per esempio, nel caso dell'analisi di malware, è fondamentale che il codice malevolo non possa identificare la presenza di uno strumento di analisi. La non invasività consiste nel non andare a modificare in nessun modo il sistema che si deve analizzare e è importante sia per le sue influenze sulla trasparenza (una modifica a una struttura fondamentale del sistema analizzato può essere facilmente identificata) che per quanto riguarda il funzionamento stesso del sistema. Infatti, modifiche eccessive alle sue strutture fondamentali potrebbero andare inavvertitamente a corrompere il funzionamento. Infine, la proprietà di essere il più possibile indipendente dal sistema che si intende analizzare permette di applicare l'approccio che gode di questa proprietà a molti sistemi diversi, senza la necessità di essere modificato per supportarli.

L'idea su cui si basa l'approccio proposto per ottenere tutte queste proprietà, altamente desiderabili per l'analisi dinamica di codice kernel, consiste nello sfruttare il supporto hardware alla virtualizzazione, recentemente introdotto sulla maggior parte dei processori moderni, il quale consente di sviluppare VMM molto efficienti e completamente trasparenti al sistema virtualizzato. Il sistema operativo che deve essere analizzato viene migrato all'interno di una virtual machine mentre è in esecuzione, senza essere riavviato, e viene installato un VMM minimale che, sfruttando il supporto hardware alla virtualizzazione, si occupa di controllare l'esecuzione del sistema

virtualizzato e di mettere a disposizione un ambiente di esecuzione per gli strumenti di analisi. In questo modo, le analisi vere e proprie vengono eseguite al livello di privilegio dell'hypervisor (cioè lo stesso del VMM), e sono completamente isolate dal sistema analizzato. Per fare ciò dinamicamente viene utilizzata una particolare caratteristica della virtualizzazione hardware-assisted, che permette di installare un VMM e di *migrare* un sistema operativo *in esecuzione* all'interno di una macchina virtuale. In questo modo, viene soddisfatta la proprietà (I), mancanza principale dei classici approcci basati su VMM, tramite cui non è possibile analizzare sistemi che non sono stati avviati all'interno di una macchina virtuale. L'utilizzo della virtualizzazione hardware-assisted permette altresì di applicare l'approccio proposto al fine di analizzare un sistema operativo indipendentemente dal fatto che quest'ultimo metta a disposizione un supporto nativo per l'analisi o dalla disponibilità del suo codice sorgente (IV). Inoltre, la tecnica proposta non è invasiva (III), dal momento che richiede solamente l'installazione di un driver minimale che si occupa di installare l'hypervisor, come sarà spiegato nel seguito di questo lavoro di tesi. Infine, facendo leva sul supporto hardware alla virtualizzazione e implementando particolari tecniche per proteggere i componenti che realizzano l'analisi vera e propria, l'approccio proposto risulta essere completamente trasparente al sistema analizzato (II). Quando l'analisi è terminata, il VMM e i componenti di analisi possono essere semplicemente rimossi dal sistema operativo analizzato, a cui viene restituito il controllo completo dell'hardware.

Per dimostrare l'efficacia della tecnica proposta, è stato implementato HYPERDBG. HYPERDBG include un hypervisor minimale che realizza l'approccio di analisi proposto in questo lavoro di tesi, unitamente a un debugger kernel interattivo. Essendo in grado di soddisfare tutte le proprietà elencate, l'approccio proposto in questo lavoro di tesi e implementato in HYPERDBG apporta un contributo allo stato dell'arte dell'analisi dinamica di codice kernel.

1.3 Organizzazione della tesi

Il Capitolo 2 fornisce le nozioni e i concetti preliminari necessari alla comprensione del presente lavoro di tesi. Vengono dapprima analizzate le principali tecniche di vir-

tualizzazione software comunemente utilizzate, evidenziandone i problemi caratteristici. In seguito, vengono presentati i dettagli del funzionamento della virtualizzazione hardware-assisted, illustrando come tale supporto hardware permetta di superare le limitazioni insite nelle tecniche di virtualizzazione software. Infine, vengono introdotti alcuni concetti relativi al debugging.

Nel Capitolo 3 si introduce il problema affrontato in questo lavoro di tesi, cioè l'analisi dinamica di codice kernel, soffermandosi sugli approcci proposti e utilizzati finora per affrontare questo problema. Viene quindi delineata un'introduzione alla soluzione proposta in questa tesi, evidenziandone le principali difficoltà concettuali e implementative.

I dettagli dell'architettura di HYPERDBG vengono presentati nel Capitolo 4, dove vengono singolarmente analizzati tutti i suoi componenti, ponendo particolare attenzione alle problematiche che insorgono dovendo virtualizzare un sistema già in esecuzione e operando al livello di privilegio dell'hypervisor. Inoltre, quando rilevante, vengono confrontate le caratteristiche di HYPERDBG e degli approcci correnti.

Il Capitolo 5 riporta alcuni dettagli riguardo all'implementazione di HYPERDBG, le limitazioni dell'implementazione attuale e alcune possibili soluzioni per tali limitazioni. Il Capitolo si conclude, quindi, elencando alcuni possibili sviluppi futuri del lavoro.

Nel Capitolo 6 vengono presentati alcuni lavori di ricerca e strumenti simili, nello scopo o nell'implementazione, a HYPERDBG. Per ognuno di questi lavori e strumenti vengono presentati i principali vantaggi e svantaggi e sottolineate le differenze rispetto a HYPERDBG.

Il Capitolo 7 conclude il lavoro di tesi.

Capitolo 2

Concetti preliminari

In questo Capitolo vengono illustrati i concetti fondamentali della virtualizzazione e del debugging. Inizialmente, viene introdotta la virtualizzazione software e le problematiche a essa relative. Dopodiché, si passa a trattare la virtualizzazione *hardware-assisted*, illustrandone i dettagli e evidenziando come permette di superare le limitazioni insite nella virtualizzazione software. Infine, vengono introdotti alcuni dei concetti basilari del debugging, dal momento che questi concetti saranno richiamati molto frequentemente nei successivi Capitoli di questo lavoro di tesi.

2.1 Virtualizzazione

La virtualizzazione delle risorse di un sistema è un concetto che risale a parecchi decenni orsono [16]. Sino a pochi anni fa, tuttavia, è rimasta confinata ai sistemi server e ai mainframe che potevano garantire un elevato livello di prestazioni, a causa dell'altissima richiesta di risorse computazionali, raramente disponibili in un sistema *end-user*. Tuttavia, con gli enormi miglioramenti in termini di potenza computazionale dei processori e con l'abbassamento dei prezzi degli stessi, unitamente all'introduzione di nuove e più efficaci tecniche a livello software, la virtualizzazione si è diffusa sempre di più e oggi è comunemente usata sulle macchine di numerosi utenti.

La virtualizzazione completa delle risorse di un sistema permette di eseguire più sistemi operativi contemporaneamente su di una sola piattaforma hardware. Contra-

riamente a quanto accade in un sistema non virtualizzato, in cui un singolo sistema operativo assume il controllo completo di tutte le risorse hardware, in un sistema virtualizzato è presente un ulteriore livello software, chiamato Virtual Machine Monitor (VMM). Il ruolo principale del VMM è quello di gestire e controllare gli accessi alle risorse hardware del sistema su cui è eseguito, chiamato in gergo “host”, in modo che tali risorse possano essere condivise tra diversi sistemi operativi eseguiti contemporaneamente, comunemente detti “guest”. Il VMM inoltre rende visibili a ogni sistema guest un insieme di risorse virtuali, che costituiscono l’entità a cui si fa spesso riferimento con il termine “macchina virtuale”. Tipicamente, il VMM è implementato come un’applicazione con componenti sia in user-space che in kernel-space, che viene eseguita all’interno del sistema operativo presente sull’host. L’implementazione di un VMM è estremamente complicata. Contrariamente a quanto succede nell’emulazione, in cui l’esecuzione ogni istruzione della CPU della macchina virtuale viene simulata, nella virtualizzazione le istruzioni vengono eseguite *nativamente*¹ sulla CPU fisica dell’host e solamente un sottoinsieme di queste viene emulato dal VMM. Ovviamente, questo impedisce di virtualizzare instruction set diversi da quello dell’architettura dell’host, ma garantisce prestazioni molto più elevate. Allo stesso tempo, però, questa tecnica introduce un insieme di problematiche particolarmente difficili da trattare. Per tentare di facilitare lo sviluppo di VMM, sia Intel che AMD, le due principali compagnie che producono processori, hanno recentemente aggiunto un particolare supporto hardware alle tecniche di virtualizzazione, rispettivamente VT-x e AMD-V [32, 1]. Nel seguito di questa Sezione verranno dapprima illustrati i dettagli della virtualizzazione software e le problematiche relative all’implementazione e all’utilizzo di un VMM software, ripercorrendo quanto introdotto dettagliatamente da Neiger *et al.* in [32]. Successivamente, verrà introdotto il supporto hardware alla virtualizzazione, entrando nelle specifiche del funzionamento di questa tecnologia, dal momento che la conoscenza di tali dettagli risulta essenziale per una comprensione completa del presente lavoro di tesi. La trattazione sarà limitata alla tecnologia di virtualizzazione hardware-assisted introdotta sui processori Intel, dal momento che è su di essa che si basa la tecnica di analisi di codice kernel qui proposta.

¹Questa tecnica di virtualizzazione è infatti nota come *direct native execution* [39].

2.1.1 Virtualizzazione Software

Come è stato brevemente accennato in precedenza, quando si usa la tecnica di virtualizzazione nota come *direct native execution* [39], solamente un sottoinsieme delle istruzioni eseguite all'interno della virtual machine deve essere emulate, mentre le rimanenti possono essere eseguite direttamente dal processore dell'host. Questa necessità di emulare alcune istruzioni è dovuta al fatto che il VMM deve essere eseguito a un livello di privilegio maggiore rispetto al sistema guest [32]. I processori Intel forniscono un meccanismo di protezione basato sull'utilizzo di quattro diversi livelli di privilegio. Come illustrato in Figura 2.1, che riporta i 4 "anelli" di protezione tipici di un architettura x86, il kernel viene eseguito al massimo livello di privilegio possibile (ring 0), mentre le applicazioni vengono eseguite nel più basso livello disponibile (ring 3). Il livello di privilegio correntemente impostato nel sistema determina se un'i-

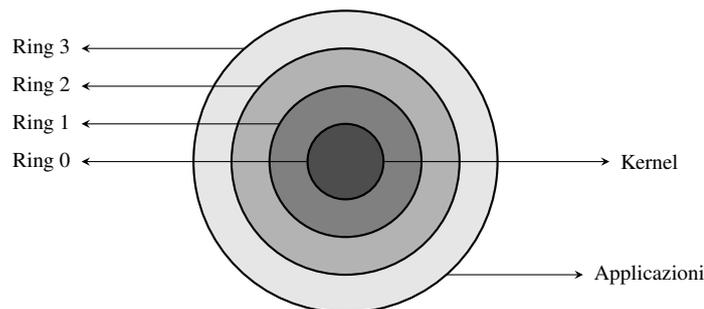


Figura 2.1: Livelli di privilegio

struzione privilegiata, come quelle che controllano le funzionalità di base della CPU, possa essere eseguita o meno. In caso negativo, l'esecuzione dell'istruzione viene interrotta. Dal momento che un sistema operativo deve essere in grado di controllare completamente la CPU, alcuni suoi componenti devono essere per forza eseguiti a ring 0. Questo evidenzia immediatamente il problema principale che un VMM si trova a dover affrontare: il sistema operativo guest, eseguito all'interno della virtual machine messa a disposizione da esso, richiede di essere eseguito al *massimo* livello di privilegio disponibile. Ovviamente, il VMM non può permettere al sistema guest di ottenere questo elevato livello di controllo, altrimenti quest'ultimo sarebbe in grado

di uscire dalla macchina virtuale creata dal VMM e di andare a interagire direttamente con l'hardware sottostante, di fatto ignorando la presenza del VMM e eludendone le politiche di accesso alle risorse. In questo scenario ipotetico, sia il VMM che il sistema operativo guest sono eseguiti a ring 0. Per evitare il problema appena illustrato, il VMM può usare la tecnica nota come *ring deprivileging*, che consiste nell'eseguire il sistema operativo guest in un livello meno privilegiato. Esistono due diversi approcci per deprivilegiare un sistema guest: eseguirlo a ring 1, secondo il modello 0/1/3, o a ring 3, secondo il modello 0/3/3. Il ring deprivileging introduce notevoli problemi, sia a livello di funzionamento che a livello di prestazioni, come illustrato nel seguito di questa Sezione.

Aliasing dei ring. La prima categoria di problemi introdotti dal deprivileging è nota come *aliasing dei ring* e racchiude tutti i problemi che vengono sollevati quando un programma viene eseguito a un livello di privilegio diverso da quello per cui è stato progettato. Un esempio di un problema introdotto dall'aliasing dei ring è il caso in cui un componente del sistema operativo guest, che si aspetta di essere eseguito a ring 0, effettui una lettura del contenuto del registro CS, per esempio tramite l'istruzione:

```
PUSH CS
```

che salva il contenuto di tale registro sullo stack. Il registro CS contiene, tra le altre cose, il livello di privilegio corrente del sistema, quindi il componente che va a leggerlo può facilmente accorgersi di non essere eseguito al livello di privilegio che si aspetta.

Compressione dello spazio di indirizzamento. Questo problema deriva dal fatto che un sistema operativo si aspetta di poter accedere liberamente a ogni zona di memoria. Il VMM, di contro, deve riservare per sé stesso alcune porzioni dello spazio di indirizzamento del guest. Infatti, è possibile eseguire il VMM completamente all'interno dello spazio di indirizzamento del sistema guest, al fine di ottenere un'elevata semplicità di accesso ai dati del guest, con lo svantaggio però di dover occupare una porzione significativa della memoria del guest per contenere il codice e i dati del VMM. Una soluzione alternativa consiste nell'eseguire il VMM in uno spazio di indirizzamento separato dal sistema guest ma, anche in questo secondo caso, il VMM

deve comunque utilizzare una minima parte dello spazio del guest, al fine di installare le strutture di controllo che gli permettono di effettuare le transizioni tra se stesso e il software guest. Indipendentemente da quale di queste alternative sia implementata, il VMM deve impedire al sistema guest di accedere alle porzioni che sta usando. In caso contrario, l'integrità stessa del VMM potrebbe essere compromessa, qualora il guest possa scrivere in tali porzioni, oppure potrebbe essere possibile per il guest accorgersi di essere eseguito all'interno di una macchina virtuale, nel caso vi possa accedere in lettura. È essenziale, quindi, che ogni tentativo del sistema guest di accedere a tali aree di memoria utilizzate dal VMM risulti in una transizione del controllo dal sistema guest al VMM, in modo che questo secondo componente possa limitarvi l'accesso, per esempio emulando l'istruzione che ha effettuato l'accesso, per modificarne il risultato in modo da "nascondersi" al sistema guest.

Accesso a informazioni sensibili. Grazie all'uso dei livelli di privilegio che sono stati illustrati precedentemente in questa Sezione, l'accesso da parte di software non privilegiato a alcuni componenti dello stato delle CPU causa un fault. Di conseguenza, dal momento che il sistema guest di una virtual machine viene eseguito a un livello di privilegio inferiore rispetto a quello previsto, ogni volta che prova a accedere a queste informazioni riservate, viene generato un fault. Intercettando tale condizione di errore, il VMM è in grado di emulare l'istruzione del sistema guest che ha causato il fault, in modo da impedire al sistema guest di accorgersi che non è eseguito a un livello di privilegio diverso da quello che si aspetta. Tuttavia, non tutte le istruzioni che accedono a informazioni sensibili causano un fault quando vengono eseguite. Per esempio, si consideri la Interrupt Descriptor Table (IDT) che contiene i puntatori alle funzioni che vengono invocate alla ricezione di determinati interrupt. Il contenuto di questa struttura è ovviamente vitale per il corretto funzionamento di un sistema operativo e, infatti, l'istruzione per scriverne una entry, la `LIDT` può essere eseguita solamente a ring 0. Al contrario, però, l'istruzione che accede in lettura a questa struttura dati, la `GIDT`, può essere eseguita da qualsiasi livello di privilegio. Qualora il VMM modifichi la IDT del sistema guest, al fine di essere in grado di intercettare alcuni eventi che devono essere analizzati prima che vengano notificati al sistema guest, un programma eseguito

all'interno del sistema guest è in grado di accorgersi di questa differenza, dal momento che può eseguire la `GIDT` senza che il VMM possa intercettarne l'esecuzione.

Degrado delle performance delle system call. L'uso di ring deprivileging può interferire anche con l'efficienza del meccanismo di accelerazione delle system call utilizzato nelle moderne architetture Intel. Questo meccanismo di accelerazione si basa sull'uso di due particolari istruzioni, la `SYSENTER` e la `SYSEXIT`, il cui scopo è quello di velocizzare la transizione dal livello user al livello kernel quando viene invocata una system call. La `SYSENTER` causa un passaggio da ring 3 a ring 0 mentre, qualora la `SYSEXIT` sia eseguita con un livello di privilegio diverso da 0, viene generato un fault. Di conseguenza, l'utilizzo di ring deprivileging fa sì che la `SYSENTER` trasferisca il controllo al VMM invece che al kernel del sistema guest, obbligando il VMM a emulare ogni esecuzione di tale istruzione. Allo stesso modo, dal momento che l'esecuzione dell'istruzione `SYSEXIT` in un guest deprivilegiato causa un fault, il VMM deve intercettare tale fault e emulare ogni esecuzione della suddetta istruzione.

Virtualizzazione degli interrupt. Tipicamente, un sistema operativo, durante la sua esecuzione utilizza la possibilità, messa a disposizione dalle architetture Intel, di mascherare gli interrupt in modo da disabilitarne la consegna quando non è pronto a riceverli. Un VMM deve poter controllare completamente gli interrupt e, conseguentemente, deve impedire al sistema operativo guest di mascherarli. Grazie al ring deprivileging, ogni tentativo del sistema guest di mascherare gli interrupt genera un fault che può essere intercettato e gestito dal VMM. Questa soluzione ha un drastico impatto sulle performance, perché il mascheramento gli interrupt è un'operazione che viene eseguita molto spesso dal sistema operativo guest. Purtroppo, il problema delle performance non è l'unico che rende difficile la gestione degli interrupt. Anche se fosse disponibile un meccanismo efficiente per impedire al sistema guest di mascherarli arbitrariamente, rimarrebbe il problema di consegnare gli interrupt al sistema guest entro un tempo adeguato. Si immagini, per esempio, la seguente situazione: il sistema guest ha richiesto di disabilitare gli interrupt perché non è pronto a riceverli. Il tentativo di disabilitazione è stato intercettato dal VMM, che ha bisogno di intercettare gli inter-

rupt e, di conseguenza, li lascia abilitati. Un device hardware invia, nel frattempo, un interrupt destinato al sistema guest che viene ricevuto dal VMM che lo trasforma in un “interrupt virtuale”. Questo interrupt virtuale non può però venire consegnato al sistema guest fintanto che questo non richiede di riabilitare gli interrupt e, di conseguenza, si corre il rischio che non venga consegnato in tempo al sistema guest. La gestione degli interrupt virtuali aumenta incredibilmente la complessità della progettazione di un VMM.

Protezione del kernel del guest. In precedenza, in questa Sezione, sono stati illustrate due diverse tipologie di ring deprivileging: il modello 0/1/3 e il modello 0/3/3. Qualora sull’host non sia possibile utilizzare il ring 1 si può adottare solo il secondo modello di deprivilegiamento. Questo implica che, all’interno della macchina virtuale, il kernel del sistema operativo guest viene eseguito allo stesso livello di privilegio (ring 3) delle applicazioni utente che, di conseguenza, posso andare a modificarlo arbitrariamente. Questa mancanza di protezione può facilmente portare a problemi nell’esecuzione dei componenti critici del sistema guest.

Accessi frequenti a risorse privilegiate. Nella presentazione dei problemi precedenti, si è spesso detto come il VMM faccia affidamento sui fault causati dall’esecuzione di istruzioni privilegiate all’interno del sistema operativo guest per intercettarle e emularle in modo che il sistema guest possa funzionare correttamente e, allo stesso tempo, non corrompa il VMM. Questo comportamento garantisce un funzionamento corretto sia del guest che del VMM ma introduce un notevole overhead se la frequenza dei fault è molto elevata, come nel sopracitato caso del mascheramento degli interrupt.

2.1.2 Soluzioni software ai problemi della virtualizzazione

Per tentare di risolvere i problemi che sono stati elencati nella Sezione 2.1.1, gli sviluppatori di VMM hanno ideato diverse soluzioni, principalmente basate su modifiche a priori del sistema guest, a livello di codice sorgente o del corrispettivo binario. Le tecniche che si affidano a modifiche del codice sorgente sono note come tecniche di

paravirtualizzazione. Due esempi di virtual machine monitor che fanno uso della paravirtualizzazione sono Xen [2] e Denali [48]. I principali vantaggi della paravirtualizzazione sono la garanzia di performance molto elevate e la possibilità di non dover modificare in alcun modo le applicazioni che vengono eseguite all'interno del sistema guest. Questo secondo vantaggio permette una grande usabilità di tali VMM che, però, hanno il problema di supportare solo un numero limitato di sistemi operativi guest, dal momento che tali sistemi devono essere modificati a priori per poter essere eseguiti all'interno di una macchina virtuale controllata da un VMM che usa la paravirtualizzazione e che non è possibile virtualizzare sistemi che non possono essere modificati. Questo problema è particolarmente rilevante nel caso si vogliano virtualizzare sistemi closed-source, come Windows. Per esempio, infatti, il VMM Xen permette di virtualizzare una versione appropriatamente modificata di Linux senza problemi, mentre riesce a virtualizzare solo parzialmente un sistema Windows.

Un secondo approccio, diverso dalla paravirtualizzazione in quanto non richiede modifiche preliminari del sistema guest, consiste nel trasformare i file binari di tale sistema al momento della virtualizzazione, per permettere al VMM di gestire correttamente tutte le problematiche evidenziate in precedenza. Per esempio, un VMM che utilizza questa tecnica è VMware [47]. Questi VMM sono in grado di gestire un numero di sistemi operativi guest maggiore rispetto alle tecniche di paravirtualizzazione ma, in compenso, non sono in grado di eguagliarne le performance, dal momento che devono effettuare le modifiche ogni volta che un sistema guest viene avviato all'interno di una macchina virtuale sotto il loro controllo.

2.1.3 Introduzione al supporto hardware alla virtualizzazione

Le difficoltà e le problematiche relative all'implementazione di un VMM software che sono state presentate nelle Sezioni precedenti di questo Capitolo hanno portato all'introduzione, sia sui processori Intel che AMD, di un supporto hardware in grado di facilitare enormemente la virtualizzazione dei processori. In questa Sezione, verranno presentati alcuni dettagli relativi a VT-x, il supporto hardware alla virtualizzazione introdotto sui processori Intel. La tecnologia VT-x mira a eliminare la maggior parte dei

problemi e delle sfide implementative che rendono la scrittura di un VMM software estremamente problematica. In questo modo, è possibile evitare di adottare soluzioni che richiedono modifiche a priori del codice del sistema operativo che si intende virtualizzare, come nella paravirtualizzazione, o del corrispondente codice binario. Il supporto hardware alla virtualizzazione permette, quindi, di implementare VMM che possono supportare molti sistemi operativi guest diversi mantenendo, allo stesso tempo, un livello di prestazioni elevato. Vengono ora illustrati alcuni dettagli della tecnologia VT-x e, in seguito, è evidenziato come tale tecnologia permetta di ovviare ai problemi illustrati nella Sezione 2.1.1.

La virtualizzazione hardware-assisted si basa principalmente sull'introduzione di due nuove modalità operative della CPU, denominate *VMX root operation* e *VMX non-root operation* [32]. La prima, spesso abbreviata in “modalità root”, è quella in cui viene eseguito il VMM. La seconda, chiamata in breve “modalità non-root”, è direttamente controllata dal VMM e è la modalità in cui viene eseguito il sistema guest di una macchina virtuale. È molto importante sottolineare come entrambe queste modalità supportino tutti e quattro i livelli di privilegio tipici di un'architettura Intel. Ciò permette di eseguire il sistema operativo guest senza doverlo deprivilegiare come invece accade nella virtualizzazione software e, allo stesso tempo, garantisce un'estrema flessibilità al VMM che, per esempio, può sfruttarli per isolare alcuni suoi componenti in un ambiente di esecuzione separato dalle parti più critiche del VMM.

Oltre a due nuove modalità operative, la tecnologia di virtualizzazione hardware-assisted definisce anche due transizioni del flusso di esecuzione. Una transizione dalla modalità root alla modalità non-root è una *VM entry*, mentre la transizione inversa è chiamata *VM exit*. Spesso, queste due transizioni sono abbreviate rispettivamente come *entry* e *exit*, per semplicità. Entrambe queste transizioni sono controllate tramite l'utilizzo di una particolare struttura chiamata *Virtual Machine Control Structure* (VMCS). Tale struttura, i cui dettagli saranno presentati nel prossimo paragrafo, contiene, tra le altre cose, due aree destinate a immagazzinare lo stato del sistema guest e dell'host, rispettivamente denominate *guest-state area* e *host-state area*. Quando viene eseguita una *entry*, la CPU si occupa di salvare lo stato dell'host nella *host-state area* e di caricare dalla *guest-state area* lo stato del sistema guest precedentemente salvato.

Viceversa, a fronte di una exit, lo stato del sistema guest viene salvato nel VMCS e viene ripristinato lo stato dell'host nei registri della CPU dalla host-state area. Prima di illustrare i dettagli del VMCS e delle entry/exit, è importante notare come, mentre in modalità root il comportamento del processore rimane sostanzialmente invariato, quando l'esecuzione è in modalità non-root, esso subisce dei cambiamenti sostanziali. Tra questi, il cambiamento più importante è che, in modalità non-root, l'esecuzione di alcune particolari istruzioni o il verificarsi di particolari condizioni causano una exit alla modalità root. Alcune istruzioni causano una exit incondizionatamente quando vengono eseguite, come per esempio la `CPUID`, e di conseguenza non possono mai venire eseguite in modalità non-root. Le altre istruzioni che potrebbero necessitare di una particolare gestione da parte del VMM e, inoltre, tutti gli eventi, causano una exit *condizionalmente*. Le condizioni che determinano se l'esecuzione di una di queste istruzioni o l'occorrenza di uno di questi eventi debba causare una exit o meno vengono configurate in una particolare area del VMCS, denominata *VM-Execution Control Fields*.

Virtual Machine Control Structure. Il VMM può accedere al VMCS facendo riferimento a un indirizzo fisico in modo da non doverlo localizzare all'interno dello spazio di indirizzamento del guest che, come sarà illustrato in seguito, può essere completamente separato da quello del VMM [32]. La regione di memoria fisica che contiene il VMCS è chiamata VMCS Region e ha una dimensione *massima* di 4-KByte mentre la sua dimensione minima è dipendente dall'implementazione del VMM che può configurare il VMCS e il suo contenuto in base alle sue necessità, in modo da ottimizzarne l'utilizzo. Inoltre, è possibile, per il VMM, definire e utilizzare un diverso VMCS per ogni macchina virtuale che intende creare, in modo da poter facilmente gestire più sistemi guest allo stesso tempo [20].

Il VMCS è organizzato in sei aree principali: *guest-state area*, *host-state area*, *VM-execution control fields*, *VM-exit control fields*, *VM-entry control fields* e *VM-exit information control fields*. Nel seguito di questa sezione vengono definite più in dettaglio tali aree e il loro scopo principale.

Guest-state area. Quest'area del VMCS è usata per contenere gli elementi dello stato del guest. Affinché il VMM funzioni correttamente, certi registri devono essere caricati a ogni exit. Tali registri includono per esempio, quelli fondamentali per il funzionamento della CPU come i registri di segmento, il registro CR3 e molti altri. Oltre a registri, la guest-state area contiene numerose altre informazioni riguardo allo stato del sistema guest che non corrispondono a registri del processore. Esempi di tali informazioni sono lo stato corrente della CPU e l'*interruptibility state*, che fornisce informazioni riguardo alla possibilità di bloccare la consegna di particolari eventi e eccezioni. Inoltre, è importante notare che la guest-state area non contiene alcuni registri che possono essere salvati e caricati dal VMM, come i registri *general purpose*. Questa scelta è dovuta al fatto che spesso non è essenziale salvare tali registri e la loro esclusione velocizza l'esecuzione di entry e exit. La gestione di questi registri è delegata al VMM, dal momento che sa meglio del processore quando è necessario salvarli e quando, invece, possono essere tralasciati.

Host-state area. Quest'area del VMCS è usata per contenere gli elementi dello stato dell'host che viene ripristinato ogni volta che avviene una exit. Contrariamente a quanto è stato detto per la guest-state area, ogni campo della host-state area corrisponde a un registro del processore. Tra tali campi si possono trovare, per esempio, il registro CR3, il registro RSP e il registro RIP. Tra questi, il registro RIP indica l'indirizzo della funzione che viene invocata quando viene eseguita una exit e corrisponde all'entry point della modalità root.

VM-execution control fields. Questi campi del VMCS controllano l'esecuzione in modalità non-root specificando quali istruzioni e quali eventi causano una exit a root mode. Dal momento che la granularità con cui è possibile controllare l'esecuzione del sistema guest tramite questi campi del VMCS è estremamente fine, in questo lavoro di tesi non verranno illustrate *tutte* le possibili configurazioni, per le quali si rimanda a [20], ma verranno analizzate solo alcune particolari condizioni che evidenziano come, effettivamente, il supporto hardware alla virtualizzazione possa facilitare l'implementazione di un VMM. Più in particolare, i VM-execution control fields includo-

no la possibilità di virtualizzare gli interrupt, permettendo di effettuare una exit ogni volta che il guest riceve un external interrupt (cioè un interrupt che non può essere mascherato tramite l'interrupt flag del registro `EFLAGS`, come i page fault) e, inoltre, permettono al VMM di richiedere che sia effettuata una exit ogniqualvolta il sistema guest sia pronto a ricevere un interrupt. Tramite queste due configurazioni è molto più immediato, per il VMM, controllare quali interrupt vadano consegnati al sistema guest e, ancora più importante, *quando* il guest è pronto a riceverli.

Tra le altre configurazioni dei VM-execution control fields, è anche importante sottolineare la possibilità di virtualizzare in modo particolarmente efficace le scritture e le letture dai registri di controllo. Tali registri, infatti, contengono alcuni bit che controllano direttamente le operazioni della CPU e è molto probabile che un VMM voglia mantenere il controllo completo degli accessi a questi particolari bit dei registri di controllo. Il VMCS contiene una maschera di bit che il VMM può utilizzare per indicare quali particolari bit di un registro di controllo vuole proteggere. Il guest può, quindi, modificare i bit non mascherati, ma qualsiasi tentativo di scrivere un bit mascherato causa una exit alla modalità root. Inoltre, per permettere eventualmente al VMM di nascondere la differenza tra il valore di un registro che il guest si aspetta di aver scritto e il suo valore effettivo, è possibile utilizzare un altro campo del VMCS il cui valore è restituito al guest ogni volta che prova a accedere in lettura a un registro di controllo i cui bit sono stati mascherati dal VMM.

Infine, per garantire un'estrema flessibilità al VMM, il VMCS contiene delle bitmap che permettono di specificare a una granularità estremamente fine qualora particolari eventi debbano causare, o meno, una exit. In particolare, sono qui riportate le due bitmap che verranno utilizzate in questo lavoro di tesi:

- **exception bitmap:** questa bitmap contiene un entry per ogni eccezione che si può verificare nel sistema. Il VMM può specificare, singolarmente per ogni eccezione, qualora essa debba causare una exit. Inoltre, per alcune eccezioni particolari, per esempio i Page Fault, è possibile anche richiedere che venga effettuata una exit solo in corrispondenza di un particolare codice di errore associato all'eccezione.

- **I/O bitmap:** questa seconda bitmap contiene un'entry per ogni porta dello spazio di I/O del sistema. Ogni volta che, all'interno del guest, viene eseguita un'operazione di I/O su di una porta il cui bit corrispondente nella bitmap è impostato, l'esecuzione di tale istruzione causa una exit a root mode.

VM-exit control fields. Questi campi controllano il comportamento delle exit. In particolare, tramite essi è possibile specificare quali informazioni aggiuntive riguardo a una exit e allo stato del sistema guest è necessario salvare nel VMCS, oltre ovviamente a quelle che vengono già salvate dall'hardware durante ogni passaggio da modalità non-root a modalità root.

VM-entry control fields. Similmente ai campi appena illustrati, i VM-entry control fields si occupano della gestione delle entry. In particolare controllano quali informazioni aggiuntive, rispetto a quelle contenute di default nel VMCS e caricate a ogni entry, vanno ripristinate nel sistema guest durante tale transizione.

VM-exit information fields. Quest'area contiene informazioni riguardo alla più recente exit avvenuta nel sistema. Indipendentemente dal tipo di exit, in tale area viene impostato dall'hardware il campo *exit reason* tramite il quale il VMM può capire qual è stato il motivo che ha causato la exit. Alcune exit, inoltre, salvano delle informazioni aggiuntive nel campo *exit qualification*. Per esempio, nel caso di un'exit scatenata da un #PF nel sistema guest, il campo *exit reason* indica che la causa della exit è stata la ricezione di un interrupt, mentre il campo *exit qualification* contiene informazioni aggiuntive che indicano, per esempio, che l'interrupt è un page fault e il relativo error code.

Entry e exit: dettagli. Come è stato evidenziato più volte in precedenza, ogni entry carica lo stato del guest dalla guest-state area del VMCS, in modo che l'esecuzione del sistema guest possa riprendere esattamente dallo stesso stato in cui era stato interrotto da una exit, a meno che, ovviamente, il VMM non cambi di proposito le informazioni contenute nel VMCS per alterare lo stato del guest. Inoltre, le entry possono essere

configurate per iniettare un evento (un'eccezione o un interrupt) nel sistema guest. Se la entry è configurata per iniettare un evento, la CPU durante il passaggio dalla modalità root alla modalità non-root, effettua tale iniezione utilizzando la IDT del sistema guest per consegnare l'evento specificato dal VMM al sistema guest. In questo modo, dal punto di vista del guest, è come se l'evento fosse accaduto appena dopo la entry.

Anche le exit si occupano di salvare e caricare gli stati, rispettivamente del guest e dell'host, nel o dal VMCS. Diversamente da quanto accade in una entry, però, in cui l'indirizzo da cui viene fatta riprendere l'esecuzione del sistema guest viene aggiornato dinamicamente nel VMCS, l'entry point da cui viene ripresa l'esecuzione del VMM è anch'esso definito nel VMCS, ma non cambia mai. Questo implica che tutte le exit utilizzano lo stesso entry point. Di conseguenza, il VMM, per identificare le ragioni che hanno causato la exit, ispeziona i VM-exit information fields precedentemente illustrati. Queste informazioni sono molto dettagliate e permettono di semplificare estremamente la progettazione dei VMM. Per esempio, si consideri una exit causata da un'istruzione che legge il contenuto del registro di controllo CR0:

```
MOV EAX, CR0
```

Il campo exit reason indica che la exit è stata causata genericamente da una lettura da un registro di controllo mentre il campo exit qualification indica specificatamente il registro di controllo coinvolto dall'istruzione (CR0), qualora l'accesso sia in lettura o in scrittura (lettura, nell'esempio) e l'ulteriore operando dell'istruzione (EAX).

2.1.4 Soluzioni ai problemi della virtualizzazione tramite VT-x

Come è stato spiegato nella Sezione 2.1.1, la maggior parte dei problemi della virtualizzazione software derivano dal fatto che è necessario deprivilegiare il sistema operativo che si vuole eseguire come guest di una macchina virtuale. Con il supporto hardware alla virtualizzazione, il guest può essere eseguito al livello di privilegio per cui è stato progettato (tipicamente ring 0), grazie all'introduzione delle due modalità operative *root* e *non-root*. In entrambe queste modalità è possibile, infatti, utilizzare tutti e quattro i livelli di privilegio tipici di un'architettura Intel. Dal momento, quindi,

che il sistema guest è già limitato dal fatto di essere eseguito in modalità non-root e che può utilizzare il massimo livello di privilegio, molti dei problemi che sono stati evidenziati in precedenza non si presentano nemmeno, come per esempio i problemi dell'*aliasing dei ring* e del degrado delle performance delle system call, o sono risolvibili molto facilmente. Vengono ora illustrati alcuni di tali problemi, spiegando come il supporto hardware per la virtualizzazione ne faciliti la gestione.

Compressione dello spazio di indirizzamento. Dal momento che il VMCS permette di specificare un valore per il registro CR3 che viene ripristinato nel sistema guest o nell'hypervisor, rispettivamente durante una entry o una exit, è possibile utilizzare due spazi di indirizzamento completamente separati, uno per il sistema guest e uno per il VMM. Di conseguenza, non è necessario che il VMM si riservi porzioni dello spazio di indirizzamento del guest con la conseguente necessità di proteggerle e nasconderle per evitare che il guest vada a accedervi e a modificarle.

Accesso a informazioni sensibili. La VT-x introduce dei cambiamenti sostanziali, che permettono di risolvere il problema dell'accesso da parte del sistema guest a informazioni privilegiate. Innanzitutto, alcuni di questi problemi non sussistono più, in quanto, per esempio, il VMM non deve più modificare la IDT del guest per intercettare gli eventi e, di conseguenza, può permettere al guest di modificare tale struttura a piacimento. Inoltre, qualora il VMM debba impedire al guest di accedere a qualche informazione normalmente a sua disposizione, il supporto hardware alla virtualizzazione fornisce la possibilità di configurare il VMCS in modo che un tentativo da parte del guest di accedere a tali informazioni risulti in una exit.

Virtualizzazione degli interrupt. Come è stato detto in precedenza, il VMM deve poter controllare completamente gli interrupt e, di conseguenza, non può permettere al sistema guest di mascherarli a piacimento. Il supporto hardware alla virtualizzazione include la possibilità di generare una exit ogni volta che il guest tenta di controllare il mascheramento degli interrupt. Intercettando questo evento, il VMM può impedire il mascheramento e, allo stesso tempo, registrare il fatto che il guest non è disposto a

ricevere interrupt. Inoltre, è possibile configurare il VMCS in modo che venga generata una exit anche quando il guest è pronto a ricevere gli interrupt. In questo modo, il VMM può eventualmente iniettare, tramite una entry come spiegato nella Sezione 2.1.3, gli interrupt di interesse per il sistema guest ricevuti mentre quest'ultimo non era disposto a accettarli.

Accessi frequenti a risorse privilegiate. La virtualizzazione hardware-assisted permette di evitare l'overhead introdotto qualora il guest acceda frequentemente a risorse privilegiate. Questo è possibile grazie all'estrema granularità con cui è possibile configurare quali particolari condizioni causano una exit che, di conseguenza, permette di evitare di dover interrompere l'esecuzione del guest per eventi che non sono di particolare interesse per il VMM.

2.1.5 Uso della virtualizzazione hardware-assisted

Nonostante il supporto hardware alla virtualizzazione sia presente sulla maggior parte dei processori Intel più moderni, a partire dai modelli Pentium 4, e nonostante i numerosissimi vantaggi che questo supporto fornisce allo sviluppo di VMM, come illustrato nella Sezione 2.1.3, questa tecnologia risulta ancora poco utilizzata al giorno d'oggi. Soltanto recentemente, infatti, i maggiori virtualizzatori software come VMware [47] e VirtualBox [45] hanno iniziato a sfruttare alcune caratteristiche della virtualizzazione hardware-assisted per migliorare le loro prestazioni.

2.2 Concetti di debugging

Il debugging è un'attività volta all'identificazione e alla correzione di errori presenti in un programma tramite l'esecuzione passo passo e l'osservazione dello stato del sistema. Nel presente lavoro di tesi saranno richiamati diverse volte concetti e pratiche comuni nel debugging, spesso anche entrando nei dettagli di *come* queste vengono implementate praticamente nei tool che consentono di effettuare questa attività. In que-

sta Sezione vengono descritte brevemente i concetti i cui dettagli potrebbero risultare meno intuitivi.

2.2.1 Breakpoint

I breakpoint sono una particolare funzionalità di debugging che permette di interrompere l'esecuzione di un programma in un punto arbitrario della sua esecuzione. Tale funzionalità risulta molto utile, se non addirittura essenziale, in quanto permette a chi effettua il debugging di interrompere l'esecuzione solamente nei punti che intende analizzare. Quando un programma raggiunge un breakpoint, infatti, la sua esecuzione viene temporaneamente bloccata, permettendo a chi sta effettuando l'analisi di andare a esplorarne lo stato, per esempio ispezionando lo stack, le locazioni di memoria utilizzate o i valori dei registri. Esistono diverse implementazioni dei breakpoint, suddivise in due categorie principali: *hardware breakpoint* e *software breakpoint*. Come si può facilmente intuire dai loro nomi, i primi richiedono un qualche supporto hardware per essere utilizzati, mentre i secondi sono implementati puramente a livello software.

Breakpoint hardware. Molti processori includono il supporto per impostare breakpoint hardware. Tale supporto consiste in un insieme di registri (debug register) e in un meccanismo a eccezioni per impostare e segnalare il raggiungimento di un breakpoint. La limitazione principale dei breakpoint hardware consiste nel fatto che, essendo appunto implementati via hardware, il loro numero è limitato e coincide con il numero di debug register messi a disposizione dal processore. Si noti, inoltre, che i debug register sono utilizzati anche per i watchpoint, come sarà spiegato in seguito, quindi il limite numerico è ancora più stringente. Di contro, però, i breakpoint hardware risultano estremamente efficienti.

Breakpoint software. Contrariamente ai loro corrispettivi hardware, è possibile impostare un numero arbitrario di breakpoint software che, di conseguenza, rappresentano una soluzione più scalabile. Esistono due varianti principali di breakpoint software. La prima variante richiede accesso al codice sorgente dell'applicazione in cui si vogliono impostare i breakpoint e l'inserimento avviene in fase di compilazione. Il

secondo metodo, invece, sfrutta una particolare istruzione che causa il sollevamento di una *breakpoint exception* quando vengono eseguite, cioè la `INT 3`, nell'istruzione set Intel. Tale istruzione è costituita da un solo opcode, che viene sostituito al primo opcode dell'istruzione su cui si vuole impostare il breakpoint, il quale viene salvato in modo da poter essere ripristinato in seguito. Quando il debugger viene notificato dal sistema della breakpoint exception scatenata dall'istruzione `INT 3`, tipicamente notifica l'evento all'utente e ripristina l'opcode originale. I metodi con cui tali breakpoint vengono resi persistenti, cioè quando vengono ripristinati ogni volta che vengono raggiunti, interrompendo quindi *ogni* esecuzione dell'istruzione su cui sono stati impostati, dipendono da debugger a debugger e non verranno analizzati. L'utilizzo dell'istruzione `INT 3` per impostare i breakpoint risulta essere una soluzione molto più dinamica della prima, in quanto permette di impostare breakpoint senza dover ricompilare il programma analizzato, però ha una seria limitazione dovuta alla necessità di dover modificare gli opcode delle istruzioni su cui si vuole impostare un breakpoint. Per un programma che vuole impedire di essere debuggato è, quindi, estremamente semplice andare a identificare la presenza di breakpoint impostati sul suo codice e, eventualmente, interrompere l'esecuzione o assumere comportamenti diversi. Tale problema è particolarmente rilevante per quanto riguarda l'analisi e il debugging di programmi malevoli.

2.2.2 Watchpoint

I watchpoint sono un'altra funzione essenziale del debugging e consistono nella possibilità di interrompere l'esecuzione dell'istruzione analizzata ogni volta che viene acceduta una particolare locazione di memoria, in lettura o in scrittura. La possibilità di monitorare tutte le scritture e le letture in una particolare area di memoria è particolarmente utile per identificare, per esempio, le istruzioni responsabili di scrivere valori non previsti in tale area o quando una zona di memoria contiene valori particolari. Come nel caso dei breakpoint, anche per i watchpoint esistono due categorie principali chiamate *hardware watchpoint* e *software watchpoint*, di cui vengono ora illustrate brevemente le caratteristiche principali.

Watchpoint hardware. Similmente all'implementazione dei breakpoint hardware, anche tale categoria di watchpoint fa affidamento ai debug register del processore. Conseguentemente, anche i watchpoint hardware soffrono delle stesse limitazioni, causate dall'esiguo numero dei debug register e dal fatto che tali registri siano condivisi anche con eventuali breakpoint hardware. Di contro, però, mentre le differenze di efficienza tra breakpoint hardware e software, per quanto presenti, non sono eccessive, l'efficienza dei watchpoint hardware non è nemmeno paragonabile a quella dei watchpoint software.

Watchpoint software. Purtroppo, dal momento che non è generalmente possibile fare in modo che l'accesso a una zona di memoria causi un'eccezione, se non adottando particolari operazioni che richiedono un livello di privilegio elevato per essere effettuate, le implementazioni tipiche dei watchpoint software risultano estremamente inefficienti. Una possibile implementazione, infatti, consiste nell'eseguire il programma sotto analisi in modalità single step, che sarà illustrata nella Sezione 2.2.3, in modo da poter controllare per *ogni* istruzione a quali zone di memoria accede, se vi accede in lettura o in scrittura e, in questo secondo caso, che valore sta scrivendo. Come si può facilmente intuire, la necessità di dover interrompere l'esecuzione a ogni istruzione al fine di interpretarla e controllare qualora stia accedendo a una zona di memoria su cui è stato impostato un watchpoint introduce un notevole overhead nell'esecuzione del programma analizzato, rendendo i watchpoint software estremamente inconvenienti rispetto ai corrispettivi basati sui debug register.

2.2.3 Single stepping

Il single stepping è una particolare modalità di esecuzione che consiste nell'eseguire un'istruzione alla volta, interrompendo l'esecuzione del programma analizzato e restituendo il controllo all'utente dopo ogni istruzione. Questa modalità è spesso utilizzata per analizzare l'evoluzione di un programma istruzione per istruzione, al fine di verificarne gli effetti sulla memoria o sui registri del processore. Inoltre, tale modalità viene usata anche per implementare altre funzionalità di debugging, come nel caso dei

watchpoint software. Per eseguire una singola istruzione, il debugger utilizza un particolare registro, chiamato EFLAGS, che contiene alcune informazioni riguardo allo stato del processore. Impostando a 1 il bit *Trap Flag* (TF) di tale registro, il processore esegue una sola istruzione e genera una debug exception che può venir intercettata dal debugger per restituire il controllo all'utente.

Capitolo 3

Scenario d'utilizzo

In questa Sezione viene presentata una breve panoramica sul problema dell'analisi di codice kernel e sulle soluzioni attuali a questo problema. Quindi, viene presentata la tecnica proposta in questo lavoro di tesi, i vantaggi ottenibili tramite essa e, brevemente, le difficoltà principali affrontate nel realizzarla.

3.1 Kernel debugging

Il *debugging* è un'attività finalizzata all'individuazione di porzioni di codice affette da errori all'interno di un programma. Questa attività è fondamentale nel processo di sviluppo di qualsiasi tipo di software e, solitamente, la sua difficoltà cresce proporzionalmente alla complessità del software analizzato. Il debugging è particolarmente critico quando viene effettuato sui componenti del *kernel* di un sistema operativo. Il kernel è la parte fondamentale di qualsiasi sistema in quanto si occupa, tra le altre cose, di gestire l'accesso alle risorse hardware di una macchina e di garantire che le applicazioni eseguite sulla macchina rispettino le politiche di protezione (accesso alle risorse) stabilite dal sistema. Come si può facilmente intuire, il kernel svolge un ruolo chiave per il corretto funzionamento di una macchina e anche un errore banale o un collo di bottiglia all'interno di un suo componente può avere effetti disastrosi. Di conseguenza, è necessario che gli sviluppatori di tali componenti pongano particolare cura nella loro implementazione e che siano dotati di strumenti adatti a verificare il loro codice.

I due principali tipi di kernel utilizzati correntemente sono: *monolitici* e *microkernel*. Nel primo caso, tutti i componenti del kernel, chiamati *moduli*, sono strettamente integrati al fine di ottenere un'ottima efficienza a scapito di semplicità e robustezza. Infatti, un difetto in un singolo modulo, per quanto separato dagli altri moduli, può facilmente portare a un crash dell'intero sistema e riuscire a risalire alla porzione di codice errata può risultare particolarmente oneroso a causa delle dimensioni di questi kernel. Gli esempi più famosi di kernel monolitici sono Linux [23] e Windows [49]. Nel caso di architetture a microkernel, come quella adottata da Minix [31], si tende a sacrificare le performance in favore di maggiore semplicità e robustezza. In tale architettura, il codice kernel è ridotto all'osso, mentre i moduli vengono implementati sotto forma di server che possono essere interrogati dai programmi in esecuzione per interagire con l'hardware sottostante. L'approccio di debugging proposto in questo lavoro di tesi si focalizza sui kernel monolitici dal momento che sono i più utilizzati dai sistemi operativi moderni e perché risultano essere i più critici da analizzare.

Gli sviluppatori kernel, per facilitare l'implementazione e l'analisi di componenti del sistema operativo, hanno a disposizione diversi strumenti e tecniche. Alcuni di questi sono specifici per l'*analisi statica* del codice, mentre altri per l'*analisi dinamica*. L'analisi statica è effettuata senza eseguire il programma da verificare ma analizzandone il codice sorgente o il corrispondente codice macchina. Tali tecniche spaziano dalla "semplice" ricerca di errori di programmazione ai più complessi *metodi formali*, che possono dimostrare matematicamente determinate proprietà del codice analizzato. L'analisi statica descrive fedelmente il comportamento di un software, indipendentemente da una specifica esecuzione ma è eccessivamente *conservativa*, nel senso che riporta proprietà più deboli di quanto siano realmente [33]. Di contro, l'analisi dinamica estrae le informazioni utili al debugging direttamente dall'esecuzione del codice da analizzare. Tale approccio garantisce estrema precisione, inibisce infatti il problema della conservatività dell'analisi statica, ma allo stesso tempo è eccessivamente specifico, infatti ogni analisi è strettamente correlata a una singola esecuzione. Uno svantaggio ulteriore dell'applicazione dell'analisi dinamica al codice kernel deriva dal fatto che un bug in un programma eseguito a livello utente (ring 3) molto probabilmente causa un crash del programma stesso, mentre un errore nel kernel por-

ta quasi sicuramente alla necessità di riavviare la macchina, rendendo quindi difficile investigare sul problema. Nonostante ciò, la precisione e la relativa semplicità dell'analisi dinamica, hanno reso molto popolari, presso gli sviluppatori kernel, i tool che adottano questo approccio.

Gli approcci esistenti per l'analisi dinamica dei sistemi operativi e del loro kernel possono essere classificati in due gruppi: basati sul kernel e basati su Virtual Machine Monitor. Le caratteristiche dei due gruppi vengono brevemente illustrate nel seguito di questa Sezione mentre, per alcuni esempi di strumenti che implementano queste tecniche, si rimanda alla Sezione 6.

3.1.1 Analisi dinamica basata sul kernel

Il primo approccio considerato per l'analisi dinamica del kernel di un sistema operativo consiste nell'inserire, all'interno del kernel, componenti che lo instrumentino al fine di fornire dei "punti di aggancio" (hook). Tipicamente, questi hook vengono inseriti in corrispondenza degli eventi più rilevanti, come la creazione di nuovi processi, l'esecuzione di system call e l'esecuzione di particolari funzioni del kernel. In risposta all'occorrenza di uno di questi eventi, vengono invocate determinate azioni per recuperare informazioni rilevanti ai fini dell'analisi [3, 42, 40, 30, 24]. Tale approccio risulta utile specialmente per quanto riguarda tipologie di analisi "passive", come il *profiling*, attività volta a raccogliere informazioni relative all'esecuzione di un programma al fine di determinare quali parti ne possono essere ottimizzate, e il *tracing*, che consiste nell'analizzare le chiamate di sistema effettuate da un programma (o le funzioni kernel invocate, nel caso di un sistema operativo).

I problemi principali di questo approccio però, sono causati proprio dall'installazione degli *hook*, dal momento che risultano particolarmente invasivi per il sistema da analizzare. Infatti, per permettere l'intercettazione dinamica degli eventi di interesse, sono necessarie diverse modifiche delle strutture del sistema operativo. In primo luogo, tali modifiche sono facilmente rilevabili quindi, per esempio nel caso sia necessario analizzare un sistema potenzialmente compromesso, un programma malevolo eseguito a ring 0 può rilevare la presenza dello strumento di analisi e tentare di corromperne i

risultati per nascondere la sua presenza. Un'ulteriore difficoltà può nascere dal fatto che, nel caso di sistemi operativi come Windows, il codice sorgente non è disponibile e le strutture del kernel non sono completamente documentate, rendendo molto difficile, se non impossibile, modificarle ai fini del debugging. Ultimo, ma non meno importante, le modalità di inserimento degli hook sono fortemente dipendenti dal sistema che deve essere analizzato. Le strutture che devono essere modificate, infatti, sono diverse per ogni tipo di kernel, rendendo quindi necessario un approccio specifico per ognuno dei sistemi da analizzare.

3.1.2 Analisi dinamica basata su Virtual Machine Monitor

La seconda tipologia di analisi dinamica consiste nell'eseguire l'intero sistema da analizzare all'interno di una Virtual Machine. In questo modo, sia il sistema operativo che le applicazioni user-space vengono eseguite sotto il controllo del Virtual Machine Monitor (VMM) e, di conseguenza, è possibile intercettare gli eventi direttamente da questo componente *senza* modificare le strutture del sistema operativo [14]. Tale possibilità garantisce una pressoché totale trasparenza al sistema sotto analisi, quindi, anche un programma, come un driver, in esecuzione con privilegi di massimo livello (ring 0), contrariamente a quanto accade nell'approccio precedente, non dovrebbe essere in grado di accorgersi della presenza di uno strumento di analisi. Un ulteriore vantaggio apportato da questo approccio è la scarsa dipendenza dal sistema operativo analizzato. Infatti, è possibile definire direttamente tramite il VMM quali eventi devono far scattare una determinata funzione di analisi, per esempio un timer nel caso di porzioni di codice che devono essere sottoposte a profiling o la registrazione di una system call in un log, nel caso del tracing. In questo caso, quindi, il VMM offre un livello di astrazione che permette di ignorare i dettagli implementativi del sistema analizzato, rendendo possibile per il debugger analizzare tali eventi indipendentemente dal sistema analizzato.

Questa soluzione, per quanto elegante, non è esente da problemi. In primo luogo, il sistema da analizzare *deve essere avviato* all'interno di una Virtual Machine. Inoltre, la necessità di eseguire il sistema all'interno di una macchina virtuale pone un pro-

blema rilevante nel caso in cui sia necessario analizzare un driver per uno specifico device hardware. Infatti, la virtualizzazione software tipicamente virtualizza i device hardware per permettere a più guest di accedere a tali risorse contemporaneamente. Di conseguenza, questo approccio non è adatto per l'analisi di componenti di un sistema operativo che richiedono di interagire direttamente con l'hardware sottostante, come un *device driver*. Infine, il terzo problema deriva dal fatto che Paleari *et al.* hanno dimostrato in [28, 29] che l'assunzione di trasparenza effettuata precedentemente riguardo ai VMM software non è valida. Inoltre, in un lavoro di ricerca complementare, gli autori dimostrano che è possibile creare piccoli programmi, denominati *red pill*, che sono in grado di determinare se stanno venendo eseguiti all'interno di una Virtual Machine o su una macchina reale [35]. Tipicamente, l'analisi di malware viene effettuata su macchine virtuali per impedire al codice maligno di provocare danni e propagare l'infezione a altre macchine. Le *red pill* possono essere utilizzate da malware particolarmente evoluti per determinare se stanno venendo eseguiti su una Virtual Machine e, conseguentemente, decidere di non effettuare nessuna operazione maligna per vanificare gli scopi dell'analisi. Quindi, questo approccio potrebbe rivelarsi inadatto all'analisi di tipologie malware che adottano queste tecniche di difesa.

3.2 Hypervisor come soluzione

Riassumendo, le due soluzioni al problema dell'analisi e debugging di codice kernel analizzate nella prima parte di questa Sezione, presentano vantaggi e svantaggi. La prima, basata sull'installazione di hook all'interno del kernel, permette di effettuare il debugging di un sistema operativo direttamente sulla macchina su cui viene eseguito, con lo svantaggio di doverne modificare le strutture, penalizzando quindi la trasparenza. La seconda soluzione, che sfrutta la virtualizzazione software, garantisce una *quasi totale* trasparenza [35], ma richiede che il sistema operativo il cui kernel deve essere verificato sia eseguito nativamente all'interno di una macchina virtuale.

La tecnica di analisi proposta in questo lavoro di tesi si propone l'obiettivo di ottenere gli stessi vantaggi di *entrambe* le soluzioni precedenti e, allo stesso tempo, di evitarne gli svantaggi caratteristici. Per raggiungere questo obiettivo, l'approccio pro-

posto sfrutta il supporto hardware per la virtualizzazione presente su ogni processore moderno che permette di sviluppare Virtual Machine Monitor particolarmente efficienti, completamente trasparenti e non invasivi per i sistemi guest. In particolare, per eliminare la debolezza intrinseca nel secondo approccio analizzato, cioè l'impossibilità di analizzare sistemi che girano al di fuori di una Virtual Machine, la tecnica proposta utilizza una caratteristica dell'hardware che permette di installare un VMM e *migrare un sistema in esecuzione all'interno di una Virtual Machine*. Inoltre, una volta terminata la sessione di analisi, il VMM può essere disabilitato così da ripristinare lo stato originale della macchina analizzata. Giunti a questo punto della trattazione, il lettore potrebbe trovare confusionario l'uso dei termini relativi alla virtualizzazione. Per maggiore chiarezza, nel prosieguo di questo lavoro, i termini *hypervisor* e *Virtual Machine Monitor* verranno utilizzati interscambiabilmente e, tranne dove diversamente specificato, faranno riferimento al relativo componente implementato sfruttando il supporto hardware per la virtualizzazione. Invece, per una dettagliata digressione sulle differenze tra un hypervisor implementato *puramente* a livello software e uno implementato tramite il sopracitato supporto hardware, si rimanda alla Sezione 2.

Per dimostrare l'efficacia della tecnica proposta, è stato sviluppato un debugger kernel interattivo che opera a livello hypervisor e che implementa la tecnica appena illustrata, chiamato HYPERDBG. I dettagli di HYPERDBG verranno presentati nella Sezione 4.

3.3 Vantaggi dell'approccio con hypervisor

Contrariamente alle due analizzate in precedenza, la tecnica di debugging tramite hypervisor proposta in questo lavoro di tesi è:

- **completamente dinamica**, non necessita infatti di ricompilare o riavviare il sistema sotto analisi. Di conseguenza, la tecnica proposta può essere utilizzata per analizzare qualsiasi sistema, indipendentemente dal fatto che sia sprovvisto di supporto nativo per l'strumentazione e che non sia eseguito all'interno di una macchina virtuale;

- **trasparente**, il codice e le strutture interne del debugger sono direttamente collegate all'hypervisor e protette da questo, quindi non è possibile, neanche per un programma che gira a ring 0, identificarne la presenza;
- **non invasiva**, le analisi possono essere effettuate su un sistema senza necessità di modificarlo e, come verrà spiegato più avanti, l'unico componente che deve essere installato sulla macchina da analizzare è un piccolo driver che si occupa di caricare l'hypervisor a runtime;
- **indipendente dal sistema operativo**, dal momento che non è necessario instrumentare il sistema analizzato, l'hypervisor e il debugger sono completamente *platform independent*. Come si illustrerà nel seguito del lavoro di tesi, la tecnica è stata migliorata con alcuni componenti dipendenti dal sistema operativo, ma questi sono completamente opzionali e forniti al solo scopo di facilitare il debugging.

Capitolo 4

HYPERDBG

In questa Sezione viene presentata in maggiore dettaglio l'architettura di HYPERDBG e le sue funzionalità principali. Inoltre, vengono presentati i motivi per cui l'approccio di debugging proposto in questo lavoro di tesi apporta innovazioni sostanziali allo stato dell'arte dell'analisi dinamica di codice kernel.

4.1 HYPERDBG: Hypervisor Debugger

L'obiettivo della tecnica di debugging proposta in questo lavoro di tesi, implementata in HYPERDBG, è quello di fornire la possibilità di analizzare e verificare codice kernel in un modo che sia allo stesso tempo *completamente dinamico, trasparente* al sistema sotto analisi, *non invasivo e indipendente* dal sistema operativo analizzato. Queste proprietà sono tutte altamente desiderabili in uno strumento di debugging, dal momento che tali strumenti sono essenziali per svariati scopi che spaziano dall'analisi di codice critico all'analisi di sistemi potenzialmente compromessi. Proprio in questo secondo caso, la proprietà di trasparenza risulta essere essenziale perché, come è stato evidenziato in Sezione 2, esistono malware che operano a livello del kernel e sono in grado di identificare la presenza di strumenti di analisi e di nascondere le loro azioni qualora si accorgano di essere analizzati. Come sarà illustrato nel seguito di questa Sezione, per garantire le proprietà sopracitate, HYPERDBG sfrutta le funzionalità di virtualizzazione hardware-assisted fornite dai processori più recenti.

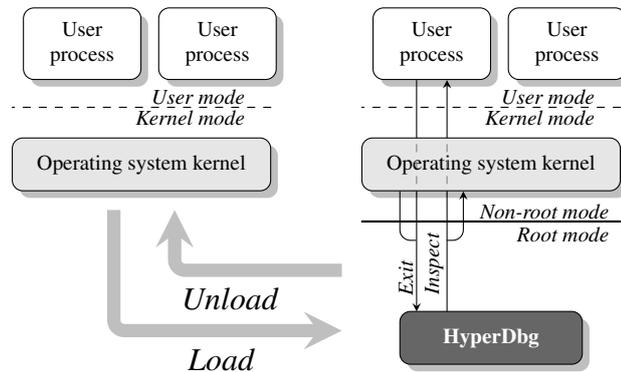


Figura 4.1: Installazione di HYPERDBG

La Figura 4.1 rappresenta, con un basso livello di dettagli, l'architettura di HYPERDBG, la fase di installazione e di rimozione e la migrazione del sistema operativo e delle sue applicazioni all'interno di una Virtual Machine. HYPERDBG è composto da due parti principali: un Virtual Machine Monitor (VMM) e il tool di analisi vero e proprio. Come nei tradizionali approcci basati su VMM, illustrati nella Sezione 3.1.2, lo strumento di analisi è eseguito all'interno del VMM in modo da essere contemporaneamente *trasparente* al sistema operativo analizzato e protetto da interferenze da parte di applicazioni eseguite all'interno della macchina virtuale. Però, contrariamente a quanto accade per gli approcci basati su VMM tradizionali, l'approccio implementato da HYPERDBG non richiede che il sistema venga avviato nativamente all'interno di una virtual machine ma permette di analizzare anche sistemi già in esecuzione. Per ottenere ciò, come abbiamo già accennato, utilizza le funzionalità di supporto alla virtualizzazione messe a disposizione dai moderni processori, tipicamente inutilizzate nella maggior parte dei sistemi. Richiamando brevemente la virtualizzazione hardware-assisted, già descritta nella Sezione 2, questa funzionalità estende un'architettura classica con due nuove modalità operative, che si aggiungono a quelle tradizionali. Queste due modalità sono, nella terminologia Intel, *VMX root operation* e *VMX non-root operation*. Tramite l'uso di queste due modalità, è possibile separare logicamente il VMM dal sistema operativo, senza richiedere modifiche a questo secondo elemento, contrariamente a quanto accade con la virtualizzazione puramente

software dove, per garantire la separazione tra le due parti si applicano tecniche che deprivilegiano il sistema operativo, che però non sono esenti da problemi, come è stato evidenziato nella Sezione 2.

Per risolvere il problema causato dalla necessità di avviare il sistema da analizzare *nativamente* dentro una virtual machine, HYPERDBG sfrutta una particolare caratteristica della virtualizzazione hardware-assisted che permette il *late launching* (lancio ritardato) delle modalità VMX. Questa caratteristica consente l'installazione di un VMM anche se il sistema è già stato avviato, di fatto risolvendo il problema delle tecniche basate su VMM puramente software. Il late launching permette di migrare, anche solo temporaneamente, un sistema operativo in esecuzione all'interno di una macchina virtuale. D'ora in avanti, il termine *guest* verrà utilizzato per riferirsi al sistema sotto analisi che è stato migrato, tramite questa tecnica, all'interno di una macchina virtuale. Una volta effettuata la migrazione, le analisi e il debugging possono essere effettuati direttamente tramite il VMM, senza nessun bisogno di modificare, né staticamente né a runtime, il codice o le strutture del guest.

4.2 Architettura

Il concetto di “migrazione” di un sistema operativo in esecuzione, utilizzato nella Sezione precedente è, in realtà, puramente virtuale. Infatti, il sistema operativo non viene spostato né modificato in nessun modo durante il caricamento dell'hypervisor. Di fatto, dopo l'attivazione delle due modalità operative aggiunte dalla virtualizzazione hardware-assisted, il sistema operativo viene semplicemente associato alla nuova modalità di esecuzione *non-root*, diventando, a tutti gli effetti, un guest della macchina virtuale controllata dall'hypervisor. Seguendo lo stesso principio dell'attivazione, l'hypervisor può essere disattivato e scaricato, in modo da ripristinare la modalità di esecuzione originale del sistema guest, semplicemente disabilitando le modalità aggiuntive VMX.

La Figura 4.2 rappresenta in maggiore dettaglio un sistema dopo l'avvio di HYPERDBG. L'esecuzione del sistema guest continua esattamente come prima, anche per quanto riguarda l'interazione con l'hardware, come mostrato dalle frecce che parto-

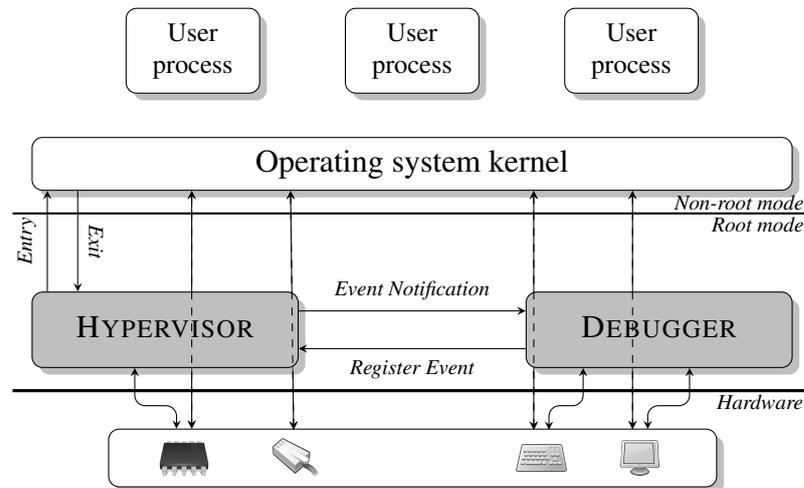


Figura 4.2: Dettaglio del sistema dopo l'avvio di HYPERDBG

no dal kernel e raggiungono direttamente i device, passando indisturbate attraverso l'hypervisor. Tuttavia, l'esecuzione del guest può essere interrotta da una *exit*. Come le eccezioni hardware, queste exit sono eventi che bloccano l'esecuzione del guest, effettuano il passaggio dalla modalità non-root alla modalità root e trasferiscono il controllo dell'esecuzione all'hypervisor. Contrariamente a quanto accade con le eccezioni hardware, però, esistono due tipi di exit, come è stato spiegato nella Sezione 2: *condizionate* e *non condizionate*. Le exit non condizionate causano *sempre* un passaggio dalla modalità non-root a root, mentre l'hypervisor può selezionare un sottoinsieme di exit condizionate che trasferiscono il controllo e inibire le restanti. In particolare, HYPERDBG è stato progettato in modo che il componente di debug possa decidere quali exit condizionate abilitare. Questa possibilità di scelta è fondamentale per permettere il debugging di un sistema, altrimenti il numero di exit sarebbe troppo elevato e renderebbe praticamente impossibile l'analisi del sistema guest. Le exit non condizionate, al contrario, vengono interamente gestite dall'hypervisor e risultano essenziali per permettere a questo componente di garantire l'integrità del proprio codice e dei propri dati. Quando si verifica una exit, esattamente come accade per le eccezioni, l'hypervisor invoca un'appropriata funzione di gestione e, eventualmente, notifica il debugger. Maggiori dettagli riguardanti la gestione delle singole exit, sia condizionate

che non condizionate, verranno forniti nella Sezione 4.3.2.

Infine, l'ultimo aspetto che va notato nella Figura 4.2 è che i due componenti di HYPERDBG interagiscono direttamente con alcuni device hardware. In particolare, l'hypervisor interagisce con la memoria (ovviamente tramite l'MMU, che non può essere bypassata a livello software) per garantire protezione per i suoi componenti e, contemporaneamente, per permettere al debugger di ispezionare aree che non corrispondono al processo correntemente schedato dal sistema guest. Il componente di debug, invece, interagisce direttamente con la tastiera e con la scheda video, per permettere l'interazione con l'utente. È stato quindi necessario implementare gli opportuni driver per interagire con i device dalla modalità root. I dettagli dell'algoritmo per la gestione della memoria e dei driver implementati verranno presentati nella Sezione 4.3.

4.3 Dettagli

In questa Sezione vengono analizzati in dettaglio i diversi componenti di HYPERDBG e le funzionalità di debugging offerte all'utente.

4.3.1 Installazione dell'hypervisor

HYPERDBG viene installato sul sistema da analizzare tramite un driver kernel minimale. Purtroppo, questo passaggio è indispensabile dal momento che le operazioni necessarie per effettuare l'installazione devono essere eseguite a ring 0 e, di conseguenza, possono essere effettuate solamente dal kernel del sistema operativo. Tuttavia, il driver è estremamente semplice e è stato sviluppato con estrema attenzione per evitare qualsiasi interferenza con il kernel. Inoltre, dal momento che, una volta caricato, HYPERDBG è completamente invisibile al sistema, il driver di installazione può essere eliminato appena dopo aver completato le operazioni necessarie.

Quando le modalità VMX vengono abilitate, viene anche creata una struttura dati chiamata *Virtual Machine Control Structure* (VMCS). Questa struttura, inizialmente, è accessibile dal driver responsabile di caricare HYPERDBG, mentre, dopo la terminazione dell'installazione, rimane accessibile solo dall'hypervisor. Il VMCS contiene

l'*Host State*, il *Guest State* e gli *Execution Control Fields*. L'*Host State* raccoglie le informazioni riguardanti lo stato del processore che viene caricato durante una *exit* e contiene lo stato di tutti i registri della CPU, con la sola esclusione dei registri general purpose (EAX, EBX, ECX e EDX). Analogamente, il *Guest State* contiene lo stato del processore che viene ripristinato durante una *entry*. I valori contenuti in quest'area vengono aggiornati automaticamente ogni volta che avviene una *exit*, così che la successiva *entry* riprenderà l'esecuzione del guest dalla medesima situazione. Infine, gli *Execution Control Fields* definiscono quali eventi devono provocare una *exit*.

Riassumendo, i compiti del driver per il caricamento di HYPERDBG consistono nell'abilitare le modalità VMX e configurare le tre parti del VMCS in modo che l'esecuzione del sistema operativo e delle applicazioni user-space continui in modalità non-root, mentre HYPERDBG viene eseguito in modalità root. Inoltre, il driver deve configurare gli *Execution Control Fields* in modo tale che tutti gli eventi che devono essere intercettati per permettere al debugger di analizzare il sistema provochino le *exit* corrispondenti. Quando queste operazioni di inizializzazione sono completate, il driver si disattiva e fa riprendere l'esecuzione del sistema guest.

Configurazione Guest State. Durante l'installazione, l'area *Guest State* è inizializzata allo stato corrente del sistema. In questo modo, quando l'installazione sarà completata, l'esecuzione del guest riprenderà in modalità non-root esattamente come se non fosse successo nulla. Il problema principale della fase di inizializzazione della modalità non-root riguarda la gestione della memoria. Infatti, è necessario evitare fin da subito che il sistema appena diventato guest possa accedere alle aree di memoria riservate dal driver all'hypervisor. Altrimenti, come sarà spiegato più dettagliatamente nel prossimo paragrafo, il guest potrebbe identificare la presenza dell'hypervisor e addirittura corromperne il funzionamento. Le CPU più moderne mettono a disposizione un meccanismo hardware per la virtualizzazione della memoria, per esempio l'estensione Intel Extended Page Table. Qualora questa estensione non sia disponibile, la virtualizzazione della memoria deve essere implementata completamente via software in modo da intercettare tutte le operazioni del guest che vanno a manipolare le Page Table, cioè le strutture che la CPU usa per effettuare la traduzione di un indirizzo vir-

tuale in fisico, come sarà dettagliatamente illustrato nel prossimo paragrafo. Tramite la virtualizzazione della memoria, l'hypervisor si può assicurare che nessuna delle aree sotto il suo controllo possa essere acceduta dal guest.

Configurazione Host State. L'Host State è inizializzato in modo che la CPU usi, in modalità root, uno spazio di indirizzamento privato e una Interrupt Descriptor Table (IDT) dedicata. Questa configurazione semplifica la separazione tra il sistema analizzato e HYPERDBG e permette di gestire gli interrupt e le eccezioni che avvengono in modalità root. Contrariamente a quanto accade per l'entry point del guest, che viene reimpostato a ogni exit in modo che la successiva entry riprenda l'esecuzione da dove era stata interrotta, l'entry point della modalità root è prefissato. Tale entry point è impostato in modo che, a ogni exit, venga eseguita una routine apposita che si occupa di verificare la tipologia di exit e di invocare la corrispondente funzione di gestione. Una descrizione più dettagliata di come vengono gestite le diverse tipologie di exit viene fornita nella Sezione 4.3.2.

Per fare in modo che lo spazio di indirizzamento dell'hypervisor sia separato da quello del sistema guest, viene allocato uno spazio sufficiente a contenere le strutture utilizzate per la gestione della memoria e vengono copiate, all'interno di questo spazio, le medesime strutture utilizzate dal driver che si occupa di effettuare il caricamento. L'indirizzo fisico di quest'area di memoria viene salvato nel Host State del VMCS in modo che venga ripristinato correttamente a fronte di ogni exit.

Configurazione Execution Control Fields. L'unica configurazione necessaria per gli Execution Control Fields riguarda gli eventi che causano exit alla modalità root. Per ridurre l'overhead introdotto nel sistema guest dalla virtualizzazione, questi campi sono impostati per minimizzare il numero di exit condizionate generate. Durante l'inizializzazione, vengono specificati solo gli eventi necessari ai fini del debugging. Eventualmente, qualora fossero richieste analisi particolari che necessitano di gestire eventi non previsti inizialmente, questi campi possono essere modificati in modo da rispondere alle nuove necessità.

4.3.2 Gestione delle VM Exit

Come è stato descritto nel Capitolo 2, le exit consistono in una transizione del controllo dal sistema guest all'hypervisor. Le exit vengono causate da particolari eventi del guest che richiedono l'attenzione dell'hypervisor, per esempio l'esecuzione di alcune istruzioni, come la `CPUID`. Indipendentemente dal fatto che una exit sia *condizionata* o *non condizionata*, la ragione che l'ha scatenata viene registrata nel VMCS. Entrando maggiormente nei dettagli, tutte le exit specificano una "exit reason" che indica, per esempio, quale istruzione ha causato il passaggio, mentre solamente alcune indicano anche una "exit qualification", che aggiunge dettagli utili alla comprensione della ragione della exit. Per esempio, una exit può essere generata da un'istruzione simile a questa:

```
MOV CR0, EAX
```

che scrive il contenuto del registro `EAX` nel registro `CR0`. In questo caso, la exit reason registrata nel VMCS indica un *accesso a un registro di controllo* mentre la exit qualification contiene (I) lo specifico registro di controllo acceduto (`CR0` nell'esempio), (II) se l'accesso è in lettura o in scrittura e (III) il registro sorgente o destinazione dell'istruzione (`EAX` nell'esempio).

Come è stato illustrato in Figura 4.2, HYPERDBG è suddiviso in due componenti principali: un core, costituito dall'hypervisor vero e proprio e un debugger, che si occupa di fornire le funzionalità di debugging e di analisi agli utenti. Questa divisione consente di adottare una gestione su due livelli delle exit. Tutte le exit, sia che siano condizionate o meno, vengono intercettate dal core di HYPERDBG. Ciò è possibile grazie al fatto che, come è stato spiegato nella Sezione 4.3.1, all'interno dell'area Host State del VMCS, viene impostato come *entry point* della modalità root, l'indirizzo di una funzione del core. Questa funzione, quando viene invocata a fronte di una exit, analizza la exit reason contenuta nel VMCS e, in base a questo campo, stabilisce qualora la exit in questione necessiti di una gestione particolare oppure se è possibile far riprendere immediatamente l'esecuzione del guest. Per esempio, è possibile che un'applicazione o un driver in esecuzione all'interno del guest, provino a eseguire delle particolari istruzioni di gestione della virtualizzazione hardware-assisted, come la

VMREAD o la VMWRITE [18]. Tali istruzioni non possono essere eseguite all'interno del guest, altrimenti potrebbe venir meno la proprietà di trasparenza dell'hypervisor, ma non richiedono nessuna particolare gestione. Infatti, il fatto stesso che venga causata una exit inibisce l'esecuzione dell'istruzione che l'ha scatenata e è sufficiente che l'hypervisor restituisca il controllo al guest facendolo ripartire dall'istruzione successiva. Alcune exit, al contrario, non possono essere liquidate così semplicemente ma richiedono una gestione più avanzata. Per esempio, l'istruzione MOV su un registro di controllo, che è stata richiamata in precedenza, scatena una exit che non può essere semplicemente ignorata. Come sarà illustrato in maggiore dettaglio poco più avanti, infatti, è necessario *emulare* l'istruzione, cioè replicarne gli effetti sul sistema. È chiaro come queste operazioni costituiscano un punto critico dell'hypervisor per quanto riguarda la trasparenza e che devono essere implementate con grande cura. Similarmente a quanto dimostrato in [35], infatti, un errore nel codice che si occupa dell'emulazione potrebbe facilmente fornire a un'applicazione eseguita nel sistema guest la possibilità di identificare la presenza dell'hypervisor. Come esempio, sia considerato nuovamente il caso dell'istruzione MOV che va a scrivere il contenuto del registro EAX in CR0. Qualora l'emulazione di questa operazione di scrittura sia implementata scorrettamente, potrebbero venire introdotte delle discrepanze tra il valore effettivo del registro CR0 dopo l'esecuzione dell'istruzione e il valore che il guest si aspetta di trovare in tale registro. Quindi, per un'applicazione interna al guest sarebbe sufficiente confrontare i due valori, reale e atteso, per accorgersi che l'istruzione non è stata effettivamente eseguita dal guest ma emulata incorrettamente dall'hypervisor.

Tutte le exit che richiedono una gestione speciale, inoltre, vengono notificate al debugger. Il debugger, per intraprendere delle azioni a fronte di una exit, deve semplicemente registrare un handler per ogni exit che intende intercettare. Quando il controllo viene trasferito alla modalità root da una exit, il core verifica se la exit reason fa parte di quelle che necessitano di una gestione particolare e, in caso positivo, notifica il debugger. La notifica avviene controllando se la exit appena avvenuta corrisponde a un evento che il debugger ha registrato e invocando il corrispondente handler, se presente. Tale handler, a sua volta, può decidere di gestire o ignorare l'evento. A seconda dei casi, quando il controllo ritorna al core di HYPERDBG, questo componen-

te decide se è necessario effettuare un'ulteriore gestione, come emulare l'istruzione, oppure se riprendere direttamente l'esecuzione del guest. Per esempio, il debugger potrebbe decidere di "nascondere" l'evento al guest oppure, al contrario, di non esserne interessato, delegandone quindi la gestione al core.

Vengono ora analizzate singolarmente le cause di una exit che necessitano di attenzioni particolari e/o vengono utilizzate dalla corrente implementazione del debugger.

Accesso ai registri di controllo. Come è già stato accennato in precedenza, qualsiasi tipo di accesso a uno dei registri di controllo CR0, CR3 e CR4 causa una exit, qualora siano stati impostati i relativi campi all'interno degli Execution Control Fields del VMCS. Dal momento che la gestione di queste exit è la stessa, indipendentemente dal registro di controllo acceduto, il core di HYPERDBG le gestisce con un'unica funzione. Prima di tutto, vengono recuperate dal campo Exit Qualification del VMCS le informazioni relative alla exit. Più precisamente, queste informazioni includono il tipo di accesso (lettura o scrittura), il registro di controllo acceduto e l'altro operando dell'istruzione (registro o locazione di memoria). A questo punto, il core notifica il debugger dell'avvenuto accesso a un registro di controllo, passandogli, inoltre, le informazioni recuperate dal VMCS. Se il debugger non gestisce tale exit, il core prosegue e, prima di restituire il controllo al guest, emula l'istruzione che ha causato la exit. L'emulazione viene effettuata replicando le modifiche sui registri e sulla memoria che l'istruzione avrebbe effettuato se non fosse stata bloccata dalla exit. Nell'implementazione attuale di HYPERDBG, questa exit non è intercettata dal debugger. Tuttavia, è comunque gestita nel core perché risulta essenziale per il meccanismo di protezione della memoria che sarà illustrato nella Sezione 4.3.3 e perché può essere rilevante per tipi di analisi diversi da quelli attualmente implementati. Per esempio, il registro CR3 cambia per ogni context switch. Qualora per l'analisi fosse rilevante identificare tali eventi, sarebbe sufficiente registrare un nuovo handler corrispondente a questa exit.

Accesso alle porte di I/O. La CPU mette a disposizione 2^{16} porte di I/O per le comunicazioni tra se stessa e le periferiche. Su un'architettura Intel, le due istruzioni utilizzate per effettuare operazioni su tali porte sono la IN, che legge da 1 a 4 byte da

una porta, e la `OUT` che scrive da 1 a 4 byte su una porta. Entrambe le istruzioni possono prendere o un registro o una locazione di memoria come secondo operando [18]. Come avviene per HYPERDBG, può essere importante intercettare queste operazioni dalla modalità `root`. Per garantire maggiore flessibilità e migliori prestazioni, non viene generata una `exit` per ogni esecuzione delle due istruzioni appena viste, con la conseguente necessità di una scrematura a livello di hypervisor. Invece, all'interno del VMCS è presente una bitmap che permette una selezione *a grana fine* di quali porte di I/O causano una `exit` quando un'istruzione prova a accedervi. In particolare, questa bitmap contiene un bit per ognuna delle 2^{16} porte e un'istruzione di I/O causa una `exit` se e solo se tenta di accedere a una porta il cui bit corrispondente è impostato nella bitmap [32].

Questa caratteristica risulta particolarmente utile nel caso di HYPERDBG. Il debugger, infatti, deve intercettare le operazioni effettuate sulla porta di I/O della tastiera PS/2 al fine di rilevare la pressione di un determinato tasto. Se non fosse possibile specificare così precisamente quali accessi causano una `exit`, sarebbe necessario controllare tutte le istruzioni di I/O per determinare quelle effettuate sulla porta relativa alla tastiera, con una conseguente introduzione di overhead inutile.

La gestione di questa `exit` differisce dalle altre, dal momento che HYPERDBG adotta una tecnica "atipica" per evitare di dover emulare tutte le `IN` e le `OUT` che non vengono gestite dal debugger. Infatti, dal momento che le diverse tipologie di queste istruzioni sono numerose, sarebbe oltremodo complicato emularle, correndo inoltre il rischio di introdurre errori. Prima di tutto, il core di HYPERDBG recupera le informazioni relative alla `exit` dal VMCS e pubblica l'evento. La funzione di gestione registrata dal debugger controlla che l'istruzione che ha causato la `exit` sia di suo interesse, cioè che corrisponda alla pressione di un tasto della tastiera. Infatti, i mouse PS/2 utilizzano la medesima porta di I/O della tastiera e quindi le `exit` generate da tali device non possono essere scremate tramite la bitmap. In caso la `exit` sia stata causata dalla tastiera, HYPERDBG emula la lettura dalla porta di I/O in questione e controlla che il tasto premuto sia quello richiesto dal debugger, come verrà spiegato nella Sezione 4.3.6. Nel caso in cui la prima condizione non si verifichi, il controllo viene restituito immediatamente al core. A questo punto, sarebbe necessario emulare l'istruzione ma, come è già

stato detto, questa operazione introdurrebbe troppa complessità nel codice del core. Di conseguenza, la soluzione adottata in HYPERDBG consiste nell'azzerare temporaneamente, nella I/O bitmap, il bit corrispondente alla porta monitorata, nell'abilitare la modalità *single-step* del processore e nel ripristinare il valore dell'istruzione pointer all'indirizzo dell'istruzione che ha causato la exit. In questo modo, quando viene ripresa l'esecuzione del guest, l'istruzione in questione verrà rieseguita senza causare la exit per l'accesso alla porta di I/O. Invece, verrà generata una exit dall'istruzione successiva, a causa dell'abilitazione della modalità *single-step*. Il core è in grado di riconoscere il fatto che il *single-step* è stato abilitato solo per eseguire l'istruzione che interagisce con la porta di I/O, quindi lo disabilita nuovamente, ripristina il bit corrispondente alla porta nella I/O bitmap e restituisce il controllo al guest. In questo modo, l'istruzione viene eseguita realmente dal guest, eliminando la necessità di emularla e evitando, di conseguenza, le complicazioni insite in questa tipologia di gestione.

Istruzione VMCALL. Questa istruzione permette a un programma in esecuzione all'interno del sistema guest, indipendentemente dal livello di privilegio in cui è eseguito, di invocare un servizio messo a disposizione dall'hypervisor. In realtà, l'invocazione di questa istruzione causa semplicemente una exit e la gestione dell'interfaccia tra il sistema guest e l'hypervisor è completamente rimandata all'implementazione del secondo componente. Per quanto possa sembrare strano, può spesso risultare utile avere a disposizione la possibilità di comunicare tra il sistema guest e l'hypervisor. Come esempio, si consideri la necessità di salvare su disco il contenuto di una particolare area di memoria durante una sessione di debugging. Per fornire tale funzionalità, è necessario che il debugger possa interagire con il file system del sistema guest. Questo componente è dipendente dal sistema operativo e sarebbe quindi necessario implementare tale funzionalità in un modo diverso per ogni sistema operativo supportato. L'idea per evitare questo problema consiste nell'eseguire all'interno del guest, dopo aver installato HYPERDBG, un piccolo programma in user-space che esegue una VMCALL per notificare al debugger la sua presenza. Il debugger, sempre relativamente all'esempio proposto in precedenza, comunica con il programma in user-space andando a scrivere la zona di memoria da salvare su disco nello spazio di indirizzamento di tale

programma, tramite le funzioni che verranno illustrate nella Sezione 4.3.3, delegando quindi al componente interno al guest il compito di scrivere il contenuto desiderato in un file. Il programma in user-space, dato che la comunicazione fornita dalla `VMCALL` è unidirezionale dal guest all'hypervisor, può fare polling su una variabile contenuta nel suo spazio di indirizzamento, in cui il debugger può scrivere, in modo da sapere quando è pronto il contenuto da salvare su disco. È evidente come tale funzionalità riduca drasticamente, se non del tutto, la proprietà di trasparenza, infatti è possibile, per un programma del guest, andare a identificare la presenza di `HYPERDBG` eseguendo un'appropriata `VMCALL`. Di conseguenza, in `HYPERDBG`, è possibile decidere se abilitare o meno questa funzionalità, a seconda che le analisi che si devono effettuare richiedano trasparenza. Per esempio, nel caso dell'analisi di malware, potrebbe risultare preferibile disabilitarla per eliminare la possibilità che il codice maligno identifichi la presenza di `HYPERDBG`.

Entrando maggiormente nei dettagli, la comunicazione tra guest e hypervisor tramite `VMCALL` in `HYPERDBG` è stata implementata con uno schema simile a quello utilizzato dalle system call, in cui l'hypervisor mette a disposizione un determinato insieme di chiamate, dette *hypercall*. Quando il core riceve una exit causata da un'istruzione `VMCALL`, considera il valore del registro `EAX` come l'identificatore della hypercall richiesta e notifica l'evento al debugger, eventualmente ignorando la exit qualora l'identificatore non corrisponda a nessuna delle hypercall disponibili. Inoltre, dal momento che sia il core che il debugger potrebbero necessitare di registrare diverse hypercall e al fine di non aumentarne eccessivamente il numero, è stato inserito un livello aggiuntivo per gestire le richieste tra il guest e il debugger. Di conseguenza, per fornire tutte le funzionalità di interazione, come quella presentata nell'esempio precedente, il debugger registra una *sola* hypercall e stabilisce quali operazioni effettuare a seconda del contenuto del registro `EBX`. Il passaggio di parametri alle hypercall, qualora ne richiedano, è stato implementato tramite l'uso dello stack del processo che effettua la chiamata. Tale processo deve solamente caricare sul proprio stack i parametri, secondo la convenzione `cdecl`, mentre `HYPERDBG` usa le funzioni di analisi e gestione della memoria, illustrate nella Sezione 4.3.3, per recuperarli.

Istruzione VMLAUNCH. Questa istruzione viene utilizzata dal driver minimale che si occupa di caricare l'hypervisor, come è stato spiegato in 4.3.1. Una volta terminata la fase di installazione, la VMLAUNCH, quando eseguita in modalità non-root, causa una exit. Per garantire la trasparenza, questa exit viene gestita in modo da emulare il comportamento di una VMLAUNCH eseguita su un sistema in cui non è stato installato un hypervisor [18]. In particolare, tale gestione consiste solamente nell'iniettare nel guest, tramite la successiva entry, un'eccezione *invalid opcode* (#UD).

Eccezioni. Similmente a quanto è stato spiegato riguardo agli accessi alle porte di I/O, la virtualizzazione hardware mette a disposizione una bitmap che permette di specificare precisamente quali eccezioni devono causare una exit [32]. Analogamente al caso precedente, quando si verifica un'eccezione, viene controllato il corrispondente bit all'interno della bitmap e, qualora sia necessario, viene generata una exit. Per garantire la protezione della memoria e per fornire alcune funzionalità di debugging, HYPERDBG abilita tramite la bitmap le exit corrispondenti alle eccezioni elencate in seguito:

debug exception (#DB): questa eccezione richiede una gestione leggermente intricata a causa del meccanismo utilizzato per gestire le istruzioni che effettuano operazioni sulle porte di I/O, spiegato in precedenza. Una #DB indica che una o più condizioni di debug si sono verificate nel sistema ma l'unica condizione interessante dal punto di vista di HYPERDBG corrisponde al caso in cui una #DB sia causata dal flag *single-step* della CPU. La complicazione nella gestione di questa eccezione deriva dal fatto che questa situazione può verificarsi in tre diversi scenari: (I) il meccanismo di gestione delle istruzioni di I/O sta eseguendo l'istruzione per evitare la necessità di emularla, (II) l'utente di HYPERDBG ha richiesto l'esecuzione di una singola istruzione tramite la GUI, o (III) un processo nel guest ha impostato il flag *single-step* della CPU. Come prima cosa, il core controlla qualora sia verificata la situazione (I) e, in caso positivo, reimposta il bit corrispondente alla porta della tastiera nella I/O bitmap. Dopodiché, notifica il debugger della exit. Il debugger, nel caso (II), permette all'utente di riprendere la sessione di debugging, altrimenti rimanda la gestione finale della exit al core.

Per concludere, il core, solo nella situazione (III), inietta l'eccezione tramite la entry mentre, nel caso (I) semplicemente azzerà il flag *single-step* e restituisce il controllo al guest senza iniettare l'eccezione, in modo che l'esecuzione del guest continui normalmente;

breakpoint exception (#BP): questa eccezione indica che è stata eseguita un'istruzione per generare breakpoint software (INT 3). Questa exit viene utilizzata unicamente dal componente di debug di HYPERDBG per dare la possibilità a un utente di impostare breakpoint software nel codice del sistema guest. Il debugger, quando riceve la notifica di questa exit, controlla che il breakpoint sia stato effettivamente impostato da lui e non da un processo interno al guest e, qualora risulti essere di sua competenza, apre la GUI e restituisce il controllo all'utente in modo che possa riprendere la sessione di debugging.

page fault exception (#PF): è necessario gestire questa eccezione per garantire la protezione della memoria dell'hypervisor e per fornire la possibilità di impostare watchpoint durante una sessione di debugging. Dal momento che è necessario introdurre il meccanismo di gestione della memoria utilizzato da HYPERDBG per illustrarne i dettagli, queste due funzionalità verranno presentate rispettivamente nei paragrafi 4.3.3 e 4.3.4.

4.3.3 Gestione della memoria

Come è stato brevemente accennato all'inizio di questa Sezione, la gestione della memoria è una parte fondamentale di HYPERDBG, per due motivi principali. Il primo motivo è la protezione da accessi indesiderati da parte del sistema guest a alcune zone di memoria. Come è stato spiegato nella Sezione 4.3.1, infatti, durante la fase di lancio dell'hypervisor, devono essere caricate in memoria alcune informazioni basilari per il funzionamento dell'hypervisor stesso, tra cui il VMCS e le sezioni contenenti i dati e il codice, per esempio. Per garantire la proprietà di trasparenza, l'hypervisor deve impedire al sistema guest di accedere sia in lettura che in scrittura a tali zone. Qualora fosse invece possibile accedervi, infatti, il sistema guest potrebbe corrompere l'hypervisor

modificando i dati contenuti all'interno del VMCS o delle sezioni sopracitate. Come è stato evidenziato nella Sezione 4.3.1, nell'area Host State del VMCS viene salvato lo stato dell'hypervisor a fronte di una entry e dalla stessa area viene recuperato lo stato durante una exit. È facile intuire quindi, che se il guest o un'applicazione in esecuzione al suo interno avessero possibilità di accedere in scrittura al VMCS, potrebbero andare a modificare il contenuto dell'area Host State e, di fatto, riuscire a controllare quello che sarà lo stato dell'hypervisor appena avverrà una exit. Al contrario, l'area Guest State e gli Execution Control Fields, anche se potessero essere modificati dal guest a causa della mancanza di adeguata protezione, non influenzerebbero il comportamento dell'hypervisor in quanto, come è stato detto in precedenza, questi campi vengono impostati direttamente dalla CPU nel passaggio dalla modalità non-root a root. Di conseguenza, ogni modifica potenzialmente effettuata dal guest in modalità non-root, verrebbe sovrascritta durante il passaggio. Oltre che dalla scrittura, è altrettanto importante che queste aree siano protette dalla lettura da parte del guest. Un'applicazione particolarmente evoluta potrebbe, infatti, ricercare all'interno della memoria tali zone per tentare di capire qualora HYPERDBG sia presente sul sistema. Per queste motivazioni, una volta abilitate le modalità VMX e caricato l'hypervisor, questo si occupa di proteggere adeguatamente le zone di memoria sensibili appena descritte, in modo da impedire eventuali accessi e corruzioni.

Memoria virtuale con paginazione. Prima di entrare nei dettagli di come viene gestita la memoria in HYPERDBG, è utile richiamare il meccanismo della memoria virtuale correntemente utilizzato dalla maggior parte dei sistemi operativi moderni. La memoria virtuale è un meccanismo che permette ai processi in esecuzione su un sistema di avere a disposizione uno spazio di indirizzamento di dimensione prefissata, indipendentemente dalle dimensioni della memoria fisica della macchina su cui è in esecuzione il sistema. Dal momento che la dimensione della memoria reale può essere minore di quella della memoria virtuale, non è possibile avere una corrispondenza 1 a 1 tra gli indirizzi delle due tipologie di memoria. Quando il *paging* è utilizzato, lo

spazio di indirizzamento¹ di un processo è suddiviso in blocchi di dimensione fissa (pagine) che possono essere associate a una pagina della memoria fisica e/o salvate temporaneamente sul disco fisso, qualora la memoria fisica non sia sufficiente a contenerli. Quando un processo prova a accedere a un indirizzo di memoria, la Memory Management Unit (MMU) si occupa di tradurre questo indirizzo nel corrispondente indirizzo fisico [19]. I dettagli del meccanismo di traduzione sono presentati nel seguito di questa Sezione.

Se la pagina contenente l'indirizzo fisico che corrisponde all'indirizzo virtuale tradotto non è correntemente nella memoria fisica, la MMU solleva un Page Fault, comunemente abbreviato come #PF. Come avviene per le altre eccezioni, il sistema operativo gestisce i #PF invocando l'appropriato *exception handler*. Il meccanismo di gestione delle eccezioni di un sistema operativo tipicamente si occupa, alla ricezione di un'eccezione, di interrompere il processo in esecuzione in quel momento e di trasferire il controllo all'handler appropriato, il cui indirizzo è contenuto nell'*Interrupt Descriptor Table* (IDT). L'handler dei #PF provvede a caricare in memoria la pagina contenente l'indirizzo che non è stato trovato, eventualmente spostando un'altra pagina sul disco per fare spazio a quella da caricare. Una volta terminata la procedura di caricamento della pagina, l'handler termina e viene ripristinato lo stato della CPU precedente all'interruzione, di fatto riprendendo l'esecuzione del processo esattamente dall'istruzione che aveva generato il #PF. In questo modo, l'istruzione verrà rieseguita e potrà andare a accedere alla pagina appena caricata senza generare ulteriori #PF.

Meccanismo di traduzione degli indirizzi lineari. Per comprendere come HYPERDBG gestisce la memoria virtuale, è necessario che sia chiaro come funziona il meccanismo di traduzione degli indirizzi lineari in indirizzi fisici, cioè come determinare se un indirizzo che fa riferimento alla memoria virtuale di un processo è effettivamente mappato in un indirizzo relativo alla memoria fisica e, inoltre, qual'è questo indirizzo fisico. Lo schema riportato in Figura 4.3 rappresenta i vari passaggi necessari per effettuare la traduzione di un indirizzo lineare nel corrispondente indirizzo fisico. Assumendo

¹Nel seguito di questa sezione, i termini “spazio di indirizzamento” e “memoria virtuale” vengono utilizzati alternativamente.

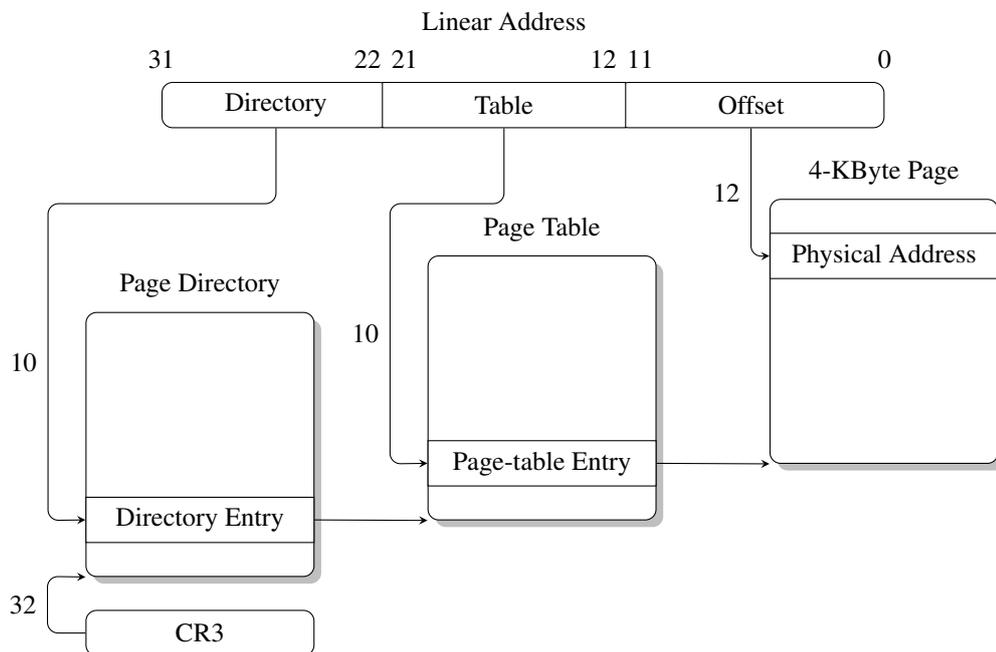


Figura 4.3: Traduzione indirizzo lineare con pagine di 4 KByte

una dimensione delle pagine pari a 4 KByte (2^{12} byte), tipicamente usata dai sistemi moderni, questo metodo può essere utilizzato per indirizzare fino a 2^{20} pagine, che corrispondono a uno spazio di indirizzamento virtuale di 2^{32} byte (4 GByte) [19]. La strategia utilizzata nei processori Intel è quella della tabella delle pagine a due livelli. La paginazione è gestita tramite tre strutture dati principali utilizzate per gestire la paginazione della memoria virtuale, rappresentate dai tre blocchi di dimensioni maggiori nella figura: la *page directory* (PD), le *page table* (PT) e le *page*. La page directory è suddivisa in 1024 *page directory entry* (PDE), ognuna delle quali fa riferimento a una page table. Le page table, a loro volta, sono suddivise in 1024 *page table entry* (PTE), ognuna delle quali fa riferimento a una pagina da 4 KByte. Uno schema riassuntivo della struttura delle PDE e delle PTE è rappresentato in Figura 4.4, in cui sono riportati solamente i bit significativi per la spiegazione della gestione della memoria di HYPERDBG.

L'ultimo elemento da considerare, prima di illustrare i vari passaggi del meccanismo di traduzione, è il registro CR3 della CPU. Questo registro di controllo contie-

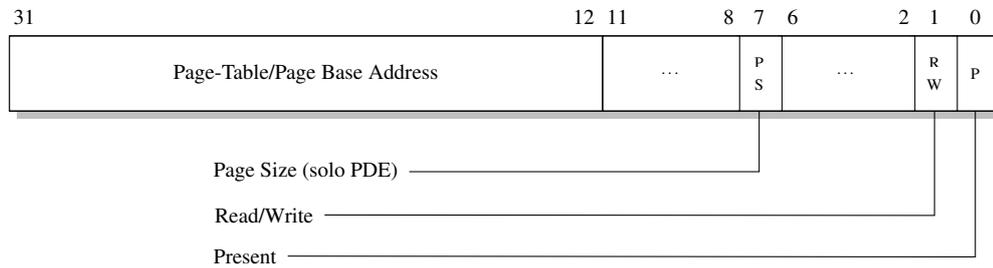


Figura 4.4: Struttura di una PDE/PTE

```

1   pde_phy_addr = (CR3 & 0xffff000) | \
2                   (((va & 0xffc00000) >> 22) * \
3                   sizeof(PDE));

```

Figura 4.5: Calcolo dell'indirizzo della PDE relativa a un indirizzo virtuale va .

ne l'indirizzo fisico a partire dal quale è mappata la PD del processo correntemente schedato. Quindi, il primo passaggio consiste nel recuperare la PDE relativa a un indirizzo virtuale che, per comodità, viene indicato come va .

Innanzitutto, viene calcolato l'indirizzo base della PD del processo corrente, prendendo i primi 20 bit di $CR3$, come si può vedere nella prima riga del codice riportato in Figura 4.5. Per calcolare l'offset della PDE all'interno della PD così ottenuta, si prendono i 10 bit più alti di va (31:22) e li si sposta di 22 posizioni verso destra. Infine, per ottenere l'allineamento corretto, si moltiplica l'offset ottenuto per la dimensione di una PDE, in questo caso 4 byte, come si può vedere nella seconda riga della Figura 4.5. Mettendo in OR i risultati così ottenuti, si ricava l'indirizzo pde_phy_addr .

```

1   pde = *pde_phy_addr;
2   pte_phy_addr = (pde & 0xffff000) | \
3                   (((va & 0x003ff000) >> 12) * \
4                   sizeof(PTE));

```

Figura 4.6: Calcolo dell'indirizzo della *page* relativa a un indirizzo virtuale va .

Tramite questo indirizzo, si va a leggere la PDE che, al suo interno, precisamente nei 20 bit superiori (bit 31:12 in Figura 4.4), contiene l'indirizzo base della PT relativa all'indirizzo `va`. Nella seconda riga del codice in Figura 4.6 è riportata l'operazione utilizzata per andare a prelevare solo i primi 20 bit della PDE. Per calcolare l'offset da aggiungere alla base della PD, si recuperano i 10 bit "intermedi" di `va` (21:12) e li si sposta di 12 posizioni verso destra. Come nel passaggio precedente, l'allineamento corretto viene ottenuto moltiplicando l'offset per la dimensione di una PTE che è, ancora una volta, di 4 byte. I due passaggi appena descritti corrispondono alle righe 3 e 4 del codice riportato in Figura 4.6. Per completare la traduzione, si prendono i 20

```

1     pte = *pte_phy_addr;
2     phy_addr = (pte & 0xffff000) | \
3                (((va & 0x0000fff) << 12));

```

Figura 4.7: Calcolo dell'indirizzo fisico corrispondente all'indirizzo virtuale `va`.

bit superiori (bit 31:12 in Figura 4.4) della PTE, che rappresentano l'indirizzo di base della pagina da 4 KByte. A questo indirizzo base, si aggiunge l'offset indicato dai 12 bit inferiori (11:0) di `va`. L'indirizzo `phy_addr` così ottenuto è l'indirizzo fisico corrispondente all'indirizzo virtuale di partenza `va`.

Durante il secondo e il terzo passaggio di traduzione, vengono effettuati i controlli sulla presenza della page table e/o della pagina cercata. Tali controlli non sono riportati nel codice nelle Figure 4.6 e 4.7 per semplificare la comprensione. Banalmente, dopo aver letto la PDE (Figura 4.6, riga 1) si controlla il bit *Present* di tale entry, che corrisponde al bit 0 riportato in Figura 4.4. Se questo bit è a 0, la page table a cui la PDE fa riferimento non è attualmente nella memoria fisica, quindi viene generato un `#PF`. Se, invece, il bit è a 1, la page table indicata dalla PDE è presente. Allo stesso modo, nel passaggio successivo, viene controllato il primo bit della PTE letta dalla memoria. Se il bit è a 0 la pagina a cui la page table fa riferimento non è presente in memoria e, come nel caso precedente, viene generato un `#PF`. Altrimenti, la traduzione va a buon fine.

Alcune architetture a memoria virtuale adottano un sistema “ibrido” che consiste nell’utilizzare pagine di 4 KByte e pagine di 4 MByte, in caso esistano pagine di memoria utilizzate molto frequentemente, come quelle in cui è contenuto il kernel di un sistema operativo, al fine di migliorare le performance di accesso a tali zone di memoria. In presenza di pagine di dimensione 4 MByte, il meccanismo di traduzione di un indirizzo virtuale v_a in indirizzo fisico è leggermente diverso. Dopo aver calcolato l’indirizzo della PDE relativo a v_a , si controlla se il flag Page Size (bit 7 in Figura 4.4) è impostato a 1 nella PDE. In caso positivo, i 10 superiori bit (31:22) della PDE indicano direttamente l’indirizzo base della pagina da 4 MByte e l’offset all’interno di questa pagina è calcolato tramite i 22 bit inferiori (21:0) di v_a .

Inoltre, è possibile che il sistema guest faccia utilizzo della *Physical Address Extension* (PAE), qualora questa sia messa a disposizione dall’hardware [19]. Questo meccanismo di paging alternativo consente di avere indirizzi fisici di 36 bit anche su un’architettura a 32 bit, aumentando di conseguenza la dimensione massima dello spazio di indirizzamento accessibile dal processore. Il meccanismo di traduzione della memoria risulta essere leggermente più complesso di quello utilizzato in assenza di questa estensione e introduce un leggero overhead causato dall’utilizzo di un livello di indizione in più rispetto al meccanismo di paging illustrato in precedenza. HYPERDBG è in grado di gestire anche sistemi che utilizzano PAE. Tuttavia, in questo lavoro di tesi non vengono illustrati i meccanismi di gestione della memoria in presenza di PAE abilitata, dal momento che le differenze con l’algoritmo di gestione della memoria presentato in questa Sezione sono puramente implementative, mentre la logica rimane sostanzialmente invariata.

Paging on-demand. Le informazioni riguardo allo spazio di indirizzamento di un processo, come illustrato precedentemente in questa Sezione, sono raggiungibili a partire dal registro della CPU CR3. Questo registro di controllo viene modificato durante i *context switch* in modo che ogni processo abbia il suo spazio di indirizzamento privato e che le traduzioni di un indirizzo virtuale in uno fisico, effettuate dalla MMU, facciano sempre riferimento alla PD e alle PT del processo corrente. Un *context switch* è un procedimento tramite cui viene ripristinato un particolare stato della CPU prece-

dentemente salvato in memoria. Per esempio, questo procedimento viene utilizzato dai sistemi operativi che permettono il multitasking, cioè permettono di eseguire più processi concorrentemente, per cambiare il processo correntemente in esecuzione con uno rimasto in attesa di essere schedulato. Inoltre, il meccanismo stesso di traduzione degli indirizzi impedisce a un processo di andare a accedere allo spazio di indirizzamento di altri processi, dal momento che nessun processo può “oltrepassare” la MMU per andare arbitrariamente a accedere a indirizzi fisici di memoria non mappati nel loro spazio di indirizzamento.

Questa protezione, per quanto essenziale per il corretto funzionamento di un sistema operativo che utilizza la memoria virtuale con paginazione, pone un problema significativo per HYPERDBG. Per fornire adeguate funzionalità di debugging dalla modalità root, infatti, HYPERDBG deve poter essere in grado di leggere dallo spazio di indirizzamento di un qualsiasi processo in esecuzione in modalità non-root. Cioè, deve poter essere in grado di “navigare” nella memoria virtuale di ogni processo, indipendentemente dal valore attuale del registro CR3. La soluzione più intuitiva per superare questa limitazione consiste nell’implementare via software il meccanismo di traduzione degli indirizzi della MMU, in modo da poterlo utilizzare partendo da un CR3 arbitrario a seconda del processo alla cui memoria HYPERDBG vuole accedere. Questa soluzione, però, introduce a sua volta un problema. Infatti, come abbiamo visto nella spiegazione del meccanismo di traduzione, il registro CR3 contiene un indirizzo fisico. Allo stesso modo, i 20 bit superiori della PDE recuperata partendo dall’indirizzo contenuto in CR3 rappresentano un indirizzo fisico. Come è già stato evidenziato, non è possibile accedere via software direttamente a un indirizzo fisico, neanche dalla modalità root. La soluzione proposta in HYPERDBG sfrutta il fatto che le strutture dati per la gestione della memoria virtuale di un processo (PD/PT) sono a loro volta mappate nello spazio di indirizzamento del processo stesso. In particolare, tali strutture sono contenute nella parte di spazio di indirizzamento riservato al kernel, che è sempre contenuta nella memoria di ogni processo. Per esempio, su un sistema Windows XP, la PD di un processo è mappata a partire dall’indirizzo virtuale 0xC0300000 mentre le PT sono mappate a partire da 0xC0000000 [37]. Queste aree di memoria vengono copiate, durante la fase di caricamento spiegata nella Sezione 4.3.1, nello spazio di

indirizzamento privato dell'hypervisor. Di conseguenza, è possibile analizzare tutte le PT dell'hypervisor, facendo riferimento solamente a indirizzi di memoria virtuali. La soluzione implementata in HYPERDBG sfrutta questa possibilità per ricercare una PT che contenga una entry libera e, una volta localizzata una entry disponibile, la modifica in modo che faccia riferimento alla pagina da 4 KByte che contiene l'indirizzo fisico da cui è necessario leggere e marca l'entry come presente. Tramite questo procedimento, si ottiene un indirizzo virtuale la cui traduzione in fisico porta esattamente alla pagina appena mappata. A questo punto, è possibile semplicemente usare questo indirizzo virtuale per accedere all'indirizzo fisico richiesto, dal momento che la MMU lo troverà mappato correttamente nello spazio di indirizzamento dell'hypervisor. Una volta terminate le operazioni che necessitano dell'indirizzo fisico, la pagina può essere eliminata, reimpostando il valore precedente nella PTE modificata.

Una volta risolto il problema di andare a leggere da un indirizzo fisico arbitrario, è possibile implementare correttamente via software il meccanismo di traduzione degli indirizzi utilizzato dalla MMU a partire da un valore del registro CR3 arbitrario. In particolare, i passi necessari per andare a leggere da un indirizzo virtuale contenuto nello spazio di indirizzamento appartenente a un processo diverso da quello correntemente schedato, sono i seguenti:

1. verificare che l'indirizzo virtuale v_a desiderato sia effettivamente presente nello spazio di indirizzamento del processo, andando a identificare la PTE corrispondente a v_a e controllando che il bit di presenza sia impostato;
2. in caso v_a sia presente, eseguire la traduzione da virtuale a fisico, per ottenere l'indirizzo fisico corrispondente a v_a ;
3. rendere accessibile nello spazio di indirizzamento dell'hypervisor, tramite il meccanismo spiegato in precedenza, l'indirizzo fisico ottenuto al passo 2;
4. leggere la quantità di memoria desiderata a partire dal nuovo indirizzo virtuale ottenuto al passo 3;
5. ripristinare la PTE modificata al passo 3, in modo da "pulire" lo spazio di indirizzamento dell'hypervisor.

Per concludere, vengono presentate alcune delle funzionalità principali di manipolazione e ispezione della memoria che sono state implementate nell'hypervisor a partire dai meccanismi spiegati precedentemente, dal momento che queste funzionalità sono essenziali per la maggior parte degli altri componenti di HYPERDBG:

MapPhysicalRegion: dato un indirizzo fisico `phy`, mappa la pagina contenente `phy` nella prima PTE inutilizzata tra le PT dell'hypervisor.

UnmapPhysicalRegion: dato un indirizzo virtuale `va` e una PTE `original`, ripristina il contenuto della PTE corrispondente a `va` con il contenuto di `original`.

ReadPhysicalRegion: dato un indirizzo fisico `phy`, un numero di byte `n` e un indirizzo di una zona di memoria di destinazione `dest`, mappa la pagina contenente `phy` all'indirizzo virtuale `va` tramite **MapPhysicalRegion**. Copia `n` byte da `va` a `dest` e elimina la pagina da cui ha letto dallo spazio di indirizzamento del processo corrente tramite **UnmapPhysicalRegion**. Qualora la zona di memoria da copiare sia più grande di una pagina, si occupa di ripetere le operazioni su tutte le pagine fisiche necessarie.

WritePhysicalRegion: dato un indirizzo fisico `phy`, un numero di byte `n` e un indirizzo di una zona di memoria sorgente `src`, mappa la pagina contenente `phy` all'indirizzo virtuale `va` tramite **MapPhysicalRegion**. Copia `n` byte da `src` a `va` e elimina la pagina da cui ha letto dallo spazio di indirizzamento del processo corrente tramite **UnmapPhysicalRegion**. Qualora la zona di memoria da copiare sia più grande di una pagina, si occupa di ripetere le operazioni su tutte le pagine fisiche necessarie.

GetPhysicalAddress: dato un indirizzo virtuale `va` e il CR3 del processo desiderato, restituisce l'indirizzo fisico corrispondente a `va`, relativamente allo spazio di indirizzamento identificato dal parametro CR3.

ReadVirtualRegion: dato un indirizzo virtuale `va`, il valore CR3 del processo desiderato, un numero di byte `n` e un indirizzo di una zona di memoria di destinazione `dest`, recupera l'indirizzo fisico `phy` corrispondente a `va` nello spazio

di indirizzamento indicato dal parametro `CR3` e utilizza la funzione **ReadPhysicalRegion** per copiare `n` byte da `phy` a `dest`.

WriteVirtualRegion: dato un indirizzo virtuale `va`, il valore `CR3` del processo desiderato, un numero di byte `n` e un indirizzo di una zona di memoria sorgente `src`, recupera l'indirizzo fisico `phy` corrispondente a `va` nello spazio di indirizzamento indicato dal parametro `CR3` e utilizza la funzione **WritePhysicalRegion** per copiare `n` byte da `src` a `phy`.

IsAddressValid: dato un indirizzo virtuale `va` e un valore `CR3` del processo desiderato, verifica che la pagina in cui `va` è mappato sia correntemente presente in memoria.

IsAddressWritable: dato un indirizzo virtuale `va` e un valore `CR3` del processo desiderato, verifica che la pagina in cui `va` è mappato sia correntemente presente in memoria e che sia scrivibile. Inoltre, effettua i dovuti controlli sul bit *Write Protect* del registro `CR0` [19].

Protezione della memoria. Come è stato anticipato nei capitoli precedenti, per garantire che l'hypervisor sia completamente isolato dal sistema guest, è necessario assicurarsi che il guest non possa accedere a nessuna delle regioni di memoria utilizzate dall'hypervisor. Tramite questa protezione, si può fare in modo che HYPERDBG sia completamente trasparente, invisibile al sistema guest e, contemporaneamente, assicurare che l'esecuzione del primo componente non possa essere in alcun modo alterata dal secondo. Tuttavia, com'è facilmente intuibile, per permettere analisi del sistema guest, è necessario che sia possibile accedervi e, potenzialmente, modificarne lo stato. Di conseguenza, l'introduzione del meccanismo di protezione, presentato nella Sezione corrente, non altera in alcun modo, né limita, le funzionalità di gestione della memoria illustrate finora.

Come è stato spiegato nella Sezione 4.3.1, durante la fase di caricamento, il VMCS viene configurato in modo che l'hypervisor abbia uno spazio di indirizzamento privato. Tuttavia, ciò non è sufficiente a garantire che nessun processo del guest possa accedere alle locazioni di memoria utilizzate dall'hypervisor. Infatti, un processo eseguito

in kernel mode nel guest potrebbe andare a leggere da qualsiasi indirizzo fisico implementando un meccanismo analogo a quello utilizzato da HYPERDBG. Assumendo che un processo interno al guest sia in grado di effettuare queste operazioni, è immediato notare come questo potrebbe andare a leggere l'intero spazio di indirizzamento privato dell'hypervisor. Per eliminare questa possibilità, è stato introdotto un ulteriore livello di protezione, basato sull'utilizzo di *shadow page table* [39]. Una shadow page table è una copia di una page table e viene utilizzata per creare una corrispondenza tra un indirizzo virtuale e uno fisico diversa rispetto a quella contenuta nella page table originale. In particolare, l'hypervisor contiene una shadow page table per ogni page table utilizzata dal sistema guest e lo "inganna" per fare in modo che utilizzi tali tabelle invece che le originali. L'hypervisor si occupa di mantenere le corrispondenze corrette tra shadow page table e page table originale, tranne per alcune entry particolari, in modo da evitare che un processo interno al guest possa accedere alla memoria dell'hypervisor. La Figura 4.8 rappresenta la memoria in presenza di shadow page table dove  indica una regione di memoria fisica non protetta dalla shadow page table mentre  denota una regione protetta.

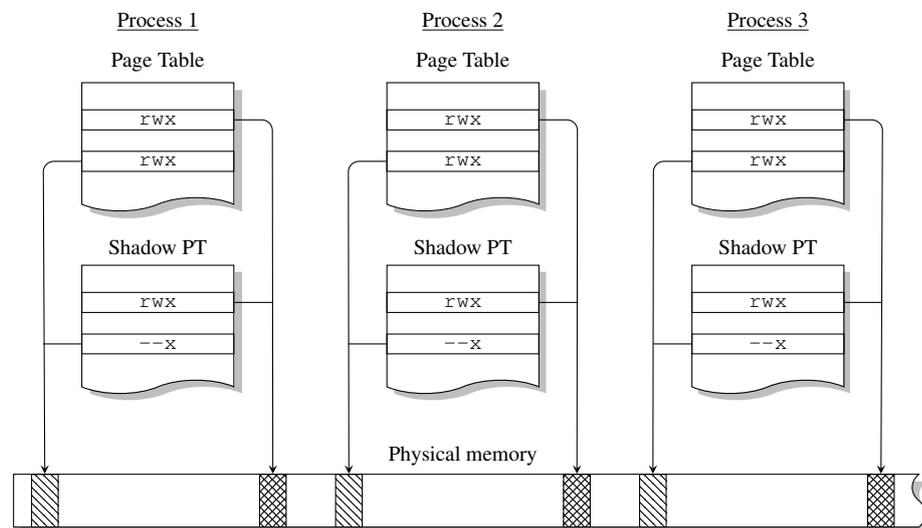


Figura 4.8: Memoria con Shadow Page Table

Il meccanismo di protezione si basa principalmente sull'intercettazione di due

eventi: le scritture sul registro CR3, che indicano che sta avvenendo un context switch, e i page fault. Il meccanismo di intercettazione di questi eventi è stato spiegato nella Sezione 4.3.2. È necessario intercettare il primo per sostituire la page table di ogni processo creato nel sistema guest con la shadow page table. Nella shadow page table installata, tutte le entry che mappano la page table del processo vengono modificate in modo da non essere né leggibili né scrivibili. In questo modo, ogni volta che il processo in questione cerca di accedere a una page table entry, per esempio quando verifica la validità di un indirizzo virtuale o durante un'allocazione di memoria, viene generato un page fault. A fronte di un #PF causato dalla protezione della page table, vengono adottate due gestioni diverse a seconda che l'accesso sia in scrittura o in lettura. Nel primo caso, l'hypervisor controlla *che cosa* il processo interno al guest sta scrivendo all'interno di una page table entry, cioè quale indirizzo fisico sta mappando nel suo spazio di indirizzamento. Qualora questo indirizzo fisico corrisponda a un'area protetta, l'hypervisor si occupa di modificarlo in modo che punti a una pagina di memoria che non contiene informazioni sensibili, prima di scriverlo nella entry della shadow page table. Per evitare che il guest si accorga della differenza tra il valore che voleva scrivere e quello che è stato effettivamente scritto, l'hypervisor salva il valore non modificato nella corrispondente entry della page table originale. In questo modo, ogni volta che viene rilevato un accesso in lettura a tale entry, l'hypervisor emula l'istruzione che l'ha effettuato in modo che il risultato rispecchi il valore contenuto nella page table originale e non quello contenuto nella shadow page table. Dal momento che il sistema guest usa la shadow page table, che è stata sostituita alla page table originale, non potrà mai accedere all'indirizzo fisico protetto e, dato che l'hypervisor maschera la differenza tra le due tabelle, non potrà neanche accorgersi di tale differenza. Nel caso in cui l'accesso in scrittura o in lettura non riguardi indirizzi sensibili, l'hypervisor utilizza un meccanismo simile a quello sfruttato per le operazioni di I/O per evitare di dover emulare tutti gli accessi. Invece, se il page fault non è stato generato dal meccanismo di protezione, viene lasciato passare al guest, in modo che questo possa gestirlo correttamente.

Tramite questo meccanismo di protezione, l'hypervisor è in grado di “mascherare” le zone di memoria sotto il suo controllo, facendo credere al sistema guest di potervi

accedere normalmente mentre in realtà viene forzato a accedere a pagine il cui contenuto può essere controllato dall'hypervisor. In questo modo, oltre a proteggere la memoria dell'hypervisor, si impedisce al guest di accorgersi della sua presenza.

4.3.4 Funzionalità di debug

HYPERDBG offre diverse funzionalità di debugging e di analisi del sistema guest. In questa Sezione vengono presentate le principali funzionalità offerte e vengono illustrati i dettagli di come sono state implementate.

Watchpoint. I watchpoint sono una delle funzionalità più complesse da implementare in modalità root. Come è stato illustrato nel Capitolo 2, le CPU moderne offrono un supporto hardware per realizzare watchpoint in maniera trasparente. Sfortunatamente, però, i watchpoint hardware sono limitati nel numero e condivisi tra le modalità root e non-root. Di conseguenza, non possono essere usati contemporaneamente dal sistema guest e dall'hypervisor, dal momento che impostare dei watchpoint hardware dalla modalità root equivarrebbe a sovrascrivere i watchpoint eventualmente impostati nel sistema guest, per esempio, da un programma di debugging. L'approccio utilizzato in HYPERDBG per implementare i watchpoint software è simile al meccanismo per la protezione della memoria dell'hypervisor illustrato nella Sezione 4.3.3. Questo approccio consiste nel proteggere dall'accesso le locazioni di memoria su cui si desidera impostare il watchpoint, in modo che qualsiasi tentativo da parte del sistema guest di accedere a tali locazioni generi un page fault. Più dettagliatamente, la pagina fisica che contiene l'indirizzo su cui viene impostato il watchpoint viene marcata come non presente in memoria e tutti i page fault che ne conseguono vengono intercettati dall'hypervisor. Il componente di debug analizza i page fault di cui viene notificato e controlla che l'indirizzo faulty, recuperabile dalle informazioni contenute nel VMCS, corrisponda a uno su cui l'utente ha impostato un watchpoint. In caso positivo, comunica all'utente che è stato raggiunto il watchpoint e visualizza l'interfaccia grafica, in modo che possa intraprendere le operazioni di analisi in risposta al watchpoint. Per garantire il corretto funzionamento del sistema guest, è necessario che l'istruzione

che ha generato il page fault sia eseguita correttamente. Qualora il watchpoint non sia permanente, si può semplicemente disabilitare la protezione della pagina e restituire il controllo al guest, facendogli eseguire l'istruzione in questione. Altrimenti, dal momento che la protezione deve essere permanente, HYPERDBG usa il medesimo meccanismo utilizzato per evitare di dover emulare le istruzioni di I/O, presentato nella Sezione 4.3.2. Analogamente, viene temporaneamente disabilitata la protezione sulla pagina contenente l'indirizzo su cui è stato impostato il watchpoint e viene eseguita l'istruzione che ha causato il page fault in modalità single step. Dopo l'esecuzione dell'istruzione, viene ripristinata la protezione della pagina e viene restituito il controllo al guest. È possibile impostare un numero arbitrario di watchpoint implementati con questo meccanismo, contrariamente a quanto accade per i watchpoint hardware.

È necessario sottolineare come, in assenza del meccanismo di protezione della memoria dell'hypervisor illustrato nella Sezione 4.3.3, un processo eseguito nel guest in modalità kernel potrebbe identificare e persino disabilitare un watchpoint impostato dall'hypervisor. Grazie al meccanismo di protezione, che maschera le differenze tra le shadow page table di un processo e le page table originali, questo non è possibile. Infatti, è sufficiente che la modifica al bit Present della PTE che mappa la pagina in cui è contenuto l'indirizzo su cui è stato impostato il watchpoint sia effettuata solamente nella shadow page table e non nella page table originale. In questo modo, qualora il guest provasse a leggere la PTE modificata, il meccanismo di protezione farebbe in modo che venga letto il valore contenuto nella page table originale, che non riflette la modifica, impedendogli quindi di identificare la presenza del watchpoint.

Breakpoint. Per motivi analoghi a quelli evidenziati per i watchpoint, non è possibile sfruttare i breakpoint hardware dall'hypervisor. Di conseguenza, sono possibili due diverse tipologie di breakpoint: una “classica”, di facile implementazione ma non trasparente, e una basata sui watchpoint. La prima è sostanzialmente equivalente a quella utilizzata dai debugger comuni e è basata sui *breakpoint software*. Richiamando brevemente quanto illustrato nel Capitolo 2, un breakpoint software è un'istruzione (INT 3) che solleva una breakpoint exception quando viene eseguita. La prima soluzione consiste, quindi, nel sostituire l'istruzione all'indirizzo su cui si vuole impostare

il breakpoint con l'istruzione appena menzionata. Quando l'esecuzione raggiunge il breakpoint, il debugger riceve una notifica dall'hypervisor e, come illustrato brevemente nella Sezione 4.3.2, verifica che il breakpoint sia stato effettivamente impostato dall'hypervisor. In caso positivo, notifica l'utente e ripristina l'istruzione originale. Se il breakpoint non è permanente, una volta intraprese le operazioni di debugging relative al breakpoint richiesto, il controllo viene restituito immediatamente al sistema guest. Altrimenti, l'istruzione su cui è stato impostato il breakpoint viene eseguita in modalità single step e, dopo l'esecuzione, viene sostituita nuovamente con l'istruzione `INT 3`, in modo che il breakpoint risulti permanente. Com'è facilmente intuibile, questa tipologia di breakpoint è tutt'altro che trasparente, dal momento che modifica direttamente il codice di un processo interno al guest e questo non dovrebbe far altro che andare a leggerlo per identificare la presenza di un breakpoint. Alcuni programmi, come Skype, per evitare di essere analizzati tramite un debugger, pre-calcolano dei checksum del proprio codice e, quando vengono eseguiti, calcolano nuovamente i checksum e controllano che corrispondano con quelli calcolati in precedenza [7]. Chiaramente, in presenza di breakpoint implementati come descritto finora, questi controlli di integrità falliscono, identificando quindi la presenza del debugger.

La soluzione alternativa, non affetta dal problema dei checksum, consiste nell'utilizzare i watchpoint per implementare i breakpoint. In questo secondo caso, è sufficiente impostare un watchpoint all'indirizzo desiderato dal momento che, quando il flusso d'esecuzione arriverà a quell'indirizzo, verrà sollevato un page fault a causa del tentativo di accesso in lettura alla pagina protetta. La gestione di tale page fault è del tutto analoga a quella discussa nel caso dei watchpoint. Questa tipologia di breakpoint, esattamente come i watchpoint, risulta totalmente trasparente al sistema guest che non può accorgersi in alcun modo della loro presenza, dal momento che non vengono effettuate modifiche esplicite al codice di un processo. La validità di questa tecnica per impostare breakpoint trasparenti, applicata all'analisi di codice maligno, è stata ampiamente discussa in [43, 44].

Single stepping. Finora abbiamo analizzato l'utilizzo della modalità single step della CPU solamente al fine di evitare di dover emulare dalla modalità root alcune istruzioni

particolari. Il single stepping risulta però essere molto utile ai fini del debugging, quindi HYPERDBG dà la possibilità di eseguire una singola istruzione per volta tramite la GUI. Quando il processore è in modalità single step, cioè quando l'apposito flag della CPU è uguale a 1, viene generata una debug exception. Questa debug exception, come è stato illustrato nella Sezione 4.3.2, viene notificata al debugger che, in caso il single step sia dovuto a una richiesta dell'utente e non a uno dei meccanismi di gestione analizzati in precedenza, aggiorna la GUI in modo che presenti le nuove informazioni riguardo al sistema e permette all'utente di proseguire la sessione di debugging. Qualora l'utente desideri eseguire un'altra istruzione in modalità single step, prima di restituire il controllo al guest, il debugger abilita i flag single step e resume, in modo che l'istruzione che ha già generato la debug exception non la sollevi nuovamente e possa essere eseguita realmente.

Disassembler. Al fine di facilitare l'attività di debugging, è possibile disassemblare porzioni del codice di un processo interno al guest. Inoltre, come sarà illustrato nella Sezione 4.3.6, la GUI include sempre una piccola porzione di codice disassemblato. Per fornire questa funzionalità, HYPERDBG si affida a un disassembler esterno open source [46], leggermente modificato per poter essere utilizzato in modalità root. Questo disassembler utilizza l'algoritmo Linear Sweep per tradurre il codice macchina in codice assembler, lo stesso utilizzato dal programma per linux *objdump* [15]. Per una discussione riguardo alle caratteristiche di questo algoritmo e a possibili alternative, si rimanda alla letteratura [38].

Stack backtrace. Grazie alle funzioni di gestione della memoria presentate nella Sezione 4.3.3, è possibile ispezionare i record di attivazione presenti sullo stack di un processo, in un qualsiasi momento della sua esecuzione. Percorrendo a ritroso i record di attivazione delle funzioni richiamate da un processo, HYPERDBG permette di visualizzare la sequenza di chiamate che ha portato il flusso di esecuzione al punto in cui è stato interrotto dal debugger. Questa tipologia di analisi risulta estremamente utile per tener traccia del flusso di esecuzione all'interno di un processo. In Figura 4.9 è riportato un semplice esempio di come possono essere analizzati i record di attivazione

presenti sullo stack utilizzando i frame pointer (EBP) salvati da ogni funzione. Inizialmente, si ottiene il puntatore al record di attivazione della funzione corrente tramite il registro EBP (riga 1). A partire da questo valore, è possibile recuperare dallo stack l'indirizzo di ritorno (riga 3) in modo da ottenere un riferimento alla funzione che ha chiamato quella corrente. Tramite questo valore si effettuano alcune analisi sulla funzione chiamante, per esempio si controlla se corrisponde a una funzione di libreria o qualora faccia parte di un driver del sistema operativo guest, tramite le analisi che saranno illustrate in seguito. Una volta terminate, si passa al record di attivazione precedente sullo stack aggiornando il valore del puntatore al frame corrente (riga 5). Effettuando iterativamente queste operazioni, è possibile andare a ispezionare tutti i record di attivazione presenti sullo stack. Si consideri, infine, che tutti i puntatori

```
1   current_frame = EBP;
2   while ( is_valid ( current_frame ) ) {
3       current_rip = *(current_frame +4);
4       // Effettua analisi: nome della funzione , ...
5       current_frame = *current_frame;
6   }
```

Figura 4.9: Algoritmo per effettuare stack backtrace.

utilizzati nello pseudocodice riportato in Figura 4.9 fanno riferimento allo spazio di indirizzamento di un processo in esecuzione all'interno del guest. Quindi, nonostante il codice riportato ci acceda direttamente per semplificare la lettura, HYPERDBG per accedervi utilizza le appropriate funzioni di accesso alla memoria del guest, descritte nella Sezione 4.3.3.

4.3.5 Analisi dipendenti dal sistema operativo

Tutte le funzionalità di HYPERDBG analizzate finora sono completamente indipendenti dal sistema operativo eseguito all'interno del guest. In molti casi, però, funzionalità dipendenti dal sistema operativo possono facilitare enormemente le analisi che si devono effettuare. Per esempio, i processi interni al guest possono essere identificati in

un modo indipendente dal sistema operativo solamente tramite l'indirizzo base delle loro page table, tipicamente contenuto nel registro CR3. Tuttavia, risulterebbe particolarmente scomodo fare riferimento a un processo solo tramite questo valore, mentre sarebbe molto più naturale utilizzare il suo *process id* (PID) o il nome del programma che esegue. Di conseguenza, HYPERDBG mette a disposizione alcune funzionalità dipendenti dal sistema operativo guest al fine di facilitare le attività di analisi e di debugging.

Queste funzionalità utilizzano tecniche di introspezione di macchine virtuali [14] per analizzare le strutture interne del sistema operativo guest al fine di tradurre le informazioni dipendenti dal sistema in un formato il più possibile user-friendly. Gli elementi estrapolati tramite le tecniche sopracitate e messe a disposizione delle attività di analisi e debugging sono:

Lista dei processi: HYPERDBG identifica la lista dei processi attualmente in esecuzione nel sistema guest, analizzandone le strutture, e estrae il PID, il nome e il valore del registro CR3 relativi al processo.

Lista dei moduli: HYPERDBG identifica la lista dei moduli e dei driver attualmente attivati nel sistema guest e ne estrae l'indirizzo base, l'indirizzo dell'entry point e il nome.

Simboli di debugging: HYPERDBG permette, utilizzando i simboli di debugging, di risolvere nomi e indirizzi, per esempio di funzioni e variabili locali. In questo modo, l'utente può richiedere l'impostazione di un breakpoint su di una funzione di libreria facendo riferimento direttamente al suo nome e non all'indirizzo e, viceversa, può controllare qualora un indirizzo specifico corrisponda a un simbolo.

4.3.6 HYPERDBG GUI

L'interfaccia grafica di HYPERDBG, permette a un utente di interagire e impartire comandi al debugger mentre si è in modalità root. Come è stato già brevemente accennato, dalla modalità root non è possibile interagire direttamente con i device driver del

```

+--[pid: 00000004; proc: System]-----[ HyperDbg ]-----
: EAX=00000058 EBX=00000001 ECX=00002ee0 EDX=00000060 ESP=005507c4 EBP=005507d8 EIP=806f58af
: ESI=00000000 EDI=805507ff CR0=e001003b CR3=00039000 CR4=000026d9 CS=0008 EFLAGS=00000246 ①
-----
hot-key pressed ②
executing command: disassemble 0x804df037
804df037: ff15c49b5580      call 0x80559bc4 <KeGdiFlushUserBatch> ④
804df03d: 58              pop %eax
804df03e: 5a              pop %edx
804df03f: ff0538f6dfff     inc 0xfffff638
804df045: 8bf2           mov %edx, %esi
804df047: 8b5f0c         mov 0xc(%edi), %ebx
804df04a: 33c9           xor %ecx, %ecx
804df04c: 8a0c18         mov (%eax,%ebx), %cl
804df04f: 8b3f           mov (%edi), %edi
804df051: 8b1c87         mov (%edi,%eax,4), %ebx
804df054: 2be1           sub %ecx, %esp
804df056: c1e902         shr $0x2, %ecx
804df059: 8bfc           mov %esp, %edi
804df05b: 3b3534f55580     cmp 0x0055f534, %esi
804df061: 0f93a9010000     jae 0x64x ③
804df067: f3a5           rep movsd
804df069: ff43           call %ebx
804df06b: 8be5           mov %ebp, %esp
804df06d: 8b0d24f1dfff     mov 0xfffff124, %ecx
804df073: 8b553c         mov 0x3c(%ebp), %edx
804df076: 899134010000     mov %edx, 0x134(%ecx)
804df07c: fa             cli
804df07d: f7457000000200   test $0x20000, 0x70(%ebp)
804df084: 7506           jnz 0x64x
804df086: f6456c01        testb $0x1, 0x6c(%ebp)
804df08a: 7458           jz 0x64x
804df08c: 8b1d24f1dfff     mov 0xfffff124, %ebx
804df092: c6432e00        movb $0x0, 0x2e(%ebx)
804df096: 807b4a00        cmpl $0x0, 0x4a(%ebx)
804df09a: 7448           .iz 0x64x
end of command: disassemble 0x804df037

executing command: backtrace 5
[current] 806f58af
[00] 005507d8 f85d14dc [i8042prt.sys] ⑤
[01] 0055081c 804dad9f <KiInterruptDispatch00+61>
[02] 00550840 f85f3062 [intelppm.sys]
[03] 005508d0 804dc0d7 <KiSwapProcess00+121>
[04] 0ffdf980 fdfdf980
end of command: backtrace 5

> ⑥

```

Figura 4.10: Interfaccia grafica di HYPERDBG

sistema guest, dal momento che si andrebbe a interferire con il loro funzionamento. È stato quindi necessario reimplementare due driver minimali, uno per la scheda video e uno per la tastiera, per fare in modo che HYPERDBG potesse interagire con tali device. I dettagli di come sono stati implementati questi due driver verranno presentati nel seguito della Sezione, mentre ora vengono illustrati i componenti principali dell'interfaccia grafica che viene utilizzata da un utente di HYPERDBG durante una sessione di debugging, facendo riferimento alla Figura 4.10, che riporta uno screenshot preso durante l'esecuzione di HYPERDBG all'interno del nostro ambiente di sviluppo.

Nella Figura, la zona identificata dal numero 1 riporta informazioni generali riguardo allo stato del guest, come il nome processo corrente, il valore dei registri general purpose e degli altri registri della CPU. L'area appena sottostante, delimitata dalle ri-

ghe tratteggiate orizzontali, viene utilizzata per visualizzare il risultato dei comandi impartiti dall'utente. HYPERDBG include la possibilità di suddividere l'output di un comando su più pagine navigabili, in modo da non limitare tale output alle dimensioni dell'area in questione. Tra le informazioni racchiuse in questa zona si possono identificare alcuni elementi particolari. Per esempio, la ragione che ha causato il passaggio da modalità non-root a root è indicata nella zona numero 2. In questo caso, la ragione del passaggio è la pressione dell'hotkey (F12) che permette di accedere alla GUI di HYPERDBG dalla modalità non-root senza la necessità che si verifichino eventi particolari. Invece, l'area numero 3 riporta il risultato di un'operazione di disassembly di alcune istruzioni a partire dall'indirizzo `0x804df037`. Come si può notare nel punto 4, HYPERDBG, quando possibile, evidenzia automaticamente se un indirizzo presente nel codice assembler può essere risolto in un simbolo, come è stato spiegato nella trattazione delle funzionalità dipendenti dal sistema guest presentate nella Sezione 4.3.4. Nella zona indicata dal numero 5, invece, si può osservare l'output riportato dal comando per visualizzare lo stack backtrace (Sezione 4.3.4) del processo corrente. Sempre tramite la risoluzione dei simboli, sono state identificate alcune delle funzioni presenti nella lista. Inoltre, si può anche notare come, in due casi particolari, è stato possibile usufruire di un'altra caratteristica dipendente dal sistema guest per andare a identificare i nomi dei moduli kernel di cui fanno parte gli indirizzi riportati nel backtrace. Dalla figura si può anche notare come il sistema guest sia stato bloccato durante l'esecuzione del driver della tastiera (`i8042prt.sys`). Questo dimostra che, grazie alla completa trasparenza al sistema guest di HYPERDBG, è possibile analizzare il driver della tastiera del sistema guest anche mentre l'utente sta utilizzando la tastiera stessa per impartire comandi al debugger. Infine, l'area indicata dal numero 6 e separata da una riga tratteggiata dalla zona di output, è utilizzata per visualizzare l'input inserito dall'utente.

Tastiera. Per permettere un'interazione trasparente con la tastiera, senza interferire con l'utilizzo di tale periferica da parte del sistema guest, è stato implementato un driver minimale. Per intercettare i tasti premuti mentre l'esecuzione è in modalità non-root, HYPERDBG sfrutta il metodo descritto nella Sezione 4.3.2 per la gestione

degli accessi alle porte di I/O mentre, per permettere all'utente di inserire comandi in modalità root, viene utilizzato il driver sopracitato. Il driver adotta un meccanismo di polling: legge ciclicamente dalle porte di I/O della tastiera per sapere se ci sono dati pronti a essere ricevuti o se si sono verificati errori. Precedentemente, si è notato come i mouse usino le stesse porte di I/O della tastiera. Un bit particolare dei valori letti indica qualora i dati provengano dalla tastiera o dal mouse, permettendo quindi al driver di ignorare i dati provenienti dal secondo device. Quando il driver rileva che ci sono dati pronti, li legge e li interpreta. È necessario interpretare i dati letti, dal momento che possono rappresentare diversi tipi di eventi, tra cui la pressione e il rilascio dei tasti e la segnalazione di eventuali errori. Qualora l'evento sia di uno questi ultimi due tipi citati, i dati vengono tradotti nel carattere corrispondente al tasto premuto oppure, nel caso di tasti speciali come *shift* o *ctrl*, viene registrato il cambiamento di stato del tasto. Questa gestione di basso livello viene completamente nascosta al debugger dal driver. Il primo componente, infatti, interagisce con la tastiera solamente tramite le funzioni di alto livello messe a disposizione dal driver. In questo modo, tutti i dettagli, per esempio la gestione degli errori di lettura dalla tastiera, vengono nascosti al debugger e gestiti unicamente dal driver. Per leggere l'input dell'utente, quindi, HYPERDBG applica ancora una volta un approccio di tipo polling, chiedendo continuamente al driver se sono disponibili caratteri letti dalla tastiera. I caratteri ricevuti vengono conservati in un buffer fino a che non viene rilevata la pressione del tasto invio. A questo punto, HYPERDBG controlla se l'input corrisponde a uno dei comandi messi a disposizione e eventualmente intraprende le azioni richieste.

Video. Il driver necessario per interagire con la scheda video è più complicato di quello utilizzato per interagire con la tastiera. Infatti, mentre le tastiere PS/2 adottano tutte lo stesso standard di comunicazione con il sistema, le schede video necessitano di una gestione molto più complicata. Tali device hardware possono fare riferimento a diversi standard (VGA, SVGA, ...), i quali a loro volta mettono a disposizione diverse modalità di utilizzo (text mode, X mode) [34]. Al fine di essere compatibile con il maggior numero di device hardware possibile, il driver implementato in HYPERDBG assume che la scheda stia operando in una modalità VGA standard. Questa modalità

è supportata dalla maggior parte delle schede video recenti, per esempio è la modalità utilizzata da Windows XP quando viene avviato in modalità provvisoria. In questa modalità standard, il framebuffer del device è mappato dal device stesso in una sezione lineare della memoria fisica che viene a sua volta mappata nella memoria virtuale dal driver del sistema guest. Per evitare confusione, si fa notare che il framebuffer sopracitato è il componente hardware di una scheda video e non l'omonima astrazione software usata per ragioni di compatibilità da alcuni sistemi operativi [13].

Il driver video di HYPERDBG si occupa, prima di tutto, di identificare la scheda video presente sul sistema. Per farlo, esegue una scansione di tutte le periferiche PCI installate sulla macchina. Per ogni device rilevato, analizza le informazioni contenute nei suoi registri PCI, in modo da identificare la scheda video. Tra le informazioni contenute nei registri PCI della scheda video, si trova un riferimento a una locazione di memoria fisica. Tale riferimento indica l'indirizzo a partire dal quale viene mappato il framebuffer del device. Per poter andare a scrivere su tale locazione di memoria, il driver di HYPERDBG la rende accessibile nello spazio di indirizzamento dell'hypervisor. La scheda video si occupa poi di riflettere le scritture effettuate su tale zona di memoria sullo schermo. In particolare, le dimensioni di questa zona dipendono dalla risoluzione dello schermo e il suo contenuto viene interpretato come il colore dei pixel dello schermo. Il numero di byte da interpretare come un singolo pixel dipende dalle impostazioni del driver e della scheda video. Il driver di HYPERDBG utilizza una modalità di colori (profondità) a 32 bit in cui, quindi, 4 byte della zona di memoria in cui è mappato il framebuffer del device corrispondono a un pixel; questa, infatti, è la modalità più comune nei sistemi moderni. Il driver si occupa di scrivere i valori appropriati all'interno di questa zona di memoria a seconda delle richieste ricevute dall'hypervisor. Per permettere la scrittura di un carattere su schermo, il driver contiene una bitmap che indica quali pixel devono essere colorati e quali no per ogni carattere. Inoltre, il driver permette di salvare il contenuto del framebuffer ogni volta che è necessario mostrare l'interfaccia grafica di HYPERDBG in modo che, quando viene restituito il controllo al sistema guest, venga ripristinato il contenuto originale.

Per facilitare l'interazione tra il componente di debug e la scheda video e per ridurre al minimo i dettagli di basso livello da gestire, il driver mette a disposizione una

funzione che permette di scrivere un carattere sullo schermo. Tale funzione prende come parametri: un carattere c , un intero C , che indica il colore desiderato, e una coppia di coordinate (x,y) che indica la posizione sullo schermo in cui si vuole disegnare il carattere. Tale funzione esegue i calcoli necessari a identificare l'offset all'interno della zona di memoria in cui è mappato il framebuffer corrispondente a (x,y) , in base alla risoluzione dello schermo e al numero di byte utilizzati per rappresentare un colore. I caratteri utilizzati da HYPERDBG sono rappresentati da un rettangolo di 8×12 pixel e la bitmap citata in precedenza indica quali di questi pixel devono essere visibili per disegnare correttamente il carattere. Partendo dall'offset iniziale, la funzione ricalcola l'offset di ogni singolo pixel del carattere c e, qualora la bitmap indichi che va disegnato, scrive il valore C in memoria, all'offset appropriato. Tramite questa funzione e tramite altre funzioni di più alto livello che si basano su questa, viene disegnata la GUI riportata in Figura 4.10. Come si può osservare, la GUI così ottenuta è graficamente molto primitiva. Questa scelta è stata effettuata per semplificare il più possibile il driver ma, in caso di necessità, è possibile aggiungere altre funzioni per disegnare componenti grafici più complessi.

Purtroppo, il driver video di HYPERDBG non è esente da problemi che derivano dal fatto che, quando l'hypervisor viene installato sulla macchina, il sistema guest ha già iniziato a interagire con la scheda video tramite il suo driver. Tali problemi verranno analizzati in dettaglio nella Sezione 5.3.2.

Capitolo 5

Implementazione e limitazioni

In questo Capitolo vengono discussi alcuni dettagli relativi all'implementazione di HYPERDBG. Inoltre, vengono presentate le limitazioni dell'implementazione corrente, potenziali soluzioni per eliminare tali limitazioni e, infine, alcune idee per i futuri sviluppi del lavoro.

5.1 Dettagli d'implementazione

L'infrastruttura di analisi e debugging basata sulla virtualizzazione hardware-assisted presentata in questo lavoro di tesi è stata implementata in un prototipo sperimentale. Allo stato attuale, il prototipo è stato sviluppato per piattaforme Windows XP. Come è stato evidenziato più volte in questo lavoro di tesi, il nucleo principale di HYPERDBG è platform-independent. È necessario però disporre di un driver opportuno che effettua il caricamento sul sistema da analizzare. Le altre parti platform-dependent riguardano le funzionalità di debugging di alto livello, elencate nella Sezione 4.3.5, che sfruttano la possibilità di investigare le strutture interne del kernel da analizzare e, dipendendo strettamente da come sono formate queste strutture, devono essere adattate a ogni kernel che si vuole analizzare. Queste componenti, però, sono completamente opzionali e non pregiudicano il funzionamento complessivo del debugger su sistemi per cui non sono state implementate.

Il codice è suddiviso in due componenti principali: l'hypervisor e il motore di debugging. Il primo componente contiene il codice necessario a abilitare le modalità VMX, a gestire le transizioni da modalità non-root a modalità root, la virtualizzazione della memoria e le funzionalità di introspezione. Il secondo, invece, contiene tutto il codice necessario a fornire all'utente le funzionalità di debug presentate in precedenza in questo lavoro di tesi: i driver della scheda video e della tastiera, l'interfaccia utente, la ricerca di simboli e l'implementazione delle funzioni di debugging, come la lettura dalla memoria, i breakpoint e la modalità single-step. Il primo componente è costituito da circa 4200 righe di codice C e da approssimativamente 300 righe di codice assembly. Il secondo conta circa 2400 righe di codice C. Da questi conteggi è esclusa la libreria di disassembling, che è stata semplicemente adattata a partire da una libreria preesistente.

Il codice di HYPERDBG è stato rilasciato sotto licenza GPL (v3.0) e è disponibile al seguente indirizzo:

<http://code.google.com/p/hyperdbg/>

5.2 Implementazione delle tecniche di introspezione

HYPERDBG include numerose tecniche di introspezione, in grado di recuperare diverse tipologie di informazioni circa il sistema analizzato. Nella presente Sezione vengono discussi i dettagli sulle funzionalità per l'analisi dei processi utente. Le altre funzionalità di introspezione, invece, presentate brevemente nella Sezione 4.3.4, sono concettualmente molto simili a quella che verrà presentata ora e, di conseguenza, non verranno illustrate in maggiori dettagli nel presente lavoro di tesi. Come è già stato brevemente delineato nella Sezione 4.3.4, le tecniche di introspezione sono fortemente dipendenti dal sistema guest e, di conseguenza, le procedure necessarie per recuperarle devono essere implementate specificamente per ogni sistema che si intende analizzare. La tecnica utilizzata da HYPERDBG per analizzare i processi utente in esecuzione all'interno del sistema guest è specifica di un sistema Windows XP, dal momento che, allo stato attuale dell'implementazione, il debugger supporta solamente questo sistema operativo.

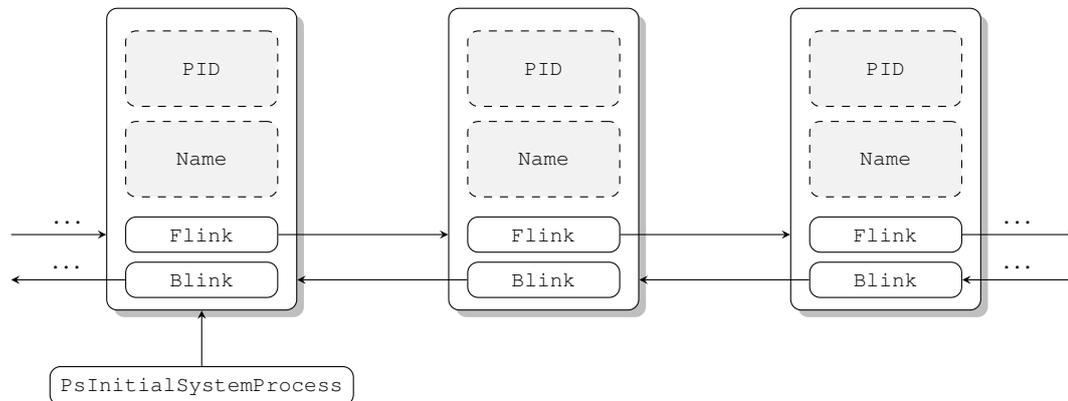


Figura 5.1: Lista di EPROCESS

Il sistema operativo Windows XP utilizza una struttura, chiamata EPROCESS, per mantenere informazioni riguardo ai processi correntemente in esecuzione al suo interno. Come si può vedere in Figura 5.1, l'insieme di queste strutture è organizzato come una lista dinamica doppiamente collegata: ogni struttura EPROCESS contiene un puntatore all'elemento successivo e all'elemento precedente all'interno della lista. Inoltre, Windows mantiene un puntatore all'entry corrispondente al processo `System`. Nella Figura sono stati riportati solamente Process ID e nome, ma la struttura EPROCESS contiene numerose altre informazioni riguardo a un processo, tra cui, per esempio, il puntatore alla lista di thread associati a esso. La procedura che elenca i processi o che ne cerca uno dato il suo PID o il suo nome è relativamente semplice. Partendo dal puntatore al processo `System`, si scorrono tutte le strutture EPROCESS utilizzando i puntatori `Flink` finché non si ritorna al punto di partenza. Durante l'attraversamento della lista, vengono recuperate le informazioni necessarie per l'analisi. Si fa notare come questa lista sia mantenuta da Windows solo per elencare i processi correntemente in esecuzione, mentre lo scheduling vero e proprio di processi e thread si basa su strutture alternative. Alcuni malware particolarmente evoluti, se eseguiti a un sufficiente livello di privilegio, sfruttano questo fatto per nascondere la loro presenza sul sistema andando a corrompere i valori dei puntatori `Flink` e `Blink` in modo da "scollegare" la struttura corrispondente al processo che vogliono nascondere dalla lista. Questo non impedisce al processo nascosto di essere schedulato correttamente e, allo stesso tem-

po, lo rende invisibile a strumenti che creano un elenco dei processi basandosi sulla lista di strutture EPROCESS. Si fa notare che, allo stato attuale dell'implementazione, anche HYPERDBG risulta essere influenzato da questa tecnica di autodifesa dei malware. Come sviluppo futuro si pianifica di affiancare, alla tecnica appena presentata, un'ulteriore procedura di analisi che elenchi i processi in esecuzione nel sistema guest basandosi su strutture che non possono venir corrotte da malware eseguiti con privilegi di livello kernel senza alterare il corretto funzionamento del processo che intendono nascondere. La possibilità di utilizzare entrambe le procedure di elenco dei processi, e di confrontarne i risultati, permette all'utente che effettua il debugging sia di fare affidamento su di una tecnica robusta (la seconda) che di effettuare *lie-detection* sul sistema. Qualora i risultati delle due tecniche non corrispondano, infatti, l'utente ha una prova effettiva che il sistema che sta analizzando è stato corrotto da un malware, o da un programma che si comporta come tale.

5.3 Limitazioni

In questa Sezione vengono presentate alcune limitazioni dello stato corrente dell'implementazione di HYPERDBG o, più in generale, dell'approccio di analisi e debugging illustrato in questo lavoro di tesi. Inoltre, vengono proposte delle potenziali soluzioni ai problemi evidenziati, le quali costituiscono anche parte degli sviluppi futuri di HYPERDBG.

5.3.1 Shadow page table e watchpoint

Al momento della stesura di questo lavoro di tesi, il codice che si occupa di effettuare la protezione della memoria non è stato ancora incluso nella versione *stabile* di HYPERDBG, cioè la versione rilasciata pubblicamente come indicato nella Sezione 5.1. Di conseguenza, nemmeno la funzionalità che permette di utilizzare i watchpoint, presentati nella Sezione 4.3.4, è attualmente disponibile, dal momento che è strettamente correlata al meccanismo di protezione. Questa limitazione è dovuta semplicemente a una misura cautelare: dal momento che si tratta di un'interazione con componenti

estremamente delicati del sistema guest (le tabelle delle pagine), un errore nel meccanismo di protezione potrebbe portare a esiti pericolosi per il sistema guest. Di conseguenza, tale porzione di codice è ancora in una fase di testing e verifica estensiva, al fine di rilevare e correggere eventuali difetti prima che venga rilasciata ufficialmente nel codice di HYPERDBG.

5.3.2 Driver video

Come è stato brevemente accennato nella Sezione 4.3.6, esistono diverse modalità in cui una scheda video può operare. Nel caso della modalità classica VGA non accelerata, il framebuffer [13] della scheda video è mappato, dal device stesso, in una zona lineare della memoria fisica della macchina. Per poter interagire con la scheda video è sufficiente, quindi, che il driver associ tale locazione di memoria fisica a una locazione nella memoria virtuale, al fine di poterla leggere e scrivere a piacimento. Per calcolare le corrispondenze tra il framebuffer e i pixel sullo schermo, come è stato spiegato in 4.3.6, è sufficiente eseguire semplici calcoli matematici basati sulla risoluzione dello schermo.

Alcuni acceleratori grafici moderni, però, al fine di ottenere elevate prestazioni, impostano la scheda video in modalità in cui non è necessario che il framebuffer sia completamente mappato in una zona lineare della memoria fisica. Tali driver usano i comandi messi a disposizione dal device hardware per effettuare il rendering vero e proprio sullo schermo, sfruttando solamente il coprocessore grafico della scheda video e evitando, in questo modo, di sovraccaricare la CPU del sistema. Per esempio, sia *addr* l'indirizzo di una porzione inutilizzata del framebuffer e (x,y) una coppia di coordinate a cui il driver accelerato vuole disegnare la finestra di un programma. Allora, invece di calcolare l'indirizzo all'interno del framebuffer basandosi su (x,y) e sulla risoluzione dello schermo, il driver accelerato semplicemente va a scrivere i valori corrispondenti alla finestra e al suo contenuto all'indirizzo *addr* e comunica al device hardware che deve disegnare sullo schermo, alla posizione (x,y) , i pixel contenuti alla locazione di memoria *addr*. Le operazioni di spostamento di una finestra sullo schermo, date per scontate da qualsiasi sistema la cui interfaccia grafica è basata

sulle finestre, costituiscono un esempio evidente di come la modalità grafica accelerata riduca incredibilmente il carico della CPU. In una modalità non accelerata, infatti, il driver video dovrebbe continuamente rilocalizzare, all'interno della zona di memoria virtuale corrispondente al framebuffer, i valori corrispondenti alla finestra e al suo contenuto, in modo da riflettere la nuova posizione sullo schermo desiderata dall'utente. Per quanto si possano adottare diversi meccanismi di ottimizzazione per diminuire gli accessi in memoria, la possibilità di impartire semplicemente un comando alla scheda video, delegando così il rendering effettivo sullo schermo al coprocessore grafico, risulta estremamente più efficiente che spostare continuamente i valori in memoria.

Come è evidente, però, il fatto che il framebuffer non sia mappato linearmente nella memoria fisica contrasta con l'assunzione effettuata dal driver video di HYPERDBG che, di conseguenza, non può funzionare correttamente quando sul sistema guest è in funzione un driver che utilizza la scheda grafica in modalità accelerata. Per ovviare al problema, è sufficiente utilizzare un driver non accelerato. Tipicamente, in sistemi operativi come Windows e Linux, è incluso un driver che opera in modalità VGA standard che viene utilizzato di default e che non causa problemi al driver video di HYPERDBG. Questa soluzione non è ottima e, di conseguenza, durante lo sviluppo sono state esplorate diverse possibili soluzioni alternative per questo problema. Sfortunatamente, tutte le soluzioni investigate dipendono in qualche misura dal sistema operativo guest o dall'hardware. La prima soluzione possibile consiste nell'interagire con il driver DirectX presente sui sistemi Windows. DirectX è una collezione di API che permette, tra le altre cose, di interagire direttamente con la scheda video, indipendentemente dall'hardware sottostante [9]. Questa soluzione, però, aggiungerebbe una dipendenza dal sistema operativo guest che non è desiderabile, dal momento che uno degli obiettivi principali di HYPERDBG è quello di essere il più possibile indipendente dal sistema guest. Una seconda soluzione consiste nell'implementare driver specifici per ogni device video che si intende supportare anche in modalità accelerata. Per farlo, tuttavia, sarebbe necessario conoscere le API messe a disposizione dai device, le quali sono, nella maggior parte dei casi, proprietarie e non rilasciate pubblicamente. Di conseguenza, l'implementazione attuale non include nessuna delle due soluzioni sopracitate, preferendo delegare all'utente il compito di disabilitare il driver accelerato

piuttosto che diminuire radicalmente la portabilità del debugger.

Questa limitazione comporta, però, l'impossibilità di utilizzare HYPERDBG per effettuare il debugging di driver video accelerati. Qualora il componente da analizzare tramite HYPERDBG sia proprio di questo tipo, è possibile adottare una terza soluzione che consiste nel disabilitare l'interfaccia grafica del debugger e utilizzare una porta seriale per comunicare l'output all'utente. Questa soluzione di ripiego causa la perdita del vantaggio rispetto a altri debugger kernel, come WinDBG [30], di poter osservare i risultati direttamente sulla macchina analizzata, però permette di sfruttare ancora gli altri punti di forza di HYPERDBG, come la completa dinamicità e la trasparenza al sistema guest.

5.3.3 Driver della tastiera

Il driver della tastiera incluso in HYPERDBG non soffre di alcuna limitazione di sorta nella gestione delle tastiere PS/2. Non è invece in grado di gestire tastiere USB. Questa limitazione è dovuta al fatto che la complessità dell'interazione con il controller USB della macchina è notevolmente maggiore rispetto a quella con una periferica PS/2. Inoltre, esistono diverse evoluzioni dei controller USB, spesso utilizzate contemporaneamente, che devono essere gestite singolarmente a causa delle differenze nel loro funzionamento. A aggravare ancora di più la situazione, deve essere considerato anche il fatto che i personal computer moderni vengono prodotti sempre più spesso senza periferiche PS/2, rendendo possibile unicamente l'utilizzo di periferiche USB. Ciò pone un problema non indifferente per l'usabilità di HYPERDBG e, di conseguenza, stanno venendo analizzate alcune possibili soluzioni che permettano a HYPERDBG di interagire con il controller USB. Prima di affrontare il problema, è utile ricordare come sia necessario intercettare la pressione dei tasti sia quando l'esecuzione è in modalità non-root, per rilevare la pressione dell'hotkey che attiva l'interfaccia grafica del debugger, che quando si è in modalità root, per poter interpretare correttamente i comandi dell'utente. Le soluzioni valutate finora sono solamente teoriche e, di conseguenza, non verranno trattate dettagliatamente all'interno di questo lavoro di tesi, ma solamente accennate.

Soluzioni in modalità non-root. Una prima alternativa per intercettare i tasti quando l'esecuzione è in modalità non-root, consiste nel modificare l'entry corrispondente agli interrupt provenienti dalla tastiera all'interno della Interrupt Descriptor Table (IDT), seguendo una tecnica spesso usata dai *keylogger*, denominata *IDT Hooking*. Inserendo nella IDT, in corrispondenza dell'entry della tastiera, l'indirizzo di un handler creato appositamente, è possibile intercettare i tasti premuti. Qualora il tasto la cui pressione ha causato l'invocazione dell'interrupt handler non corrisponda all'hotkey di HYPERDBG, l'handler si occupa di notificare il sistema, altrimenti abilita l'interfaccia grafica del debugger. Questa tecnica, come si può facilmente intuire, è molto invasiva dal momento che modifica una delle strutture fondamentali del sistema guest. Tuttavia, le versioni più moderne della virtualizzazione hardware includono la possibilità di configurare l'hypervisor in modo che l'esecuzione dell'istruzione `SIDT` causi una `exit`. Quest'istruzione permette di leggere il contenuto di una entry dell'IDT e, di conseguenza, la possibilità di interrompere l'esecuzione consente di mascherare la differenza causata dall'IDT Hooking. Ciononostante, questa soluzione non è molto efficace perché rende la trasparenza di HYPERDBG dipendente dall'hardware del sistema da analizzare.

Una seconda soluzione da considerare per intercettare i tasti premuti dall'utente deriva, ancora una volta, da una tecnica utilizzata dai *keylogger*. Senza entrare troppo a fondo nei dettagli tecnici, è possibile sfruttare la scarsa velocità di una transazione di dati tra una periferica USB e il sistema per intercettare i tasti *prima* che vengano processati dal sistema [5]. Il controller UHC, prima generazione dei controller USB, processa una transazione ogni millisecondo [17]. Qualora il sistema operativo guest sia Windows, è possibile sfruttare una *race condition* nella catena di procedure che il sistema invoca per gestire la transazione e intercettare in questo modo i tasti premuti. Come si può facilmente intuire, questa tecnica è fortemente dipendente dal sistema operativo guest. Inoltre, è anche dipendente dal tipo di controller USB dal momento che è specifica per controller UHC e potrebbe non essere adatta per i controller più evoluti (EHC, OHC).

Infine, un terzo approccio considerato, più semplice dei due precedenti, consiste nell'utilizzare un programma eseguito in user space nel sistema guest per intercettare

la pressione dell'hotkey di HYPERDBG. Per notificare l'hypervisor della pressione dell'hotkey, il programma in user space può effettuare una *hypercall*, come spiegato nella Sezione 4.3.2. Naturalmente, l'utilizzo di un programma user space all'interno del guest non è trasparente e, in caso la trasparenza sia essenziale per i compiti di analisi che si intendono effettuare tramite HYPERDBG, è necessario integrare il codice dell'hypervisor con particolari tecniche atte a nascondere processi eseguiti in user space, similmente a quanto fanno i malware con capacità di rootkit.

Soluzioni in modalità root. Contrariamente a quanto accade in modalità non-root, dove è necessario semplicemente identificare la pressione di un singolo tasto predefinito, quando si è in modalità root si deve leggere l'input dell'utente e, di conseguenza, è necessaria una completa interazione con il device hardware. Una prima soluzione per identificare i tasti premuti dall'utente consiste nell'analizzare le strutture dati utilizzate per il trasferimento di dati tra un device USB e il sistema. Assumendo che il controller USB a cui è collegata la tastiera sia di tipo UHC, come avviene nella maggior parte delle configurazioni hardware, tale trasferimento di dati viene effettuato tramite una zona di memoria fisica condivisa tra il sistema e la periferica (*Memory Mapped I/O*) in cui viene creato un array, chiamato *Frame List*, le cui entry contengono puntatori a altre strutture dati, i *Transfer Descriptor* (TD), che contengono le informazioni di trasferimento vere e proprie [17]. Dal momento che il controller USB è una periferica PCI, è possibile eseguire una scansione del bus PCI per recuperare le informazioni riguardo all'indirizzo fisico a cui è associato l'array *Frame List* e un contatore che indica l'entry correntemente utilizzata. Una volta ottenute queste informazioni, dalla modalità root è possibile utilizzare le funzioni di interazione con la memoria per accedere alla memoria condivisa. Dal momento che, al fine di evitare numerose complicazioni, gli interrupt sono disabilitati quando l'esecuzione è in modalità root, non è possibile sapere quando il controller USB ha dei dati pronti da trasferire e li ha inseriti nelle appropriate code di TD. Di conseguenza, l'unica soluzione possibile per sapere se sono stati premuti dei tasti è effettuare uno scan continuo delle strutture dati contenute nella memoria condivisa per andare a verificare la presenza dei TD corrispondenti alla tastiera e, eventualmente, leggerli.

Per evitare la necessità di effettuare polling sulla memoria condivisa tra il controller USB e il sistema si possono lasciare abilitati gli interrupt anche in modalità root. Come abbiamo già detto, questo introduce una notevole complessità, perché richiede di riprogrammare completamente l'*Advanced Programmable Interrupt Controller (APIC)* in modo tale che consegni tutti gli interrupt relativi alle periferiche USB e accodi gli altri fin quando il controllo dell'esecuzione non ritorna al guest. Questo approccio introduce a sua volta due ulteriori problemi. Il primo deriva dal fatto che gli interrupt accodati dall'APIC potrebbero essere scartati se l'utente di HYPERDBG rimane in modalità root per troppo tempo, causando malfunzionamenti visibili nel guest. Il secondo problema, invece, è causato dal fatto che il controller USB invia il medesimo interrupt indipendentemente da quale device è pronto a effettuare una transizione dati, aumentando così il rischio che HYPERDBG interferisca con le transizioni degli altri device USB presenti sul sistema, potenzialmente corrompendone irreparabilmente i risultati, per esempio interrompendo il trasferimento di un file a un unità disco esterna. Per evitare la complessità causata dalla necessità di riprogrammare completamente l'APIC, una soluzione alternativa potrebbe consistere nel lasciarlo inalterato, abilitare gli interrupt anche in modalità root e predisporre tutte le entry della IDT dell'hypervisor in modo che utilizzino tutte lo stesso handler che si occupa di creare e gestire una coda di interrupt, che verranno iniettati nel sistema guest alla prima entry, possibilità che è stata illustrata nel Capitolo 2. Ovviamente, anche in questo caso è necessario predisporre un handler apposito per gli interrupt provenienti dal controller USB. Questo approccio, pur semplificato dal fatto di non dover riprogrammare l'APIC, incontra i medesimi problemi evidenziati nella soluzione precedente.

Per concludere, viene evidenziato nuovamente come il problema posto dalle periferiche USB sia ancora aperto e, nel prossimo futuro dello sviluppo di HYPERDBG, verranno valutate le soluzioni proposte che, eventualmente, saranno implementate e integrate nel codice del debugger al fine di aumentarne l'usabilità.

5.4 Ulteriori sviluppi futuri

L'eliminazione delle limitazioni elencate nelle Sezioni precedenti è sicuramente uno degli sviluppi futuri prioritari di HYPERDBG, ma non è l'unico. Oltre a questo, infatti, è prevista l'aggiunta del supporto per altre piattaforme, sia hardware che software. Per quanto riguarda le piattaforme hardware, è prevista l'aggiunta del supporto per AMD-V, la tecnologia di virtualizzazione hardware-assisted sulle famiglie di processori AMD. Il codice di HYPERDBG è stato progettato in modo che sia possibile aggiungere il supporto per questa seconda piattaforma hardware senza dover modificare parti di codice già esistenti. Una volta creato il modulo che mette a disposizione le funzioni dipendenti dalla piattaforma di virtualizzazione hardware, è sufficiente compilarlo e istruire il codice che utilizza queste funzioni affinché usi la versione corrispondente all'hardware per cui si vuole compilare HYPERDBG. Per nascondere al resto del codice le differenze tra tali moduli, in modo da minimizzare le modifiche necessarie, è stata realizzata un'interfaccia comune alle due diverse piattaforme hardware. Tale interfaccia espone sempre il medesimo insieme di funzioni, indipendentemente dalla piattaforma, e si occupa di richiamare le versioni appropriate a seconda di quale supporto, VT-x o AMD-V, è stato selezionato a compile-time. Grazie a questa ingegnerizzazione del codice, quindi, per aggiungere il supporto per la virtualizzazione AMD, sarà sufficiente implementare il modulo che contiene le funzioni che dipendono in una qualsiasi maniera dalla piattaforma hardware.

Un ulteriore sviluppo futuro riguarda l'adattamento di HYPERDBG a altri sistemi guest. Come è già stato detto più volte, la maggior parte del codice, sia del componente hypervisor sia del componente di debugging, è indipendente dal sistema che si intende analizzare. Tra le parti non indipendenti, solamente una è fondamentale, cioè il driver minimale che si occupa di caricare e avviare HYPERDBG, mentre le altre, che implementano le funzionalità di debugging e di introspezione dipendenti dal sistema guest, sono opzionali. Quindi, inizialmente, è previsto lo sviluppo del driver minimale come un modulo del kernel di Linux, in modo che siano utilizzabili almeno le funzionalità essenziali di debug. Una volta ottenuto ciò, saranno aggiunte anche le funzionalità di introspezione, implementate al momento solo per ispezionare le strut-

ture interne di un sistema Windows XP. Per quanto le strutture interne del kernel di Linux siano diverse da quelle utilizzate da Windows, l'idea alla base delle tecniche di introspezione è la medesima e, di conseguenza, non verrà illustrata in ulteriori dettagli nel presente lavoro di tesi.

Capitolo 6

Lavori correlati

In questo Capitolo vengono presentati alcuni lavori di ricerca e strumenti simili, nello scopo o nell'implementazione, a HYPERDBG e, più genericamente, al modello di analisi proposto in questo lavoro di tesi. Inizialmente, si analizzano alcuni framework che danno la possibilità di instrumentare dinamicamente il codice kernel, per poi passare a analizzare i principali debugger a livello kernel attualmente disponibili. Successivamente si presentano alcuni framework di analisi basati su macchine virtuali e la possibilità di applicare il paradigma di programmazione aspect-orientated al codice kernel. Per ognuno di questi lavori e strumenti vengono presentati i principali vantaggi e svantaggi e sottolineate le differenze rispetto a HYPERDBG.

6.1 Instrumentazione dinamica del kernel

L'instrumentazione tipicamente consiste nella possibilità di integrare un frammento di codice con funzionalità non previste originalmente, spesso al fine di monitorare e analizzare performance, diagnosticare errori e tracciare l'esecuzione di diverse tipologie di componenti, a svariati livelli di granularità: dal singolo programma all'intero kernel di un sistema. Esistono diversi tipi di instrumentazione che possono essere applicati al codice sorgente o al corrispondente file compilato, sia staticamente, cioè andando a modificare il programma *prima* che questo venga eseguito, che dinamicamente, cioè integrando le modifiche a *run-time*.

In questa Sezione vengono descritti alcuni approcci di strumentazione dinamica di codice kernel introdotti in precedenti lavori di ricerca, dal momento che sono quelli che presentano più somiglianze all'approccio di debugging e analisi proposto in questo lavoro di tesi.

6.1.1 DTrace

DTrace è un servizio incluso nel kernel di Solaris che permette di effettuare strumentazione dinamica di un sistema [3]. I punti di forza principali di DTrace sono l'efficienza e la flessibilità. Infatti il framework di strumentazione non introduce nessun overhead e, allo stesso tempo, fornisce decine di migliaia di punti di strumentazione. Le azioni che devono essere effettuate a fronte del raggiungimento di un punto di strumentazione possono essere espresse in un linguaggio di controllo di alto livello, simile al C, che permette di effettuare un'ampia gamma di operazioni a un dato punto di strumentazione. Inoltre, uno dei principali punti di forza di DTrace consiste nel fatto che esso garantisce che le azioni specificate tramite il linguaggio di controllo siano *sicure (safe)* per il sistema analizzato. In particolare, effettuando diversi controlli su tali azioni e intercettando gli errori che possono avvenire nei punti strumentati a run-time, per esempio accessi illegali alla memoria, DTrace impedisce che le azioni effettuate introducano errori fatali nel sistema sotto analisi corrompendone il funzionamento.

Tramite i punti di strumentazione messi a disposizione da DTrace è possibile fornire le stesse funzionalità di analisi e debugging offerte da HYPERDBG. Tuttavia, DTrace soffre di una limitazione molto significativa dovuta dalla necessità che alcuni punti di strumentazione esistano a priori nel kernel. Chiaramente questa necessità lo rende estremamente dipendente dal sistema operativo analizzato e, contemporaneamente, gli impedisce di essere applicato a sistemi operativi di cui non è disponibile il codice sorgente. HYPERDBG non è affetto da queste limitazioni, dal momento che permette di analizzare il codice kernel di un sistema senza dipendere da alcun suo componente preesistente, né dalla disponibilità del suo codice sorgente.

6.1.2 KernInst

KernInst è un framework che permette l'strumentazione dinamica di codice kernel [42]. Il codice può essere inserito, o rimosso, dinamicamente nei punti di instrumentazione messi a disposizione dal framework. Per utilizzare KernInst non è necessario modificare o ricompilare il kernel da analizzare. Per fornire le funzionalità di instrumentazione, il framework include un demone e un modulo, che devono essere installati sulla macchina di cui si vuole instrumentare il kernel. Per specificare le operazioni da intraprendere a un dato punto di instrumentazione, KernInst mette a disposizione una API C++ che nasconde tutti i dettagli di comunicazione tra il demone e il programma che instrumenta effettivamente il kernel. Sono state implementate diverse applicazioni utilizzando le API di KernInst, tra cui si può notare Kperfmon, uno strumento che permette di raccogliere precisi dati riguardo alle performance di un kernel.

L'utilizzo di un demone e di un modulo del kernel permette a KernInst di poter essere usato senza dover modificare a priori il codice del kernel sotto analisi. Tuttavia, questa caratteristica non lo rende indipendente dal sistema su cui viene utilizzato né permette di utilizzarlo con sistemi closed source. Infatti, il componente che fornisce le funzionalità di instrumentazione è un modulo che deve essere reimplementato per ogni kernel che si deve instrumentare e non è possibile implementarlo per sistemi di cui non si ha a disposizione il codice sorgente o un'ampia documentazione riguardo ai suoi meccanismi interni. Si fa notare che, come descritto in precedenza, anche HYPERDBG necessita di un driver kernel per poter essere caricato sul sistema da analizzare. Tuttavia, il driver di HYPERDBG contiene esclusivamente il codice necessario a caricare l'hypervisor mentre il motore di analisi vero e proprio è rinchiuso nell'hypervisor stesso. La realizzazione di un driver che effettua il caricamento di HYPERDBG su sistemi diversi da quelli previsti originalmente è triviale. Inoltre, un ulteriore vantaggio di KernInst nei confronti di HYPERDBG consiste nel fatto che i componenti (modulo e demone) del primo framework devono rimanere in esecuzione sul sistema da analizzare per l'intera durata delle analisi, rendendo il framework facilmente identificabile. Invece, come è stato spiegato dettagliatamente nel Capitolo 4, una volta

caricato, HYPERDBG risulta essere completamente trasparente al sistema sotto analisi.

6.2 Debugger a livello kernel

Negli ultimi anni sono stati compiuti numerosi sforzi per sviluppare debugger a livello kernel che fossero sia efficienti che affidabili. Infatti, tali applicazioni sono essenziali per molte attività, come lo sviluppo di driver. Uno dei primi e più utilizzati debugger kernel è stato SoftICE [40], ma il progetto è stato abbandonato. Tuttavia sono disponibili alternative sia commerciali, come Syser [41], che open source, come rr0d [36]. Inoltre, le versioni più moderne di Windows includono nativamente un sistema di debugging per il kernel [30]. Purtroppo, per sfruttare completamente le capacità dell'infrastruttura di debugging kernel di Microsoft, il sistema sotto analisi deve essere collegato fisicamente tramite un cavo seriale a un'altra macchina, da cui vengono condotte le analisi. Tutti questi approcci sono affetti dallo stesso problema: per analizzare codice a livello kernel fanno affidamento su di un modulo del kernel stesso. Come è stato già evidenziato più volte nei capitoli precedenti, questa limitazione permette a processi eseguiti a un sufficiente livello di privilegio di identificare la presenza dello strumento di analisi e, potenzialmente, di corromperne i risultati. Inoltre, è necessario implementare il modulo in questione diversamente per ogni sistema operativo che si vuole analizzare. HYPERDBG, come è stato illustrato, non è identificabile da processi che sono eseguiti nel sistema analizzato, indipendentemente dal livello di privilegio a cui sono eseguiti, né è dipendente in alcun modo da tale sistema. Queste due proprietà lo rendono, di fatto, superiore agli approcci di debugging illustrati in questa Sezione.

6.3 Strumenti di analisi basati su macchine virtuali

Nella Sezione 3.1.2 è stata illustrata la possibilità di analizzare dinamicamente un sistema tramite l'uso di macchine virtuali. Questo approccio consiste nell'eseguire il sistema da analizzare all'interno di una macchina virtuale e nell'effettuare le analisi dall'esterno [14]. In questa Sezione vengono brevemente illustrati alcuni framework di analisi basati su macchine virtuali.

In alcuni lavori di ricerca [22, 10], gli autori propongono particolari macchine virtuali che permettono di effettuare *execution replaying*: un utente può navigare avanti e indietro attraverso una “storia” dell’esecuzione dell’intero sistema. Questa possibilità risulta molto utile dal punto di vista del debugging perché libera l’utente dalla necessità di ricreare un problema al fine di analizzarlo, permettendogli di navigare a ritroso nella storia di esecuzione, fino al punto in cui si è manifestato il difetto. Questa peculiarità risulta essere molto utile per analizzare errori in componenti che vengono invocati solamente in situazioni particolari, come in risposta a un interrupt o alla ricezione di un input esterno, oppure per problemi che richiedono lunghe esecuzioni prima di manifestarsi. Invece, dal punto di vista della sicurezza, la possibilità di investigare i dettagli dell’esecuzione “passata” di un sistema può essere molto utile per capire come è stato effettuato un attacco. Più in particolare, King *et al.* propongono un debugger che sfrutta funzionalità di *execution replaying* per permettere a un utente, tra le altre cose, di impostare breakpoint in punti dell’esecuzione precedenti a quello in cui è stato fermato il sistema e di effettuare single-step all’indietro nella “storia” delle istruzioni eseguite dal sistema. ReVirt invece, presentato da Dunlap *et al.*, è un framework che si propone di ovviare al problema dell’integrità e della consistenza dei risultati ottenuti da strumenti di logging e tracing su di un sistema compromesso da malware in grado di modificare il kernel del sistema sotto analisi e, conseguentemente, di corrompere tali risultati al fine di nascondere le proprie tracce. L’approccio proposto in questo lavoro di ricerca consiste nell’eseguire il sistema da analizzare all’interno di una macchina virtuale, in modo da impedire a un eventuale malware eseguito a livello kernel di corrompere i risultati delle analisi effettuate. Inoltre, sfrutta tecniche di *execution replaying* per garantire un’estrema completezza dei risultati ottenuti tramite le analisi.

Infine, Chow *et al.* propongono Aftersight [4], uno strumento che permette di effettuare analisi di un sistema eseguito all’interno di una macchina virtuale, introducendo un overhead minimo nel sistema guest. Infatti, per non influenzare eccessivamente le prestazioni del sistema analizzato, Aftersight disgiunge la registrazione degli eventi rilevanti dalle analisi vere e proprie, che vengono effettuate da un motore separato dal framework di virtualizzazione. Inoltre, si possono aggiungere tipologie di analisi diverse da quelle pensate originalmente, senza intaccare il carico del siste-

ma virtualizzato. Aftersight fa parte ufficialmente del framework di virtualizzazione VMWare [47].

Gli approcci presentati finora sono tutti affetti da due problemi principali. Il primo deriva dall'assunzione che gli autori fanno riguardo alla correttezza dei Virtual Machine Monitor di cui fanno uso. Come è già stato accennato brevemente e come è stato dimostrato in [28, 29], quest'assunzione è tutt'altro che valida. Infatti, un attaccante potrebbe sfruttare la presenza di errori nei Virtual Machine Monitor per identificarne la presenza. Il secondo problema è causato dalla necessità che il sistema da analizzare sia eseguito *nativamente* all'interno di una virtual machine, che impedisce di applicare questi approcci a sistemi che richiedono configurazioni hardware particolari o che non possono essere spenti e riavviati come guest di una macchina virtuale. HYPERDBG non implementa le funzionalità di execution replaying illustrate precedentemente ma potrebbe essere facilmente esteso per includerle e, essendo in grado di *migrare* un sistema operativo *in esecuzione* all'interno di una macchina virtuale, non sarebbe affetto dalla limitazione che è stata appena evidenziata. Inoltre, HYPERDBG non interferisce in alcun modo con l'utilizzo dell'hardware da parte del sistema da analizzare e, di conseguenza, è utilizzabile anche in presenza di sistemi hardware complessi o non comuni.

6.4 Analisi di malware tramite virtualizzazione hardware assisted

Dinaburg *et al.* in [8] propongono Ether, un analizzatore di malware che presenta diversi punti in comune con HYPERDBG. Ether, infatti, sfrutta la virtualizzazione hardware-assisted per effettuare le analisi da un ambiente più privilegiato del sistema operativo, isolato da questo e completamente trasparente. Tutte queste, infatti, risultano caratteristiche fondamentali per effettuare analisi di malware, dal momento che esemplari particolarmente evoluti sono in grado di identificare la presenza degli analizzatori classici che, tipicamente, risiedono nel sistema infettato o creano un ambiente di analisi apposito tramite emulazione, risultando particolarmente facili da identifica-

re. Ether, invece, operando completamente al livello di privilegio dell'hypervisor, non ha parti che possano essere facilmente identificate da un malware eseguito all'interno dell'ambiente di analisi. I concetti di base di Ether sono molto simili a quelli di HYPERDBG, però, contrariamente a quest'ultimo, non include un hypervisor creato ad-hoc ma utilizza quello di Xen [2]. Questo comporta alcune limitazioni per Ether, dal momento che Xen utilizza la tecnica nota come paravirtualizzazione (presentata nella Sezione 2.1.2) che richiede che il codice sorgente del sistema operativo venga modificato prima di poter essere virtualizzato. Ne consegue che Ether può essere utilizzato per effettuare analisi di malware solo in sistemi che sono predisposti per essere eseguiti all'interno di una macchina virtuale Xen. Inoltre, Xen richiede che il sistema operativo da virtualizzare sia *avviato* nativamente all'interno di una macchina virtuale e, conseguentemente, Ether può essere utilizzato solo su sistemi già virtualizzati. HYPERDBG, grazie alle sue caratteristiche di trasparenza e isolamento dal sistema operativo analizzato, potrebbe essere utilizzato per l'analisi di malware con gli stessi vantaggi di Ether, con l'aggiunta, però, della possibilità di analizzare sistemi operativi già in esecuzione *non* all'interno di una macchina virtuale e di non richiedere modifiche a priori di tali sistemi.

6.5 Programmazione aspect-oriented

La programmazione aspect-oriented è un paradigma di programmazione che mira a aumentare la modularità incapsulando i *cross-cutting concern*, cioè parti di codice volte a risolvere uno specifico problema, ma disperse nell'intero programma in modo trasversale alla decomposizione funzionale [21]. Un tipico esempio di cross-cutting concern sono le funzioni di logging, dal momento che sono spesso usate nella maggior parte dei componenti di un sistema e una modifica in tali funzioni riguarda (*concern*) tutti i componenti che le richiamano. I costrutti aspect-oriented permettono di descrivere in maniera compatta queste computazioni trasversali, incapsulandole in unità di codice dette "aspetti". Ogni aspetto contiene una porzione di codice imperativo, chiamata "advice", che viene automaticamente "intessuta" nel codice originale del programma dal compilatore. Per stabilire dove vengono aggiunti gli advice, il programmatore

aspect-oriented può specificare dei punti, detti “joint-point” e delle condizioni (before, after, around). L’unione di joint-point e condizioni può formare, a sua volta, condizioni anche estremamente complesse, dette “point-cut”.

Negli ultimi anni sono stati proposti molti progetti e lavori di ricerca basati sul paradigma aspect-oriented e finalizzati al debugging, all’analisi e all’instrumentazione sia di programmi [26] che di sistemi operativi [6, 25]. Questi approcci però, prevedono sempre una prima fase di compilazione del programma o del sistema da analizzare, al fine di integrare il codice contenuto negli aspetti e il codice originale dell’applicazione. Più recentemente, Engel *et al.* in [11] hanno proposto TOSKANA, che permette di applicare il paradigma di programmazione aspect-oriented a funzioni contenute nel kernel, tramite un approccio più dinamico dei suoi predecessori. Infatti, l’implementazione degli aspetti è contenuta all’interno di moduli kernel che possono essere scambiati a run-time, al fine di sostituire dinamicamente il codice degli advice.

HYPERDBG può essere usato per ottenere facilmente funzionalità simili a quelle offerte dall’applicazione del paradigma aspect-oriented al codice kernel. Per esempio, è possibile specificare azioni che devono essere eseguite a fronte dell’esecuzione di una particolare funzione del kernel che si intende analizzare. Inoltre, contrariamente a quanto accade per l’aspect-orientation, queste azioni possono essere specificate in maniera completamente dinamica, senza modificare in alcun modo il sistema sotto analisi.

Conclusioni

Nel presente lavoro di tesi è stato proposto un approccio innovativo per l'analisi dinamica di codice kernel. Il metodo proposto è in grado di assicurare le stesse potenzialità degli approcci di analisi dinamica finora proposti nell'ambito della ricerca scientifica o come strumenti di supporto per gli sviluppatori, senza però soffrire delle limitazioni caratteristiche di tali approcci. Per essere in grado di ottenere ciò, la tecnica proposta in questo lavoro di tesi sfrutta il supporto hardware alla virtualizzazione, presente sulla maggior parte dei processori moderni. Grazie all'uso di questa tecnologia innovativa, l'approccio proposto per l'analisi dinamica di codice kernel non richiede *nessuna* modifica del sistema operativo da analizzare, né necessita che questo sia riavviato, permettendo quindi di analizzare anche sistemi di produzione. Inoltre, contrariamente agli approcci correntemente usati, è completamente trasparente al sistema analizzato. L'approccio, molto brevemente, consiste nell'installare un virtual machine monitor minimale e nel migrare il sistema da analizzare all'interno di una macchina virtuale a *run-time*, senza richiedere che questo sia riavviato. Una volta installato il VMM, le procedure di analisi vengono eseguite al livello di privilegio del VMM, in modo da non poter essere influenzate dal sistema analizzato. Una volta concluse le analisi, sia il VMM che il motore di analisi possono essere rimossi completamente e il controllo dell'hardware torna nelle mani del sistema operativo analizzato.

L'approccio proposto in questo lavoro di tesi è stato applicato sperimentalmente per creare un debugger kernel, chiamato HYPERDBG. La realizzazione di tale strumento

costituisce una prova effettiva della validità dell'approccio di analisi dinamica di codice kernel. Applicando la tecnica proposta, infatti, HYPERDBG è in grado di superare le potenzialità e le prestazioni dei simili strumenti finora utilizzati dagli sviluppatori kernel.

Inoltre, le caratteristiche di dinamicità e trasparenza dell'approccio proposto lo rendono estremamente adattabile a un ampio insieme di possibilità che sono state solo vagamente accennate in questo lavoro di tesi, sia nel campo dei sistemi operativi che della sicurezza. Si ritiene, di conseguenza, che la tecnica proposta come contributo originale di questo lavoro di tesi possa costituire, oltre a una sostanziale innovazione nel campo dell'analisi dinamica di codice kernel, anche un'ottima base di partenza per futuri sviluppi di ricerca legati all'ambito della virtualizzazione e della sicurezza.

Parte del lavoro oggetto di questa tesi è stato presentato alla *25th International Conference on Automated Software Engineering (ASE)* [12].

Bibliografia

- [1] AMD, Inc. AMD virtualization. <http://www.amd.com/us/products/technologies/virtualization/Pages/virtualization.aspx>.
- [2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, October 2003.
- [3] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of USENIX Annual Technical Conference*, pages 15–28, Boston, MA, U.S.A., June 2004.
- [4] J. Chow, T. Garfinkel, and P. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, Boston, MA, U.S.A., June 2008.
- [5] USB Keyboard Sniffing with TD. <http://beist.org/>.
- [6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference*, pages 88–98, Vienna, Austria, September 2001.
- [7] F. Desclaux and K. Kortchinsky. Vanilla Skype part 1, June 2006.
- [8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, Alexandria, Virginia, USA, October 2008.

- [9] DirectX. <http://en.wikipedia.org/wiki/DirectX>.
- [10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI '02: 5th Symposium on Operating Systems Design and Implementations*, Boston, MA, U.S.A., December 2002.
- [11] M. Engel and B. Freisleben. TOSKANA: A toolkit for operating system kernel aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:182–226, 2006.
- [12] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 417–426, Antwerp, Belgium, September 2010.
- [13] Framebuffer. <http://en.wikipedia.org/wiki/Framebuffer>.
- [14] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Symposium on Network and Distributed Systems Security*, San Diego, CA, U.S.A., February 2003.
- [15] GNU Project - Free Software Foundation. objdump. <http://www.gnu.org/manual/binutils-2.10.1/htmlchapter/binutils4.html>.
- [16] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, pages 34–45, June 1974.
- [17] Intel Corporation. *Universal Host Controller Interface (UHCI) Design Guide*, March 1996.
- [18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B*, September 2008.
- [19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, September 2008.
- [20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, September 2008.

- [21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 1997.
- [22] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference*, pages 71–84, Anaheim, CA, U.S.A., April 2005.
- [23] Linux Kernel. <http://kernel.org/>.
- [24] Linux Trace Toolkit. <http://lttng.org/>.
- [25] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, pages 49–54, Málaga, Spain, June 2002.
- [26] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–256, Washington, DC, U.S.A., April 2002.
- [27] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, pages 21–40, Bonn, Germany., July 2010. Springer.
- [28] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, pages 261–272, Chicago, IL, U.S.A., July 2009. ACM.
- [29] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA)*, pages 171–182, Trento, Italy., July 2010.
- [30] Microsoft Corporation. Debugging Tools for Windows. <http://www.microsoft.com/whdc/devtools/debugging/default.aspx>.
- [31] The MINIX 3 Operating System. <http://www.minix3.org/>.

- [32] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.
- [33] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
- [34] OSDev. VGA Hardware. http://wiki.osdev.org/VGA_Hardware.
- [35] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, August 2009. ACM.
- [36] Rasta ring 0 debugger. <http://rr0d.droids-corp.org/>.
- [37] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, U.S.A., 2004.
- [38] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, U.S.A., 2002. IEEE Computer Society.
- [39] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.
- [40] SoftICE. <http://en.wikipedia.org/wiki/SoftICE>.
- [41] Syser Kernel Debugger. <http://www.sysersoft.com/>.
- [42] A. Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin-Madison, 2001.
- [43] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *21st Annual Computer Security Applications Conference*, pages 381–392, Tucson, AZ, U.S.A., December 2005.
- [44] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *In Proceedings of 2006 IEEE Symposium on Security and Privacy*, pages 264–279, Oakland, CA, U.S.A., May 2006.

-
- [45] VirtualBox. <http://www.virtualbox.org>.
- [46] Vivek Thampi. `udis86`. <http://udis86.sourceforge.net/>.
- [47] VMWare. <http://www.vmware.com>.
- [48] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.
- [49] Windows Research Kernel. <http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.aspx>.