UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA TRIENNALE IN INFORMATICA

STUDIO E IMPLEMENTAZIONE DI UN INTEGRITY-CHECKER HARDWARE-ASSISTED

Relatore: Prof. Danilo BRUSCHI
Correlatore: Dott. Aristide FATTORI

Tesi di Laurea di:
Riccardo SCHIRONE
Matricola 776567

Anno Accademico 2012–13

Indice

Capitolo 1

Introduzione

I malware sviluppati al giorno d'oggi sono sempre più sofisticati e difficili da rilevare. Oltre a compromettere un sistema, molti malware puntano a nascondere le loro attività così da poterlo infettare per il più lungo tempo possibile. Inoltre, a causa della grandezza e della complessità dei programmi utilizzati su un computer, è molto probabile che questi contengano delle vulnerabilità sfruttabili da un attaccante per compromettere la macchina.

Tradizionalmente, nei sistemi desktop e server, vengono utilizzati strumenti di protezione, come gli anti-virus, eseguiti come applicazioni all'interno del sistema operativo (OS), nel tentativo di proteggere il sistema da possibili attacchi. Nonostante questa difesa potesse essere accettabile anni fa, ormai non è più sufficiente, poiché diversi tipi di attacchi mirano non solo a compromettere le applicazioni, ma anche il sistema operativo. In un ambiente in cui quest'ultimo, che ha il controllo totale della macchina su cui è eseguito, viene compromesso, le analisi effettuate da un comune tool anti-malware risultano inutili. Un malware, infatti, può operare silenziosamente all'interno del sistema operativo, rimanendo invisibile alle applicazioni utente o ad altre parti del sistema. Difatti, il codice eseguito a livello *kernel*, il nucleo dell'OS, possiede un privilegio maggiore di qualsiasi altra applicazione utente, che gli permette di comandare direttamente l'hardware e gestire le risorse della macchina a proprio piacimento. Le normali applicazioni, invece, hanno bisogno di fare affidamento sul sistema operativo per interagire con il resto del sistema e perciò sono facilmente soggiogabili da questo.

Per tale motivo sono state sviluppate tecniche che sfruttano i livelli di protezione offerti dalle architetture Intel, così da poter avere tool di rilevazione malware più affidabili. L'idea alla base di questa scelta è quella di proteggere e isolare il software di analisi dal software potenzialmente vulnerabile o maligno, così che il malware non possa interferire con il meccanismo di protezione oppure corromperlo. Uno fra i tanti modi di rilevare la presenza di malware è quello di fare integrity checking, controllando che le parti di sistema che non dovrebbero essere modificate rimangano, appunto, inalterate. È molto difficile, infatti, infettare un sistema senza modificare parti dello stesso. Il programma che controlla l'integrità viene chiamato *integrity checker* o *integrity monitor* e può essere eseguito in diversi modi. Come spiegato in precedenza, eseguirlo con lo stesso privilegio o con privilegio inferiore del sistema operativo, vorrebbe dire renderlo soggetto a codice malevolo eseguito all'interno di quest'ultimo. Per tale motivo sono state utilizzate altre tecniche, tra cui una che prevede l'utilizzo di tecnologie hardware assisted, in particolare Intel VT-x [25] e SMM [11].

L'hypervisor è un software eseguito con un livello di privilegio maggiore rispetto al sistema operativo, che permette di eseguire più OS su una sola macchina fisica. Il supporto hardware alla virtualizzazione, Intel VT-x, permette di creare hypervisor particolarmente resistenti e trasparenti, più facilmente che tramite approcci puramente software. Ciò li rende adatti allo sviluppo, tra altre cose, di sistemi di integrity checking. Infatti, in ambito di ricerca, sono state sviluppate diverse soluzioni fra cui [22] [5]. Soluzioni basate sul software, invece, sono vulnerabili a diversi attacchi [24] [10] [19].

Questo lavoro di tesi si concentra sulla seconda tecnologia menzionata, cioè System Management Mode, una particolare modalità operativa delle CPU Intel, che ha il controllo massimo sulla macchina e un livello di privilegio maggiore di qualsiasi altro software [15]. SMM è raggiungibile esclusivamente attraverso la generazione di un System Management Interrupt (SMI) e, quando questo viene ricevuto, il processore cambia ambiente di esecuzione, andando a eseguire il codice all'interno della System Management RAM (SMRAM). Tale area di memoria può essere resa inaccessibile a qualunque altro software, incluso l'hypervisor, garantendo in questo modo l'integrità del codice e dei dati al suo interno. Perciò SMM risulta luogo ideale per un tool di analisi del sistema, ma anche per un malware, poiché vorrebbe dire rimanere invisibile

alle applicazioni utente, al sistema operativo e all'hypervisor [12].

In questi ultimi anni sono stati sviluppati molti integrity checker che sfruttano le caratteristiche di SMM per raggiungere un alto grado di affidabilità. Fra tutte, quelle di maggior interesse per questo lavoro di tesi sono *AppCheck* [28], *Spectre* [31], *HyperCheck* [26] e *HyperSentry* [6]. HyperSentry e HyperCheck si preoccupano di controllare solamente l'integrità di hypervisor e, entro alcuni limiti, del kernel del sistema operativo. HyperSentry, in particolare, presenta una buona soluzione per la generazione del System Management Interrupt, attraverso l'utilizzo di un canale di comunicazione fuori dal controllo della CPU. Spectre, invece, fornisce un framework per l'analisi di applicazioni e il rilevamento di attacchi basati sulla memoria, ma utilizza un'assunzione troppo rigida sul posizionamento dei dati del kernel all'interno della memoria principale. Inoltre, tutte queste soluzioni hanno il difetto di gestire solo sistemi a 32 bit, senza preoccuparsi di quelli a 64 bit che, però, sono ormai molto diffusi.

Invece, il software sviluppato in questo lavoro di tesi, chiamato *Yet Another SMM Integrity Checker* (YASIC), è un integrity checker che sfrutta i vantaggi offerti da SMM per controllare l'integrità di processi e librerie eseguite su un sistema a 64 bit, facendo affidamento esclusivamente sulle poche informazioni a disposizione per analizzare il contenuto della memoria principale. Nella modalità System Management, il software ha il controllo diretto sull'hardware e sulla macchina, senza che si debba fare affidamento su altre parti del sistema che potrebbero essere state compromesse. Il compito di YASIC è controllare l'integrità dei processi che sono in esecuzione sul sistema, per evitare che questi vengano modificati da malware o, comunque, in modi non permessi. È, infatti, una tecnica comune per i malware infettare lo spazio di memoria di altri processi, ritenuti fidati dal sistema, così da introdurre codice malevolo al loro interno e rimanere invisibili ai tool di rilevazione. YASIC, invece, controlla proprio che i processi rimangano integri mentre sono in esecuzione, tramite controlli periodici. Esso può essere suddiviso in tre parti: un modulo di acquisizione dello stato del sistema, un modulo di comunicazione e uno di analisi. I primi due vengono eseguiti sulla macchina che si intende analizzare, l'ultimo, invece, viene eseguito su un altro computer che ha il compito di controllare che i dati raccolti dal primo modulo, e inviati dal secondo,

siano integri.

L'utilizzo di un integrity checker basato esclusivamente su SMM e sull'hardware, permette di ottenere delle analisi accurate e affidabili, perché viene ridotto al minimo indispensabile l'insieme di componenti che vengono sfruttate. Così facendo, se anche il sistema operativo o l'hypervisor dovessero venire compromessi, YASIC sarebbe in grado di analizzare la macchina, senza che le sue analisi possano essere facilmente aggirate. Inoltre, YASIC ha la caratteristica di gestire sistemi su cui è in esecuzione un sistema operativo a 64 bit e in cui sono presenti più di 4 GB di memoria principale, problema ignorato dalle soluzioni esistenti.

Il lavoro di tesi è organizzato come segue: nel capitolo 2 verranno forniti i concetti di base per poter comprendere il lavoro svolto, analizzando le funzionalità messe a disposizione dall'architettura Intel e sfruttate dall'integrity checker. Il capitolo 3 fornisce uno fra i possibili scenari d'uso di YASIC, ne spiega meglio la sua architettura, le assunzioni fatte nello svilupparlo e le sue limitazioni. Nel capitolo 4 viene studiato nel dettaglio il funzionamento di YASIC, suddividendolo nei tre moduli che lo compongono e spiegando in che modo il software realizzato sfrutta l'architettura utilizzata. Per finire, il capitolo 5 conclude questo lavoro di tesi, fornendo degli spunti per delle possibili migliorie all'integrity checker.

Capitolo 2

Concetti preliminari

In questo capitolo vengono presentati i concetti fondamentali utili per la comprensione del lavoro di tesi. Vengono analizzate l'architettura di un PC e le principali funzionalità e caratteristiche delle CPU Intel utilizzate nella realizzazione di YASIC.

2.1 Data Integrity

Verificare l'integrità di dati vuol dire controllare che essi non siano stati modificati rispetto a uno stato conosciuto, considerato *stato base*. L'integrity checker, chiamato anche integrity monitor, è l'entità che si occupa di monitorare lo stato dei dati per controllarne l'integrità e segnalarne, in caso, l'eventuale modifica. Una modifica al dato in questione non significa obbligatoriamente un'alterazione non permessa allo stesso, ma può derivare da una modifica autorizzata e non propriamente gestita dall'integrity monitor. Infatti, è un problema comune per gli integrity monitor la gestione dell'aggiornamento dello stato base.

Non tutti gli integrity monitor funzionano allo stesso modo, in particolare si possono differenziare per frequenza di monitoraggio. È possibile eseguire il monitor:

- *quando deciso dall'utente*: è l'utente a decidere, manualmente, quando eseguire l'integrity monitor e quindi verificare l'integrità dei dati. Questo metodo ha il pregio di consumare poche risorse di sistema, dato che il monitor è per la

maggior parte del tempo in attesa, ma ha il difetto di richiedere, appunto, l'interazione con l'utente. Inoltre, nel caso in cui l'alterazione sia temporanea, è probabile che quest'ultima non venga rilevata, poiché il controllo dei dati avviene in conseguenza della richiesta dell'utente.

- *Real-time*: l'integrity monitor è sempre in esecuzione e controlla continuamente l'integrità dei dati, riuscendo a rilevare alterazioni anche temporanee. Non è richiesta l'interazione con l'utente, ma vengono consumate molte risorse di sistema, dovendo effettuare continuamente il monitoraggio.

- *Periodicamente*: l'integrity monitor viene svegliato a intervalli regolari. Può essere considerato come un buon compromesso tra l'attivazione manuale e l'esecuzione a real-time, dato che il monitor non viene eseguito continuamente, ma solo dopo un intervallo fissato. Il problema di questo metodo è la possibilità di capire, per un malware, l'intervallo con cui viene eseguito il monitor. Infatti, se ciò dovesse avvenire, il malware potrebbe alterare i dati mentre il monitor non è in esecuzione e reimpostarli allo stato base poco prima della successiva esecuzione del monitor, rimanendo, in tale maniera, invisibile all'integrity monitor, come esposto in [27].

- *In istanti casuali*: eseguendo il controllo di integrità in istanti casuali si riesce a evitare che un malware riesca a individuare quando sarà il prossimo controllo. Perciò diventa più complesso, per un malware, rimanere invisibile nascondendo le proprie alterazioni al sistema.

L'integrity monitor, per poter svolgere il proprio compito, deve mantenere un database contenente lo stato base, oppure poter accedere a una copia dello stesso, così da poter confrontare, ogni volta che viene eseguito, lo stato corrente con quello salvato. Nel caso, però, di dati di grandi dimensioni, quali i file o i processi, mantenere lo stato base risulta oneroso da un punto di vista di memoria e, allo stesso modo, lo è eseguire il confronto ogni volta. Per questo, è un metodo comune per gli integrity checker operare su delle *impronte* dei dati da analizzare, piuttosto che sui dati stessi. Tali impronte sono solitamente calcolate attraverso una funzione crittografica di hash, che trasforma

messaggi di lunghezza arbitraria in una stringa di dimensione fissa, che rappresenta l'impronta del messaggio. I dettagli sull'argomento vengono forniti nella sezione 2.5. Sfruttando gli hash è possibile mantenere un database più piccolo e calcolare, di volta in volta, l'hash del dato allo stato corrente, confrontandolo con quello memorizzato.

L'integrity checker può essere usato per controllare l'integrità di diversi tipi di dati, nonostante fino a qualche tempo fa venisse usato prevalentemente per monitorare i file. L'idea alla base di questa scelta è che risulta difficile compromettere un sistema senza alterare in qualche modo i file di applicazioni utente o del sistema operativo. Sebbene questa tecnica possa essere utile per rilevare modifiche non autorizzate al sistema, un malware, o in generale un attaccante, non ha obbligatoriamente bisogno di modificare i file per compromettere un sistema. È possibile, infatti, modificare i processi in esecuzione su una macchina per raggiungere lo stesso scopo. Quindi, sono stati sviluppati integrity monitor in grado di controllare l'integrità di processi, sistemi operativi e hypervisor. YASIC, il software realizzato in questo lavoro di tesi, monitora i processi in esecuzione sul sistema e le librerie utilizzate, verificandone l'integrità.

2.2 Architettura tipica di un PC

In figura 2.1 viene mostrata l'architettura tipica di un PC monoprocessore. In alto si trova la Central Processing Unit (CPU), l'unità centrale di elaborazione che si occupa di eseguire le istruzioni presenti nella Random Access Memory (RAM). La CPU è collegata attraverso il Front Side Bus al Memory Controller Hub (MCH), a cui invia gli indirizzi che deve leggere o scrivere. Sebbene la maggior parte degli indirizzi inviati dalla CPU al MCH siano rediretti verso la RAM, gli indirizzi fisici vengono anche utilizzati per comunicare con i dispositivi del PC, come la scheda grafica, le schede PCI o la memoria flash in cui è contenuto il BIOS. È il Memory Controller Hub a decidere dove redirigere la richiesta di un indirizzo, attraverso una mappa della memoria, che associa ad ogni regione di memoria fisica un dispositivo che la possiede e ne gestisce le richieste.

Una mappa di memoria tipica per i primi 4 GB, in un architettura Intel, è mostrata in figura 2.2, ma gli indirizzi e gli intervalli specifici possono variare in base al chip-

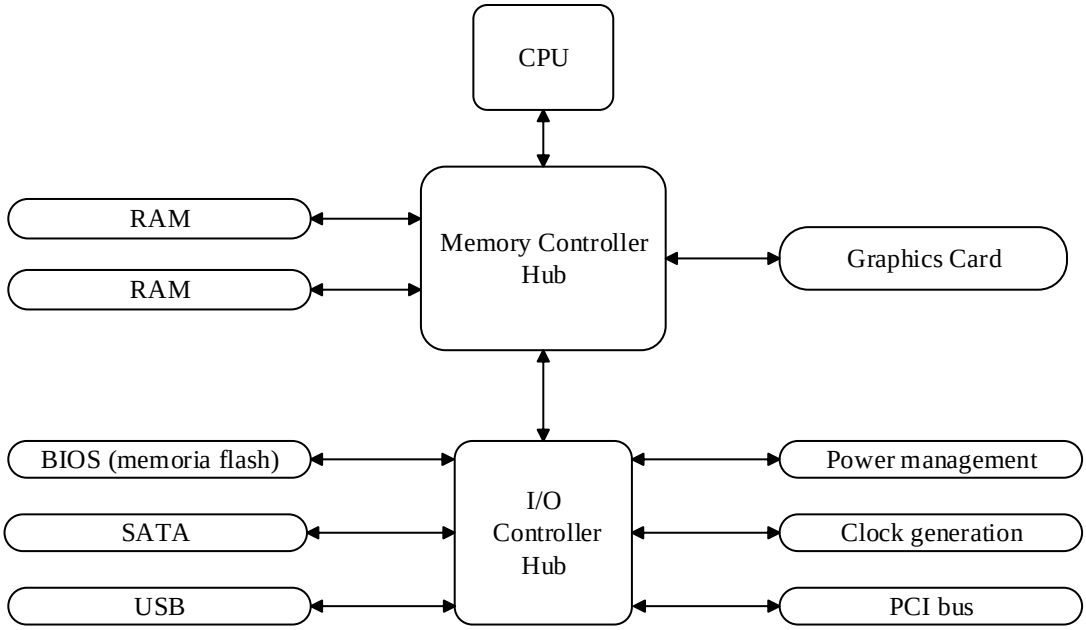


Figura 2.1: Architettura tipica di un PC

set. In particolare, però, possiamo notare che alcuni intervalli di indirizzi di memoria fisica vengono rediretti alla memoria flash, in cui è contenuto il BIOS. Le memorie flash sono particolari tipi di memoria non volatile, spesso utilizzate come Read-Only Memory (ROM), cioè in sola lettura. All'accensione del PC, il chipset è configurato per mappare la memoria flash negli indirizzi alti dei 4 GB e al di sotto dell'indirizzo `0xFFFFF` per ragioni di retro-compatibilità. La CPU preleva la prima istruzione da eseguire all'indirizzo `0xFFFFFFF0`, proprio nell'area mappata all'interno della memoria flash, in quanto il resto della RAM non contiene nessun dato rilevante. In questo modo, il BIOS è il primo codice ad essere eseguito sulla macchina.

Altri intervalli di indirizzi fisici vengono mappati nei registri dei dispositivi e, quindi, il MCH inoltra gli indirizzi appartenenti a queste aree verso l'I/O Controller Hub (ICH), che si occupa di interagire con le periferiche connesse al PC.

Negli ultimi anni Intel ha sviluppato un nuovo tipo di chipset, che sostituisce la classica architettura ad Hub, con Northbridge e Southbridge, a favore del Platform Controller Hub (PCH). In questa architettura, le funzionalità di MCH e ICH vengono riordinate e gran parte delle operazioni del Northbridge vengono spostate all'interno

della CPU, lasciando al PCH il resto delle funzioni.

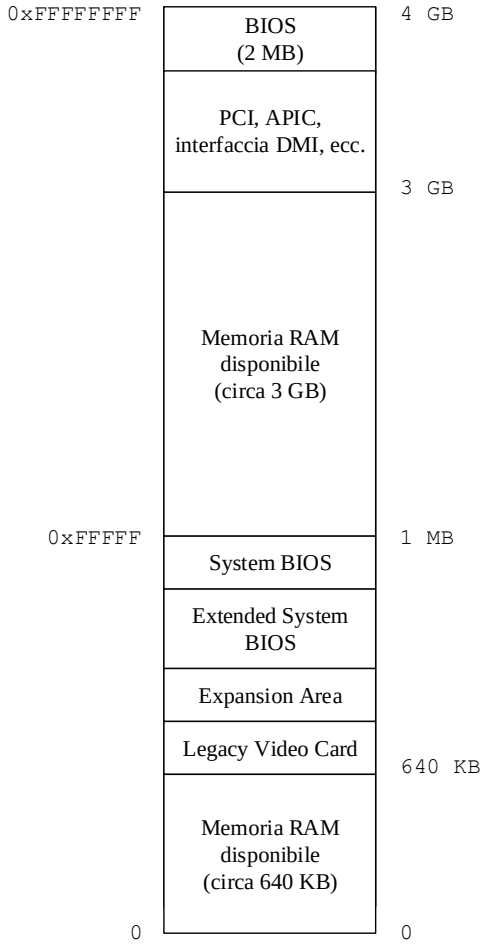


Figura 2.2: Layout della memoria dei primi 4 GB in un'architettura Intel

2.3 Modalità operative dei processori Intel/AMD

In questa sezione vengono analizzate le modalità operative di una CPU x86-64 per studiarne le diverse caratteristiche, soffermandosi, in particolare, su System Management Mode e sulla modalità IA-32e.

La modalità operativa in cui si trova il processore in un dato istante determina l'insieme di istruzioni che il software in esecuzione può utilizzare e le funzioni ar-

chitetturali offerte dal sistema. Un processore x86 supporta tre modalità operative di base:

- *Protected mode*: è la modalità operativa principale del processore, che fornisce grande flessibilità, performance e retro-compatibilità. In questa modalità sono attive diverse funzioni dell'architettura, come la segmentazione e la paginazione, ed è possibile accedere alla memoria fisica fino a 4 GB.

- *Real mode*: implementa l'ambiente esecutivo del processore Intel 8086, con, in aggiunta, la possibilità di abilitare la modalità protetta. All'avvio del processore o in seguito ad un reset, il processore viene posto in questa modalità, in cui può accedere ad un massimo di 1 MB di memoria fisica.

- *System management mode (SMM)*: modalità utilizzata solitamente per implementare funzionalità legate alla piattaforma e dipendenti da questa, come la gestione dell'alimentazione. È stata introdotta a partire dal Intel 386 SL.

Con l'avvento dell'architettura Intel 64, è stata aggiunta la nuova modalità IA-32e, che presenta, a sua volta, due sotto-modalità:

- *Compatibility mode*: permette alla maggior parte del codice scritto a 16 o 32 bit di essere eseguito senza modifiche su un sistema operativo a 64 bit. Un sistema operativo a 64 bit può supportare applicazioni a 64 bit, ma anche applicazioni a 32 bit eseguite in modalità compatibile.

- *64 bit mode*: permette alle applicazioni che fanno uso di questa sotto-modalità di accedere ad uno spazio degli indirizzi lineare a 64 bit. Aumenta il numero di registri generali offerti dall'architettura e estende quelli già presenti a 64 bit, invece che a 32. In questa modalità la dimensione di default di un indirizzo è 64 bit, mentre la dimensione degli operandi di 32.

Nell'utilizzare la memoria, il processore non usa direttamente indirizzi di memoria fisica, ma può adoperare uno fra questi tre modelli:

- *memoria flat*: il programma vede la memoria come un unico e continuo spazio di memoria, chiamato **spazio degli indirizzi lineari**.

- *Memoria segmentata*: la memoria appare al programma come un insieme di spazi degli indirizzi indipendenti, chiamati **segmenti**. Tipicamente questo modello permette di separare codice, dati e stack di un programma in segmenti separati. Gli indirizzi generati dal programma vengono chiamati **indirizzi logici** e consistono di un *selettore di segmento* e un *offset*. Il selettore seleziona uno fra i segmenti a disposizione, mentre l'offset identifica un byte all'interno del segmento.

- *Real-address mode*: è la modalità di memoria utilizzata dal processore Intel 8086 ed è supportata per retro-compatibilità. L'indirizzo fornito dal programma consiste di un selettore ed un offset, analogamente a quanto effettuato per il modello di memoria segmentata. La differenza sta nel calcolo dell'indirizzo lineare che viene ottenuto moltiplicando il selettore per 16 e sommandoci l'offset.

Utilizzando il modello flat o il modello di memoria segmentata è possibile abilitare anche la *paginazione*, che permette di mappare blocchi di indirizzi lineari, chiamati **pagine**, in blocchi di indirizzi fisici, chiamati **frame**. Nel momento in cui si abilita la paginazione viene abilitata la Memory Management Unit (MMU), un componente hardware che si occupa di tradurre gli indirizzi lineari in indirizzi fisici. Nella sezione 2.3.3 viene visto in dettaglio la traduzione degli indirizzi.

2.3.1 System Management Mode

System Management Mode (SMM) è una modalità operativa molto particolare utilizzata solitamente per gestire funzioni di sistema, come il controllo dell'hardware, la gestione dell'alimentazione e funzionalità specifiche definite dai produttori. Il principale vantaggio nell'utilizzo di SMM è che fornisce un ambiente di esecuzione isolato, che opera in maniera trasparente al sistema operativo o a qualunque altro software in esecuzione sul sistema.

Il processore passa in SMM soltanto conseguentemente alla ricezione di un System Management Interrupt (SMI) e può uscirvi esclusivamente attraverso l'esecuzione dell'istruzione `rsm`, un istruzione eseguibile soltanto in SMM. Nel momento in cui viene

ricevuto uno SMI, la CPU finisce di gestire l'istruzione in esecuzione, dopodiché salva lo stato corrente e cambia ambiente di esecuzione, eseguendo poi lo SMI handler, precedentemente caricato dal BIOS. Lo stato del processore viene salvato nella State Save Map, una tabella contenuta in una particolare area di memoria chiamata System Management RAM (SMRAM).

L'isolamento e la trasparenza dell'ambiente di esecuzione in cui viene eseguito lo SMI handler, vengono garantiti attraverso i seguenti meccanismi:

- l'unico modo per entrare in SMM è attraverso la ricezione di uno SMI
- il processore esegue il codice SMM in uno spazio degli indirizzi separato, che può essere reso inaccessibile a qualunque altro software
- durante l'ingresso in SMM, la CPU salva il contesto del sistema operativo o del task precedentemente in esecuzione, permettendo di ripristinarlo successivamente, attraverso l'esecuzione dell'istruzione `rsm`
- tutti gli interrupt sono disabilitati non appena entrati in SMM
- l'istruzione `rsm`, che permette di uscire dalla modalità SMM e ripristinare la precedente modalità, è eseguibile solo in SMM.

Lo SMI può essere generato in qualunque delle modalità operative precedentemente esposte e, quindi, la modalità di System Management può essere raggiunta qualunque sia la modalità precedentemente attiva, come mostrato meglio in figura 2.3.

Non appena entrati in SMM, si ottiene un ambiente di esecuzione molto simile a quello che si ha in Real Mode, attraverso la disabilitazione automatica della paginazione e della modalità protetta. A differenziare SMM da Real Mode è la possibilità di indirizzare la memoria fino a 4 GB, con l'uso di *prefissi* che permettono di modificare la dimensione degli operandi e degli indirizzi delle istruzioni, così da poter accedere alla memoria sopra il primo Megabyte.

SMRAM Entrati in SMM, la paginazione e la modalità protetta vengono disabilitati, permettendo l'accesso ai 4 GB inferiori della memoria fisica. Codice e dati dello SMI

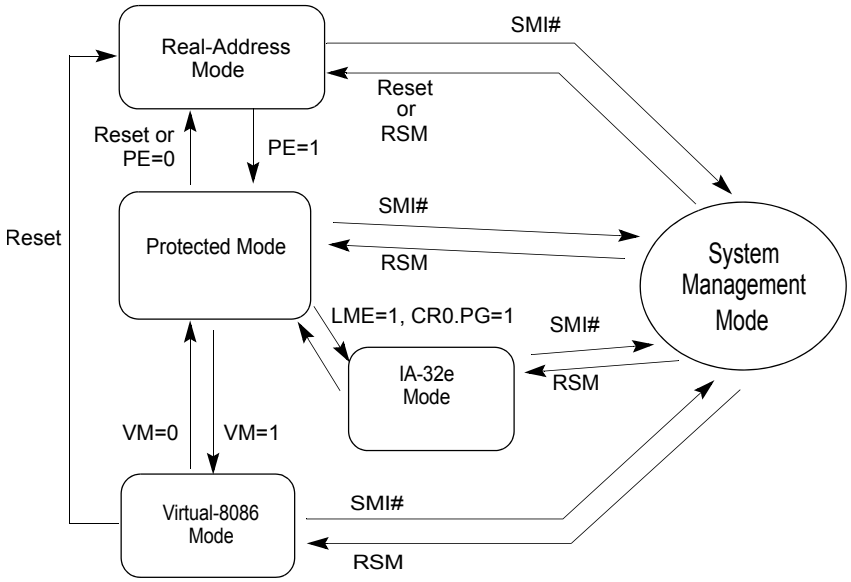


Figura 2.3: Transizioni fra le modalità operative Intel

handler risiedono in un'area particolare di memoria, chiamata System Management RAM (SMRAM), in cui, tra le altre cose, viene memorizzata la State Save Map. Tale area di memoria ha una dimensione che può variare dai 32 KB ai 4 GB, nonostante di default sia di 64 KB, e comincia ad un indirizzo nella memoria fisica che viene chiamato SMBASE e che, all'accensione o al reset della CPU, vale 0x30000. La SMRAM può trovarsi fisicamente nella memoria di sistema oppure in una memoria RAM separata.

La prima istruzione dello SMI handler viene eseguita all'indirizzo SMBASE + 0x8000, mentre la State Save Map viene salvata dall'indirizzo SMBASE + 0xFE00 fino a SMBASE + 0xFFFF. Il valore di SMBASE è contenuto in un registro interno del processore e non ci si può accedere se non attraverso il valore che viene salvato nella State Save Map durante l'ingresso in SMM. Per cambiare SMBASE è sufficiente modificare il valore nella State Save Map, così che il processore aggiorni il proprio registro interno durante l'esecuzione dell'istruzione `rsm`, che ripristina il contesto corrente attraverso l'utilizzo dei valori precedentemente salvati. In questo modo, al successivo SMI verrà utilizzato il nuovo valore di SMBASE. Nonostante la possibilità di rilocare la SMRAM attraverso la configurazione dei registri della CPU, è necessario fare atten-

zione poiché, come precedentemente spiegato, è il Memory Controller Hub a filtrare gli indirizzi fisici a cui la CPU vuole accedere.
La SMRAM si trova di solito:

- all'indirizzo 0xA0000, in quella che viene chiamata "SMRAM compatibile"
- all'indirizzo 0xFEDA0000, nella "High SMRAM"
- in altre locazioni nella "Extended SMRAM", specificate attraverso il segmento di memoria TSEG.

Quindi, è importante capire, prima di rilocare la SMRAM, che la sua protezione è distribuita tra CPU e chipset. Quando, ad esempio, viene utilizzata la SMRAM compatibile e la CPU richiede un indirizzo fisico appartenente a quella regione, il chipset redirige la richiesta verso la SMRAM solo se il processore si trova in SMM, altrimenti la redirige verso la memoria della scheda video. Soltanto la CPU, però, conosce qual è l'attuale valore di SMBASE, mentre il chipset può solo proteggere l'area di memoria in cui "crede" che sia. Fortunatamente, negli ultimi anni è stato introdotto un nuovo registro all'interno dell'architettura Intel, chiamato System Management Range Register, che permette di centralizzare i meccanismi di protezione relativi alla SMRAM, e sono state apportate migliorie alla gestione di questa particolare area di memoria, per evitare attacchi come quelli proposti da Rutkowska in [30], relativi alla cache.

YASIC è stato sviluppato per essere eseguito nella SMRAM compatibile, nell'area di memoria che normalmente viene utilizzata per comunicare con la scheda video. In questo modo si risolvono alcune problematiche relative alla gestione delle cache per la SMRAM.

2.3.2 SMRAM State Save Map

Come accennato precedentemente, la CPU salva il proprio contesto all'interno della SMRAM, in una tabella chiamata State Save Map. A seconda che il processore supporti o meno l'architettura Intel 64, la State Save Map è composta diversamente. Tale tabella permette, al codice eseguito in SMM, di sapere lo stato della macchina prima

della ricezione dello SMI e, entro alcuni limiti, di modificarlo. Non tutti i valori registrati all'interno della State Save Map, infatti, sono accessibili in scrittura. All'interno della tabella vengono salvati quasi tutti i registri del processore, garantendo, in questo modo, il ripristino del contesto una volta usciti da SMM e la trasparenza di questa modalità rispetto a tutte le altre. In particolare, sono memorizzati i valori di CR3 e di GDTR, che vengono utilizzati dal modulo di analisi dello stato di YASIC.

2.3.3 IA-32e

La modalità IA-32e viene introdotta con l'architettura Intel 64 che, oltre a supportare tutte le modalità previste dal IA-32, supporta anche questa nuova modalità. Quest'ultima si suddivide in due sotto-modalità, chiamate modalità compatibile e modalità a 64 bit. La modalità compatibile serve principalmente, come suggerisce il nome, per fornire un ambiente retro-compatibile per il codice a 16 e 32 bit, permettendo la sua esecuzione senza modifiche. Quella a 64 bit, invece, fornisce i vantaggi veri e propri della nuova architettura Intel 64, come uno spazio degli indirizzi lineari a 64 bit e il supporto per uno spazio degli indirizzi fisici più grande di 64 GB. Per determinare se il processore si trova in modalità IA-32e è sufficiente analizzare il bit LMA del registro IA32_EFER. IA32_EFER è un Model Specific Register (MSR), che permette di abilitare o disabilitare funzionalità particolari della CPU, come ad esempio la modalità IA-32e. Prima di provare ad abilitare tale modalità, è necessario essere in Protected Mode, quindi con la segmentazione abilitata. Una volta in modalità IA-32e, gli effetti della segmentazione dipendono dalla sotto-modalità utilizzata: in modalità compatibile la segmentazione viene utilizzata come in Protected Mode, mentre in modalità a 64 bit la segmentazione è quasi completamente disabilitata. Viene imposto dall'architettura, infatti, un modello di memoria flat per lo spazio degli indirizzi lineari a 64 bit, azzerando i registri di segmento CS, DS, ES, SS così che l'indirizzo logico sia uguale a quello lineare. I segmenti vengono comunque utilizzati per gestire la protezione e i privilegi.

La sotto-modalità che è in esecuzione in un dato istante è determinata sulla base dei segmenti. In memoria viene mantenuta una tabella di descrittori di segmenti, chiamata

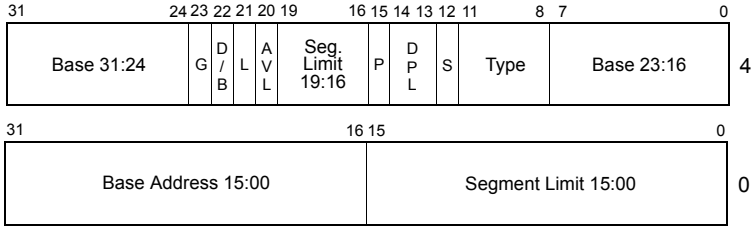


Figura 2.4: Segment descriptor

Global Descriptor Table (GDT), in cui, per ogni segmento vengono indicati l'indirizzo base, la dimensione, i privilegi di accesso e poche altre informazioni. In particolare, come evidenziato in figura 2.4, è possibile notare i bit D/B ed L, che indicano rispettivamente la dimensione di default degli operandi e l'abilitazione della modalità IA-32e per il segmento. Se L è uguale a 1 e D/B è azzerato, il segmento viene eseguito in modalità a 64 bit, mentre se L è uguale a 0, e la modalità IA-32e è attiva, il segmento è eseguito in modalità compatibile.

Paginazione La paginazione è un meccanismo che converte indirizzi lineari in indirizzi fisici, determinando quali sono permessi e qual è la modalità di cache utilizzata. Permette al sistema operativo di creare uno spazio degli indirizzi separato per ogni processo, conosciuto come "spazio degli indirizzi virtuale". In questo meccanismo, lo spazio degli indirizzi lineari e quello degli indirizzi fisici vengono divisi in blocchi di uguale dimensione, chiamati rispettivamente *pagine* e *frame*. La traduzione dall'indirizzo virtuale a quello fisico avviene tramite un insieme di *tabelle delle pagine*, che permettono di mappare pagine in frame.

Utilizzando il paging, è possibile proteggere i processi l'uno dall'altro, dato che essi non possono accedere direttamente alla memoria fisica, ma sono costretti a sottostare alle traduzioni imposte dal sistema. Infatti, una volta abilitata la paginazione, tutti gli indirizzi lineari sono trattati come indirizzi virtuali che vengono convertiti dalla Memory Management Unit (MMU) in indirizzi fisici.

La traduzione effettuata dalla MMU avviene tramite un insieme gerarchico di tabelle. Ogni tabella contiene delle entry che puntano alla tabella di livello inferiore, che a sua volta può contenere centinaia di entry che puntano alla tabella gerarchicamente

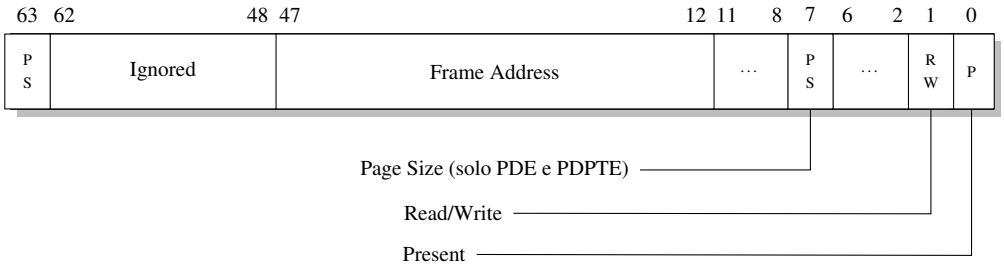


Figura 2.5: Caratteristiche principali di una entry della tabella delle pagine

più bassa e così via, fino ad arrivare all'indirizzo fisico convertito. L'indirizzo della prima tabella della gerarchia è contenuto nel registro CR3 del processore.

In modalità IA-32e, la dimensione dei frame e delle pagine può essere: di 4 KB, di 2 MB o 1 GB, nonostante quest'ultima dimensione non sia supportata da tutti i processori. Nella figura 2.6 viene mostrata la traduzione di un indirizzo virtuale con pagine di 4 KB, mentre in figura 2.5 sono mostrati i campi essenziali di una entry delle tabelle. Per ogni pagina è possibile specificare se è accessibile in scrittura, da *user mode* e, inoltre, se è presente o meno nella memoria RAM, attraverso il bit P. Quest'ultimo bit permette di capire se una pagina è ancora in memoria oppure se è stata spostata, ad esempio per liberare la memoria. Ogni struttura di paging è grande 4 KB e contiene 512 entry, nella modalità IA-32e. Le entry sono selezionate, all'interno di ogni tabella, usando come indice parte dell'indirizzo virtuale e ognuna contiene l'indirizzo fisico del frame in cui trovare la tabella di paging di livello inferiore.

Nel dettaglio, la traduzione di un indirizzo virtuale, in modalità a 64 bit con pagine di dimensione 4 KB, avviene come segue:

1. attraverso l'indirizzo fisico contenuto nel registro CR3 viene individuata la Page Map Level 4 (PML4)

2. i bit dal 47 al 39 dell'indirizzo virtuale vengono utilizzati come indice all'interno della tabella, selezionando una delle entry

3. si accede alla Page Directory Pointer Table (PDPT), grazie all'indirizzo fisico contenuto nella entry selezionata al passo precedente, e si utilizzano i bit dal 38 al 30 come indice all'interno della tabella

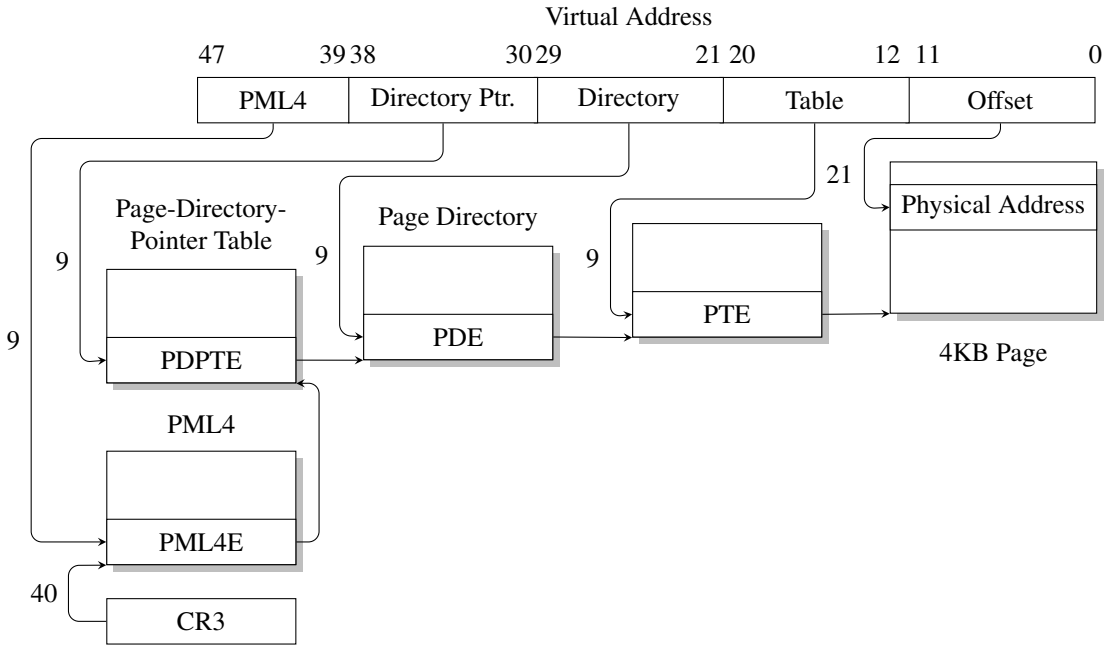


Figura 2.6: Meccanismo di traduzione degli indirizzi in modalità IA-32e e pagine da 4 KB

4. si ottiene l'indirizzo fisico della Page Directory, attraverso la entry precedente, e si seleziona una entry della tabella attraverso i bit dal 29 al 21 dell'indirizzo virtuale

5. trovata la Page Table si accede a una entry con i bit dal 20 al 12 dell'indirizzo virtuale e si ottiene, quindi, l'indirizzo fisico del frame in cui è mappata la pagina a cui appartiene l'indirizzo virtuale

6. i bit dal 11 allo 0 dell'indirizzo virtuale richiesto vengono utilizzati come offset all'interno del frame, indirizzando il dato vero e proprio.

Nel caso di pagine di dimensione 2 MB, il bit Page Size (PS) della Page Directory Entry è impostato a 1 e l'indirizzo fisico ottenuto è quello del frame. In questo caso, i restanti bit, dal 20 allo 0, sono utilizzati come offset all'interno del frame.

2.4 Coreboot e il BIOS

Il Basic Input/Output System, chiamato solitamente BIOS, è un software che ha il compito di inizializzare e controllare i componenti hardware del sistema e, successivamente, cedere il controllo al *boot loader*. Quest'ultimo è un programma che si incarica di caricare in memoria il *kernel* del sistema operativo e lasciare il controllo ad esso.

La prima operazione eseguita dal BIOS è il Power On Self-Test (POST), cioè l'insieme di operazioni che identificano e inizializzano i vari dispositivi connessi al sistema, come la quantità di RAM, la scheda video, la Network Interface Card (NIC) e le periferiche esterne. In seguito, analizza i supporti come l'hard disk, il CD e l'USB alla ricerca del boot loader da mandare in esecuzione. Quest'ultimo cercherà e caricherà in memoria il sistema operativo. Tutto il processo viene chiamato *bootstrap*.

Il BIOS è memorizzato in una particolare memoria, chiamata Read Only Memory (ROM), che viene mappata nella memoria RAM dal chipset, già dall'accensione del PC. Nonostante il nome, queste memorie sono ormai riscrivibili elettronicamente, attraverso quello che viene chiamato *flashing*, per permettere, ad esempio, l'aggiornamento del BIOS.

Tra gli altri compiti del BIOS, c'è anche quello di inizializzare la SMRAM e adoperare le dovute precauzioni per renderla inaccessibile a qualunque software all'infuori di quello eseguito in SMM. In particolare, il BIOS dovrà configurare il registro System Management RAM Control (SMRAMC) del Memory Controller Hub e abilitare il bit D_LCK, che rende la SMRAM inaccessibile e, soprattutto, blocca qualsiasi scrittura sui bit di configurazione del chipset riguardanti la SMRAM. Il bit D_LCK è scrivibile una volta sola e viene azzerato solo in seguito ad un reset della macchina, garantendo così un meccanismo per l'integrità e la protezione della SMRAM.

Coreboot è un software open source che cerca di sostituire i BIOS proprietari. Dovendo configurare l'hardware, il codice è fortemente dipendente dalla piattaforma e al giorno d'oggi Coreboot supporta più di 230 schede madri differenti. È ideale per sistemi embedded, server e normali PC, visto il suo sviluppo attento e modulare, che permette di includere solo le funzionalità realmente necessarie, riducendo la quantità di codice eseguita e lo spazio richiesto nella memoria ROM. La prima operazione

svolta da Coreboot è l'inizializzazione dell'hardware, dopodiché viene eseguita la logica di boot, racchiusa all'interno del *payload*. I tipi di payload a disposizione sono differenti. È possibile: eseguire direttamente il kernel Linux, eseguire un boot loader o una applicazione oppure utilizzare SeaBIOS. Quest'ultimo è un'implementazione dei BIOS tradizionali ed è la base di YASIC. La maggior parte del codice di Coreboot e SeaBIOS è scritto in linguaggio C, a eccezione di poche istruzioni eseguite all'avvio e di alcune interfacce fornite da SeaBIOS per compatibilità con i BIOS tradizionali.

YASIC è parte integrante del codice di SeaBIOS ed è da esso inizializzato durante la fase di boot. È suo compito, per esempio, inizializzare la SMRAM e installarne il software all'interno.

2.5 SHA-1 e accenni crittografici

Una funzione hash crittografica è una funzione che prende in input una stringa di lunghezza arbitraria e ne restituisce una di lunghezza fissa. La funzione hash crittografica ideale h dovrebbe soddisfare le seguenti caratteristiche:

- dato un messaggio m, l'hash $h(m)$ può essere calcolato molto velocemente
- dato un hash y, è computazionalmente impossibile trovare un messaggio m tale che $h(m)$ sia uguale a y
- è computazionalmente impossibile trovare due messaggi m_1 e m_2 con $h(m_1)$ uguale a $h(m_2)$

Per le caratteristiche appena elencate, una funzione hash crittografica è indicata per calcolare l'impronta di un oggetto su cui fare integrity. Dato che, però, l'insieme dei possibili messaggi in input è più grande delle stringhe che si possono ottenere con un numero finito di caratteri, è naturale che ad alcuni hash corrispondano più messaggi. Quando due o più messaggi generano lo stesso hash si parla di *collisione* e la difficoltà con cui si riescono a trovare due messaggi con lo stesso hash è un indice di qualità per la funzione hash in esame. Le funzioni hash utilizzate principalmente sono MD5

e SHA, nonostante la funzione MD5 presenti ormai diversi attacchi che permettono di trovare collisioni in poco tempo.

SHA-1 produce un hash di 160 bit, attraverso una procedura iterativa. Il messaggio *m* in input viene suddiviso in blocchi di uguale lunghezza, che sono processati attraverso una sequenza di *round*, i quali applicano una funzione di compressione *h'*, che combina il blocco corrente con il risultato dei round precedenti. In questo modo, partendo da un valore iniziale X_0 e calcolando $X_j = h'(X_{j-1}, m_j)$ si ottiene l'hash del messaggio all'ultimo passaggio.

YASIC utilizza SHA-1 come impronta dei dati che deve controllare, così da avere un buon compromesso tra efficienza e robustezza.

2.6 User Datagram Protocol

Lo User Datagram Protocol o UDP è uno dei principali protocolli di rete utilizzati in Internet. Con UDP le applicazioni possono inviare messaggi, chiamati *datagrammi*, attraverso la rete ad altri host sulla rete IP, senza bisogno di stabilire un canale di comunicazione. Difatti, è definito come un protocollo *connectionless* ed è solitamente considerato meno affidabile di altri protocolli come TCP che, al contrario di UDP, gestiscono la ritrasmissione di pacchetti persi o errati e il loro riordino. Nonostante queste lacune, UDP è ideale per applicazioni in cui è fondamentale la velocità di trasferimento e per cui la perdita di qualche pacchetto non compromette l'utilizzo dell'applicazione stessa.

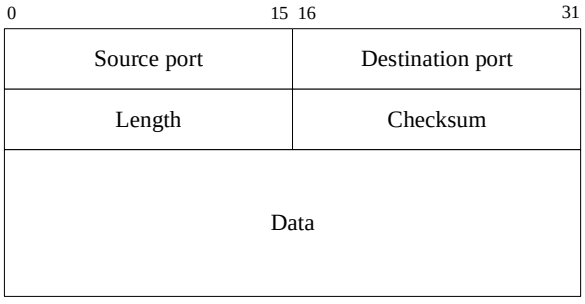


Figura 2.7: Header UDP

YASIC utilizza UDP per la sua semplicità di implementazione, dato che, come appena spiegato, il protocollo non deve garantire funzionalità particolari e anche gli header dei pacchetti sono elementari, come mostrato in figura 2.7.

Capitolo 3

Scenario d'utilizzo

In questo capitolo viene fornita un'idea dei possibili scenari di utilizzo di YASIC e della sua architettura. In particolare, nella prima sezione vengono analizzati i diversi modi in cui può agire un integrity checker, dopodiché si spiegano i vantaggi e gli svantaggi di usare System Management Mode per sviluppare un tale software. Nella terza sezione è esposta l'architettura di base mentre, nelle ultime due sezioni, vengono fatte alcune assunzioni e spiegate le limitazioni di YASIC.

3.1 Integrity checking

Un dato si considera integro in un certo istante quando non è stato alterato rispetto all'ultima modifica autorizzata. Controllare l'integrità di un dato vuol dire quindi verificare che questo sia integro. Solitamente, questa operazione viene eseguita confrontando delle *impronte* dei dati, piuttosto che i dati stessi. Le impronte possono essere calcolate attraverso una funzione hash, cioè una funzione che ha come input una stringa di bit di lunghezza arbitraria e come output una stringa di caratteri a lunghezza fissa. La caratteristica principale di queste funzioni è che anche la modifica di un solo bit nella stringa d'ingresso produce un output molto differente. Per i dettagli riguardo le funzioni hash e il controllo d'integrità si rimanda, rispettivamente, alla sezioni 2.5 e 2.1.

YASIC è un integrity checker eseguito periodicamente che controlla l'integrità di processi e/o librerie eseguite su un sistema. Verificare che un processo sia integro vuol dire controllare che venga eseguito effettivamente il processo atteso, senza alterazioni non previste. È infatti un metodo diffuso, per il codice malevolo, quello di compromettere un sistema iniettandosi nello spazio degli indirizzi di un altro processo [23]. È possibile suddividere gli integrity monitor principalmente in due categorie, a seconda del tipo di rilevazione che effettuano: quelli statici e quelli dinamici [8].

Un integrity monitor statico focalizza la propria analisi sui file binari di un programma, analizzando il codice e il file come sono memorizzati nel sistema. Tale tecnica permette di rilevare alterazioni eseguite sui file del sistema, atte ad introdurre, rimuovere o comunque modificare codice o dati in un file binario di un programma. Ad esempio, è una tecnica diffusa fra i primi rootkit - strumenti il cui scopo è quello di nascondere la presenza di attività maligne su un sistema - sostituire i programmi comunemente utilizzati per visualizzare lo stato della macchina, così da nascondere la presenza di backdoor o processi relativi a malware [17]. Sebbene l'analisi statica possa rilevare questo tipo di modifiche, non permette di controllare l'integrità dei processi una volta che questi sono in esecuzione.

Un integrity monitor di tipo dinamico, invece, ha lo scopo di verificare l'integrità del codice di un processo mentre è in esecuzione. Per farlo deve analizzare la memoria principale del sistema, dove i processi risiedono, piuttosto che i file. Inoltre questa tipologia include gli integrity monitor che verificano l'integrità di altre parti del sistema, come il sistema operativo o un possibile hypervisor.

Il software realizzato in questo lavoro di tesi è di tipo dinamico e, infatti, analizza il contenuto della memoria principale per verificare che i processi rimangano integri durante la loro esecuzione. In questo modo, controlla che non venga iniettato codice malevolo nella memoria di un processo o che, comunque, questa non venga alterata.

3.2 Modalità operativa

L'architettura x86 è un'architettura basata su una protezione fatta a livelli gerarchici, comunemente chiamati *ring*. Ogni ring ha un privilegio differente sul sistema, dipen-

dente dal livello di accesso alle risorse. In un tipico sistema moderno il kernel lavora a ring 0, il livello più privilegiato nelle prime architetture x86, mentre i processi utente operano a ring 3, il livello meno privilegiato. Inoltre, a partire dal processore Intel 386SL, è stata introdotta una nuova modalità operativa chiamata System Management Mode (SMM) [13], che detiene il livello di privilegio massimo sulla macchina. Tale modalità viene chiamata da alcuni "ring -2" ed è stata successivamente copiata da AMD ed inserita nella propria architettura. Nelle recenti CPU Intel e AMD è stato introdotto un ulteriore livello di protezione, chiamato ring -1, con un privilegio maggiore del sistema operativo, ma comunque inferiore rispetto a SMM. Lo scopo di questa nuova modalità, chiamata anche *hypervisor mode*, è quello di fornire supporto hardware per la virtualizzazione. Affinché i vari ring possano comunicare, esistono particolari interfacce che permettono ad un ring più esterno di accedere, in una maniera predefinita, alle risorse di un ring più interno, evitando di fornire ai livelli meno privilegiati l'accesso diretto e incontrollato.

Per un malware è importante compromettere un sistema al livello di privilegio maggiore, poiché questo permette di accedere a più risorse, senza dover sottostare ai metodi di accesso imposti dai ring più interni. Visto nell'ottica di un attaccante, compromettere un sistema a ring 3, per esempio, vorrebbe dire dover fare continuamente affidamento sul sistema operativo per svolgere ogni operazione che coinvolga l'hardware. Un rilevatore interno al sistema operativo, se individua l'attività maligna, può facilmente inibire le operazioni richieste e rendere innocuo l'attaccante, poiché il sistema operativo ha un privilegio maggiore sull'intero sistema di quanto non lo abbia un processo nello spazio utente. Se per un attaccante è importante ottenere l'accesso ai ring più interni per evitare di essere rilevato, lo stesso vale per un integrity checker, poiché rende più efficace e affidabile la sua rilevazione. Infatti, meno privilegi ha il software, più possibilità ci sono che processi, sistema operativo o hypervisor compromessi possano nascondere o alterare il risultato del programma.

Proprio per i motivi appena illustrati, YASIC è stato pensato per essere eseguito in SMM. Il beneficio principale di usare SMM è che offre un ambiente d'esecuzione isolato dal resto del sistema e trasparente, sia alle varie applicazioni, che al sistema operativo e all'hypervisor. Il codice eseguito in SMM ha il controllo completo sulla

macchina e non ha bisogno di interfacciarsi con nessun altro ring per comunicare con l'hardware. Di conseguenza risulta il luogo ideale in cui eseguire un integrity monitor, poiché viene eseguito con il livello di privilegio più alto possibile su un architettura x86 e si fa affidamento direttamente sull'hardware per ogni operazione, senza sfruttare altro codice che potrebbe essere stato compromesso. Inoltre, se l'hardware non è stato manomesso (e il codice dell'integrity monitor non contiene bug), si può considerare affidabile l'output del software, proprio perché basato solo sull'uso diretto dell'hardware.

Sebbene lavorare in SMM porti diversi vantaggi, introduce anche un problema: la perdita della semantica delle informazioni. Infatti, lavorando direttamente con l'hardware, non si ha nessuna informazione sul significato del contenuto della memoria principale ed è compito del software ricostruire la semantica per poter poi fare il controllo d'integrità. Tale problema è noto in letteratura come *"semantic gap"* [9].

3.3 Architettura

YASIC è un software che controlla l'integrità di processi e librerie di un sistema ed è eseguito nella modalità di System Management. Il software realizzato deve essere installato sul sistema che si intende controllare, quello che d'ora in poi sarà chiamato *"sistema target"*. YASIC si può suddividere principalmente in tre moduli:

- *modulo di acquisizione dello stato*: è la parte più corposa del sistema sviluppato, che ricostruisce la semantica associata alle informazioni contenute nella memoria principale. In seguito, analizza le strutture dati individuate per trovare i processi e le librerie da analizzare.

- *Modulo di comunicazione*: si occupa di comunicare con la macchina *monitor*, un sistema considerato sicuro e collegato attraverso la rete alla macchina target. Praticamente, il modulo è un driver per la Network Interface Card (NIC) che si occupa di costruire i pacchetti dati da inviare al monitor.

- *Modulo di analisi*: risiede nella macchina monitor e riceve i dati inviati dal modulo di comunicazione, eseguito sul target. Ricevute le informazioni relative ai

processi, le confronta con quelle in suo possesso per verificare l'integrità degli oggetti in analisi.

L'integrity checker è stato testato attraverso QEMU [3], un software open-source in grado di emulare diverse architetture tra cui x86, AMD64 e ARM. Il sistema emula un architettura Intel con CPU Intel64 single core, e utilizza una versione di Coreboot [1] modificata per installare l'integrity checker sviluppato, piuttosto che un comune SMI handler. Tra i vari dispositivi emulati da QEMU, si è utilizzata la NIC RTL8139 per collegare la macchina emulata alla rete, così da poter comunicare con il sistema monitor.

3.4 Assunzioni

In questo e nei prossimi capitoli vengono fatte alcune assunzioni sullo stato del sistema e sui possibili attacchi che non verranno presi in considerazione.

Innanzitutto si suppone di avere un BIOS aggiornato, che ponga le dovute protezioni alla SMRAM, un'area di memoria protetta, normalmente accessibile solamente quando il processore si trova in SMM. In particolare, è necessario che il BIOS imposti il bit D_LCK, che si trova nel SMRAM Control Register, un registro del Memory Controller Hub. Se così non dovesse essere, qualunque software con la possibilità di fare I/O potrebbe accedere alla SMRAM, anche quando il processore non si trova in SMM, e modificare facilmente il codice dell'integrity monitor, come dimostrato in [11].

Come seconda assunzione si suppone che non vengano effettuati attacchi fisici alle macchine target e monitor, come anche il semplice spegnimento dei sistemi, e che la macchina monitor non sia compromessa.

Per finire, si assume non sia installata nel sistema target una NIC manomessa, che potrebbe alterare o non inviare i dati. Attacchi di questo genere non sarebbero rilevabili dall'integrity-checker, che è sviluppato basandosi proprio sull'hardware e facendo completo affidamento solo su questo.

3.5 Limitazioni

YASIC controlla l'integrità di processi e librerie. Più precisamente il modulo di acquisizione dello stato recupera i processi che devono essere verificati e ne analizza lo spazio di memoria alla ricerca delle sezioni di codice e delle sezioni statiche. È solo di queste sezioni che il software verifica l'integrità, senza considerare le sezioni dati o dinamiche in cui non vengono rilevate alterazioni di alcun genere. Quest'ultime, evolvendo assieme al processo, possono contenere ad ogni esecuzione, e in ogni istante, dati differenti di cui non si può controllare l'integrità, per il semplice fatto che non si hanno dei dei dati di base con cui eseguire il confronto. Per controllare che tali aree di memoria vengano usate correttamente esistono tecniche differenti, che non verranno spiegate in questa sede poiché si discostano dallo scopo di questo lavoro di tesi.

Pertanto YASIC è in grado di rilevare se un malware o un attaccante modificano in qualche modo il codice dei processi o ne iniettano nel loro spazio di memoria, ma non è in grado di rilevare altri tipi di minaccia, quali ad esempio *buffer overflow* o *format string* per i quali è possibile sfruttare altri tool.

Capitolo 4

Sviluppo e implementazione

In questo capitolo vengono studiati in dettaglio il funzionamento e l'implementazione di YASIC. Nella prima parte si analizzano il modulo di acquisizione dello stato e le configurazioni utilizzate durante lo sviluppo del software, nella seconda il modulo di comunicazione e nella terza quello di analisi. Per finire, vengono introdotti due problemi, e le relative soluzioni, in questo tipo di integrity checker sviluppati in SMM: come generare lo SMI per attivare il software e come stabilire una comunicazione sicura tra le parti.

4.1 Introduzione

YASIC è un integrity checker hardware-assisted che controlla l'integrità di applicazioni utente e librerie di sistema. È eseguito sul sistema target con il compito di analizzarne lo stato, trovare i dati che devono essere verificati e comunicarli al "sistema monitor". Il monitor deve controllare che i dati inviati dal target appartengano effettivamente a processi integri.

Il software realizzato viene eseguito a intervalli regolari, attraverso la generazione periodica di uno SMI, il quale permette la transizione del processore dalla modalità operativa in cui si trova, alla modalità di System Management, dove viene eseguito l'integrity checker. Una volta cambiata modalità d'esecuzione, viene eseguito YASIC, che ricostruisce la semantica dei dati presenti nella RAM e analizza i processi in esecu-

zione. Trovati i processi da controllare, accede al loro spazio di memoria per ottenere le sezioni statiche al loro interno e recuperare il codice delle librerie utilizzate. Infine, viene calcolato un hash delle diverse sezioni e vengono creati dei pacchetti UDP da inviare al monitor, contenenti le informazioni dei processi. Il monitor, ricevuti i pacchetti, confronta gli hash calcolati dal target con quelli in suo possesso, facilmente calcolabili attraverso l'immagine in memoria degli stessi processi analizzati. Se qualcuno di questi hash dovesse differire da quello atteso, il monitor può segnalare l'alterazione della macchina target.

Per poter realizzare questo sistema di controllo di integrità, attraverso l'uso di SMM, è stato necessario risolvere alcuni problemi riguardanti in particolare l'acquisizione dello stato della macchina e il superamento del cosiddetto *"semantic gap"*.

4.2 Modulo di acquisizione dello stato

In questa sezione viene spiegato nel dettaglio il modulo di acquisizione dello stato, utile a recuperare la semantica dei dati, non visibile da SMM. In particolare, vengono prima studiate le configurazioni necessarie per poter analizzare il sistema e sviluppare efficacemente il software, e successivamente in che modo sono effettivamente recuperate le informazioni riguardo i processi.

4.2.1 Impostazione ambiente di base

Quando una CPU x86-64 entra in SMM, in seguito alla ricezione di uno SMI, la traduzione degli indirizzi da virtuali a fisici viene disabilitata e l'indirizzamento diventa simile a quello che si ha in Real Mode, qualunque sia la modalità precedentemente utilizzata. A differenziare la modalità Real Mode da SMM sono i limiti di segmento che in SMM sono di 4 GB, piuttosto che di 64 KB. Per i dettagli su come sia possibile indirizzare fino a 4 GB System Management Mode, si rimanda alla sezione 2.3.1.

Per poter realizzare una soluzione di integrity checking che sia in grado di gestire i processi e le librerie eseguiti su sistemi operativi moderni a 64 bit, è necessario abilitare la modalità IA-32e non appena viene eseguito lo SMI handler. Così facendo, YASIC

è in grado di accedere alla memoria fisica oltre i 4 GB, limite ormai superato dalla maggior parte dei computer utilizzati oggi, oltre che a registri altrimenti inaccessibili.

Abilitare la modalità IA-32e richiede l'inizializzazione di alcune strutture dati e che la CPU sia già in Protected Mode con la paginazione abilitata. YASIC deve quindi, per prima cosa, occuparsi di attivare la modalità protetta. Per farlo, è sufficiente abilitare il bit PE del registro CR0, dopo aver costruito alcune strutture dati di partenza, quali:

- una Interrupt Descriptor Table (IDT), per indirizzare i diversi handler di interrupt mentre il processore si trova in modalità protetta;
- una Global Descriptor Table (GDT), che deve contenere un descrittore di segmento nullo e almeno un descrittore di segmento codice e uno di segmento dati, da utilizzare come stack.

Sebbene normalmente richiesto, non è necessario configurare una IDT nel caso in analisi, perché, non appena la CPU entra in SMM, gli interrupt vengono disabilitati. Non essendo prevista la gestione degli interrupt da parte di YASIC, la IDT viene ignorata sia durante l'attivazione della modalità protetta, sia durante l'attivazione della modalità IA-32e.

Successivamente, è possibile inizializzare IA-32e come previsto dal manuale Intel [15]. Nella pratica ciò che va fatto è:

1. disabilitare la paginazione azzerando il bit PG del registro CR0.
2. Abilitare l'estensione PAE attraverso l'attivazione del bit CR4.PAE, così da poter supportare indirizzi fisici di memoria più grandi di 32 bit.
3. Caricare in CR3 l'indirizzo fisico di una tabella di quarto livello, chiamata anche Level-4 Page Map Table (PML4), che sarà necessaria per mappare indirizzi virtuali in indirizzi fisici, quando la paginazione sarà riattivata. Affinché il sistema sia funzionante, è fondamentale che la PML4 sia inizializzata opportunamente. In particolare bisogna utilizzare il cosiddetto "mapping identità", che permette di avere indirizzi virtuali e fisici uguali, per l'area di memoria relativa alla SMRAM. In questo modo, appena sarà abilitata la paginazione, il processore potrà

continuare ad eseguire le istruzioni dello SMI handler, dato che gli indirizzi virtuali saranno mappati negli stessi indirizzi fisici.

4. Abilitare IA-32e impostando IA32_EFER.LME a 1. IA32_EFER è uno dei Machine Specific Registers (MSR), che sono un insieme di registri che permettono di controllare le performance e le funzionalità specifiche dei vari tipi di processore. IA32_EFER gestisce le funzionalità relative alla modalità IA-32e.

5. Abilitare la paginazione impostando CR0.PG a 1.

Eseguita l'ultima operazione, la CPU effettua dei controlli di consistenza per verificare che i diversi passaggi siano stati eseguiti correttamente, dopodiché entra a tutti gli effetti in modalità IA-32e. Per evitare che il sistema generi un'eccezione, è fondamentale impostare correttamente il mapping identità, che permette di accedere ad un indirizzo valido subito dopo l'attivazione della paginazione. Infatti, per recuperare l'istruzione successiva da eseguire, la CPU esegue una somma tra l'indirizzo dell'istruzione che ha eseguito e la dimensione della stessa. Appena abilitata la paginazione, però, l'indirizzo ottenuto non è più un indirizzo fisico, ma uno virtuale che, quindi, deve essere tradotto dalla MMU. Grazie al mapping identità tale indirizzo sarà uguale a quello fisico, permettendo al software di continuare la sua esecuzione.

In IA-32e esistono due sotto-modalità: quella compatibile e quella a 64 bit. Quale delle due sotto-modalità è attiva dipende dai descrittori di segmento presenti nella GDT, in cui sono presenti due bit che permettono di scegliere una modalità piuttosto che l'altra, come spiegato meglio nella sezione 2.3.3. Per questo è necessario aggiornare la Global Descriptor Table così da configurare la modalità a 64 bit, che sarà utilizzata da YASIC. Aggiornata la GDT, prima di poter eseguire il controllo d'integrità vero e proprio, è necessario eseguire l'istruzione *far jmp*, un istruzione dell'architettura x86/x64 che trasferisce incondizionatamente il controllo a un nuovo indirizzo, forzando l'aggiornamento del registro di segmento codice, necessario per abilitare effettivamente la sotto-modalità a 64 bit.

4.2.2 Configurazione tool per utilizzare il linguaggio C

Quanto spiegato nelle sezioni precedenti e nelle successive è stato implementato sfruttando sia il linguaggio assembly che il linguaggio C. Nonostante inizializzare la modalità IA-32e sia un'operazione piuttosto semplice, anche se scritta interamente in linguaggio assembly, si è preferito impostare fin da subito un ambiente adeguato per poter successivamente scrivere le diverse funzioni di YASIC attraverso l'uso del linguaggio C. I vantaggi di questa scelta sono, ovviamente, la rapidità di sviluppo e la maggiore leggibilità del codice sorgente, oltre che una fase di testing e debugging più facile.

Prima di utilizzare il suddetto linguaggio, è stata abilitata la modalità protetta ed è stato impostato uno *stack*, un area di memoria leggibile e scrivibile, necessario per le chiamate a funzioni all'interno del codice sorgente scritto in C e per memorizzare dati che non possono essere salvati nei registri della CPU. Utilizzare il linguaggio C ha introdotto alcuni problemi relativi, in particolare, alla corretta configurazione dei tool usati durante lo sviluppo.

Per comprendere meglio i problemi affrontati, è necessario capire come avviene lo sviluppo di un BIOS e in particolare di Coreboot, il cui codice è stato utilizzato come base per la creazione di YASIC. All'accensione di un PC, il primo software a venire eseguito è quello posizionato all'interno di una particolare memoria, chiamata Read-Only Memory (ROM), in cui è contenuto il BIOS. Il BIOS memorizzato nella ROM contiene codice e dati, in un formato assoluto, per poter eseguire i vari compiti del BIOS e cedere, successivamente, il controllo al boot loader. Il risultato finale del processo di sviluppo di Coreboot, quindi, non è altro che un unico file che andrà memorizzato nella ROM.

Nonostante ciò, sviluppare il codice del BIOS interamente in assembly, sebbene possibile, porta ad uno sviluppo più lento, difficile e meno modulare, caratteristica che, invece, risulta principale in Coreboot. Di conseguenza, il codice sorgente è scritto per la maggior parte in linguaggio C e compilato per produrre un file "raw", cioè senza nessun formato eseguibile, che andrà copiato direttamente nella memoria ROM e deve, quindi, rispettare alcuni vincoli imposti dall'architettura.

Innanzitutto, nel generare il file raw bisogna tenere conto che la ROM viene mappata negli indirizzi più alti della memoria fisica. L'intervallo di indirizzi che va da `0xFFF80000` a `0xFFFFFFFF` è sempre mappato nella ROM, mentre altre sezioni della ROM, nel caso questa sia più grande di 512 KB, possono essere abilitate successivamente attraverso la configurazione del chipset [14]. Inoltre, si deve considerare che la prima istruzione a essere eseguita da una CPU Intel deve trovarsi all'indirizzo fisico `0xFFFFFFF0`, quindi mappato all'interno della memoria ROM [15].

Affinché questi ed altri vincoli vengano rispettati, è necessario istruire specificatamente il compilatore e il linker, cosicché l'output finale del processo di compilazione e linking abbia codice e dati posizionati correttamente. Quasi ogni programma, infatti, è composto da più file sorgente che vengono compilati separatamente dal *compilatore*, il cui scopo è quello di tradurre un linguaggio ad alto livello in codice macchina. Il risultato della compilazione è un *relocatable object file* ed è suddiviso in diverse *sezioni* che possono contenere diversi tipi di dati o codice. Visto che il compilatore traduce i file singolarmente, non è in grado di risolvere i riferimenti a funzioni o variabili definiti in altri file. Per questo viene utilizzato il *linker*, che prende in input i diversi *object file* generati dal compilatore e crea un unico file eseguibile. Il linker ha il compito di rilocare le diverse sezioni contenute nei codici oggetto, sostituendo i simboli utilizzati con gli indirizzi di memoria adeguati, per riflettere i cambiamenti dovuti alla rilocazione. Il file eseguibile generato dal linker è, solitamente, in un formato eseguibile (es. PE o ELF) che contiene, oltre al codice e ai dati del programma, delle informazioni aggiuntive che vengono fornite al *loader*. Quest'ultimo è un componente del sistema operativo che carica nella memoria principale un programma con le diverse sezioni posizionate correttamente. In particolare, all'interno del formato ELF, che è utilizzato solitamente sui sistemi Unix-like, esistono due informazioni per ogni sezione del file eseguibile, che si chiamano Virtual Memory Address (VMA) e Load Memory Address (LMA). VMA indica l'indirizzo base con cui è stato rilocato il codice dal linker, mentre LMA è un indicazione per il loader su dove caricare, nella memoria del processo, la sezione.

Per istruire correttamente il linker è possibile utilizzare uno *script*, cioè un file di testo contenente una serie di comandi, che vengono interpretati dal linker durante la sua esecuzione. All'interno dello script per il linker è possibile specificare, per ogni

sezione, il Load Memory Address (LMA) e il Virtual Memory Address (VMA). Dato che il software realizzato non viene caricato da nessun loader, come avviene, invece, per una comune applicazione a livello utente, è necessario creare un file di output che rispecchi l'immagine che il software avrà in memoria. Il linker va, quindi, impostato per generare direttamente l'immagine del programma e non un file eseguibile, come avviene normalmente. Così facendo, LMA è l'indirizzo in cui verrà posizionata la sezione all'interno del file raw, mentre VMA è l'indirizzo che avrà la sezione durante l'esecuzione del file eseguibile generato. Nello sviluppo di applicazioni a livello utente, questi due indirizzi sono di norma uguali, ma nello sviluppare kernel o BIOS è comune che LMA e VMA differiscano. Ad esempio, è possibile che il BIOS copi parte del proprio codice dalla ROM (mappata nella RAM nell'intervallo di indirizzi `[0xFFF80000-0xFFFFFFFF]`) alla memoria principale, ad un indirizzo differente. Affinché il codice copiato nella RAM venga eseguito correttamente, è necessario che la sezione contenente il codice sia stata rilocata dal linker per essere eseguita a quell'indirizzo (VMA), nonostante questa, inizialmente, si trovasse ad un indirizzo differente (LMA).

Coreboot utilizza uno script per il linker che è prodotto automaticamente, attraverso l'analisi delle sezioni dei file oggetto, permettendo di riposizionare le varie sezioni con maggior criterio.

Con l'uso del linguaggio C anche per la parte di codice relativa allo SMI handler, è necessario modificare il tool che analizza le sezioni dei file oggetto. Come spiegato nella sezione 2.4 relativa a Coreboot, è il BIOS che copia il codice e i dati relativi a SMM all'interno della SMRAM, occupandosi successivamente di renderla inaccessibile a qualunque altro software. Per fare ciò, Coreboot deve conoscere l'indirizzo di memoria, all'interno della ROM, in cui trovare i dati di SMM e l'indirizzo, all'interno della SMRAM, in cui copiarli. Lo script del linker permette di fornire tali indirizzi al codice del BIOS, che potrà inizializzare la SMRAM in maniera appropriata. Le sezioni copiate all'interno della SMRAM, però, vanno considerate con cura, poiché anche essa ha una struttura particolare. Nello specifico, è necessario che il BIOS copi la prima istruzione che andrà eseguita dallo SMI handler all'indirizzo `0xA8000` e che eviti di scrivere nell'area di memoria dedicata alla State Save Map.

Attraverso l'uso di uno script per il linker e tenendo conto delle considerazioni appena fatte, è possibile avere un ambiente di esecuzione correttamente configurato per poter sfruttare tutti i vantaggi offerti dal linguaggio C, fin dalle prime fasi dello sviluppo. Nel dettaglio, è necessario configurare opportunamente LMA e VMA delle sezioni relative allo SMI handler, poiché, sebbene non sia fondamentale il posizionamento delle sezioni all'interno della ROM (LMA), lo è la loro localizzazione nella SMRAM (VMA). I dati, statici e non, sono stati posizionati a partire dall'indirizzo `0xA0000`, la routine di inizializzazione all'indirizzo `0xA8000`, così che venga eseguita appena ricevuto lo SMI, e le sezioni contenente codice a 64 bit consecutivamente a questa. In tal modo, si lascia intatta la State Save Map e si ottiene un ampio spazio per lo stack utilizzato dallo SMI handler, dall'indirizzo `0xAFE00` in giù. Inoltre, lo script per il linker fornisce dei simboli utilizzabili dai vari file sorgenti per indirizzare codice definito in altri file. In particolare, questa funzionalità risulta utile per dare al codice C che opera in modalità protetta, l'indirizzo della funzione da richiamare non appena abilitata la modalità IA-32e.

4.2.3 Gestione della memoria

Appena la CPU riceve uno SMI entra in SMM potendo accedere direttamente alla memoria fisica fino a 4 GB. Sebbene in questa modalità sia banale accedere ad una qualsiasi area di memoria fisica, è necessario, come precedentemente spiegato, abilitare la modalità IA-32e e la paginazione per poter superare il limite dei 4 GB di memoria fisica e accedere a parti di registri altrimenti inaccessibili. Una volta abilitata la paginazione, ogni indirizzo utilizzato viene convertito dalla MMU in un indirizzo fisico, utilizzando la tabella delle pagine puntata dal registro CR3. La conversione da indirizzo virtuale a indirizzo fisico avviene come spiegato nella sezione 2.3.3. Quindi, per accedere a una particolare area di memoria fisica, bisogna utilizzare un indirizzo virtuale che venga mappato nell'indirizzo fisico desiderato.

L'integrity checker ha bisogno di accedere alla memoria fisica per recuperare le strutture dati del kernel del sistema operativo in esecuzione e, perciò, è stato creato un gestore della memoria con il compito di gestire la tabella delle pagine attiva durante

l'esecuzione di YASIC.

Il sistema si occupa di gestire la paginazione, fornendo indirizzi virtuali con cui è possibile accedere ad un particolare indirizzo fisico. Nel dettaglio, viene fornito un indirizzo fisico al sistema, che controlla se esiste già un indirizzo virtuale che permette di accedere a tale indirizzo fisico: se è così viene restituito l'indirizzo virtuale, altrimenti viene modificata la tabella delle pagine per creare un nuovo mapping che permetta di accedere all'area desiderata. È compito del gestore della memoria sapere quali frame sono mappati in quale pagine virtuali e viceversa, così che solo per gli indirizzi fisici per cui non è già disponibile una mappatura ne venga creata una nuova. Per rendere il sistema il più piccolo possibile, vengono sfruttati solo i 64 KB di SMRAM disponibili di default e, all'interno di questa memoria, è necessario posizionare le tabelle delle pagine utilizzate dal gestore della memoria. Risulta, quindi, fondamentale tenere traccia di quali frame siano già mappati in indirizzi virtuali e quali no.

Un'altra operazione molto comune nella ricostruzione della semantica delle informazioni è l'accesso all'indirizzo fisico puntato da un indirizzo virtuale del sistema operativo in esecuzione. Tali indirizzi, infatti, non sono facilmente accessibili, perché la tabella delle pagine usata in SMM è differente da quella utilizzata precedentemente dal sistema operativo. Ciò che va fatto per recuperare i dati è:

1. recuperare il valore del registro CR3 utilizzato dal sistema operativo precedentemente in esecuzione;

2. simulare tramite software il funzionamento della MMU per ottenere l'indirizzo fisico associato all'indirizzo virtuale ricercato;

3. sfruttare la funzionalità sopra esposta, relativa al sistema della memoria, per ottenere un indirizzo virtuale valido durante l'esecuzione dello SMI handler, che sia mappato nell'indirizzo fisico trovato al passo precedente.

Simulare la MMU è un'operazione fondamentale per accedere alle informazioni del sistema operativo. A livello logico, il funzionamento è lo stesso utilizzato dalla MMU fisica. L'unica differenza è il metodo d'accesso alle diverse tabelle delle pagine, che sono collegate l'una all'altra attraverso indirizzi fisici. Ad esempio, il registro

CR3 contiene l'indirizzo fisico in cui trovare la tabella PML4 composta da 512 entry, ognuna delle quali contiene un indirizzo fisico per ottenere la tabelle di livello inferiore e così via, fino a raggiungere la tabella delle pagine. Dato che le tabelle sono collegate fra loro attraverso indirizzi fisici, bisogna tenere conto che, per accedere a ognuna di queste, è necessario utilizzare la funzionalità sopra esposta, che permette di ottenere un indirizzo virtuale utilizzabile durante l'esecuzione dello SMI handler.

Quindi, risulta essenziale la realizzazione di un gestore della memoria. Ogni accesso a un indirizzo virtuale del sistema operativo richiede, infatti, di attraversare diverse tabelle delle pagine dell'OS e successivamente modificare la tabella delle pagine attuale per poter accedere all'indirizzo desiderato.

Per concludere, si noti che, sia nell'attraversare le tabelle delle pagine, sia nell'accedere ad alcune parti dello spazio di memoria dei processi, è necessario controllare sempre la presenza effettiva della pagina in memoria principale. Il sistema operativo, infatti, può copiare una pagina dalla RAM all'hard disk nel caso in cui abbia bisogno di liberare memoria fisica. Per superare questo problema, si evita di accedere alle pagine in cui è azzerato il bit Present, contenuto in ogni voce delle varie tabelle delle pagine. Su queste pagine non è effettuato il controllo d'integrità e sono, quindi, ignorate.

Il sistema della memoria è usato soprattutto nel modulo di acquisizione dello stato, che necessita un continuo accesso ai dati del sistema operativo.

4.2.4 Recupero informazioni sui processi

Una volta abilitata la modalità IA-32e, è possibile ricostruire la semantica delle informazioni. Infatti, attraverso una buona conoscenza del sistema operativo precedentemente in esecuzione e grazie alla possibilità di leggere qualsiasi dato nella memoria principale, YASIC può ricostruire il significato dei dati memorizzati nella RAM. In particolare, dovendo controllare l'integrità di processi e librerie, è necessario sapere prima di tutto quali sono i processi in esecuzione.

Un processo è un programma in esecuzione, associato ad un insieme di risorse quali i file aperti, i segnali pendenti, lo stato del processore, uno spazio di indirizzi di memoria e altre informazioni utili al kernel. Nei moderni sistemi operativi, come

Linux, vengono fornite due virtualizzazioni: un processore virtuale e una memoria virtuale. La prima fornisce al processo l'illusione di avere il monopolio del sistema, nonostante il processore venga condiviso potenzialmente con altri centinaia di processi. La memoria virtuale, invece, permette al processo di allocare e gestire la memoria come se avesse tutta la memoria del sistema a propria disposizione [16].

Il kernel Linux gestisce i processi attraverso quello che viene chiamato *descrittore di processo*, che è una struttura di tipo *task_struct* i cui campi principali sono mostrati in figura 4.1, in cui sono presenti tutte le informazioni sopra esposte e molte altre. Per poter analizzare i processi in esecuzione, i *task* sono collegati fra loro e formano una lista doppiamente concatenata, chiamata *task list*. Tutte le *task_struct*, assieme ad altre strutture dati, si trovano nell'area di memoria dedicata al kernel, un'area della memoria virtuale di ogni processo, che viene mappata nella memoria fisica in cui è contenuto effettivamente il nucleo del sistema operativo. In particolare, in un sistema con un unico processore, esiste sempre uno e un solo *task* corrente.

Per poter ottenere la lista dei processi in esecuzione sul sistema, quindi, è sufficiente ottenere uno fra i *task* del sistema operativo e seguire i collegamenti che formano la *task* list.

Dato che è un'operazione comune, per il kernel, ottenere la *task_struct* relativa al processo corrente, Linux utilizza una macro chiamata *current* che calcola l'indirizzo del descrittore di processo corrente. Per farlo, sfrutta la struttura *thread_info* mostrata in figura 4.2, che contiene un puntatore a una *task_struct*. In ogni istante esiste una struttura *thread_info* che punta al *task* corrente e che si trova nello stack del kernel, a uno spiazzamento costante.

Quindi, YASIC, ottenuto il *task* corrente, può attraversare la *task list* per analizzare tutti i processi in esecuzione. Per recuperare il *task* corrente, però, bisogna innanzitutto trovare l'inizio dello stack del kernel e le tabelle delle pagine attive durante l'esecuzione del sistema operativo, così da poter convertire l'indirizzo dello stack in un indirizzo accessibile in SMM.

È possibile ottenere le informazioni necessarie attraverso la State Save Map, una tabella di 512 byte all'interno della SMRAM in cui il processore salva il proprio stato durante il trasferimento del controllo a SMM. Come spiegato nella sezione 2.3.1 relati-

```
struct task_struct {
        volatile long state;
        void *stack;
        ...
        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;

        struct list_head tasks;
        struct mm_struct *mm, *active_mm;

        int exit_state;
        int exit_code, exit_signal;

        pid_t pid, tgid;

        struct task_struct *real_parent;
        struct task_struct *parent;
        struct list_head children;
        struct list_head sibling;

        char comm[TASK_COMM_LEN];
        ...
};
```

Figura 4.1: Struttura *task_struct*

va a SMM, all'interno della State Save Map si trovano i registri di segmento, i registri di controllo e i registri generali, insieme ad altri dati utili per garantire un ambiente di esecuzione isolato e trasparente dal resto del sistema. In particolare, può essere utilizzata dal gestore di SMI per recuperare:

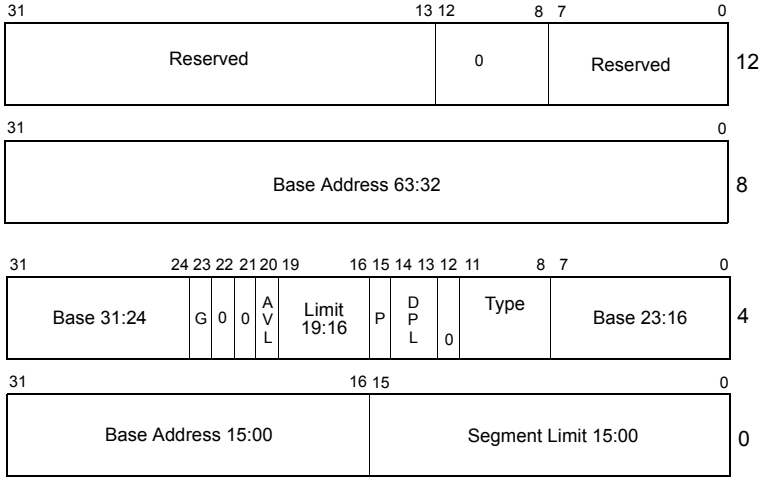
- il registro CR3, che contiene l'indirizzo fisico della tabella PML4 utilizzata dall'OS. Tale indirizzo è fondamentale per accedere ad ogni dato relativo al sistema operativo, poiché viene utilizzato dal sistema della memoria di YASIC per attraversare le tabelle delle pagine e recuperare l'indirizzo fisico associato ad un indirizzo virtuale dell'OS.

```
struct thread_info {
        struct task_struct      *task;
        struct exec_domain      *exec_domain;
        __u32                   flags;
        __u32                   status;
        __u32                   cpu;
        int                     preempt_count;
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        void __user             *sysenter_return;
        unsigned int            sig_on_uaccess_error:1;
        unsigned int            uaccess_err:1;
};
```

Figura 4.2: Struttura *thread_info*

- L'indirizzo base della GDT utilizzata dal sistema operativo. Con tale indirizzo e con il selettore del Task Register, anche esso contenuto nella State Save Map, è possibile trovare il descrittore del Task State Segment (TSS), il cui formato è mostrato in figura 4.3, in cui è presente l'indirizzo del TSS vero e proprio. Questa struttura è utile poiché contiene al suo interno gli indirizzi virtuali degli stack utilizzati dai *ring* dell'architettura Intel e, in special modo, l'indirizzo dello stack usato in *kernel mode*.

Attraverso i due dati ottenuti grazie alla State Save Map, e al gestore delle memoria spiegato nel precedente capitolo, si può accedere allo stack del kernel, ottenere la struttura *thread_info* relativa al *task* corrente e, quindi, recuperarne la struttura *task_struct*. In questo modo, YASIC può analizzare tutti i processi in esecuzione sul sistema attraversando la *task list*. Per ogni processo in esecuzione che si vuole controllare viene eseguita una scansione più approfondita, che coinvolge la memoria virtuale del processo.



AVL	Available for use by system software
B	Busy flag
BASE	Segment Base Address
DPL	Descriptor Privilege Level
G	Granularity
LIMIT	Segment Limit
P	Segment Present
TYPE	Segment Type

Figura 4.3: TSS descriptor

4.2.5 Analisi della memoria virtuale dei processi

Per ogni processo selezionato durante l'attraversamento della *task list*, che identifica tutti i processi in esecuzione, viene eseguita un'analisi più approfondita che analizza la memoria virtuale del processo, per poter eseguire, successivamente, il controllo d'integrità solo su alcune parti dello spazio degli indirizzi. In questa sezione si vede come il kernel Linux gestisce la memoria dei processi e in che modo sfruttarla per controllarne l'integrità.

Lo spazio di indirizzamento di un processo è l'insieme degli indirizzi di memoria virtuale che un processo può utilizzare. Nell'architettura Intel 64, ad ogni processo è assegnato uno spazio di indirizzi a 64 bit. Nonostante ciò, al giorno d'oggi, l'architettura ha definito un meccanismo per mappare solo indirizzi virtuali a 48 bit in indirizzi fisici a 52 bit, lasciando la traduzione di indirizzi virtuali a 64 bit per una specifica futura [15]. I bit più significativi di un indirizzo a 64 bit, che altrimenti non sarebbero usati al momento, vengono utilizzati per assicurarsi che gli indirizzi virtuali rispetti-

no quella che viene chiamata *forma canonica degli indirizzi*, che consiste nell'avere uguali gli ultimi bit dell'indirizzo. I 48 bit meno significativi, invece, sono utilizzati per accedere alle voci delle tabelle delle pagine, come spiegato nella sezione 2.3.3.

Oltre alle limitazioni imposte dall'architettura, è possibile comunque che un processo non possa indirizzare completamente il proprio spazio degli indirizzi, perché, ad esempio, non ha i permessi per accedere ad alcune aree oppure perché queste non sono state mappate. Gli intervalli di indirizzi mappati nello spazio di memoria del processo sono chiamati *aree di memoria* [16]. Un processo può richiedere al kernel di aggiungere o rimuovere aree di memoria dal proprio spazio degli indirizzi. Ogni area di memoria ha dei permessi, come quelli di lettura, scrittura ed esecuzione, e un processo può indirizzare solo aree di memoria valide, cioè aree su cui possiede i permessi adeguati all'operazione che intende eseguire.

Un'area di memoria può contenere diversi tipi di dati, tra cui:

- l'immagine del codice eseguibile del file binario, chiamata sezione *text*;
- l'immagine delle variabili globali inizializzate, contenute nel file eseguibile, chiamata sezione *data*;
- l'immagine di una pagina avente solo 0, che contiene le variabili globali non inizializzate, chiamata sezione *bss*;
- l'immagine di una pagina avente solo 0, usata come stack per il processo;
- immagini di sezioni *text*, *data* e *bss* per ogni libreria condivisa caricata nello spazio degli indirizzi del processo;
- un file mappato in memoria;
- un segmento di memoria condiviso con un altro processo;
- un mapping chiamato "anonimo", che non fa riferimento a nessun altro file o segmento, ma utilizzato per chiamate come la *malloc()*.

Ogni task contiene un puntatore a un *descrittore di memoria*, che mantiene tutte le informazioni relative allo spazio degli indirizzi associato ad un task. Il descrittore di

memoria è una struttura di tipo *mm_struct* e, nel dettaglio, memorizza l'inizio e la fine di ogni segmento di memoria del processo, il numero di pagine utilizzate, la quantità di spazio di indirizzi virtuale che è stata usata e altre informazioni utili per gestire la memoria virtuale di un processo. In particolare, però, contiene un insieme di aree di memoria virtuale (rappresentate attraverso la struttura *vm_area_struct*) e la tabella delle pagine del processo.

Il modulo di acquisizione dello stato, avendo a disposizione le *task_struct* dei processi da analizzare, può facilmente accedere al descrittore di memoria relativo, utilizzando le funzionalità offerte dal gestore della memoria. Successivamente, può scorrere la lista di aree di memoria del processo, analizzando solo quelle su cui si intende eseguire il controllo d'integrità. Nello specifico, le aree di memoria che saranno controllate sono quelle non scrivibili, che quindi contengono o dati in sola lettura o codice, come le sezioni *text* e *data*.

Come precedentemente spiegato, le aree di memoria contengono anche le immagini delle sezioni relative alle librerie condivise utilizzate dal processo. Quindi, è possibile controllare l'integrità delle librerie, assicurandosi che non vengano alterate. Per sapere a quale libreria è associata una sezione di codice o dati, è sufficiente utilizzare il campo *vm_file* della struttura *vm_area_struct*, che è un puntatore a una struttura *file*. Attraverso tale struttura si può recuperare il nome del file associato all'area di memoria e, nel caso sia un'area in cui è mappata una sezione di libreria, si ottiene il nome della libreria condivisa.

4.3 Modulo di comunicazione

In questa sezione viene studiato il modulo di comunicazione di YASIC, che ha il compito di comunicare con il *sistema monitor* per fornirgli le informazioni necessarie a verificare l'integrità dei processi in esecuzione sul *sistema target*.

4.3.1 Calcolo hash

Un metodo per permettere al sistema monitor di controllare l'integrità delle aree di memoria, ottenute grazie al modulo di acquisizione dello stato, è inviare tutto il contenuto delle aree, lasciando al monitor il compito di controllare che quelle aree combacino con quelle a sua disposizione. Tale metodo, però, ha lo svantaggio di dover trasmettere una grande quantità di dati e sebbene fattibile, risulta poco efficiente. Un'altra soluzione, che è quella utilizzata da YASIC, è usare il contenuto delle aree di memoria come input di una *funzione hash*, spiegata in maniera più approfondita nella sezione 2.5. La caratteristica fondamentale di una buona funzione hash è che anche cambiando un singolo bit della stringa in input, si ottiene un output differente. In particolare, si è scelto di utilizzare la funzione SHA1, per ottenere un buon compromesso tra efficienza e sicurezza. La funzione MD5, un'altra funzione molto usata fino a qualche anno fa, presenta diversi tipi di attacchi che permettono di trovare in poco tempo delle *collisioni* [29], e viene ormai considerata poco sicura.

Sebbene sia possibile calcolare l'hash su tutta l'area di memoria, questo significherebbe dover mappare tutta l'area nella memoria virtuale di SMM. Ciò introdurrebbe una serie di problemi, come la possibilità che non siano disponibili abbastanza voci nella tabella delle pagine utilizzata dall'integrity checker per mappare tutta l'area. Per questo, l'area di memoria viene mappata nella memoria virtuale di SMM a blocchi di 4 KB (o 2 MB nel caso l'area sia mappata utilizzando pagine di questa dimensione), l'hash viene calcolato sui diversi blocchi e successivamente spedito al modulo di comunicazione vero e proprio.

Dato che, come detto più volte, nella modalità di System Management non esiste alcun supporto generalmente offerto dal sistema operativo o dalle librerie, è stato necessario introdurre all'interno dello SMI handler il codice per calcolare la funzione hash SHA1, che, si fa notare, è circa il 60% di tutto il codice sviluppato.

4.3.2 Invio dati su rete

Il sistema target comunica con il sistema monitor attraverso la rete. Si presuppone, quindi, che nel momento in cui viene eseguito il controllo d'integrità, i due PC siano

connessi.

Quando, al passo precedente, viene collezionato un hash di un blocco di area di memoria, si può mandare il dato al monitor. Infatti tutte le informazioni necessarie sono state raccolte e necessitano solo di essere inviate. In particolare sono due i problemi che vanno risolti:

- cosa trasmettere e come organizzare i dati
- come inviare i dati attraverso la rete.

Per quanto riguarda il primo punto, si è scelto di creare un pacchetto dati con il formato in figura 4.4. Nel dettaglio, il campo *task_name* conterrà il nome del task relativo al blocco di dati e sarà di lunghezza 16 byte, la stessa lunghezza utilizzata da Linux per mantenere il nome del processo all'interno della struttura *struct_task*, mentre il campo *file_name* avrà, se presente, il nome del file associato all'area di memoria. *Address* conterrà l'indirizzo virtuale, a 64 bit, del blocco di cui si è calcolato l'hash, mentre *size* la dimensione del blocco (4 KB o 2 MB). Infine, il campo *hash* manterrà l'output della funzione SHA1 calcolata sul blocco analizzato. Attraverso queste informazioni il modulo di analisi, spiegato meglio nella prossima sezione, sarà in grado di controllare se il blocco di memoria preso in considerazione è integro oppure no.

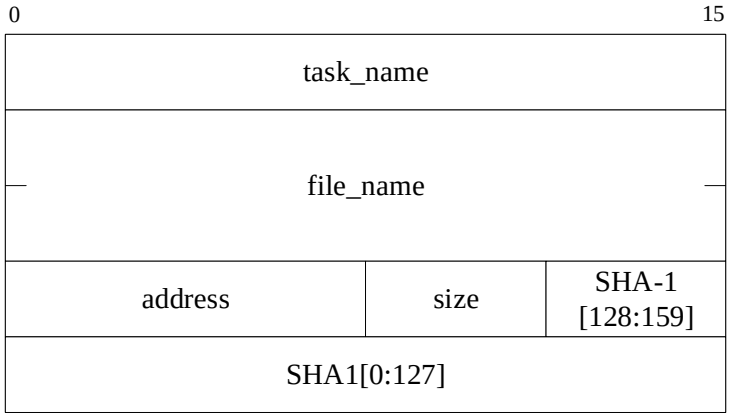


Figura 4.4: Formato payload pacchetto UDP

Per risolvere il secondo problema non è possibile fare affidamento sul sistema operativo come farebbe un comune software, ma è necessario implementare un driver ele-

mentare che sia in grado di comandare la Network Interface Card (NIC) presente sul sistema target. Sfruttando QEMU, si è scelto di utilizzare la NIC RTL8139 [20] per la sua semplicità di utilizzo. YASIC comunica, quindi, direttamente con l'hardware per inviare i propri pacchetti al sistema monitor. Il protocollo di comunicazione utilizzato è il protocollo UDP, spiegato nella sezione 2.6.

4.4 Modulo di analisi

Il modulo di analisi risiede nel sistema monitor e ha il compito di ricevere i pacchetti UDP trasmessi dal modulo di comunicazione e controllare l'integrità delle parti di processo che sono state trasmesse. Questo modulo non ha bisogno di utilizzare la modalità di System Management, ma può essere eseguito in kernel mode. Infatti, vista l'assunzione fatta in precedenza relativa alla sicurezza del sistema monitor, è possibile sfruttare le funzionalità già offerte dal sistema operativo per eseguire il controllo d'integrità.

Il modulo di analisi, per funzionare, deve possedere gli hash delle sezioni statiche dei processi da controllare e delle librerie utilizzate sul target. Se il monitor ha una configurazione analoga a quella del target, è anche possibile calcolare gli hash solo quando necessario, altrimenti il monitor deve collezionarli in un istante precedente al controllo d'integrità, quando il target non è considerato compromesso, e confrontarli nel momento in cui riceve i pacchetti UDP dal target.

Realizzando il modulo in kernel mode risulta semplice analizzare i processi e il loro spazio degli indirizzi, avendo a disposizione tutte le funzioni di gestione dei processi e della memoria.

Il modulo di analisi verifica che le sezioni ricevute abbiano lo stesso hash di quello calcolato sul sistema monitor. Nel caso in cui non sia così, segnala la possibile compromissione del target.

4.5 Generazione dello SMI

L'intero procedimento che permette di effettuare il controllo d'integrità dei processi può avvenire solo se si riesce a generare un System Management Interrupt (SMI). Lo SMI è un interrupt esterno che non può essere ignorato, con la priorità più alta di qualunque altro interrupt esterno, incluso il Non-Maskable Interrupt (NMI). Esistono due modi per generare uno SMI: uno software e uno hardware. Il software può generare uno SMI attraverso la scrittura su una particolare porta specificata dal chipset. In alternativa, il metodo hardware prevede la configurazione dell' I/O Controller Hub (ICH), anche chiamato Southbridge, così da far generare uno SMI in corrispondenza di alcuni eventi. L'Intel Platform Controller Hub serie 7 [14], ad esempio, prevede circa 40 modi diversi di innescare uno SMI, che vanno da eventi legati alla gestione dell'alimentazione, a quelli legati all'Universal Serial Bus (USB) o alla scadenza di un timer.

Il metodo software permette a un programma in grado di eseguire I/O sulla macchina di generare uno SMI, attraverso la scrittura sulla porta `0xB2`. È possibile, ad esempio, implementare un modulo del kernel che scriva periodicamente su tale porta così da garantire sempre l'esecuzione dell'integrity checker.

Nella realizzazione di YASIC sono, quindi, varie le soluzioni adottabili, sebbene quest'ultima richieda una modifica del kernel in uso e, inoltre, se il kernel dovesse venire compromesso, è possibile che venga alterato per evitare di scrivere sulla porta `0xB2` e prevenire l'esecuzione dello SMI handler, oppure è possibile che il sistema venga ripulito dalle modifiche effettuate prima di cedere il controllo all'integrity checker, come mostrato in [27]. Altre soluzioni prevedono l'uso di un dispositivo Peripheral Component Interconnect (PCI) e la relativa configurazione dell'ICH per la generazione dello SMI.

Un altro metodo è l'utilizzo dell'Intelligent Platform Management Interface (IPMI), un'interfaccia di gestione del sistema presente su molti server e direttamente implementata nell'hardware, indipendente dal processore, dal BIOS e dal sistema operativo. IPMI può essere usato per generare uno SMI, come dimostrato in [6]. Il vantaggio principale di questa soluzione è la possibilità, da parte del monitor, di deci-

dere i momenti in cui eseguire il controllo d'integrità, attraverso l'invio di pacchetti opportunamente forgiati.

In ogni caso, un software compromesso con elevati privilegi (es. sistema operativo, hypervisor) potrebbe disabilitare o redirigere lo SMI ad una routine sotto il proprio controllo, evitando l'esecuzione dell'integrity checker [27]. La soluzione migliore sarebbe raggiungibile solo con l'aiuto dedicato da parte dell'hardware e con l'introduzione di un dispositivo appositamente sviluppato.

4.6 Comunicazione sicura tra target e monitor

YASIC utilizza la rete per far comunicare il sistema target con il sistema monitor. Come già detto, si assume che un potenziale attaccante non sia in grado di manomettere o sostituire l'hardware del target e, in particolare, la NIC utilizzata. Nonostante ciò, l'utilizzo della rete estende la superficie d'attacco al sistema realizzato in questo lavoro di tesi.

Per esempio, un attaccante nella rete potrebbe intercettare i pacchetti inviati e modificarne il contenuto per nascondere le tracce dell'alterazione della memoria di un processo. Per prevenire questo tipo di attacco è necessario stabilire una chiave segreta condivisa tra sistema target e sistema monitor, così che il monitor possa essere certo che i dati ricevuti provengano effettivamente dal sistema target e non siano, invece, stati modificati. Una soluzione banale e poco efficiente è quella di inserire la chiave direttamente nel codice dello SMI handler, all'interno della SMRAM. Sebbene la SMRAM, una volta inizializzata dal BIOS, venga resa inaccessibile, il contenuto dello SMI handler viene configurato e copiato nella SMRAM dal BIOS. Ciò significa che il codice dell'integrity checker e, quindi, la chiave segreta, si trovano anche essi all'interno del BIOS, rendendo la sicurezza della chiave soggetta ad attacchi di *reverse engineering*.

Una soluzione a questo problema è quella di utilizzare una smart card sul sistema target, in cui memorizzare la chiave privata [21]. Attraverso tale device si evita che la chiave privata sia accessibile da dispositivi che non siano la smart card stessa. È possibile, quindi, calcolare un hash del pacchetto dati da trasmettere al sistema

monitor e inviarlo alla smart card, che restituisce l'hash cifrato attraverso la chiave segreta. In questo modo si assicura l'autenticità dei pacchetti inviati attraverso la rete. Dato che l'integrity checker viene eseguito in SMM, è necessario, come per la NIC, implementare un driver in grado di comandare la smart card.

Un'altra soluzione riguarda l'utilizzo del Trusted Platform Module (TPM), un processore crittografico presente su alcune piattaforme, in grado di memorizzare chiavi crittografiche e attestare l'identità e lo stato *fidato* della piattaforma. TPM fornisce un'interfaccia per sigillare dei dati associandoli alla configurazione corrente della piattaforma, sia hardware che software, o a parte di questa. I dati vengono cifrati dal TPM, che restituisce in output il dato cifrato. Quest'ultimo può essere decifrato solo dal Trusted Platform Module e solo se la configurazione fornita è la stessa con cui è stato associato il dato. Questa funzionalità viene chiamata *sealing* [18] e può essere sfruttata per cifrare la chiave segreta tra monitor e target, in modo da associarla alla configurazione della piattaforma che si ottiene quando la CPU è in SMM. Così facendo, si può memorizzare la chiave cifrata nella SMRAM e farla decifrare alla prima esecuzione di YASIC, senza renderla visibile a nessun software oltre a quello eseguito in SMM.

Nel caso in cui non sia presente il TPM è possibile comunque stabilire una chiave segreta tra sistema target e monitor. Infatti, il lavoro realizzato presuppone che il BIOS non sia compromesso, ma sia fidato. Si può, quindi, implementare nel BIOS un protocollo di comunicazione tra i due sistemi per stabilire la chiave segreta e memorizzarla subito dopo nella SMRAM, resa successivamente inaccessibile, come spiegato in [31].

Supponendo di avere una comunicazione sicura tra sistema target e sistema monitor, è possibile, comunque, che alcuni pacchetti vadano persi nella rete oppure vengano affetti da errori di trasmissione. Per quanto riguarda gli errori di trasmissione si può usare il *checksum* previsto dal protocollo UDP. Il monitor, calcolato il checksum sul pacchetto ricevuto, lo confronta con quello memorizzato nell'header per verificare la correttezza del dato e, nel caso non siano uguali, segnala al target l'errore. Riguardo, invece, la perdita di pacchetti nella trasmissione dal sistema target al monitor, è necessario stabilire un protocollo che permetta al target di sapere quali pacchetti vanno inviati nuovamente e quali no. Tale protocollo è lasciato come sviluppo futuro e possibile estensione del software realizzato.

Capitolo 5

Conclusioni

In questo lavoro di tesi è stata sviluppata una tecnica di integrity checking basata sulla modalità di System Management offerta dai processori Intel. Nonostante l'idea di eseguire un integrity monitor in SMM non sia completamente nuova , la maggior parte dei lavori svolti nel campo della ricerca riguardano il controllo d'integrità del sistema operativo e dell'hypervisor. YASIC, invece, controlla direttamente le applicazioni eseguite nel sistema operativo, lasciando il problema dell'integrità di altre parti del sistema a tool differenti. Inoltre, il software sviluppato è in grado di gestire sistemi operativi eseguiti su architetture Intel 64, mentre molte delle soluzioni relative a SMM tralasciano questo aspetto.

Quanto realizzato mostra come sia possibile analizzare un sistema in esecuzione avendo a disposizione solamente il contenuto della RAM, attraverso la quale YASIC riesce a ricostruire la semantica delle informazioni relative ai processi da controllare. Proprio per questo motivo, l'integrity monitor sviluppato in questo lavoro di tesi non può essere portabile, in quanto, per realizzarlo e, in particolare, per recuperare le informazioni dei processi, si è fatto affidamento su una conoscenza approfondita del kernel. YASIC è stato sviluppato per analizzare un sistema Linux, ma un procedimento analogo potrebbe essere seguito per analizzare i processi in esecuzione su altri sistemi operativi.

A differenza, però, di molti altri lavori che sono stati sviluppati utilizzando SMM, YASIC è stato realizzato fin dall'inizio con l'idea di operare su sistemi che supporta-

no l'architettura Intel 64. Ciò ha introdotto la necessità di realizzare il gestore della memoria spiegato nella sezione 4.2.3, che ha portato una maggiore complessità che sarebbe altrimenti assente se si potesse accedere direttamente alla memoria fisica.

È possibile aggiungere alcune migliorie all'interno dell'integrity monitor. Si potrebbe, ad esempio, sviluppare un protocollo di comunicazione con il monitor che sia sicuro e affidabile, come accennato nella sezione 4.6. In questo modo, ci si assicura che un attaccante che agisce nella rete e si pone tra i sistemi target e monitor non possa intercettare e falsificare i pacchetti inviati.

Come seconda cosa, sarebbe necessario permettere al sistema monitor di decidere quali processi devono essere analizzati, effettuando una comunicazione fra monitor e target non appena YASIC viene mandato in esecuzione. Il software realizzato fino a questo momento, invece, permette di analizzare solo quei processi che sono stati selezionati durante lo sviluppo, rendendo la soluzione poco flessibile.

L'integrity monitor sviluppato, come detto più volte, si occupa soltanto di controllare le sezioni di memoria statiche dei processi, tralasciando, invece, quelle che contengono dati dinamici. In realtà, questa lacuna di YASIC è facilmente risolvibile includendo all'interno del modulo di analisi dello stato la capacità di inviare tali sezioni al monitor e includendo, all'interno di quest'ultimo, uno o più moduli che permettono di analizzare le aree ricevute, come fatto in [31]. Tale funzionalità è, comunque, fuori dallo scopo di questo lavoro di tesi, dato che per l'analisi delle aree di memoria dinamiche, come *stack* e *heap*, non è sufficiente e non è possibile eseguire un banale controllo di integrità come, invece, è possibile fare con le sezioni statiche.

In ogni caso, YASIC è in grado di analizzare lo stato corrente di un sistema ed estrarre tutte le informazioni di cui si potrebbe aver bisogno per realizzare un integrity checker. Il modulo di analisi dello stato, infatti, è stato sviluppato per poter ottenere i dati dal kernel del sistema operativo, riuscendo a superare il "semantic gap" di cui si è parlato in precedenza. La modalità di System Management viene sfruttata per ottenere un ambiente trasparente ed isolato in cui eseguire il software, garantendo un controllo d'integrità che sia affidabile, dato che le uniche cose su cui YASIC fa affidamento sono il BIOS e l'hardware. Quanto è stato realizzato può essere pensato come base per un *framework* di analisi eseguito in System Management Mode.

Bibliografia

[1] Coreboot. http://www.coreboot.org/Welcome_to_coreboot.

[2] National institute of standards, nist. http://nvd.nist.gov.

[3] Qemu - open source processor emulator. http://wiki.qemu.org/Main_Page.

[4] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, September 2012.

[5] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. Hima: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009.

[6] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[7] J. Butler and K. Kendal. Blackout: What really happened. *Black Hat USA*, 2008.

[8] F. Chen, Y. Li, T. Zhang, and K. Wu. A static-dynamic conjunct windows process integrity detection model.

[9] P. M. Chen and B. D. Noble. When virtual is better than real. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[10] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[11] L. Duflot, D. Etiemble, and O. Grumelard. Using cpu system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

[12] S. Embleton, S. Sparks, and C. C. Zou. Smm rootkit: a new breed of os independent malware. *Security and Communication Networks*, 2010.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, August 2012.

[14] Intel Corporation. *Intel® 7 Series / C216 Chipset Family Platform Controller Hub (PCH) - Datasheet*, June 2012.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A, 3B & 3C): System Programming Guide*, March 2013.

[16] R. Love. *Linux kernel development*. Pearson Education, 2010.

[17] McAfee® Proven Security™. Rootkits, part 1 of 3: The growing threat. `http://download.nai.com/Products/mcafee-avert/whitepapers/akapoor_rootkits1.pdf`, April 2006.

[18] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[19] 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 USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[20] Realtek Semiconductor Corp. *Single-chip multi-function 10/100Mbps ethernet controller with power management - Datasheet*, August 2005.

[21] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. When hardware meets software: a bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 79–88. ACM, 2012.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007.

[23] M. Sikorski, A. Honig, and S. Lawler. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[24] M.-K. Sun, M.-J. Lin, M. Chang, C.-S. Laih, and H.-T. Lin. Malware virtualization-resistant behavior detection. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 912–917. IEEE, 2011.

[25] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[26] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[27] J. Wang, K. Sun, and A. Stavrou. An analysis of system management mode (smm)-based integrity checking systems and evasion attacks. Technical report, Technical report, George Mason University, 2011.

[28] J. Wang, K. Sun, and A. Stavrou. Hardware-assisted application integrity monitor. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5375–5383. IEEE, 2012.

[29] X. Wang and H. Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[30] R. Wojtczuk and J. Rutkowska. Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[31] F. Zhang, K. Leach, K. Sun, and A. Stavrou. Spectre: A dependable introspection framework via system management mode. In *Proceedings of The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.