

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E SVILUPPO DI UN
SISTEMA PER LA VERIFICA
DELL'INTEGRITÀ HARDWARE E
FIRMWARE DI UN SISTEMA

Elaborato in
PROGRAMMAZIONE DI RETI

Relatore
GABRIELE D'ANGELO

Presentata da
MARCO MANCINI

Prima Sessione di Laurea
Anno Accademico 2015 – 2016

PAROLE CHIAVE

Integrità firmware

Tracciamento Hardware

Integrità Hardware

Information Security

Linux

Alla mia famiglia e ai miei compagni di corso, è grazie al loro supporto che ho avuto la forza di continuare e terminare questo primo percorso universitario.

Indice

1	Integrità firmware e hardware	1
1.1	Panoramica integrità firmware e hardware	1
1.2	Obbiettivi	2
2	Conoscenze di base	3
2.1	Nozioni di hardware e firmware	3
2.2	Nozione di kernel	4
2.3	Nozione di integrità e crittografia	4
2.3.1	Funzioni crittografiche di hash	5
2.3.2	MD5	7
2.3.3	SHA	7
3	Analisi firmware e hardware	9
3.1	Firmware nel mondo Linux	9
3.1.1	Evoluzione del kernel	10
3.1.2	Richiesta dei firmware	10
3.1.3	Distribuzione dei firmware	11
3.1.4	Progetto FWUPD	12
3.2	Firmware nei dischi rigidi	13
3.2.1	Localizzazione e caricamento del firmware	13
3.2.2	Casi di dump	14
3.2.3	Equation Group	16
3.3	Firmware BIOS	17
3.3.1	BIOS e UEFI	17
3.3.2	LightEater	19
3.3.3	Sistema Virustotal	20
3.3.4	Tool per dump del firmware	20
3.4	Tracciamento Hardware	23
3.4.1	PSN Intel	24
3.4.2	Analisi dei comandi	25
3.4.3	Analisi dettagliata dmidecode	30

4	Progettazione e sviluppo del sistema	33
4.1	Funzionalità	33
4.2	Scelta del linguaggio e dell'IDE di sviluppo	35
4.3	Suddivisione del sistema e panoramica	35
4.4	Progettazione e sviluppo del sottosistema hardware	35
4.4.1	Modellazione classi HWDevice e Command	36
4.4.2	Implementazione dei dispositivi specifici	42
4.4.3	Modellazione e sviluppo DeviceManager	43
4.4.4	Implementazioni specifiche DeviceManager	49
4.4.5	Modellazione HWManager	55
4.4.6	Test del sottosistema	57
4.5	Progettazione e sviluppo del sottosistema firmware	63
4.5.1	Modellazione di un Firmware	64
4.5.2	Modellazione e sviluppo FirmwareManager	65
4.5.3	Test del sottosistema	69
4.6	Modellazione e implementazione della classe ShellMenuParser	71
4.7	Sviluppi futuri	72
5	Conclusioni	73
A	Link GitHub al progetto	75
	Bibliografia	77

Capitolo 1

Integrità firmware e hardware

1.1 Panoramica integrità firmware e hardware

Il tema che andrò a trattare è l'integrità firmware e hardware il quale, in questi ultimi anni, si è fatto sempre più spazio nel campo dell'Information Security [1].

Per *integrità* [36] intendiamo la protezione di dati e informazioni a fronte di modifiche dolose o meno del loro contenuto.

Sono molteplici i casi reperibili in rete di attacchi verso terminali avvenuti attraverso la violazione dell'integrità dei firmware installati sulle macchine stesse, i quali hanno causato ingenti danni. Tanto per citarne uno, si può fare riferimento ad un gruppo di hacker denominato Equation Group [2], di cui parleremo più in dettaglio successivamente, il quale è stato capace di progettare ed implementare un tool in grado di infettare i firmware di diversi modelli di dischi rigidi di imprese come Seagate, Western Digital, IBM, Toshiba, Samsung e Maxtor. Una volta infettato il sistema, il tool riusciva a creare partizioni non visibili dall'utente ma accessibili dagli attaccanti, le quali erano utilizzate per memorizzare informazioni che il tool stesso reperiva dal sistema infetto.

Il problema principale per evitare e scovare questo tipo di attacchi è proprio il livello a cui essi operano, come si può notare in Figura 1.1 il livello firmware si trova appena sopra quello hardware e risulta essere quindi alla base della gerarchia di qualsiasi sistema informatico.

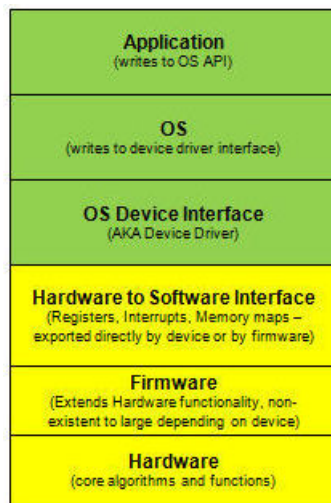


Figura 1.1: In questa immagine vengono descritti i diversi livelli architetturali di un generico sistema informatico. Immagine tratta da [37]

Un attacco a questa profondità è difficilmente rilevabile, soprattutto dai generici antivirus e, se ben progettato, può essere in grado di sopravvivere a riavvii e formattazioni del sistema stesso.

Risulta quindi interessante poter verificare l'autenticità dei firmware installati in una determinata postazione e, inoltre, poter monitorare costantemente lo stato dell'hardware installato.

1.2 Obiettivi

L'obiettivo posto è quello di riuscire a creare uno strumento software in grado di verificare che non ci siano state alterazioni hardware o firmware in una generica postazione utente. In pratica, vogliamo essere in grado, di determinare eventuali variazioni (es. compromissione del firmware) con un buon livello di confidenza.

Capitolo 2

Conoscenze di base

In questo capitolo si vogliono fornire le conoscenze di base dei principali concetti che verranno trattati e con i quali il lettore potrebbe non avere dimestichezza.

2.1 Nozioni di hardware e firmware

Quando parliamo di **hardware** intendiamo tutti i componenti fisici presenti all'interno di una macchina. La componentistica a cui facciamo riferimento può essere di natura elettronica, meccanica (e.g. dischi rigidi), magnetica (e.g. floppy disk) e ottica (e.g. lettore ottico). Essendo componenti di vario genere e con scopi diversi alcuni di essi risultano sicuramente più interessanti da un punto di vista della sicurezza, analizzeremo in seguito suddetti elementi.

Un **firmware** (anche chiamato **microcodice**) può essere visto come una sorta di software contenuto in una memoria permanente (e.g. ROM, EPROM o memorie flash) di un determinato dispositivo. Attraverso esso è possibile avviare il dispositivo su cui è installato ed interagirci, può essere quindi visto come un intermediario tra le componenti logiche e fisiche di un sistema. Essendo una sorta di applicativo molto low-level, nel corso della sua vita potrebbe richiedere aggiornamenti di varia natura (e.g. aggiunta di servizi, correzione di bug), questo tipo di operazione è chiamata comunemente *flash* ed è considerata molto rischiosa in quanto, se qualcosa andasse storto durante questa fase, il dispositivo sul quale si sta svolgendo l'aggiornamento del firmware potrebbe smettere di funzionare o non funzionare più correttamente. Nella maggior parte dei casi l'aggiornamento di un firmware avviene attraverso tool specifici distribuiti dai produttori dell'hardware stesso i quali possono offrire anche funzionalità di backup del firmware, in modo da poter recuperare ad eventuali danni. Un'altra operazione che può essere svolta sui firmware è il cosiddetto

dump, cioè il recupero del firmware stesso da un determinato device, vedremo in seguito casi interessanti di questo tipo.

2.2 Nozione di kernel

Nell'ambito informatico, per **kernel**, si intende il nucleo del sistema operativo il quale ha il completo controllo di tutto ciò che si verifica nel sistema e fornisce ai processi attivi sulla macchina un accesso sicuro e controllato sull'hardware. Il kernel è la prima componente del sistema operativo ad essere caricata durante l'avvio del sistema e rimane attiva per tutta la durata della sessione visto che i suoi servizi sono continuamente richiesti. Le funzionalità che offre sono molteplici, tipicamente include la gestione della memoria, la gestione dei processi, la gestione Input/Output e molto altro. Tutti questi servizi sono utilizzabili dal sistema attraverso opportune interfacce chiamate *system calls* (chiamate a sistema). I kernel possono essere divisi in due macrofamiglie, quelli monolitici e i microkernel. I primi contengono tutte le funzionalità principali del sistema operativo e le interfacce per l'utilizzo dei dispositivi e, proprio per questi motivi, rendono le dimensioni del codice molto ampie ed il kernel stesso difficilmente estendibile a fronte di aggiunte di dispositivi hardware. Quest'ultima operazione, infatti, comporterebbe all'aggiunta nel kernel dell'interfaccia utilizzata per uno specifico device, tuttavia i kernel monolitici odierni (come il kernel Linux [3]) sono in grado di caricare moduli in fase di esecuzione permettendo un'estensione dinamica del kernel stesso. I microkernel, invece, offrono solamente le funzioni basilari per il corretto funzionamento di un sistema, come la definizione dello spazio di indirizzamento di memoria, la comunicazione tra processi e la gestione di questi ultimi permettendo al sistema operativo una maggiore flessibilità nell'implementazione di ulteriori funzionalità.

2.3 Nozione di integrità e crittografia

In informatica per **integrità** intendiamo uno dei cinque pilastri dell'Information Assurance (IA) [4], gli altri quattro sono autenticazione, disponibilità, riservatezza e non ripudiabilità. Questo termine è composto da due concetti [5] correlati tra loro, che sono:

- **Integrità dei dati:** Assicura che le informazioni vengano modificate solo da utenti autorizzati e secondo determinate modalità.

- **Integrità del sistema:** Assicura che un sistema svolga le funzionalità previste in maniera sicura, cioè senza che esso sia stato soggetto a manipolazioni intenzionali o accidentali.

La perdita di integrità dei dati può avvenire accidentalmente, per esempio la rottura di un disco rigido in cui i dati sono contenuti, o intenzionalmente, cioè attraverso attacchi informatici. Un attacco può essere generalmente visto come una violazione della politica di sicurezza adottata da un sistema avvenuto violando i meccanismi progettati appositamente per rendere sicuro il sistema.

Uno dei metodi maggiormente adottati nell'IA per far fronte ad attacchi informatici e cercare di garantire anche l'integrità dei dati è la **crittografia**. Attraverso quest'ultima è possibile codificare i dati in un formato non accessibile da un utente non autorizzato, permettendo di trasmettere dati senza che nessuno, a parte il diretto interessato, possa decodificare o compromettere le informazioni.

Questo campo è composto da un'ampia famiglia di algoritmi crittografici tra i quali i più importanti sono:

- **Crittografia a chiave simmetrica:** Utilizzata per crittare messaggi facendo utilizzo di una sola chiave.
- **Crittografia a chiave pubblica:** Utilizzata per crittare messaggi utilizzando una coppia di chiavi, una pubblica e una privata.
- **Funzioni hash crittografiche:** Utilizzano una particolare trasformazione, la quale varia a seconda dell'algoritmo utilizzato, per crittare i dati presi come input.

2.3.1 Funzioni crittografiche di hash

Tra le tre macrocategorie appena citate porremo più attenzione sulle funzioni crittografiche di hash visto che sono adatte per verificare l'integrità dei dati.

Come si può notare in Figura 2.1 una generica funzione di hash prende come input un messaggio di lunghezza variabile e produce una stringa di dimensione fissa chiamata valore di hash (o digest) la quale lunghezza varia a seconda dell'algoritmo utilizzato.

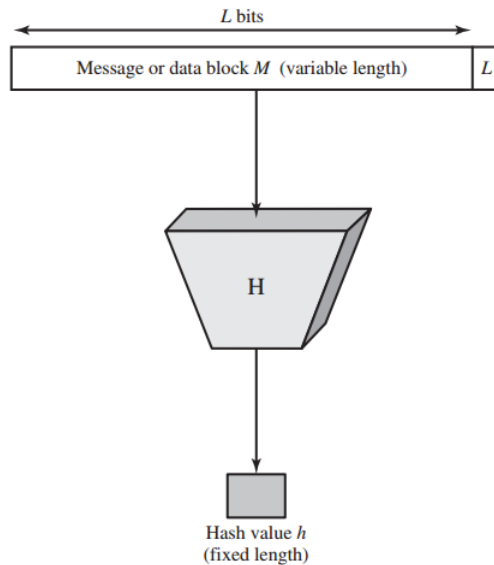


Figura 2.1: Funzionamento di una generica funzione di hash
Immagine tratta da [5]

Una generica funzione di hash, per essere considerata tale, deve rispettare le seguenti proprietà:

1. Deve poter essere applicata ad un blocco di dati di qualsiasi grandezza.
2. Deve produrre un output di lunghezza fissa.
3. Dato un input deve essere computazionalmente facile calcolare il suo digest.
4. Dato un valore di hash deve essere computazionalmente difficile, se non impossibile, risalire al messaggio che lo ha generato. Questa proprietà prende il nome di *non invertibilità della funzione*.
5. Deve essere computazionalmente difficile modificare un messaggio senza modificare il relativo valore di hash. La proprietà in questione è anche chiamata *resistenza debole alle collisioni*.
6. Deve essere difficile, sempre in termini computazionali, trovare due input che, passati alla funzione, generano lo stesso valore di hash. In questo caso parliamo di *resistenza forte alle collisioni*.

Dopo aver definito le proprietà che qualsiasi funzione hash dovrebbe rispettare possiamo ad analizzare la sicurezza di una generica funzione. Esistono due approcci di attacco per queste funzioni, il primo è basato sulla crittoanalisi

mentre il secondo è un approccio di tipo brute-force. La crittoanalisi di una funzione di hash si basa sullo studio di quest'ultima con lo scopo di ricercare falle logiche che poi verranno sfruttate per corrompere la sicurezza della funzione, l'unico modo per evitare attacchi del genere è progettare la funzione nel modo più sicuro possibile.

La resistenza di una funzione a fronte di attacchi brute-force, invece, deriva unicamente dalla lunghezza dell'output prodotto. Se una funzione rispetta le proprietà precedentemente descritte di resistenza alle collisioni allora, secondo il birthday-paradox¹, un eventuale attaccante riuscirebbe a trovare una collisione eseguendo $2^{n/2}$ computazioni. Andiamo ora ad analizzare due delle più utilizzate funzioni crittografiche di hash ponendo particolare attenzione alla loro sicurezza.

2.3.2 MD5

La funzione crittografica di hash MD5 [6] fu progettata da Ronald Rivest nel 1994 come miglioramento della precedente versione MD4 [7], sempre ideata da lui.

L'algoritmo in questione prende come input un messaggio di lunghezza arbitraria e produce un digest di 128-bit. Anche se, già dalla sua nascita, sono state trovate diverse debolezze [8] risulta ancora molto utilizzato nella crittografia come firma digitale, integrità dei dati ed autenticazione degli utenti.

Sono numerose le ricerche effettuate sulla sicurezza di questa funzione, nel 2005 un ricercatore cinese di nome Jie Liang [9] è riuscito a progettare un algoritmo in grado di trovare in poche ore due sequenze di stringhe lunghe 128 byte che, passate alla funzione MD5, producono lo stesso valore di hash, rompendo così la resistenza alle collisioni dell'algoritmo. Alla luce di queste ricerche posso affermare che la funzione crittografica di hash MD5 non è una funzione sicura e che non può sempre garantire l'integrità dei dati.

2.3.3 SHA

La seconda famiglia di funzioni di hash che trattiamo è la Secure Hash Algorithm (SHA) [10], questa comprende cinque diversi algoritmi, che sono SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512 dei quali il primo produce un digest di 160 bit, mentre i restanti producono valori di hash con lunghezza in bit pari al valore riportato nella loro sigla.

Soffermiamoci ora sulla sicurezza che il primo algoritmo di questa famiglia offre; anche in questo caso, come per l'MD5, sono stati effettuati diversi test di attacco uno [11] dei quali ha dimostrato che è possibile trovare una collisione

¹https://en.wikipedia.org/wiki/Birthday_problem

con una complessità minore di 2^{69} operazioni di hash. Tutte le altre funzioni della famiglia, invece, risultano essere più resistenti alle collisioni e quindi migliori per la verifica dell'integrità dei dati.

Capitolo 3

Analisi firmware e hardware

In questo capitolo verranno affrontate diverse tematiche riguardanti il mondo del firmware e dell'hardware. Si procederà trattando inizialmente la gestione dei firmware in ambienti Linux per poi analizzare in dettaglio il funzionamento dei firmware di dispositivi particolarmente delicati nell'ambito della sicurezza, come BIOS e dischi rigidi. Durante tutta questa fase di analisi verranno mostrati diversi esempi di attacchi e progetti interessanti per comprendere al meglio l'importanza di questo ambito.

3.1 Firmware nel mondo Linux

In questa prima sezione verrà definito come e dove, nei moderni sistemi Linux, i firmware vengano caricati facendo anche riferimento ad un interessante progetto nato da poco ma, prima di entrare nel dettaglio, è meglio effettuare alcune precisazioni.

Il mondo Linux comprende un grande insieme di distribuzioni (circa 300) le quali si basano sul Kernel Linux [3], il quale ricordiamo essere monolitico, ma che si differenziano per diversi fattori. Uno di questi, tra i più importanti, è l'organizzazione dei filesystem, cioè il meccanismo secondo il quale i file di un generico Sistema Operativo vengono organizzati all'interno del dispositivo di memorizzazione. Molte distribuzioni seguono il Filesystem Hierarchy Standard (FHS) [12], cioè uno standard nato nel 1993 ed evolutosi nel tempo, il quale si pone come obiettivo quello di standardizzare l'organizzazione dei filesystem in sistemi operativi Unix-like, tra cui i sistemi Linux. Altre distribuzioni, invece, hanno preferito non seguire questo standard ed adottare approcci differenti. Ci troviamo quindi in un mondo molto frammentato e difficile da coprire nella sua interezza.

3.1.1 Evoluzione del kernel

Per raggiungere al meglio l'obiettivo posto in questa sezione è utile vedere come il Kernel Linux si sia evoluto nel tempo. Il Kernel Linux nacque più di 20 anni fa e, anno dopo anno, ha visto aggiungersi diverse funzionalità, tra tutte le versioni noi ci concentreremo sulla 2.6. Da questa versione, rilasciata nel Dicembre del 2003, è possibile identificare dinamicamente i dispositivi che vengono collegati al sistema e riconoscere le relative proprietà (e.g. codice identificativo del produttore, id del device, ecc), questa funzionalità è stata implementata attraverso il sistema udev [13] e l'aiuto di hotplug, che andremo presto a definire.

Prima di questa versione il Kernel Linux utilizzava un filesystem statico chiamato Device File System (DevFS), il quale era eseguito esclusivamente in kernel mode. Con questo filesystem la directory `/dev` era popolata sin dall'installazione del sistema operativo da tutti i possibili device che un sistema poteva controllare, ovviamente questa metodologia non era la migliore e comportava un enorme spreco di spazio di memorizzazione. La versione presa in analisi ha portato alla sostituzione del DevFS con Udev; udev è un sistema composto da un demone¹ e da diversi servizi kernel, tra cui hotplug il quale può essere eseguito in userspace, al contrario del suo predecessore. È proprio l'avvento di hotplug che permette di gestire dinamicamente l'aggiunta e la rimozione dei dispositivi, caricare i driver appropriati e ricercare i firmware per determinati device. Sarà proprio questa funzionalità ad essere presa come caso di studio in seguito.

3.1.2 Richiesta dei firmware

Dopo aver definito come l'avvento di udev e hotplug abbia rivoluzionato il Kernel Linux passiamo ora ad analizzare come un generico sistema Linux effettui le varie richieste dei firmware [14].

Ogni volta che il driver di un determinato dispositivo necessita del firmware viene effettuata una chiamata alla funzione `request_firmware(fw_entry, FIRMWARE, device)` questa funzione, come si può notare, accetta tre parametri che andiamo ora a definire. Il primo parametro rappresenta il puntatore all'immagine del firmware in questione sul quale, come vedremo successivamente, verrà caricato il firmware, il secondo parametro indica il nome del file contenente il firmware mentre il terzo è il device per il quale viene richiesto il firmware.

Non appena questa chiamata a funzione viene effettuata, il kernel effettua una ricerca dell'immagine del firmware attraverso il suo nome, il quale ricorda-

¹un demone è un software eseguito in background il quale tipicamente ha lo scopo di fornire servizi all'utente

mo essere passato come secondo parametro, nelle directory `/lib/firmware` e `/lib/firmware/updates` dedicate appositamente a contenere i diversi firmware. Se l'esito della ricerca è positivo allora l'immagine sarà collegata al puntatore passato come primo parametro ed il driver sarà in grado di accedervi per poi rilasciare le risorse. Nel caso in cui il kernel non sia stato in grado di trovare l'immagine allora si passa in userspace generando la directory `/sys/class/firmware/X/loading,data`, in questo path la X identifica il nome del firmware che si sta ricercando mentre loading e data sono due attributi. Loading è utilizzato per gestire lo stato di caricamento del firmware e può assumere tre diversi valori, 1, 0 o -1. Il valore 1 indica che il processo sta caricando il firmware, con il valore 0 viene indicato che il processo di caricamento ha raggiunto il termine mentre, se l'attributo loading assume il valore -1 allora significa che il processo è stato abortito. Il secondo attributo, invece, è utilizzato esclusivamente per contenere l'immagine del firmware.

Appena finita la creazione della cartella l'attributo loading viene impostato ad 1 così da poter cominciare la ricerca del firmware. Dopo questa fase il processo torna in kernel mode dove il kernel si occuperà di terminare tutti i caricamenti parziali che sono in corso, per poi tornare in userspace. Ora è compito dell'utente effettuare l'upload del firmware interessato nella cartella `/sys/class/firmware/X/data`, una volta che anche questa operazione è stata completata entra in gioco lo script hotplug il quale non farà altro che recuperare l'immagine nell'apposita cartella e passarla al kernel il quale, a sua volta, si occuperà di reindirizzarlo al driver che ha generato la chiamata.

Il processo appena descritto, però, deve avvenire entro un certo limite di tempo, questo limite è impostabile ed è definito in termini di secondi nel file `/sys/class/firmware/timeout`, nel caso in cui l'esecuzione del processo non avvenga entro questo limite, hotplug lo ferma impostando l'attributo loading a -1 e facendo così restituire un errore alla funzione `request_firmware`. Se tutto è andato bene ed è quindi stato possibile reperire il firmware allora il driver ha tutto ciò che gli serve nella struttura `fw_entry` appositamente popolata con l'immagine e può procedere con il caricamento dell'immagine all'interno del device desiderato. Una volta caricato potrà effettuare una chiamata alla funzione `release_firmware(fw_entry)` rilasciando l'immagine e le relative risorse.

3.1.3 Distribuzione dei firmware

In questa sezione verranno analizzate le possibili modalità di distribuzione dei firmware prendendo come caso di studio le distribuzioni Debian/ Ubuntu.

Nei sistemi Linux i firmware possono essere distribuiti da diverse origini. Alcuni firmware sono già contenuti nei kernel Linux sin dalla loro installazione, altri possono essere reperibili da fonti on-line mentre molti altri, non avendo

una licenza di distribuzione libera, non possono essere distribuiti liberamente come il resto del sistema.

Nelle distribuzioni Ubuntu i firmware possono essere ottenuti dalle seguenti fonti:

1. linux-image package, package contenente il kernel linux e i firmware autorizzati.
2. linux-firmware package, ulteriore package contenente firmware autorizzati.
3. linux-nonfree package, in questo package sono contenuti i firmware con licenze commerciali.
4. un package driver separato
5. altre fonti come siti web, CD o email.

Le prime due fonti (linux-image package e linux-firmware package) sono installate di default dall'installer della distribuzione mentre le restanti sono reperibili dall'utente. Tutti i file contenenti i firmware sono contenuti all'interno della cartella `/lib/firmware`, proprio per questo motivo il kernel ogni qualvolta debba cercare firmware da reindirizzare all'opportuno driver analizza completamente la suddetta cartella.

3.1.4 Progetto FWUPD

Fwupd [15] è un progetto open-source nato nel 2015 che si pone come scopo quello di automatizzare l'aggiornamento dei firmware nelle macchine Linux in modo sicuro ed affidabile. Il progetto in questione è molto interessante in quanto, al giorno d'oggi, l'aggiornamento di un qualsiasi firmware richiede sia che il produttore fornisca informazioni riguardanti i diversi aggiornamenti specifici per l'hardware, sia un meccanismo che consenta l'installazione del firmware, in pratica non esiste ancora un metodo standard che possa comprendere tutti gli aggiornamenti di questo tipo.

Questo sistema è stato progettato come un servizio D-Bus [16] il quale viene attivato ogni qualvolta sia richiesto. D-Bus è un demone open-source di comunicazione tra processi (IPC) ed offre alle applicazioni la possibilità di comunicare tra loro e con il sistema, riuscendo a captare anche eventi generati dal kernel come, per esempio, il collegamento di nuove periferiche.

Essendo nato da poco, fwupd, ha ancora un numero limitato di firmware supportati, questo è causato anche dal fatto che i produttori di firmware che aderiscono o vogliono aderire a questo progetto devono effettuare l'upload dei

loro firmware i quali saranno sottoposti a diversi controlli prima di poter essere pubblicati ed utilizzati dagli utilizzatori finali.

Può risultare interessante spendere due parole sullo standard adottato per l'identificazione dei firmware. Un produttore che voglia effettuare l'upload dei propri firmware, oltre all'immagine del firmware stesso, deve allegare un file XML che segua una determinata struttura, così facendo, il servizio, è in grado di memorizzare e gestire tutti i firmware con più facilità ed omogeneità. Per quanto riguarda la sicurezza che questo servizio offre, ogni aggiornamento firmware da loro proposto è firmato e accompagnato da un checksum inoltre forniscono un file XML dal quale è possibile ricavare tutte le informazioni riguardanti l'aggiornamento stesso.

Il progetto in questione è molto interessante e, con il giusto supporto dei produttori, potrebbe prendere piede in non molto tempo.

3.2 Firmware nei dischi rigidi

In questa sezione verranno trattati i firmware dei dischi rigidi. Inizialmente verrà mostrato dove gli odierni dischi rigidi memorizzano i firmware per poi analizzare due interessanti casi di dump ed infine citare un importante attacco che ha coinvolto questa componente.

Risulta triviale dire che un dispositivo come il disco rigido assume rilevante importanza all'interno dei sistemi odierni in quanto funge come contenitore della maggior parte di informazioni presenti in un sistema e, come vedremo in seguito, un attacco a questa componente potrebbe causare ingenti danni.

3.2.1 Locazione e caricamento del firmware

Gli odierni dischi rigidi sono composti da diverse componenti una delle quali può essere vista come la "scheda madre" dell'hard disk, la componente a cui stiamo facendo riferimento prende il nome di Printed Circuit Board (PCB). Quest'ultima è composta da una serie di chip e da una piccola, ma molto importante, parte del microcodice del disco rigido. In Figura 3.1 è possibile notare come la parte mancante del firmware sia situata all'interno dei dischi dell'hard disk, in un'area riservata e non accessibile.

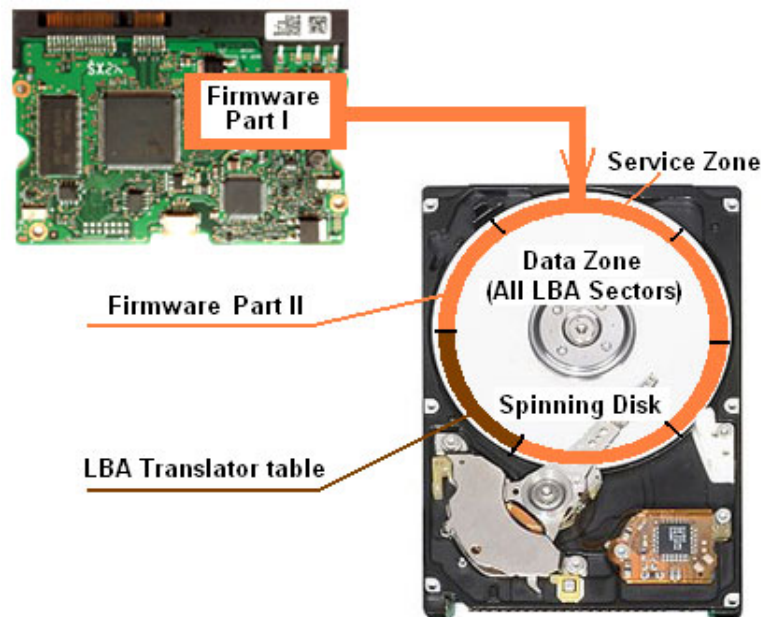


Figura 3.1: Posizionamento firmware in un generico hard disk.
Immagine tratta da [38]

Questa area viene anche chiamata Service Area, ed è inoltre utilizzata per tutti i servizi e le informazioni utili ad un hard disk come le tabelle di configurazione, informazioni di sicurezza (password del disco, ecc.), informazioni sul disco (numero seriale, produttore, ecc..) e altro.

Quando il disco rigido viene avviato la prima porzione di firmware, contenuta nel PCB, avvia l'intero processo di avvio del sistema, quindi il firmware presente all'interno della Service Area viene caricato nella memoria del disco rigido. Successivamente, il codice appena caricato, verrà abbinato alla sua parte mancante presente all'interno del PCB e, se in questa fase di abbinamento si dovessero aver problemi, allora significa che probabilmente l'integrità del firmware in questione è stata violata.

La Service Area contenente tutti i moduli utili al disco rigido e parte del firmware è accessibile solo attraverso opportuni comandi ATA[40] che variano da produttore a produttore e difficilmente reperibili. Risulta quindi molto complesso, ma non impossibile, eseguire operazioni in questa specifica area.

3.2.2 Casi di dump

Sarebbe sicuramente interessante riuscire ad estrapolare il firmware di una componente così rilevante per poi sottoporre l'output restituito ad un sistema in grado di verificare la sua autenticità ma, dalle precedenti analisi, è facile

capire come eseguire operazioni di questo tipo non sia un lavoro facile in quanto ci troviamo in un mondo molto frammentato e privo di standard.

Un software reperibile in rete che offre una funzionalità di questo tipo è PC3000 [17]. Il tool in questione è frutto di circa 20 anni di esperienza nell'ambito del recupero dati e riparazione di dischi rigidi ed offre numerose funzionalità, tra cui:

1. Diagnosi completa del disco rigido.
2. Verifica e recupero della Service Area.
3. Lettura e scrittura del firmware .
4. Modifica delle tabelle di configurazione.
5. Modifica delle impostazioni e dei parametri di identificazione.
6. Reset completo del disco rigido.

Come si può notare, tra le tante funzionalità offerte è presente anche la lettura e la scrittura del firmware. Questo software, inoltre, supporta una gran vastità di modelli dei maggiori produttori di dischi rigidi odierni; a fronte delle analisi effettuate precedentemente è facile capire come un tool che dia tutte queste possibilità debba conoscere il set di comandi ATA utilizzati da ogni produttore per poter interagire con la Service Area dei dischi rigidi. Purtroppo PC3000 si può reperire in commercio per un prezzo che si aggira intorno ai 4500\$, il quale risulta essere inadatto ai nostri scopi.

Un ulteriore metodo utilizzato per effettuare il dump del firmware di un Hard Disk Drive è attraverso l'utilizzo dell'interfaccia JTAG [18]. JTAG altro non è che un protocollo utilizzato da più di 200 imprese produttrici di circuiti integrati e circuiti stampati utilizzato per il test funzionale dei dispositivi. Attraverso questa interfaccia possono essere attivate varie funzioni sviluppate dai costruttori del componente, una delle più interessanti è la possibilità di testare, leggere e programmare il microcode del dispositivo. È proprio questa funzionalità che può essere sfruttata per effettuare il dump del firmware. Per effettuare un dump attraverso JTAG è necessario disassemblare l'hard disk dal proprio sistema e avere una buona conoscenza del protocollo JTAG, cioè essere in grado di riconoscere i pin del device utilizzati da questo protocollo ed i comandi utilizzabili. In rete è possibile trovare diversi articoli riguardanti questo metodo di dump uno² dei quali viene effettuato da un ragazzo il quale mostra come sia possibile infettare il firmware di un disco rigido partendo da

²<http://www.malwaretech.com/2015/04/hard-disk-firmware-hacking-part-1.html>

una copia del firmware che viene recuperata attraverso l'interfaccia appena descritta.

3.2.3 Equation Group

Un caso di attacco interessante da analizzare è quello effettuato da un gruppo di hacker denominato Equation Group [2]. Il gruppo in questione è considerato uno dei più sofisticati gruppi di cyber-attacco nel mondo ed è coinvolto in numerose operazioni illecite. Diverse ricerche effettuate dalla Kaspersky [19] hanno portato alla luce diversi malware da loro sviluppati ed utilizzati, tra cui:

- **EquationDrug** - Piattaforma di attacco molto complessa in grado di supportare un sistema modulare di plugin aggiornabile dinamicamente dagli attaccanti.
- **DoubleFantasy** - Un validatore progettato per identificare le vittime.
- **TripleFantasy** - Sistema completo utilizzato per effettuare attacchi basati su backdoor.
- **GrayFish** - La piattaforma d'attacco più sofisticata dell'Equation Group, è eseguita in fase di avvio del sistema ed è in grado di sopravvivere a fronte di formattazioni e riavvii del sistema stesso.
- **Fanny** - Un sistema utilizzato per raccogliere informazioni riguardanti vittime risidenti nel Medio Oriente e Asia.
- **EquationLaser** - Uno dei primi tool di attacco da loro sviluppato, usato tra il 2001 ed il 2004.

Tra i numerosi attacchi che il suddetto gruppo è riuscito a portare a termine noi ci concentreremo su uno in particolare, il quale coinvolge i firmware dei dischi rigidi.

La piattaforma d'attacco **EquationDrug** precedentemente citata offre due funzionalità molto interessanti: la possibilità di riprogrammare il firmware del disco rigido con un payload modificato appositamente dal gruppo e, inoltre, fornisce una serie di API memorizzate all'interno di settori nascosti del disco rigido. Un attacco di questo genere può comportare due grandi problemi:

- Un'estrema persistenza dell'infezione a fronte di formattazioni del disco o del sistema operativo.
- Un'area di memorizzazione accessibile da remoto, persistente ed invisibile all'interno del disco rigido.

Le ricerche effettuate dalla Kaspersky hanno inoltre dimostrato che il sistema in questione è in grado di infettare dischi rigidi di produttori come Western Digital, Samsung, Seagate, Maxtor, Toshiba e Micron. Una volta che il malware è riuscito ad insidiarsi nel computer della vittima è in grado di riconoscere il modello del disco rigido installato e riprogrammare il suo firmware utilizzando una serie di comandi ATA specifici per poi riuscire ad accedere da remoto e creare partizioni invisibili su cui poter memorizzare dati provenienti dalla macchina della vittima.

3.3 Firmware BIOS

In questa sezione analizzeremo più in dettaglio i firmware dei BIOS. Inizialmente verrà trattata la sua evoluzione in UEFI ponendo attenzione al lato sicurezza che questa implementazione ha cercato di apportare, poi passeremo ad una rassegna di un caso di attacco il quale ha coinvolto questa componente ed infine analizzeremo un interessante sistema progettato da VirusTotal [20].

3.3.1 BIOS e UEFI

Il Basic Input/Output System (BIOS) è composto da un insieme di routine software contenenti tutte le funzionalità per controllare la componentistica installata in un sistema. All'avvio del sistema esso è il primo componente ad essere avviato e si occupa dell'inizializzazione e del controllo di tutte le componenti hardware installate sulla macchina, dopo aver effettuato questa operazione si occupa di preparare tutto il necessario affinché il sistema operativo possa avviarsi correttamente per poi avviarlo. Per fare in modo che il suo contenuto sia persistente ma che sia anche possibile eseguire aggiornamenti di questa componente, essa è posizionata all'interno di memorie riprogrammabili non volatili.

Con il passare degli anni l'evoluzione tecnologica ha portato a richiedere funzionalità per le quali i BIOS non erano progettati (e.g. gestione remota della sicurezza, monitoring della potenza e della temperatura del sistema, ecc..). I BIOS, come si può facilmente intuire, hanno preso vita molteplici anni fa e quindi non erano progettati ed implementati per essere estesi. Per questi motivi nel 2003 Intel ha deciso di presentare una nuova tecnologia chiamata Unified Extensible Firmware Interface (UEFI), che si poneva come obiettivo quello di sostituire gradualmente i vecchi BIOS. Una miglioria che questo sistema ha apportato è la risoluzione al problema riguardante le limitazioni di memorizzazione che il BIOS imponeva, quest'ultimo utilizzava il Master Boot Record (MBR) per salvare tutti i dati mentre UEFI utilizza la GUID Partition

Table. La differenza sostanziale tra questi due elementi risiede nella loro capacità di memorizzazione, il Master Boot Record utilizza entry dalla dimensione di 32-bit limitando il numero di partizioni totali a 4 mentre il metodo adottato dall'UEFI utilizza entry estese a 64-bit offrendo quindi una capacità di memorizzazione più ampia. Un'ulteriore caratteristica apportata da UEFI è la capacità di ridurre notevolmente i tempi di caricamento del sistema operativo, questo incremento della velocità è dovuto al fatto che UEFI è stato progettato per essere modulare e, durante l'avvio del sistema, carica solo i moduli strettamente necessari al corretto funzionamento del sistema senza preoccuparsi di tutto ciò che reputa inutile.

Un importante funzionalità apportata dall'UEFI è sicuramente il Secure Boot, un meccanismo che nasce con l'obiettivo di garantire l'integrità della piattaforma e controllare tutti gli stati del processo di avvio del sistema. Durante la prima fase di boot del sistema esso è il primo elemento ad essere avviato ed è composto da tre elementi chiave che andiamo ora a descrivere.

- **Verifica delle immagini** - Una delle principali innovazioni che l'UEFI ha portato è la possibilità di utilizzare la crittografia per la verifica delle immagini che esso contiene attraverso un'infrastruttura a chiave pubblica. Le chiavi utilizzate sono memorizzate all'interno del meccanismo di Secure Boot e seguono la seguente gerarchia:
 1. *Platform Key (PK)*: È la chiave radice ed è specifica per ogni sistema, l'accesso a questa chiave (generalmente dato ai produttori) consente la modifica di tutte le chiavi facenti parte della gerarchia.
 2. *Key Exchange Key (KEK)*: Il proprietario di questa chiave (generalmente il produttore del SO) può effettuare aggiornamenti sui database di chiavi db/dbx.
 3. *db* - database contenente tutti i certificati e valori di hash leciti.
 4. *dbx* - database contenente i certificati e i digest proibiti.
- **Variabili di autenticazione UEFI** - Esse forniscono un mezzo per garantire l'integrità delle diverse variabili che potranno essere memorizzate all'interno dei diversi database. Come si può notare in Figura 3.2 alle variabili di autenticazione vengono aggiunti in testa dei dati di autenticazione che verranno utilizzati per garantire l'integrità.

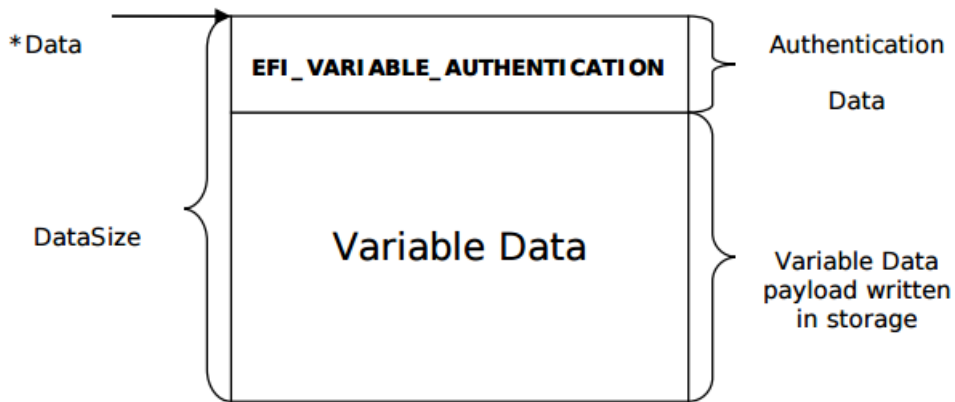


Figura 3.2: Layout di una variabile di autenticazione.

- **Aggiornamento sicuro dell'UEFI** - Questa componente è usata per verificare l'aggiornamento delle immagini dei firmware. L'intero processo di verifica avviene mediante l'utilizzo di firme digitali.

Essendo il primo elemento ad essere avviato esso è in grado di controllare tutti i binari in suo possesso controllando l'esistenza dei relativi valori di hash all'interno del database dbx. Se la ricerca ha esito positivo allora viene generato un errore ed il relativo binario non verrà caricato altrimenti, il sistema, effettuerà un'ulteriore ricerca nel database contenente i valori di hash leciti. Se quest'ultima ricerca termina senza che il valore sia stato trovato allora il binario non potrà essere caricato e sarà comunque generato un messaggio di errore mentre, se il valore di hash viene trovato all'interno di questo database, il binario potrà essere caricato senza problemi. Il meccanismo appena descritto offre sicuramente maggiore sicurezza rispetto al normale funzionamento del BIOS ma ricerche come [21] hanno comunque dimostrato come sia possibile, attraverso meccanismi non banali, riuscire ad infettare sistemi con il Secure Boot abilitato.

3.3.2 LightEater

Un caso interessante di attacco informatico che ha preso di mira il BIOS è quello presentato da due ricercatori durante la conferenza CanSecWest dello scorso 2015. Ciò che hanno fatto è stato mostrare come un generico BIOS possa essere compromesso in soli due minuti attraverso un malware da loro progettato che prende il nome di LightEater [22]. Il malware in questione è in grado di sfruttare alcune vulnerabilità del BIOS dirottando il System Management Mode (SMM). Il SMM è una modalità operativa dei processori Intel utilizzata

dal BIOS per determinate funzioni le quali necessitano di privilegi di sistema di alto livello (e.g. power management, controllo della velocità della CPU, controllo delle ventole, ecc). Attraverso l'attacco ad una componente così importante sono riusciti ad ottenere dei privilegi di alto livello attraverso i quali sono riusciti a riscrivere il contenuto del BIOS installando un sistema in grado di fornire loro un entry point. Una volta che il sistema è stato infettato il malware è in grado di leggere completamente il contenuto della memoria della macchina e fornire il tutto agli attaccanti. È facile capire come un malware di questo tipo possa causare danni notevoli, soprattutto se, sulle macchine infette, il sistema operativo in esecuzione sia Tails [23]. Questo comporterebbe un accesso, da parte degli attaccanti, alle chiavi crittografiche utilizzate come forma di criptaggio garantendo quindi l'accesso a tutti i dati e i documenti presenti in memoria.

3.3.3 Sistema Virustotal

Un recente articolo [24] pubblicato da VirusTotal ha mostrato come essi siano stati in grado di sviluppare un sistema capace, data l'immagine del firmware del BIOS, di estrarre tutti gli eseguibili ed i file contenuti in esso per poi effettuare un'analisi dettagliata di ognuno di essi ed avere una visione complessiva di tutto ciò che è contenuto all'interno del proprio BIOS. L'articolo comprende anche scansioni effettuate su diversi firmware dalle quali si può notare come molti di questi comprendano non solo eseguibili BIOS ma anche eseguibili Windows potenzialmente dannosi, alcuni dei quali classificati come Trojan da molti antivirus, questo conferma quanto precedentemente detto sulla sicurezza dei BIOS ed i diversi binari da esso contenuti.

Sarebbe quindi interessante poter effettuare un dump del firmware del BIOS per poi sottoporlo al sistema appena descritto, è proprio per questo motivo che ora verranno trattati due diversi tool da me utilizzati per cercare di effettuare questa operazione.

3.3.4 Tool per dump del firmware

Prima di partire con l'analisi dei tool è importante capire come il dump del firmware di una componente così delicata sia rischioso, se per qualche motivo durante questa fase qualcosa dovesse andare storto (e.g. shutdown della macchina, tool non configurati ad-hoc) il rischio che il BIOS possa essere corrotto e non più funzionante è molto alto, occorre quindi prestare molta attenzione.

Il primo tool che ho utilizzato per cercare di effettuare il dump del firmware del BIOS prende il nome di Flashrom [25]. Flashrom è progettato per

identificare, leggere, scrivere, verificare e resettare diversi chip. Naturalmente il tool in questione non è in grado di supportare tutti i chipset³ esistenti in commercio, proprio per questo motivo dal sito web dei produttori del tool è possibile sfogliare la lista di chipset supportati, per questo motivo prima di provare ad effettuare il dump ho deciso di cercare informazioni riguardanti il chipset installato sul mio sistema. Per effettuare questa operazione ho sfruttato il comando *lspci*⁴ già installato sulla mia piattaforma. Come si può notare in Figura 3.3 l'output restituito dall'esecuzione del comando è riuscito a fornirmi diverse informazioni tra le quali è presente anche l'Host Bridge che il mio sistema possiede.

```
doomdiskday@doomdiskday:~$ lspci
00:00.0 Host bridge: Intel Corporation Xeon E3-1200 v3/4th Gen Core Processor DRAM Controller (rev 06)
00:01.0 PCI bridge: Intel Corporation Xeon E3-1200 v3/4th Gen Core Processor PCI Express x16 Controller (rev 06)
00:02.0 VGA compatible controller: Intel Corporation 4th Gen Core Processor Integrated Graphics Controller (rev 06)
00:03.0 Audio device: Intel Corporation Xeon E3-1200 v3/4th Gen Core Processor HD Audio Controller (rev 06)
00:14.0 USB controller: Intel Corporation 8 Series/C220 Series Chipset Family USB XHCI (rev 05)
00:16.0 Communication controller: Intel Corporation 8 Series/C220 Series Chipset Family MEI Controller #1 (rev 04)
00:1a.0 USB controller: Intel Corporation 8 Series/C220 Series Chipset Family USB EHCI #2 (rev 05)
00:1b.0 Audio device: Intel Corporation 8 Series/C220 Series Chipset High Definition Audio Controller (rev 05)
00:1c.0 PCI bridge: Intel Corporation 8 Series/C220 Series Chipset Family PCI Express Root Port #4 (rev d5)
00:1c.4 PCI bridge: Intel Corporation 8 Series/C220 Series Chipset Family PCI Express Root Port #5 (rev d5)
00:1d.0 USB controller: Intel Corporation 8 Series/C220 Series Chipset Family USB EHCI #1 (rev 05)
00:1f.0 ISA bridge: Intel Corporation HM86 Express LPC Controller (rev 05)
00:1f.2 SATA controller: Intel Corporation 8 Series/C220 Series Chipset Family 6-port SATA Controller 1 [AHCI mode] (rev 05)
00:1f.3 SMBus: Intel Corporation 8 Series/C220 Series Chipset Family SMBus Controller (rev 05)
01:00.0 VGA compatible controller: NVIDIA Corporation GK107M [GeForce GT 755M] (rev a1)
07:00.0 Ethernet controller: Qualcomm Atheros QCA8171 Gigabit Ethernet (rev 10)
08:00.0 Network controller: Broadcom Corporation BCM4313 802.11bgn Wireless Network Adapter (rev 01)
```

Figura 3.3: Output restituito da *lspci*.

Partendo da questa informazione ed effettuando diverse ricerche sono riuscito a trovare il chipset corrispondente all'Host Bridge in questione che risulta essere il modello C226 della famiglia Intel.

Cercando il suddetto modello tra i chipset supportati da Flashrom ho poi scoperto che non è ancora stato testato ma questo non dovrebbe causare grossi problemi e il tool potrebbe funzionare, per questi motivi ho deciso di eseguire il test.

Flashrom è utilizzabile da linea di comando e l'unico parametro di cui necessita è il *programmer* cioè la famiglia di chip su cui deve agire. Nel mio caso, tra le diverse opzioni, il *programmer* selezionato è stato l'*internal* in quanto è quello contenente il chipset installato sul mio sistema. Purtroppo l'esecuzione del comando ha restituito un messaggio di errore (Figura 3.4) secondo il quale avrei provato ad eseguire il tool in un laptop non supportato e che mi avvertiva del fatto che anche solo una semplice lettura del firmware avrebbe potuto causare problemi.

³il chipset è quell'insieme di circuiti integrati di una scheda madre i quali si occupano dello smistamento delle informazioni passanti dal bus di sistema, nelle moderne architetture esso è composto da due componenti principali, l'Host Bridge ed il SouthBridge

⁴*lspci* è un comando utilizzato nelle piattaforme Unix-like per elencare tutte le periferiche e i bus PCI installati sul sistema stesso

```
doomdiskday@doomdiskday:~$ sudo flashrom --programmer internal
[sudo] password for doomdiskday:
flashrom v0.9.7-r1852 on Linux 4.2.0-35-generic (x86_64)
flashrom is free software, get the source code at http://www.flashrom.org

Calibrating delay loop... OK.
Active config mode, unknown reg 0x20 ID: 85.
Please send the output of "flashrom -V -p internal" to
flashrom@flashrom.org with W836xx: your board name: flashrom -V
as the subject to help us finish support for your Super I/O. Thanks.
=====
WARNING! You seem to be running flashrom on an unsupported laptop.
Laptops, notebooks and netbooks are difficult to support and we
recommend to use the vendor flashing utility. The embedded controller
(EC) in these machines often interacts badly with flashing.
See the manpage and http://www.flashrom.org/Laptops for details.

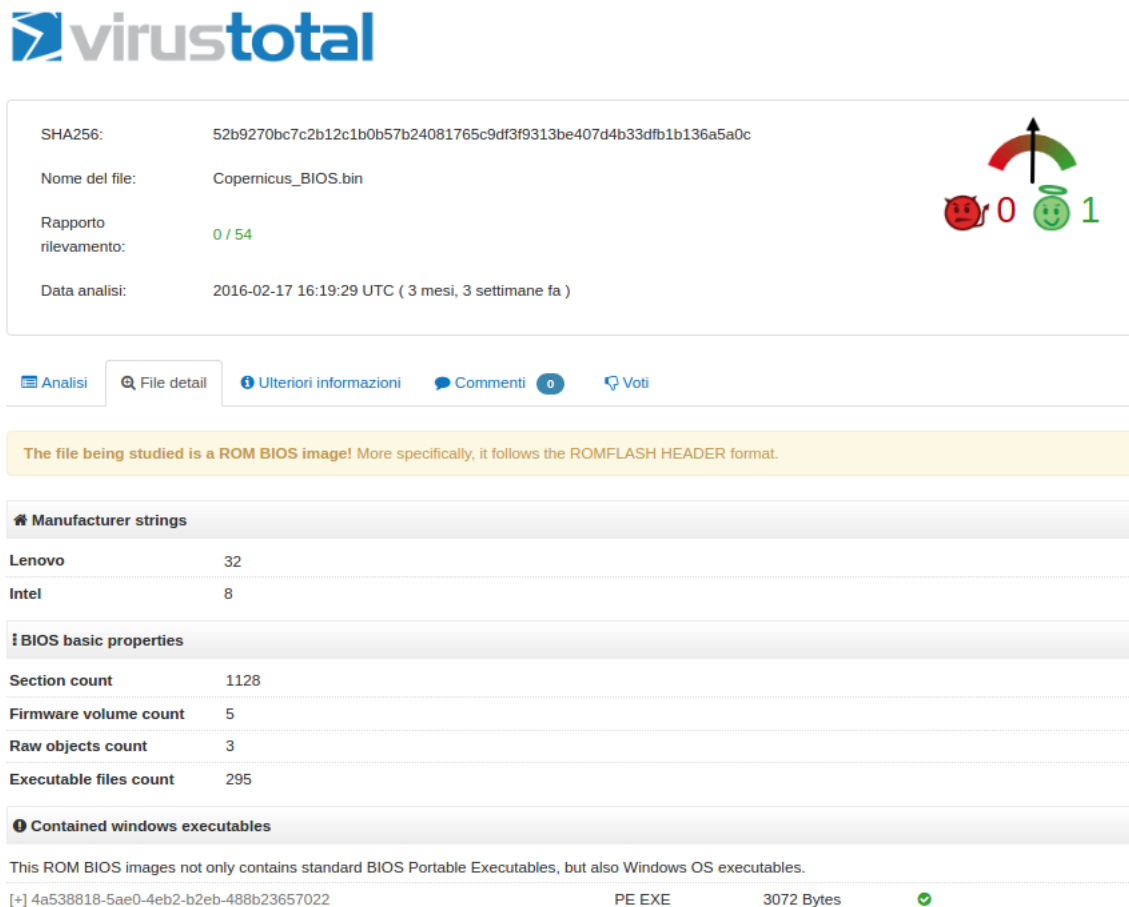
If flash is shared with the EC, erase is guaranteed to brick your laptop
and write may brick your laptop.
Read and probe may irritate your EC and cause fan failure, backlight
failure and sudden poweroff.
You have been warned.
=====
Aborting.
Error: Programmer initialization failed.
```

Figura 3.4: Errore Flashrom.

Eseguendo diverse ricerche ho scoperto che questa avvertenza poteva essere bypassata attraverso un parametro aggiuntivo ma ho comunque deciso di non testare il tool onde evitare problemi.

Il secondo software testato è Copernicus [26]. Copernicus è un tool in grado di valutare la sicurezza del BIOS in macchine Intel. Tra le funzionalità che offre da anche la possibilità di eseguire il dump del BIOS ma, purtroppo, è eseguibile solo in sistemi Windows indi per cui per eseguire il test ho dovuto accedere alla partizione Windows della mia macchina. Al contrario del precedente tool, Copernicus non fornisce una lista di chipset supportati ma offre solamente un manuale di utilizzo sul quale vengono spiegati tutti gli step da eseguire per effettuare il dump del firmware. Dopo aver seguito tutti i passi che la guida proponeva sono riuscito finalmente ad estrapolare l'immagine del firmware senza incappare in particolari problemi.

Successivamente ho deciso di effettuare l'upload dell'immagine al servizio VirusTotal precedentemente descritto per verificare la sicurezza relativa al mio BIOS. Come si può notare in Figura 3.5 il risultato fornitomi dalla scansione è privo di elementi dannosi.



The image shows a screenshot of the VirusTotal analysis page for a file named 'Copernicus_BIOS.bin'. The file's SHA256 hash is 52b9270bc7c2b12c1b0b57b24081765c9df3f9313be407d4b33dfb1b136a5a0c. The analysis report shows 0 detections out of 54 engines. The analysis was performed on 2016-02-17 at 16:19:29 UTC, 3 months and 3 weeks ago. A yellow banner indicates that the file is a ROM BIOS image. The 'Manufacturer strings' section lists 'Lenovo' (32) and 'Intel' (8). The 'BIOS basic properties' section shows 1128 sections, 5 firmware volume counts, 3 raw objects, and 295 executable files. The 'Contained windows executables' section shows one PE EXE file of 3072 bytes.

SHA256:	52b9270bc7c2b12c1b0b57b24081765c9df3f9313be407d4b33dfb1b136a5a0c
Nome del file:	Copernicus_BIOS.bin
Rapporto rilevamento:	0 / 54
Data analisi:	2016-02-17 16:19:29 UTC (3 mesi, 3 settimane fa)

The file being studied is a ROM BIOS image! More specifically, it follows the ROMFLASH HEADER format.

Manufacturer strings

Lenovo	32
Intel	8

BIOS basic properties

Section count	1128
Firmware volume count	5
Raw objects count	3
Executable files count	295

Contained windows executables

This ROM BIOS images not only contains standard BIOS Portable Executables, but also Windows OS executables.

[+] 4a538818-5ae0-4eb2-b2eb-488b23657022	PE EXE	3072 Bytes	✓
--	--------	------------	---

Figura 3.5: Porzione di analisi effettuata da VirusTotal

3.4 Tracciamento Hardware

Procediamo ora con un'analisi dettagliata riguardante il tracciamento dell'hardware in un generico sistema Linux. Inizialmente verrà analizzato un tentativo, da parte di Intel, di identificazione univoca dei processori da loro prodotti poi passeremo ad una rassegna dei comandi disponibili nei moderni sistemi Linux ed utilizzabili per recuperare informazioni relative all'hardware installato, infine verranno analizzati nel dettaglio due dei comandi descritti.

Prima di proseguire definiamo meglio il concetto di tracciamento hardware e la relativa importanza. Con il suddetto concetto intendiamo la possibilità di tenere costantemente traccia dell'hardware installato sul proprio sistema e riuscire a rilevare eventuali variazioni le quali potrebbero essere lecite (e.g. upgrade di una determinata componente) oppure illecite (e.g. sostituzione di una componente *trusted* con una alterata). L'importanza di riuscire ad

avere un tracciamento hardware all'interno del proprio sistema è alla base del concetto di integrità in quanto non è possibile avere un sistema *trusted* senza poter fare affidamento sull'hardware installato.

3.4.1 PSN Intel

Un caso interessante da citare è sicuramente il tentativo da parte di Intel di inserire all'interno dei propri processori un identificativo univoco. Con la produzione dei primi processori Pentium III, Intel ha introdotto una nuova funzionalità chiamata Processor Serial Number [27] che si pone come scopo quello di identificare univocamente i microprocessori sui quali viene installato e, sempre secondo i loro obiettivi, non da la possibilità di essere modificato ma è solo accessibile in lettura. Il PSN è un numero composto da 96-bit e stampato nel silicio del processore durante la sua produzione, è proprio questo il motivo per il quale non può subire variazioni. I primi 32-bit forniscono informazioni riguardanti la famiglia del processore mentre i rimanenti 64 bit sono differenti da processore a processore e sono utilizzati come forma di identificazione univoca. L'Intel, inoltre, forniva assieme al processore un software da loro progettato attraverso il quale l'utente aveva la possibilità di abilitare/disabilitare il PSN e di leggere il suo valore, le stesse operazioni potevano essere effettuate direttamente dal BIOS. Secondo l'Intel questo meccanismo avrebbe apportato molti benefici, tra cui:

- Nascita di sistemi di identificazione e autenticazione più sicuri.
- Maggiore sicurezza dei dati.
- Una migliore gestione delle violazioni delle licenze software.

L'idea iniziale di Intel era quella di commercializzare i processori Pentium III con il PSN abilitato ma, proprio questa motivazione, ha causato l'insurrezione dei membri della Science and Technology Options Assessment (STOI)⁵ i quali sostenevano che attraverso Processor Serial Number sarebbe stato facile tracciare i movimenti degli utilizzatori sul web infrangendo la loro privacy, oltre a questo è stata anche messa in discussione la sicurezza offerta da questo meccanismo. Effettivamente il PSN è sì stampato su silicio ma la relativa lettura avviene o attraverso un software oppure attraverso il BIOS ma, sfortunatamente, la maggior parte degli utenti non è familiare con quest'ultima componente e sicuramente preferisce l'utilizzo di uno strumento software il quale però potrebbe essere facilmente soggetto ad attacchi. Inoltre ricordiamo che, anche se

⁵La Science and Technology Options Assessment è un comitato di membri del parlamento europeo che si occupa di tutti i problemi relativi alle tecnologie in commercio.

con una probabilità minore, anche i BIOS potrebbero subire attacchi. A seguito di questa denuncia l'Intel decise di non implementare questo meccanismo e lo mise quindi fuori dal commercio. A parer mio l'idea in questione sarebbe stata molto interessante ma, se avesse preso piede, avrebbe sicuramente dovuto evolvere il sistema di sicurezza per limitare al minimo la possibilità di subire attacchi.

3.4.2 Analisi dei comandi

In questa sezione saranno analizzati diversi comandi che possono essere utilizzati per l'estrapolazione di informazioni riguardanti l'hardware installato su una generica macchina Linux per poi selezionare ed analizzare nel dettaglio quelli che, a seguito di un confronto, risultano migliori.

Il primo tool preso in analisi è `dmidecode` [28], questo comando, a differenza di molti altri in circolazione, riesce ad recuperare le informazioni relative all'hardware richiedendole direttamente al BIOS sfruttando gli standard SMBIOS/DMI [29]. Offrendo questo tipo di interazione l'attendibilità delle informazioni restituite è molto alta. Eseguendo diversi test sulla mia macchina e analizzando l'output restituito (Figura 3.6) posso affermare che il tool è in grado di restituire un numero di informazioni alquanto elevato.

```
doomdiskday@doomdiskday:~$ sudo dmidecode
[sudo] password for doomdiskday:
# dmidecode 2.12
# SMBIOS entry point at 0x8cebef98
SMBIOS 2.7 present.
69 structures occupying 2938 bytes.
Table at 0x8CEBC000.

Handle 0x0000, DMI type 0, 24 bytes
BIOS Information
    Vendor: LENOVO
    Version: 74CN44WW(V3.05)
    Release Date: 09/18/2013
    Address: 0xE0000
    Runtime Size: 128 kB
    ROM Size: 4608 kB
    Characteristics:
        PCI is supported
        BIOS is upgradeable
        BIOS shadowing is allowed
        Boot from CD is supported
        Selectable boot is supported
        EDD is supported
        Japanese floppy for NEC 9800 1.2 MB is supported (int 13h)
        Japanese floppy for Toshiba 1.2 MB is supported (int 13h)
        5.25"/360 kB floppy services are supported (int 13h)
        5.25"/1.2 MB floppy services are supported (int 13h)
        3.5"/720 kB floppy services are supported (int 13h)
        3.5"/2.88 MB floppy services are supported (int 13h)
        8042 keyboard services are supported (int 9h)
        CGA/mono video services are supported (int 10h)
        ACPI is supported
        USB legacy is supported
        BIOS boot specification is supported
        Targeted content distribution is supported
        UEFI is supported
    BIOS Revision: 0.44
    Firmware Revision: 3.5
```

Figura 3.6: Porzione di output restituito dopo l'esecuzione del comando dmidecode

Un ulteriore valore aggiunto a questo comando è dato dal fatto che è supportato dai seguenti sistemi operativi:

- Linux i386, x86-64, ia64
- FreeBSD i386, amd64
- NetBSD i386, amd64
- OpenBSD i386, amd64
- BeOS i386
- Cygwin i386
- Solaris x86

- Haiku i586

Risulta quindi molto portabile da un sistema ad un altro senza dover incappare in grossi problemi.

Un ulteriore comando utilizzabile per gli scopi proposti è `lscpu` [30]. Come si può notare in figura 3.7 l'output che questo comando restituisce, però, è limitato alle informazioni riguardanti la CPU installata sul sistema, per fare ciò non fa altro che controllare il contenuto del file `/proc/cpuinfo`

```
doomdiskday@doomdiskday:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):         1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz
Stepping:              3
CPU MHz:               2700.000
CPU max MHz:           3400,0000
CPU min MHz:           800,0000
BogoMIPS:              4788.37
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):    0-7
```

Figura 3.7: Output restituito dall'esecuzione del comando `lscpu`

Un'altro tool molto interessante è `lshw` [31]. Il comando in questione permette di estrarre informazioni dettagliate sulla configurazione hardware della macchina ed è in grado di restituire l'output in diversi formati (plain text, HTML o XML). Dopo aver eseguito il comando sulla mia macchina (Figura 3.8) posso affermare che il tool è in grado di restituire un numero elevato di informazioni riguardanti la maggior parte dei dispositivi installati (configurazione di memoria, configurazione della scheda madre, versione della CPU e relativa velocità, configurazione della cache, configurazione delle schede di rete e delle schede video).

```

doomdiskday@doomdiskday:~$ sudo lshw
[sudo] password for doomdiskday:
doomdiskday
  description: Computer
  width: 64 bits
  capabilities: smbios-2.7 vsyscall32
*-core
  description: Motherboard
  physical id: 0
*-memory
  description: System memory
  physical id: 0
  size: 11GiB
*-cpu
  product: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz
  vendor: Intel Corp.
  physical id: 1
  bus info: cpu@0
  size: 2420MHz
  capacity: 3400MHz
  width: 64 bits
  capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca c
mov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp x86-64 cons
tant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqd
q dtes64 monitor ds_cpl vmx est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 movbe popcnt tsc_de
adline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb pln pts dtherm tpr_shadow vnmi flex
priority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt cpufreq
*-pci
  description: Host bridge
  product: Xeon E3-1200 v3/4th Gen Core Processor DRAM Controller
  vendor: Intel Corporation
  physical id: 100
  bus info: pci@0000:00:00.0
  version: 06
  width: 32 bits
  clock: 33MHz

```

Figura 3.8: Porzione di output restituito dall'esecuzione del comando lshw

Effettuando diverse ricerche purtroppo ho scoperto che in alcuni sistemi potrebbe causare problemi con l'individuazione del numero di core presenti nella cpu⁶.

L'ultimo tool analizzato è hwinfo [32]. Questo comando, come i restanti, riporta informazioni riguardanti unità hardware come CPU, controller del disco rigido, controller USB, schede di rete e schede grafiche. Dopo aver testato il comando ed analizzato l'output restituito (Figura 3.9) mi sono accorto che genera un elevato numero di informazioni, molte delle quali risultano poco utili.

⁶Per core intendiamo il nucleo della CPU, cioè quello/quegli elementi che si occupano dell'elaborazione delle informazioni.

```
doomdiskday@doomdiskday:~$ hwinfo | more
===== start debug info =====
libhd version 21.19u (x86-64) [7688]
using /var/lib/hardware
kernel version is 4.2
---- /proc/cmdline ----
  BOOT_IMAGE=/boot/vmlinuz-4.2.0-35-generic.efi.signed root=UUID=5f6886ff-18e8-4374-9db7-1e8ed7a20c
0c ro quiet splash vt.handoff=7
---- /proc/cmdline end ----
debug = 0xff7ffff7
probe = 0x15938fcdaa17fcf9fffe (+memory +pci +isapnp +net +floppy +misc +misc.serial +misc.par +misc
c.floppy +serial +cpu +bios +monitor +mouse +scsi +usb -usb.mods +modem +modem.usb +parallel +paral
lel.lp +parallel.zip -isa -isa.isdn +isdn +kbd +prom +sbus +int +braille +braille.alva +braille.fhp
+braille.ht -ignx11 +sys -bios.vbe -isapnp.old -isapnp.new -isapnp.mod +braille.baum -manual +fb +
pppoe -scan +pcmcia +fork -parallel.imm +s390 +cpuemu -sysfs -s390disks +udev +block +block.cdrom +
block.part +edd +edd.mod -bios.ddc -bios.fb -bios.mode +input +block.mods +bios.vesa -cpuemu.debug
-scsi.noserial +wlan -bios.crc -hal +bios.vram +bios.acpi -bios.ddc.ports=0 +modules.pata -net.eep
om +x86emu=dump -max -lsrc)
shm: attached segment 2555921 at 0x7f455c6bc000
>> hal.1: read hal data
>> floppy.1: get nvram
>> floppy.2: klog info
>> bios.1: cmdline
>> bios.1.1: apm
>> bios.2: ram
  bios: 0 disks
>> bios.2: rom
>> bios.3: smp
---- BIOS data 0x00400 - 0x004ff ----
 400 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 430 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 490 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
 4a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 "....."
```

Figura 3.9: Porzione di output restituito dall'esecuzione del comando hwinfo

Avendo testato e confrontato i diversi output restituiti dai comandi sopracitati posso affermare che `dmidecode` è sicuramente il più completo, affidabile e portabile di tutti in quanto viene supportato da un gran numero di sistemi operativi e, interagendo direttamente con il BIOS, riesce a recuperare un gran numero di informazioni attendibili. Anche `hwinfo` risulta molto completo ma, leggendo il relativo manuale, ho notato che potrebbe non essere in grado di individuare tutti i dispositivi hardware presenti nel sistema, questo problema potrebbe essere causato dal fatto che il recupero delle informazioni avviene attraverso la libreria `libhd` che potrebbe non essere supportata in tutti i sistemi Linux.

3.4.3 Analisi dettagliata dmidecode

Analizziamo ora il comando `dmidecode` il quale è risultato essere il migliore tra quelli precedentemente descritti.

Dmidecode raggruppa i diversi devices secondo le categorie SMBIOS (figura 3.10) ed associa ad ognuna di esse un intero univoco.

0	BIOS
1	System
2	Base Board
3	Chassis
4	Processor
5	Memory Controller
6	Memory Module
7	Cache
8	Port Connector
9	System Slots
10	On Board Devices
11	OEM Strings
12	System Configuration Options
13	BIOS Language
14	Group Associations
15	System Event Log
16	Physical Memory Array
17	Memory Device
18	32-bit Memory Error
19	Memory Array Mapped Address
20	Memory Device Mapped Address
21	Built-in Pointing Device
22	Portable Battery
23	System Reset
24	Hardware Security
25	System Power Controls
26	Voltage Probe
27	Cooling Device
28	Temperature Probe
29	Electrical Current Probe
30	Out-of-band Remote Access
31	Boot Integrity Services
32	System Boot
33	64-bit Memory Error
34	Management Device
35	Management Device Component
36	Management Device Threshold Data
37	Memory Channel
38	IPMI Device
39	Power Supply
40	Additional Information

Figura 3.10: Categorizzazione device SMBIOS.
Tabella tratta da [39]

Come si può notare, la suddivisione in categorie offerta da SMBIOS è molto ampia e copre praticamente tutti i device possibilmente reperibili in un sistema.

Dmidecode, inoltre, offre un ulteriore livello di raggruppamento attraverso le seguenti keyword:

Tabella 3.1: Raggruppamento attraverso keyword offerto da dmidecode
Tabella tratta da [39]

Keyword	Types
bios	0
system	1, 12, 15, 23, 32
baseboard	2, 10, 41
chassis	3
processor	4
memory	5, 6, 16, 17
cache	7
connector	8
slot	9

Le keyword appena mostrate non fanno altro che accorpare alcune delle categorie di dispositivi precedentemente descritte. Passiamo ora ad analizzare l'output restituito dall'esecuzione del comando. Dmidecode genera, per ogni device riconosciuto, un record il quale segue la seguente struttura:

- **Handle** - identificativo univoco del device in questione utilizzato per referenziare diversi records tra loro.
- **Type** - intero che rappresenta la categoria di hardware a cui il device appartiene.
- **Size** - rappresenta la grandezza del record in questione.
- **Decoded Values** - insieme di valori dipendenti dal device analizzato i quali rappresentano le caratteristiche del dispositivo.

È proprio l'insieme dei decoded value che contiene tutte le informazioni riguardanti i diversi dispositivi.

Capitolo 4

Progettazione e sviluppo del sistema

In questa sezione tratteremo la progettazione e lo sviluppo del sistema posto come obiettivo. Partiremo definendo le funzionalità che il sistema dovrà offrire per poi passare ad un'analisi dettagliata della sua struttura, tutta la descrizione del lavoro svolto sarà accompagnata da diagrammi UML in modo da facilitare la comprensione.

4.1 Funzionalità

Per prima cosa andiamo ora a definire le funzionalità che lo strumento software dovrà svolgere. Come si può notare nel diagramma dei casi d'uso in Figura 4.1 il tool dovrà fornire all'utente le seguenti possibilità:

- Mostrare a video tutte le informazioni riguardanti l'hardware installato sulla macchina.
- Generare dei log¹ contenenti le informazioni della configurazione hardware.
- Controllare l'hardware installato sul sistema e confrontarlo con i file di log restituendo un risultato contenente le eventuali differenze.
- Mostrare a video tutte le informazioni riguardanti le diverse immagini dei firmware contenute nel sistema.
- Generare dei log contenenti le informazioni sui firmware contenuti sulla propria macchina.

¹Per log intendiamo dei file contenenti le registrazioni degli eventi.

- Controllare i firmware in possesso del sistema e confrontarli con i file di log rilevando eventuali differenze.

Offrendo queste funzionalità ad un generico utente, egli sarà in grado di riuscire a tenere costantemente traccia dei dispositivi hardware e delle immagini firmware presenti sul proprio sistema, e poter rilevare eventuali cambiamenti e/o manomissioni.

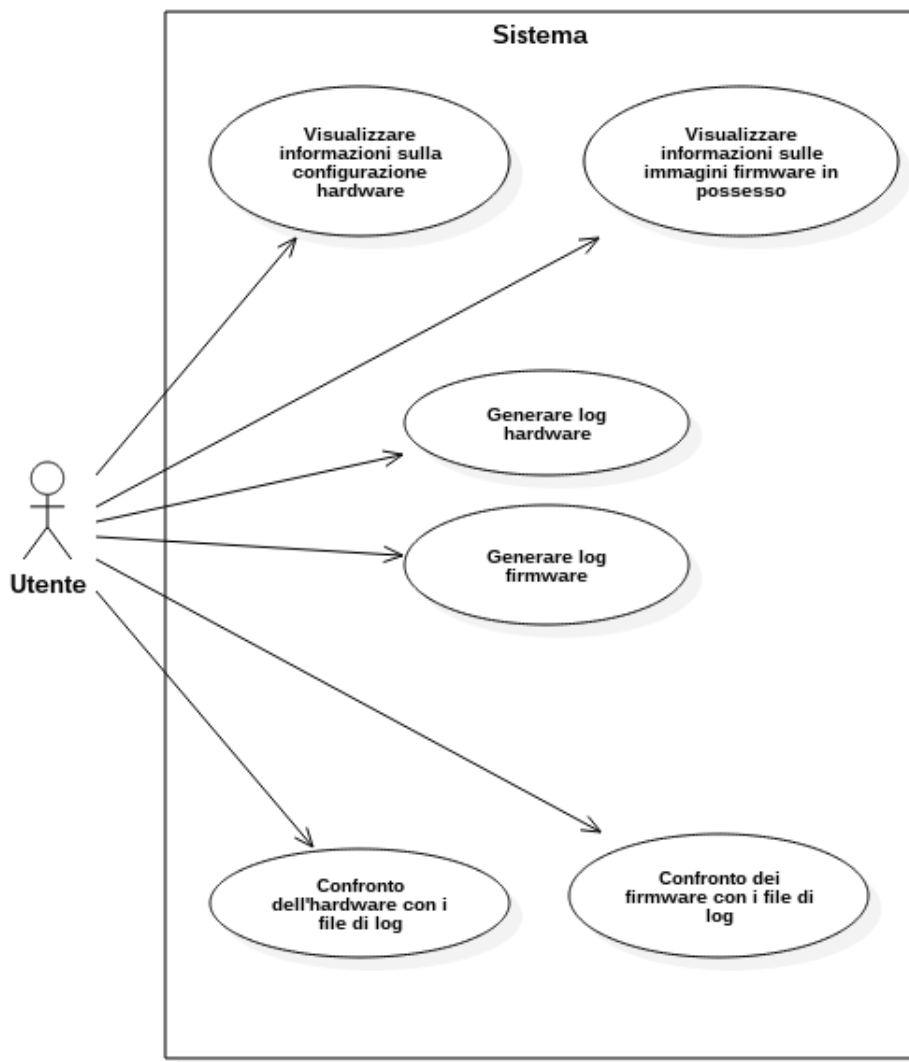


Figura 4.1: Diagramma dei casi d'uso del sistema.

4.2 Scelta del linguaggio e dell'IDE di sviluppo

Per quanto riguarda la scelta del linguaggio di programmazione utilizzato si è deciso di optare per un linguaggio ad oggetti come il C++ visto che offre un'alta riusabilità ed un buon livello di astrazione attraverso il quale sarà possibile modellare tutte le entità presenti nel sistema. L'IDE scelto per lo sviluppo del software è Eclipse [41] poichè precedentemente utilizzato per diversi progetti e ritenuto completo.

Un'ulteriore parentesi sulla scelta del sistema operativo va aperta, come detto nella sezione 3.1 il mondo Linux comprende una gran vastità di distribuzioni e, purtroppo, non offre una completa omogeneità per quanto riguarda l'organizzazione del FileSystem, per questi motivi ho deciso di sviluppare lo strumento software facendo riferimento al Filesystem Hierarchy Standard [12] il quale offre un'ampia copertura dei sistemi Linux odierni.

4.3 Suddivisione del sistema e panoramica

Come si può facilmente notare, il sistema, è composto da due macrosezioni, la prima riguardante le componentistiche hardware e la seconda riguardante le immagini firmware. Per questi motivi ho deciso di suddividere il sistema in due sottosistemi, sviluppando e testando un sistema alla volta. Da qui, la descrizione del lavoro, proseguirà inizialmente analizzando la porzione di sistema che si occupa della gestione dell'hardware e poi proseguirà con un'analisi del sottosistema firmware riuscendo a dare una visione complessiva di tutto il sistema. Entrambe le analisi appena citate definiranno inizialmente come raggiungere l'obiettivo e quali strumenti verranno utilizzati, poi passeranno ad un'analisi dettagliata di tutti gli elementi che compongono il sottosistema, ed infine verranno mostrati diversi test effettuati sulla mia macchina.

4.4 Progettazione e sviluppo del sottosistema hardware

Come precedentemente detto, questa porzione di sistema deve raggiungere i tre seguenti obiettivi principali:

1. Mostrare a video le informazioni riguardanti la configurazione hardware.
2. Generare dei log con le informazioni estrapolate.
3. Controllare eventuali differenze tra le componenti attualmente installate e le componenti contenute nei file di log.

Per raggiungere gli obiettivi appena citati sicuramente bisogna prima trovare un modo per recuperare il maggior numero di informazioni possibili dal sistema. Per conseguire quest'ultimo scopo le possibili opzioni sono diverse, si potrebbero utilizzare librerie reperibili in rete oppure utilizzare API che offrono, appunto, questa funzionalità. Effettuando diverse ricerche non sono riuscito a trovare alcun tipo di libreria o API in grado di svolgere il recupero di informazioni hardware ed ho quindi optato per l'utilizzo dei comandi Linux. Le analisi da me effettuate in precedenza (sezione 3.4.2) hanno classificato, tra un insieme ampio di comandi, `dmidecode` come il più adatto a svolgere questa funzionalità, per queste motivazioni la maggior parte delle informazioni che il tool dovrà estrarre sarà effettuata attraverso l'utilizzo di questo comando e la relativa gestione dell'output restituito.

Occorre spendere due parole anche sul tipo di device sui quali il tool dovrà essere in grado di lavorare. La prima versione del software sarà in grado di supportare tutti i dispositivi più delicati da un punto di vista della sicurezza e che risultano essere più soggetti a manomissioni che risultano essere:

- BIOS
- Schede di rete
- Schede video
- Memoria (RAM)
- Dischi Rigidi

Naturalmente si dovrà tenere conto degli sviluppi futuri e fare in modo che aggiungere dispositivi supportati non sia un problema.

Passiamo ora a definire la modellazione degli elementi del sottosistema.

4.4.1 Modellazione classi `HWDevice` e `Command`

Per modellare correttamente tutti i possibili device di un sistema si è deciso di creare un'interfaccia `HWDevice` la quale rappresenta un generico dispositivo hardware, uno specifico device, per essere tale, non dovrà fare altro che implementare questa interfaccia.

Come si può notare in Figura 4.2 l'interfaccia in questione offre un solo campo, `deviceInfos`, il quale altro non è che un array di `BaseProperty` che si occupa di rappresentare tutte le possibili informazioni di un generico device e che andremo presto a definire. Inoltre, l'interfaccia, offre un metodo `getProperty` il quale prende in ingresso un parametro indicante la posizione della

proprietà all'interno dell'array di informazioni del device e ritorna l'opportuna informazione.

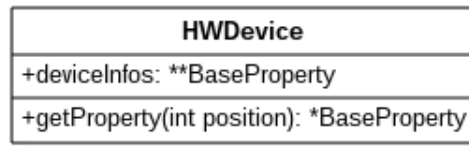


Figura 4.2: Diagramma delle classi di un generico device hardware.

Come appena detto la classe *BaseProperty* è designata per la rappresentazione di una generica proprietà di un dispositivo hardware. Prima di procedere con la sua descrizione è meglio spendere due parole per motivare la scelta di modellare le proprietà dei device come delle classi piuttosto che come campi specifici per le diverse implementazioni. Durante la fase di progettazione delle diverse funzionalità del sottosistema in questione mi sono accorto che le operazioni da svolgere sui dispositivi hardware erano sempre le stesse ma, la loro implementazione, sarebbe stata dipendente unicamente dai campi sul quale lavorava i quali potevano essere di tipi differenti (stringhe piuttosto che interi) quindi mi è servito un metodo per riuscire a modellare un insieme di proprietà potenzialmente tutte differenti, proprio per questi motivi la modellazione delle proprietà è avvenuta attraverso la classe *BaseProperty* e l'utilizzo dei generici, vedremo poi come è stato possibile lavorare su queste proprietà attraverso l'utilizzo del pattern Visitor.

La modellazione di una generica proprietà avviene, come detto, attraverso la classe *BaseProperty*. Per raggiungere lo scopo preposto dalla classe l'idea iniziale era quella di effettuare la relativa modellazione mediante una stringa rappresentante la proprietà ed un campo generico rappresentate il valore della proprietà ma, purtroppo, la natura del linguaggio non permette l'inserimento, all'interno della classe, di campi e/o metodi generici a meno che tutta la classe non sia generica. Modellare questa classe come una classe generica avrebbe portato ad avere, all'interno di un generico *HWDevice* un array di *BaseProperty* di un solo tipo e quindi inadatto agli scopi. Per questi motivi, come si può notare in Figura 4.3, ho deciso di modellare la classe *BaseProperty* come una classe astratta contenente solamente la proprietà rappresentata come stringa e dei metodi astratti che verranno descritti a breve, questa classe è poi estesa ed implementata dalla classe *BaseInfo* la quale contiene un campo generico che ha come scopo quello di rappresentare il valore della proprietà a cui fa riferimento.

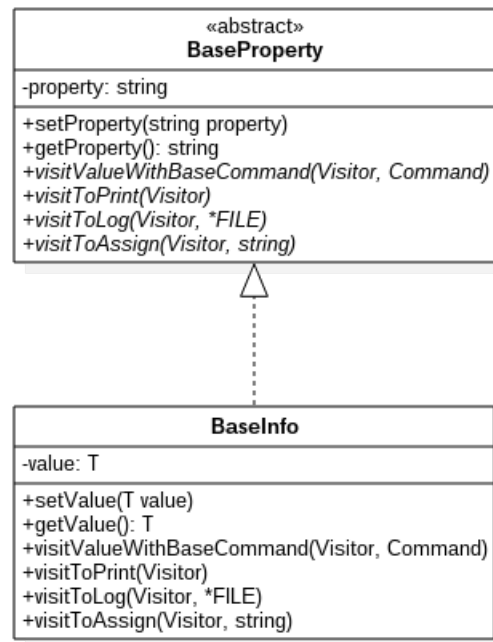


Figura 4.3: Diagramma delle classi della classe BaseProperty.

Seguendo una modellazione di questo tipo è possibile creare, per ogni specifico device, un set di proprietà che possono variare tra loro in base al tipo della proprietà stessa.

Per la gestione di queste proprietà ho deciso di applicare il pattern comportamentale Visitor [33] in quanto risulta essere adatto agli scopi. Per applicare il pattern ho dovuto, innanzitutto, modellare una classe *Visitor* astratta (Figura 4.4) contenente tutto l'insieme dei metodi utili da utilizzare per svolgere le diverse funzionalità su una generica classe *BaseInfo* (e.g. assegnamento del valore, mostrare a video la proprietà, scrivere su file le proprietà). La classe *Visitor* è poi implementata dalla classe *BaseInfoVisitor* che contiene tutte le logiche delle funzionalità descritte nella classe padre.

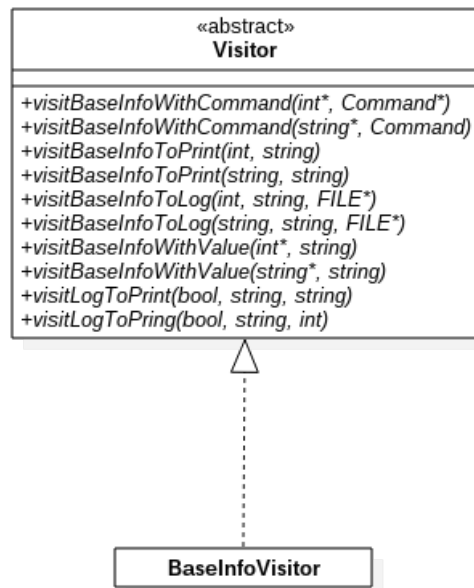


Figura 4.4: Diagramma delle classi della classe Visitor e BaseInfoVisitor.

Come si può notare dalla Figura 4.4 la classe *Visitor* contiene diversi metodi sui quali è stato effettuato l’overloading, questa scelta è dovuta dal fatto che le proprietà da modellare possono essere di diversi tipi e, quindi, servono diverse modalità attraverso le quali è possibile operare su di esse.

Ora risulta facile capire come siano state gestite le diverse funzionalità applicabili ad una proprietà che sono:

- **Assegnamento del valore generico** - Può avvenire attraverso i metodi *visitValueWithBaseCommand* e *visitToAssign* della classe. Con il primo metodo il valore viene assegnato sulla base dell’esecuzione di un comando (vedremo poi come questo sia possibile) mentre, con il secondo metodo, il valore viene assegnato attraverso un valore passato come parametro.
- **Stampa della proprietà** - Avviene attraverso il metodo *visitToPrint*.
- **Stampa su file della proprietà** - Questa funzionalità viene implementata dal metodo *visitToLog*.

Come abbiamo precedentemente affermato, il recupero delle informazioni riguardanti un generico device, avviene attraverso l’utilizzo dei comandi Linux, in particolare del comando `dmidecode`. Analizzando attentamente l’output restituito dall’esecuzione del comando in questione mi sono accorto che le entry dei decoded values descritti in 3.4.3 seguono il formato *Proprietà : Valore*,

dove *Proprietà* rappresenta il nome dell'informazione restituita mentre *Valore* modella il valore relativo alla proprietà, è proprio sulla base di queste informazioni che i diversi device che dovranno essere modellati saranno popolati con le giuste proprietà.

Ho deciso di modellare un generico comando attraverso un'interfaccia *Command* (Figura 4.5) la quale è caratterizzata da tre campi di tipo stringa. Il primo campo modella il comando da eseguire, il secondo ed il terzo campo, invece, rappresentano due filtri che vengono utilizzati per fare in modo che l'esecuzione di un comando restituisca solo ed unicamente il valore interessato. Per comprendere al meglio la natura di questi ultimi due filtri è meglio fare un esempio, l'esecuzione del comando "dmidecode -type bios | grep "ROM Size"" è in grado di restituire la grandezza della ROM in possesso dal BIOS installato sul sistema. L'output che questo comando genera segue generalmente il seguente pattern: *ROM Size: X kB* dove **X** identifica il numero di kByte in possesso ed è l'unica informazione a noi utile, per filtrarla basterà applicare due filtri, il primo (campo `mainFilter` della classe) settato a ":" permetterà di ricavare la sottostringa contenente il valore della ROM installata e la sua unità di misura, con il secondo filtro (campo `secondFilter`) impostato a " " sarà possibile filtrare nuovamente il contenuto ed ottenere solamente l'informazione necessaria.

Strutturando un generico comando in questa maniera sarà facile, in futuro, effettuare aggiornamenti al software a fronte di variazioni degli output restituiti maneggiando e modificando opportunamente i valori dei campi appena descritti.

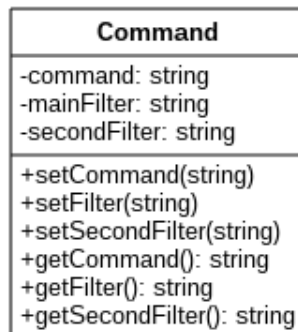


Figura 4.5: Diagramma delle classi della classe *Command*.

Oltre alla modellazione di un generico comando è stato necessario anche modellare un'entità in grado di eseguire un comando e restituire il valore filtrato attraverso i due filtri precedentemente descritti. Come si può notare in

Figura 4.6 la modellazione di questa entità è stata effettuata attraverso una classe generica *CommandManager* che tiene conto del tipo di dato da restituire e che estende una classe padre *helper_cmdManager* contenente il metodo *GetStdoutFromCommand* il quale si occupa, dato un comando, di restituire il relativo output in formato stringa. Sarà poi l'implementazione specifica del *CommandManager* a gestire, attraverso il metodo *getInfoFromCommand*, l'opportuno cast al tipo desiderato.

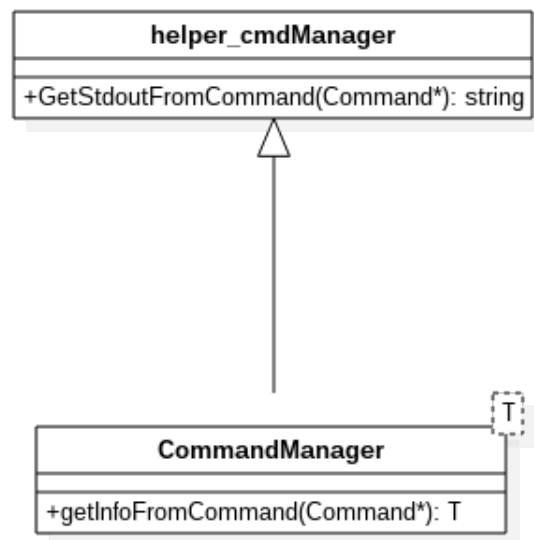


Figura 4.6: Diagramma delle classi della classe *CommandManager*.

Durante lo sviluppo dello strumento software sono incappato in diversi problemi riguardanti le stringhe (e.g. trim dei risultati, generazione dei timestamp, ecc) ed è quindi sorta la necessità di avere un elemento in grado di gestire tutte queste problematiche, lo *StringManager*. Come si può notare in Figura 4.7 questa entità contiene tutta una serie di metodi che, come vedremo dopo, saranno utili per la manipolazione e generazione di stringhe.

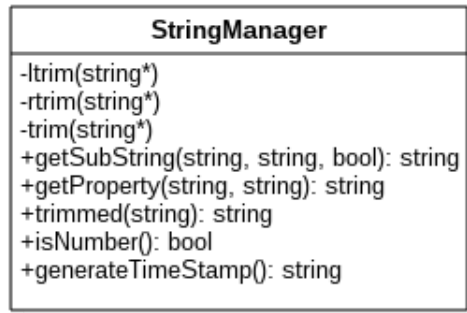


Figura 4.7: Diagramma delle classi della classe StringManager.

4.4.2 Implementazione dei dispositivi specifici

Come abbiamo precedentemente detto la classe HWDevice modella un generico device, sarà poi ogni implementazione specifica di ogni device a dover realizzare la classe astratta HWDevice e a dover definire l'insieme delle sue proprietà. Definiamo ora, per ogni device che il tool dovrà supportare (definiti in 4.4) l'insieme delle sue proprietà e il tipo di valore ad esse associate.

Tabella 4.1: Proprietà del BIOS

Proprietà	Tipo del valore
Vendor	string
Version	string
Release Date	string
ROM Size	int
Revision	string
Firmware Revision	string

Tabella 4.2: Proprietà di una generica scheda video

Proprietà	Tipo del valore
Product	string
Vendor	string
Physical ID	int
Clock	int

Tabella 4.3: Proprietà di un generico disco rigido

Proprietà	Tipo del valore
Serial Number	string
Model	string
Firmware revision	string

Tabella 4.4: Proprietà di un generico dispositivo di memoria (RAM)

Proprietà	Tipo del valore
Serial Number	string
Manufacturer	string
Speed	int
Type	string
Size	int

Tabella 4.5: Proprietà di una generica scheda di rete

Proprietà	Tipo del valore
Serial Number	string
Product	string
Vendor	string
Description	string

4.4.3 Modellazione e sviluppo DeviceManager

Passiamo ora a definire la modellazione della classe cardine dell'intero sottosistema.

Considerando che, nel caso più generale, una postazione utente può contenere più dispositivi dello stesso tipo (e.g. due schede video) si è deciso di modellare un'entità capace di gestire correttamente tutta la categoria di dispositivi associata ad essa ed in grado di svolgere le operazioni per le quali il tool è stato progettato. Come si può notare in Figura 4.8 questa classe è molto corposa e copre un insieme di campi e metodi molto ampio, per questo motivo l'analisi di questa classe verrà suddivisa analizzando le quattro principali funzionalità che offre in modo da poter capire al meglio lo scopo dei diversi metodi e campi offerti. Le funzionalità appena citate sono:

1. **Recupero delle informazioni riguardanti i device**
2. **Stampa delle informazioni**

3. Generazione dei log
4. Controllo dei device attualmente installati con dei file di log

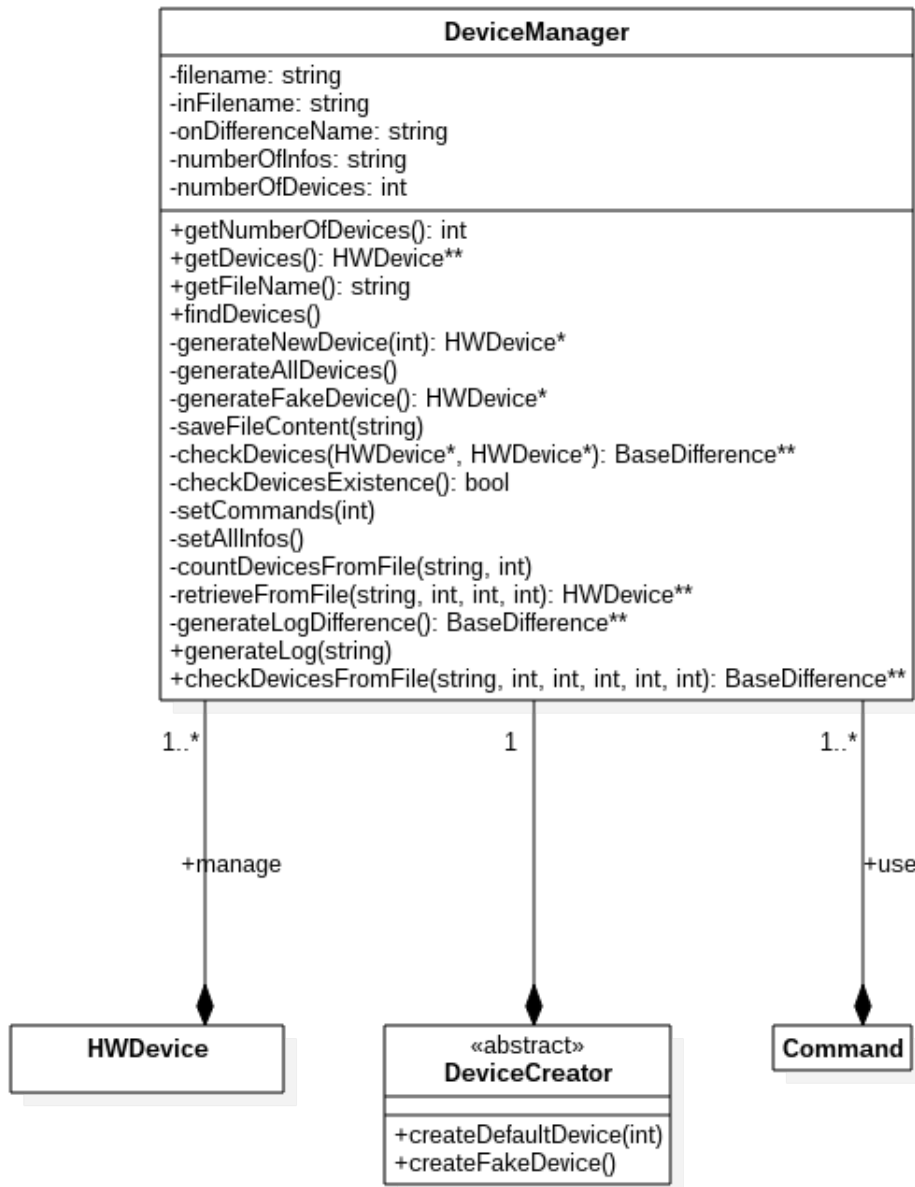


Figura 4.8: Diagramma delle classi della classe `DeviceManager`.

La procedura di recupero delle informazioni riguardanti i device ha come scopo quello di popolare l'array di `HWDevice` contenuto all'interno del `De-`

viceManager e fa largo uso del pattern comportamentale Template Method [35]. Per riuscire a conoscere la grandezza dell'array appena citato e quindi il numero di dispositivi che verranno identificati (campo *numberOfDevices*) la procedura che ci accingiamo a descrivere è preceduta da una fase attraverso la quale viene prima definito un comando in grado di effettuare questo conteggio e poi, sulla base del valore restituito da questo comando, viene generato un array di *HWDevice* della corretta dimensione.

Come precedentemente detto tutta la fase di recupero avviene mediante l'utilizzo dei comandi quindi, ogni specifico *DeviceManager*, dovrà definire il set di comandi da utilizzare per svolgere questa funzione, per fare ciò la classe in questione offre un metodo virtuale *setCommands* che dovrà obbligatoriamente essere definito da tutte le implementazioni specifiche della classe. Come si può notare, il metodo in questione, accetta in ingresso un intero che definisce la posizione del device all'interno del set di dispositivi contenuti nella classe, per il quale sono stati impostati i comandi, vedremo poi come questo parametro possa risultare utile nelle diverse implementazioni specifiche di *DeviceManager*.

Una volta che i comandi sono stati correttamente impostati si può procedere ciclando ogni comando tenendo conto che, nel caso più generale, ad ogni comando eseguito corrisponde una proprietà del dispositivo modellato. Proprio sulla base di quest'ultima affermazione ogni *BaseProperty* viene visitata da un *BaseInfoVisitor* il quale accetta un comando attraverso il quale è in grado di impostare il valore della proprietà. Il procedimento appena descritto è contenuto all'interno del metodo virtuale *setAllInfos*, ho deciso di rendere questo metodo virtuale in modo da dare la possibilità di definire una diversa modalità di recupero delle informazioni alle implementazioni future dei *DeviceManager* e non legare strettamente questa procedura all'utilizzo dei comandi. Infine arriviamo al metodo centrale utilizzato per offrire questa funzionalità, cioè *findDevices*, con i metodi descritti fin'ora siamo in grado di popolare un generico dispositivo con le corrette proprietà quindi, quest'ultimo metodo, non dovrà fare altro che ciclare per tutti i dispositivi richiamando l'operazione *setAllInfos*, facendo così, una volta completato il ciclo, avremo il set di *HWDevice* completo e correttamente popolato.

La procedura di stampa delle informazioni avviene attraverso il metodo *printAllInfos*, ciò che fa è semplicemente visitare tutti i dispositivi, precedentemente popolati dalla funzione appena descritta, e utilizzare un *BaseInfoVisitor* su ogni *BaseProperty* del device. Sarà poi compito del Visitor appena utilizzato stampare la proprietà ed il relativo valore.

Veniamo ora alla descrizione della procedura di generazione dei log. Attra-

verso questa funzionalità si vuole fornire all'utente la possibilità di generare dei log in una posizione specifica del filesystem oppure utilizzare la directory di default `/var/log/`, dando la possibilità di tener traccia di tutte le scansioni effettuate. Per raggiungere questo obiettivo ho implementato la funzione `generateLog` la quale prende in ingresso una stringa rappresentante il percorso dal quale si vogliono generare i log. Ogni volta che questa funzione viene chiamata, prima di procedere con la scrittura su file, si occupa di salvare l'intero contenuto del file (funzione `saveFileContent`) in modo da poterlo appendere successivamente al termine della scansione. Una volta che questa fase di backup è terminata la funzione procede scrivendo inizialmente un timestamp (generato dallo `StringManager` precedentemente descritto) al quale susseguono, per ogni device, le relative informazioni le quali sono scritte su file attraverso la funzione `visitToLog` del `BaseInfoVisitor`. Non appena tutte le proprietà ed i relativi valori di ogni dispositivo gestito dal manager sono state stampate, il metodo, non fa altro che appendere il contenuto precedente sotto la scansione appena effettuata ed infine chiudere il file. Con un'implementazione del genere si dà la possibilità all'utente di trovare sempre le ultime scansioni effettuate in cima al file di log e, come vedremo tra poco, è utile per riuscire a dare la possibilità all'utilizzatore di controllare l'hardware presente nella propria macchina con scansioni effettuate anche parecchi giorni precedenti.

Passiamo, infine, alla descrizione della funzionalità più importante, cioè controllare i device attualmente in possesso con quelli contenuti all'interno di un file di log. Per fare ciò però è meglio procedere per gradi analizzando le diverse fasi e componenti che compongono questa procedura, che sono:

1. **Generazione di un set di *HWDevice* da un file di log** - Questa funzionalità è stata implementata attraverso la funzione `retrieveFromFile`. Ciò che fa è, prima di tutto, contare il numero di dispositivi contenuti all'interno del file di log (funzione `countDevicesFromFile`), facendo così è in grado di generare un array di `HWDevice` della corretta dimensione. Una volta generato l'array l'algoritmo parte cominciando facendo il parsing l'intero file e, facendo sempre uso del `BaseInfoVisitor`, è in grado di assegnare correttamente i valori alle proprietà di ogni dispositivo ed infine restituire il set di device appena popolato. Il processo appena descritto è inoltre in grado di tenere conto delle diverse scansioni contenute all'interno di un file di log e selezionare unicamente la versione che viene passata come parametro alla funzione.
2. **Modellazione delle differenze di due generici *HWDevice*** - Definiamo ora il concetto di differenza tra due dispositivi facenti parte della stessa categoria. Due generici `HWDevice` sono differenti tra loro se e solo

se almeno un valore relativo ad una loro proprietà risulta essere diverso tra i due, risulta quindi utile avere un'entità in grado di modellare questa differenza. La classe a cui si sta facendo riferimento è *BaseDifference* (Figura 4.9), come si può notare anche in questo caso, come per la modellazione delle *BaseProperty*, si è fatto uso di una classe generica che estende *BaseDifference*, in questo modo è possibile tener traccia di un set di differenze potenzialmente di tipo diverso (stringhe piuttosto che interi o viceversa). La classe in questione presenta un campo booleano il quale ha come scopo quello di definire la presenza o meno di una differenza, se impostato a true allora vuol dire che la corrispondente proprietà, indicata dal campo *property* della classe, ha subito una variazione, altrimenti significa che la proprietà è rimasta invariata. Per le operazioni di assegnamento e stampa della differenza è stato utilizzato, anche in questo caso, un Visitor il quale opera secondo le stesse logiche precedentemente descritte.

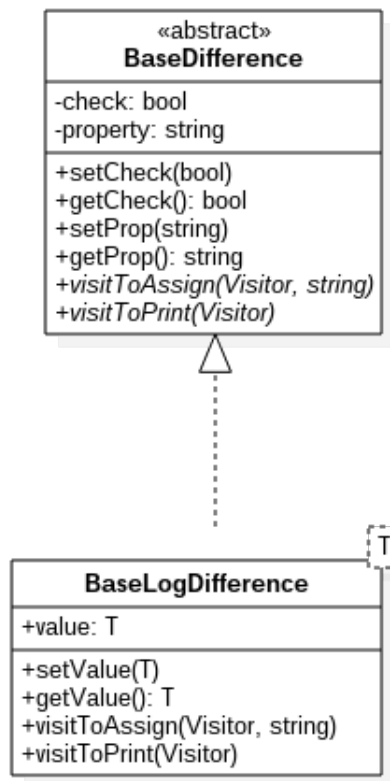


Figura 4.9: Diagramma delle classi della classe *BaseDifference*.

3. **Confronto di due generici *HWDevice*** - Il confronto di due dispositivi avviene mediante l'utilizzo della funzione *checkDevices*. La funzione non fa altro che ciclare tutte le informazioni dei due device e passare tutto in pasto ad un comparatore da me progettato. Il comparatore in questione è il *BaseInfoComparator* (Figura 4.10), esso offre il metodo pubblico *compare* il quale non fa altro che, date due *BaseProperty*, capire in fase di run-time il tipo specifico della proprietà e richiamare la corretta implementazione del metodo *visitToCompare* il quale effettua il confronto e, in caso le due proprietà siano differenti, assegna il valore della proprietà da controllare alla *BaseDifference*. Facendo così, la funzione è in grado di generare un set di *BaseDifference* della stessa dimensione del numero di informazioni del dispositivo e contenente, per ogni proprietà, il risultato del confronto.



Figura 4.10: Diagramma delle classi della classe *BaseInfoComparator*.

4. **Confronto di tutti i dispositivi presenti con il file di log** - Prima di procedere con l'analisi della funzione che è in grado di svolgere questa funzionalità è meglio effettuare una precisazione che mi ha permesso di implementare la suddetta funzione. La struttura della fase di recupero delle informazioni dei dispositivi impone il fatto che i device vengano identificati e memorizzati secondo un certo ordine quindi anche la procedura di log avviene secondo le stesse logiche. Durante la fase di confronto dei dispositivi attualmente presenti e quelli contenuti all'interno dei log ogni dispositivo in posizione X viene confrontato con il device nella medesima posizione nel file di log facendo uso della funzione di confronto precedentemente descritta e riuscendo quindi a generare un insieme di differenze tra i due. Rimane però da definire la modalità attraverso la quale è possibile identificare eventuali device aggiunti o rimossi. Questa operazione viene effettuata semplicemente confrontando le grandezze del set di dispositivi sul quale si sta eseguendo il confronto, se i device presenti attualmente nel sistema sono in un numero maggiore rispetto a quelli loggati allora vorrà dire che è stato aggiunto almeno un dispositivo mentre, se il set dei dispositivi installati nel sistema è minore

rispetto ai device presenti nel file di log vorrà dire che qualche device è stato rimosso. Implementando le suddette logiche la funzione *checkDevicesFromFile* è in grado di restituire l'insieme delle differenze tra tutti i dispositivi installati e loggati le quali possono essere stampate a video e quindi informare l'utente in caso qualcosa non sembri integro.

Avendo definito e descritto le quattro funzionalità del *DeviceManager* dovrebbe essere più chiaro il significato dei suoi campi e dei suoi metodi. L'unico elemento che potrebbe non essere compreso è il *DeviceCreator*. Questa classe è nata per far fronte al problema della generazione dei diversi *HWDevice*, infatti ogni implementazione specifica del *DeviceManager* dovrà contenere un'implementazione specifica del *DeviceCreator* in modo da poter generare le corrette istanze di *HWDevice*. Come si può notare in Figura 4.11 questa classe astratta definisce solamente i metodi *createDefaultDevice* e *createFakeDevice*. Attraverso il primo ogni istanza specifica della classe in questione sarà in grado di generare uno specifico *HWDevice* mentre il secondo metodo è utilizzato per generare un dispositivo fasullo ed è unicamente utilizzato all'interno della procedura di confronto descritta in precedenza.

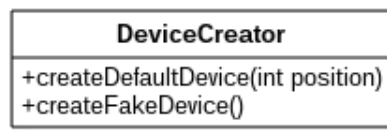


Figura 4.11: Diagramma delle classi della classe *DeviceCreator*.

4.4.4 Implementazioni specifiche *DeviceManager*

Avendo strutturato il *DeviceManager* facendo in modo che il più del codice sia riusato da tutte le relative implementazioni specifiche risulta molto semplice e veloce definire queste implementazioni in quanto, per ognuna di esse, è sufficiente effettuare l'overriding dei metodi *setCommands* e *generateLogDifference*. L'overriding del primo metodo è dovuto al fatto che, come detto in precedenza, ogni manager utilizza un set di comandi differenti per effettuare il recupero delle informazioni mentre con la seconda funzione ogni manager definisce il set di *BaseDifference* che vuole utilizzare. Inoltre ogni manager deve definire un'implementazione specifica del *DeviceCreator* la quale deve incapsulare le logiche di creazione degli oggetti. In Figura 4.12 è possibile vedere come siano stati modellati cinque diversi manager che andiamo ora a descrivere ponendo particolare attenzione ai comandi utilizzati per effettuare il recupero delle informazioni dei dispositivi.

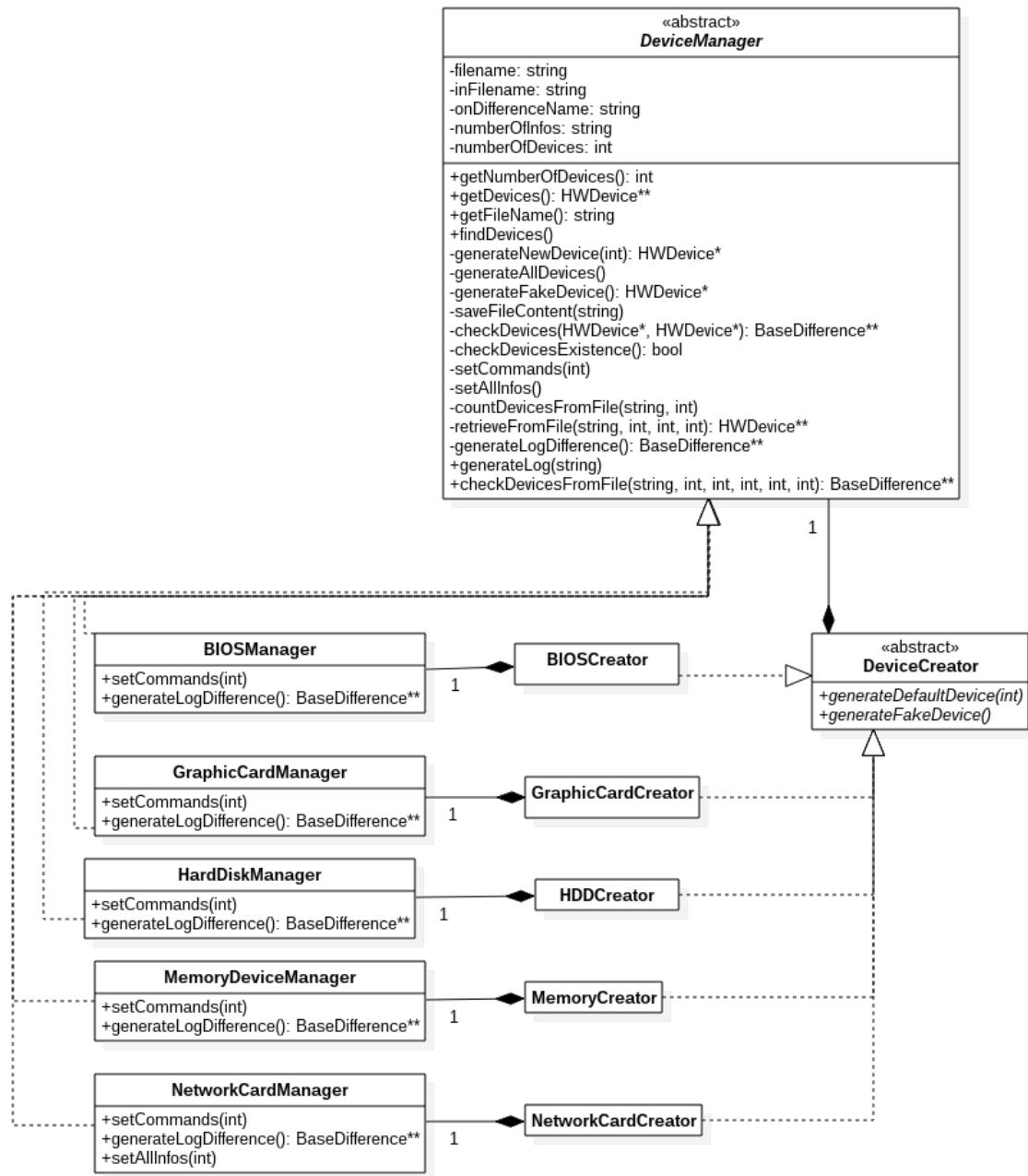


Figura 4.12: Diagramma delle classi delle classi specifiche di DeviceManager.

Partiamo con l'analisi del *BIOSManager*, come si può facilmente capire, questo manager si occupa della gestione di tutti i BIOS presenti all'interno del sistema. Dalla tabella 4.1 possiamo notare come un generico BIOS, secondo la modellazione seguita, sia composto da 6 diverse proprietà. Come abbia-

mo detto in precedenza, nel caso più generale possibile, ad ogni proprietà corrisponde un comando in grado di recuperare il relativo valore, analizziamo quindi ora i sei *Command* utilizzati per estrapolare le informazioni riguardanti i BIOS:

Proprietà	Comando
Vendor	<code>dmidecode -type BIOS grep -i 'vendor' sed -n Xp</code>
Version	<code>dmidecode -type BIOS grep -i 'version' sed -n Xp</code>
Release Date	<code>dmidecode -type BIOS grep -i 'release date' sed -n Xp</code>
ROM Size	<code>dmidecode -type BIOS grep -i 'ROM size' sed -n Xp</code>
BIOS Revision	<code>dmidecode -type BIOS grep -i 'BIOS revision' sed -n Xp</code>
Firmware Revision	<code>dmidecode -type BIOS grep -i 'firmware revision' sed -n Xp</code>

Come si può notare, per ogni proprietà, è stato utilizzato il comando `dmidecode`. Come detto in 3.4.3 questo comando offre una gran vastità di categorie di dispositivi, per filtrare solo la categoria relativa al BIOS è bastato parametrizzare il comando con la categoria *BIOS*. Per ricavare il valore di ogni proprietà è bastato poi concatenare al comando il comando *grep*² opportunamente parametrizzato con l'opportuna proprietà. Come si può notare ogni comando è inoltre formato da un ulteriore comando finale, cioè `sed -n Xp`. Questo è dovuto al fatto che, nel momento in cui esistano più dispositivi dello stesso tipo (in questo caso BIOS) l'esecuzione del comando deve riuscire a restituire solo l'informazione relativa al dispositivo in questione e, attraverso l'utilizzo del comando `sed`³ opportunamente parametrizzato questo è possibile. Ora dovrebbe risultare più chiaro l'utilità del parametro *position* che la funzione *setCommands* richiede, questo parametro è utilizzato appunto per identificare la posizione del dispositivo all'interno del set del *DeviceManager* e comporre opportunamente i diversi comandi in quanto sarà proprio questo intero a definire il parametro del comando `sed` appena descritto.

Passiamo ora alla definizione dei comandi utilizzati dal *MemoryDeviceManager* il quale si occupa di tutti i dispositivi di memoria presenti nel sistema. Un generico dispositivo di memoria è composto da cinque proprietà (descritte in 4.4) e anche in questo caso, come nel precedente, ad ogni proprietà corrisponde ad un comando che andiamo ora a definire:

²`grep` è un comando dei sistemi Unix e Unix-like che è in grado di ricercare all'interno di un file o di un output di un altro comando tutte le linee che contengono un determinato valore passato come parametro.

³`sed` è un comando Linux che permette di filtrare un testo in base a diversi parametri (es. numero di riga)

Proprietà	Comando
Serial Number	<code>dmidecode -type 17 grep -i 'Serial Number: ' sed -n Xp</code>
Size	<code>dmidecode -type 17 grep -i 'Size: ' sed -n Xp</code>
Speed	<code>dmidecode -type 17 grep -i 'Speed: ' sed -n Xp</code>
Type	<code>dmidecode -type 17 grep -i 'Type: ' sed -n Xp</code>
Manufacturer	<code>dmidecode -type 17 grep -i 'Manufacturer: ' sed -n Xp</code>

Proprio come per i BIOS anche per i dispositivi di memorizzazione è stato utilizzato il comando `dmidecode` in combinazione con i comandi `grep` e `sed`. Ciò che varia dal caso precedente è naturalmente l'insieme delle proprietà e quindi dei parametri passati al comando `grep` e la categoria utilizzata per `dmidecode` la quale risulta essere la numero 17 tra le 40 presenti.

La classe *HardDiskManager*, invece, si occupa della gestione di tutti gli dischi rigidi presenti sul sistema. Analizziamo il set di comandi utilizzato da questo manager per recuperare tutte le informazioni riguardanti gli hard disk, le quali possono essere visualizzate nella tabella 4.3.

Proprietà	Comando
Model Number	<code>hdparm -I /dev/sda grep -i 'Model Number: ' sed -n Xp</code>
Serial Number	<code>hdparm -I /dev/sda grep -i 'Serial Number: ' sed -n Xp</code>
Firmware Revision	<code>hdparm -I /dev/sda grep -i 'Firmware revision: ' sed -n Xp</code>

In questo caso al posto del comando `dmidecode` è stato utilizzato il comando `hdparm`. Questa scelta è stata forzata dal fatto che `dmidecode`, purtroppo, non è stato in grado di estrapolare le tre proprietà utili alla modellazione di un generico disco rigido e ho quindi dovuto optare per il comando `hdparm`. Arricchendo il comando con il parametro `-I /dev/sda` esso è in grado di restituire tutte le informazioni riguardanti i dischi rigidi installati sul sistema. Anche in questo caso è stata necessario l'utilizzo dei comandi `grep` e `sed` per filtrare opportunamente l'output.

Analizziamo ora i comandi utilizzati dal *GraphicCardManager*, cioè il manager in grado di gestire tutte le schede video presenti sul sistema. Come affermato nella tabella 4.2 ogni scheda video è caratterizzata da quattro proprietà le quali vengono popolate attraverso i comandi che andiamo ora a definire:

Tabella 4.6: My caption

Proprietà	Comando
Product	<code>lshw -c video grep -i 'product: ' sed -n Xp</code>
Vendor	<code>lshw -c video grep -i 'vendor: ' sed -n Xp</code>
Physical ID	<code>lshw -c video grep -i 'physical id: ' sed -n Xp</code>
Clock	<code>lshw -c video grep -i 'clock: ' sed -n Xp</code>

Come si può notare, anche in questo caso, non si è fatto uso del comando `dmidecode` in quanto, come per i dischi rigidi, quest ultimo non è stato in grado di recuperare tutte le informazioni necessarie. Al posto di `dmidecode` ho deciso di utilizzare `lshw` visto che, come descritto in 3.4.2, è risultato completo ed efficiente. Anche `lshw`, come `dmidecode`, offre la possibilità di filtrare i device per categoria, in questo caso, quindi, è bastato parametrizzare il comando in questione con la categoria *video* per avere tutte le informazioni riguardanti le schede video presenti nel sistema. Applicando poi i filtri *grep* e *sed* sono riuscito ad avere tutte le proprietà necessarie.

Arriviamo ora all'analisi dell'ultimo *DeviceManager* cioè il *NetworkCardManager* il quale compito è quello di occuparsi delle schede di rete presenti nel sistema. Come si può notare dal diagramma in Figura 4.12 questo manager oltre all'overriding delle due funzioni *setCommands* e *generateLogDifference* effettua un ulteriore overriding, quello della funzione *setAllInfos* cioè il metodo incaricato all'assegnamento di tutte le proprietà le quali, normalmente, vengono popolate attraverso l'utilizzo di uno specifico comando per ognuna di esse in questo caso, invece, l'approccio che ho dovuto seguire è diverso. Purtroppo anche in questo caso `dmidecode` non è stato in grado di fornire tutte le informazioni necessarie per la modellazione di una generica scheda di rete quindi ho dovuto optare all'utilizzo del comando `lshw`. Analizzando l'output del comando in questione opportunamente parametrizzato per filtrare solo le schede di rete mi sono accorto che le informazioni che generava per ogni scheda di rete non erano omogenee. Come si può notare in Figura 4.13 per le prime due schede identificate è presente la proprietà *Product* mentre la terza scheda la proprietà non è presente, l'utilizzo del comando `lshw` opportunamente parametrizzato e filtrato attraverso l'utilizzo di *grep* e *sed*, come fatto fin'ora, avrebbe causato problemi e le proprietà delle diverse schede video non sarebbero state correttamente popolate.

```

doomdiskday@doomdiskday:~$ sudo lshw -c network
[sudo] password for doomdiskday:
*-network
   description: Ethernet interface
   product: QCA8171 Gigabit Ethernet
   vendor: Qualcomm Atheros
   physical id: 0
   bus info: pci@0000:07:00.0
   logical name: enp7s0
   version: 10
   serial: 28:d2:44:6e:e3:96
   capacity: 1Gbit/s
   width: 64 bits
   clock: 33MHz
   capabilities: pm pciexpress msi msix bus_master cap_list ethernet physical tp 10bt 10bt-fd 100bt 100
bt-fd 1000bt-fd autonegotiation
   configuration: autonegotiation=on broadcast=yes driver=axl latency=0 link=no multicast=yes port=twis
ted pair
   resources: irq:33 memory:c2500000-c253ffff ioport:3000(size=128)
*-network
   description: Network controller
   product: BCM4313 802.11bgn Wireless Network Adapter
   vendor: Broadcom Corporation
   physical id: 0
   bus info: pci@0000:08:00.0
   version: 01
   width: 64 bits
   clock: 33MHz
   capabilities: pm msi pciexpress bus_master cap_list
   configuration: driver=bcma-pci-bridge latency=0
   resources: irq:16 memory:c2400000-c2403fff
*-network
   description: Wireless interface
   physical id: 1
   logical name: wlp8s0b1
   serial: 9c:d2:1e:54:d4:46
   capabilities: ethernet physical wireless
   configuration: broadcast=yes driver=brcmsmac driverversion=4.2.0-35-generic firmware=610.812 ip=192.
168.1.69 link=yes multicast=yes wireless=IEEE 802.11bgn

```

Figura 4.13: Output lshw filtrato per restituire le schede di rete.

Per questi motivi l'approccio che ho seguito si discosta dai precedenti, per ogni scheda di rete ho deciso di identificare nell'output restituito dal comando `lshw -class network` il blocco di informazioni riguardanti la scheda stessa il quale poi viene analizzato dallo *StringManager* facendo in modo di restituire le corrette proprietà. Per fare ciò ho dovuto ideare tre diversi comandi, uno per la localizzazione del punto di partenza del blocco, uno per identificare il punto finale del blocco e, infine, un comando in grado di combinare i due precedenti e restituire il blocco opportuno, andiamo ora ad analizzare i tre comandi appena citati:

1. **Comando per l'identificazione del punto di partenza** - Come si può notare in Figura 4.13 ogni scheda di rete viene identificata nell'output dalla stringa `*-network` quindi, per trovare il punto iniziale del blocco di informazioni, basterà identificare la linea in cui questa stringa è contenuta tenendo conto di quale scheda si sta analizzando. Per fare ciò ho utilizzato il comando `lshw -class network | grep -n *-network | cut -d ':' -f 1 | sed -n Xp`, con le prime due porzioni di comando sono stato in grado di filtrare le schede di rete e poi filtrare ulteriormente solo le linee contenenti la stringa di identificazione. Attraverso il comando `cut -d ':'`

-f 1 ho poi isolato il numero di linea contenente la riga di identificazione ed, infine, è stato utilizzato il comando *sed* per estrapolare solamente l'informazione riguardante la scheda di rete in considerazione.

- 2. Comando per l'identificazione del punto terminale** - Per identificare il punto terminale del blocco di informazioni il comando utilizzato può variare tra due implementazioni leggermente differenti tra loro. La prima è `lshw -class network | grep -n *-network | cut -d ':' -f 1 | sed -n (X+1)p` la quale ricalca molto il comando precedentemente utilizzato, infatti questo comando permette l'identificazione dell'inizio del prossimo blocco di informazioni e, quindi, la fine di quello corrente, esso è utilizzato in tutti i casi in cui la scheda in analisi non è l'ultima del blocco restituito dal comando. La seconda implementazione di questo comando serve proprio per coprire l'ultimo caso citato, cioè quello in cui la scheda di rete presa in considerazione è l'ultima del blocco, in questo caso il comando utilizzato è stato `lshw -class network | wc -l` il quale non fa altro che ritornare il numero di righe totali dell'output.
- 3. Estrapolazione del blocco di informazioni** - Avendo definito i due estremi del blocco non rimane che utilizzare il comando `lshw -class network | tail -n +START | head -n END` dove il parametro `START` è definito dall'output del comando descritto al punto 1 mentre il secondo parametro `END` è ricavato sottraendo all'output del comando descritto al punto 1 il valore di `START`. Attraverso il comando `tail -n +START` l'output viene filtrato dalla riga `START` in poi e combinando il comando `head -n END`, il quale permette di filtrare le prime `END` righe dall'output, sono riuscito ad estrarre il blocco di informazioni riguardanti l'opportuna scheda di rete.

A questo punto è bastato effettuare l'overriding del metodo `setAllInfos` facendo in modo che eseguisse il comando appena descritto e, attraverso lo `StringManager`, ricercasse le opportune proprietà all'interno dell'output generato dal comando.

4.4.5 Modellazione HWManager

Passiamo ora all'analisi della classe `HWManager`. Questa classe si occupa della gestione di tutti i possibili `DeviceManager` e, quindi, gestisce tutte le operazioni che possono essere effettuate su di essi. Come si può notare in Figura 4.14 esso è composto da diversi `DeviceManager` e da una serie di metodi che rispecchiano le funzionalità che il sottosistema dovrà offrire.

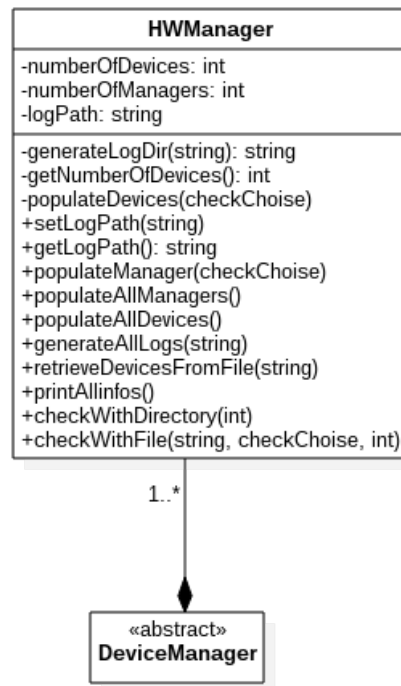


Figura 4.14: Diagramma delle classi della classe HWManager.

Analizziamo ora le funzionalità offerte da questo manager in modo da comprendere al meglio come esso utilizzi i metodi che offre:

- Popolazione dei device** - Questa funzionalità è espressa attraverso la funzione *populateAllDevices* la quale non fa altro che, per ogni manager gestito, richiamare l'opportuno metodo *findDevices*. In questo modo ogni manager contenuto dal *HWManager* sarà in possesso di tutti i dispositivi recuperati. Naturalmente per poter usufruire di questo metodo è necessario prima popolare il set di manager attraverso la funzione *populateAllManager*. Entrambi i metodi appena descritti sono disponibili in due versioni, una che permette la popolazione di tutti i manager e tutti i device ed un'altra che permette la creazione di uno specifico manager e, quindi, solamente dei relativi device.
- Stampa dei dispositivi** - La funzionalità in questione è implementata dal metodo *printAllInfos*. Questo metodo si occupa di ciclare tutti i manager e, per ognuno di essi, richiamare la funzione *printAllInfos*. In questo modo si ha la possibilità di stampare a video tutte le informazioni recuperate dai diversi dispositivi.

- **Generazione dei file di log** - Questa operazione è possibile grazie all'utilizzo del metodo *generateAllLogs* il quale, prima genera la directory (se non presente) sulla quale verranno generati i file di log, e poi cicla tutti i manager richiamando l'opportuno metodo *generateLog* precedentemente descritto.
- **Confronto dispositivi** - Quest'ultima funzionalità è stata implementata attraverso le funzioni *checkWithFile* e *checkWithDirectory*. Il primo metodo accetta un percorso contenente il file di log con cui effettuare il confronto, la scelta del manager da utilizzare e la versione di scan con cui effettuare il confronto. Ciò che fa è richiamare la funzione *checkDevicesFromFile* dell'opportuno manager e, sulla base delle *BaseDifference* restituite da questa funzione, stampare a video il risultato del confronto. Il metodo *checkWithDirectory*, invece, cicla tutti i manager richiamando la funzione appena descritta, così facendo è possibile mostrare all'utente i risultati di tutti i confronti effettuati.

Ora dovrebbe essere chiaro come sia stato possibile modellare ed implementare tutte le entità del sottosistema in modo da offrire all'utilizzatore tutte le funzionalità preposte come obiettivo.

4.4.6 Test del sottosistema

In questa sezione verrà mostrato il funzionamento del sottosistema appena descritto attraverso dei test, ognuno dei quali si incentra su una funzionalità specifica del sistema.

Il primo test effettuato mira alla funzionalità di recupero e visualizzazione delle informazioni riguardanti BIOS, schede video, RAM, schede di rete e dischi rigidi installati sul sistema. Come si può notare in Figura 4.15 questa funzionalità è stata espressa attraverso il parametro *-hw* e, attraverso la sua esecuzione, il tool è stato in grado di recuperare un gran numero di informazioni. Come si può notare, per alcune proprietà (e.g. tipo e velocità delle RAM) il tool non è stato in grado di recuperare il relativo valore, questo è dovuto al fatto che il comando utilizzato per recuperare queste informazioni non è riuscito a recuperarle ed ha quindi restituito un valore fasullo.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw
Number of devices retrieved: 11
[1]---BIOS INFOS---
Vendor: LENOVO
Version: 74CN44WW(V3.05)
Release date: 09/18/2013
ROM Size: 4608
Revision: 0.44
FW Revision: 3.5

[1]---GRAPHIC CARD INFO---
Product: GK107M [GeForce GT 755M]
Vendor: NVIDIA Corporation
Physical ID: 0
Clock: 33
[2]---GRAPHIC CARD INFO---
Product: 4th Gen Core Processor Integrated Graphics Controller
Vendor: Intel Corporation
Physical ID: 2
Clock: 33

[1]---MEMORY INFOS---
Serial Number: 0F409137
Manufacturer: Unknown
Speed: 1600
Type: DDR3
Size: 8192
[2]---MEMORY INFOS---
Serial Number: Empty
Manufacturer: Empty
Speed: 1600
Type: Unknown
Size: -1
[3]---MEMORY INFOS---
Serial Number: 217B6A7D
Manufacturer: Unknown
Speed: -1
Type: DDR3
Size: 4096
[4]---MEMORY INFOS---
Serial Number: Empty
Manufacturer: Empty
Speed: -1
Type: Unknown
Size: -1

[1]---NETWORK CARD INFO---
Serial: 28:d2:44:6e:e3:96
Product: QCA8171 Gigabit Ethernet
Vendor: Qualcomm Atheros
Description: Ethernet interface
[2]---NETWORK CARD INFO---
Serial: NOT FOUND
Product: BCM4313 802.11bgn Wireless Network Adapter
Vendor: Broadcom Corporation
Description: Network controller
[3]---NETWORK CARD INFO---
Serial: 9c:d2:1e:54:d4:46
Product: NOT FOUND
Vendor: NOT FOUND
Description: Wireless interface

[1]---HDD INFO---
Serial Number: ST1000LM014-SSHD-8GB
Model Number: W3817K5J
Firmware Revision: LVD3
```

Figura 4.15: Esecuzione del comando di recupero delle informazioni.

Il secondo test, invece, è incentrato sulla funzionalità di generazione dei log la quale, ricordiamo, può essere effettuata sia nella directory di default `/var/log/hwfwinfo/HW/` che in una qualsiasi cartella passata come parametro al tool. Come si può notare in Figura 4.16 questa funzionalità viene espressa attraverso il parametro `-hw -l` il quale può essere ulteriormente arricchito attraverso il passaggio di un ulteriore parametro identificante il path sotto il quale si vogliono generare i log.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw -l
Logs generated under /var/hwfwinfo/HW/ directory
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw -l /home/doomdiskday/Desktop/testHW/
```

Figura 4.16: Esecuzione del comando di generazione dei log.

I log generati dal tool possono essere visti in Figura 4.17, come si può notare, in ogni file di log, ogni scansione è identificata da un timestamp e dalla serie di informazioni riguardanti i device.

```
MemoryModuleInfos.txt x
---Scan Date: 18-6-2016 10:31:41
---MEMORY INFOS---
Serial Number: 0F409137
Manufacturer: Unknown
Speed: 1600
Type: DDR3
Size: 8192
---MEMORY INFOS---
Serial Number: Empty
Manufacturer: Empty
Speed: 1600
Type: Unknown|
Size: -1
---MEMORY INFOS---
Serial Number: 217B6A7D
Manufacturer: Unknown
Speed: -1
Type: DDR3
Size: 4096
---MEMORY INFOS---
Serial Number: Empty
Manufacturer: Empty
Speed: -1
Type: Unknown
Size: -1

BIOSInfos.txt x
-Scan Date: 18-6-2016 10:31:41
-BIOS INFOS---
ndor: LENOVO
rsion: 74CN44HW(V3.05)
lease date: 09/18/2013|
M Size: 4608
vision: 0.44
Revision: 3.5

HDDInfos.txt x
---Scan Date: 18-6-2016 10:31:41
---HDD INFO---
Serial Number: ST1000LM014-SSHD-8GB
Model Number: W3817K5J
Firmware Revision: LVD3
---Scan Date: 18-6-2016 10:30:56
---HDD INFO---
Serial Number: ST1000LM014-SSHD-8GB
Model Number: W3817K5J
Firmware Revision: LVD3

Network Card Infos.txt x
---Scan Date: 18-6-2016 10:31:41
---NETWORK CARD INFO---
Serial: 28:d2:44:6e:e3:96
Product: QCA8171 Gigabit Ethernet
Vendor: Qualcomm Atheros
Description: Ethernet interface
---NETWORK CARD INFO---
Serial: NOT FOUND
Product: BCM4313 802.11bgn Wireless Network Adapter
Vendor: Broadcom Corporation
Description: Network controller
---NETWORK CARD INFO---
Serial: 9c:d2:1e:54:d4:46
Product: NOT FOUND
Vendor: NOT FOUND
Description: Wireless interface
---Scan Date: 18-6-2016 10:30:56
---NETWORK CARD INFO---
Serial: 28:d2:44:6e:e3:96
Product: QCA8171 Gigabit Ethernet
Vendor: Qualcomm Atheros
Description: Ethernet interface
---NETWORK CARD INFO---
Serial: NOT FOUND
Product: BCM4313 802.11bgn Wireless Network Adapter
Vendor: Broadcom Corporation
Description: Network controller
---NETWORK CARD INFO---
Serial: 9c:d2:1e:54:d4:46
Product: NOT FOUND
Vendor: NOT FOUND
Description: Wireless interface

Graphic Card Infos.txt x
---Scan Date: 18-6-2016 10:31:41
---GRAPHIC CARD INFO---
Product: GK107M [GeForce GT 755M]
Vendor: NVIDIA Corporation
Physical ID: 0
Clock: 33
---GRAPHIC CARD INFO---
Product: 4th Gen Core Processor Integrated Graphics Controller
Vendor: Intel Corporation
Physical ID: 2
Clock: 33
```

Figura 4.17: File di log generati.

Passiamo ora al test della funzionalità di confronto analizzando il comportamento del tool in tre diversi casi che sono:

1. Nessun device è stato alterato rispetto all'ultima scansione.
2. È stato alterato almeno un device.
3. È stato rimosso o aggiunto un device.

Per effettuare il primo test dei tre appena citati è bastato parametrizzare il software con *-hw -c* subito dopo aver generato i log, naturalmente non è stata cambiata alcuna componentistica nel frattempo ed il tool, infatti, ha riportato esattamente questo risultato (Figura 4.18).

```
doomdiskday@doomdiskday:~/wspace3/hwfwInfo/Debug$ sudo ./hwfwinfo -hw -c
BIOSes Checks
[0] Vendor not changed
[0] Version not changed
[0] Release date not changed
[0] ROM Size not changed
[0] Revision not changed
[0] FW Revision not changed

Graphic Cards Checks
[0] Product not changed
[0] Vendor not changed
[0] Physical ID not changed
[0] Clock not changed
[1] Product not changed
[1] Vendor not changed
[1] Physical ID not changed
[1] Clock not changed

Memory devices Checks
[0] Serial Number not changed
[0] Manufacturer not changed
[0] Speed not changed
[0] Type not changed
[0] Size not changed
[1] Serial Number not changed
[1] Manufacturer not changed
[1] Speed not changed
[1] Type not changed
[1] Size not changed
[2] Serial Number not changed
[2] Manufacturer not changed
[2] Speed not changed
[2] Type not changed
[2] Size not changed
[3] Serial Number not changed
[3] Manufacturer not changed
[3] Speed not changed
[3] Type not changed
[3] Size not changed

Network Cards Checks
[0] Serial Number not changed
[0] Product not changed
[0] Vendor not changed
[0] Description not changed
[1] Serial Number not changed
[1] Product not changed
[1] Vendor not changed
[1] Description not changed
[2] Serial Number not changed
[2] Product not changed
[2] Vendor not changed
[2] Description not changed
```

Figura 4.18: Test del confronto dei device con risultato di non alterazione.

Per effettuare i due rimanenti test verrà utilizzato come caso di studio il BIOS in quanto, prendere in considerazione tutti i dispositivi, sarebbe ripetitivo e poco sensato visto che il comportamento che il tool assume nei confronti

dei diversi dispositivi è sempre lo stesso.

Non potendo realmente alterare il mio sistema il caso dell'alterazione del device è stato simulato semplicemente modificando i file di log generati in precedenza, è bastato modificare il valore di un paio di proprietà e vedere come il software rispondesse a queste modifiche. Dai risultati mostrati in Figura 4.19 si può notare come il software sia stato in grado di trovare le alterazioni e mostrarle a video.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw -c
[sudo] password for doomdiskday:
      BIOSes Checks

[0] Vendor not changed
[0] Version not changed
[0] Release date not changed
[0] ROM Size changed in: 2304
[0] Revision changed in: 0.46
[0] FW Revision changed in: 4.20
```

Figura 4.19: Test del confronto dei device con risultato di alterazione.

L'ultimo test, cioè quello riguardante l'aggiunta o la rimozione di un dispositivo è stato simulato sempre alterando il file di log. È bastato aggiungere e rimuovere dal log del BIOS il blocco di informazioni riguardanti il BIOS stesso, attraverso la rimozione del blocco di informazioni appena citato si è voluto simulare l'aggiunta di un dispositivo rispetto alla configurazione precedente mentre, attraverso l'aggiunta di un blocco di informazioni al file di log, si è simulata la rimozione completa di un dispositivo. Come si può vedere in Figura 4.20 il software è riuscito, in entrambi i casi, ad accorgersi e riportare all'utente le informazioni di alterazione.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw -c
BIOSes Checks

It would seem that 1 BIOSes were added

WARNING! These BIOSes seems to be added from the last scan!
[0] Vendor changed in: LENOVO
[0] Version changed in: 74CN44WW(V3.05)
[0] Release date changed in: 09/18/2013
[0] ROM Size changed in: 4608
[0] Revision changed in: 0.44
[0] FW Revision changed in: 3.5
```

(a) Test di aggiunta di un BIOS

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -hw -c
BIOSes Checks

It would seem that 1 BIOSes were removed

[0] Vendor not changed
[0] Version not changed
[0] Release date not changed
[0] ROM Size not changed
[0] Revision not changed
[0] FW Revision not changed

WARNING! These BIOSes seems to be removed from the last scan!
[1] Vendor changed in: ASUS
[1] Version changed in: 344432AL
[1] Release date changed in: 07/13/2011
[1] ROM Size changed in: 2306
[1] Revision changed in: 0.44
[1] FW Revision changed in: 3.0
```

(b) Test di rimozione di un BIOS

Figura 4.20: Test di aggiunta e rimozione di un BIOS

Arrivati a questo punto posso dire che tutti i test riguardanti il sottosistema hardware hanno restituito i risultati attesi e quindi questa porzione di sistema risulta completa ed è in grado di fornire all'utente tutte le funzionalità poste come obiettivo.

4.5 Progettazione e sviluppo del sottosistema firmware

In questa sezione verrà trattata la progettazione e lo sviluppo del sottosistema firmware. Come detto in precedenza questa porzione di sistema deve fornire all'utente le seguenti funzionalità:

1. Mostrare a video le informazioni riguardanti i firmware presenti sul sistema.

2. Generare dei log con le informazioni estrapolate.
3. Controllare eventuali differenze tra i firmware attualmente in possesso e quelli presenti nei log.

Per raggiungere questi obiettivi innanzitutto bisogna definire da dove le immagini dei firmware vengano recuperate. Come mostrato dalle ricerche effettuate e descritte in 3.1 la maggior parte delle immagini in questione sono contenute all'interno della directory `/lib/firmware/`, ho quindi deciso di implementare tutte le funzionalità appena citate appoggiandomi al contenuto della suddetta cartella.

4.5.1 Modellazione di un Firmware

La modellazione di un generico firmware è avvenuta attraverso la classe *Firmware*. Come si può notare in Figura 4.21 la classe in questione è composta da tre campi di tipo stringa. Il campo *file_name* identifica il nome del file, il campo *directory* rappresenta il path attraverso il quale è possibile raggiungere il file mentre, l'ultimo campo, *sha256* viene utilizzato per memorizzare, appunto, l'sha256 del file in questione.

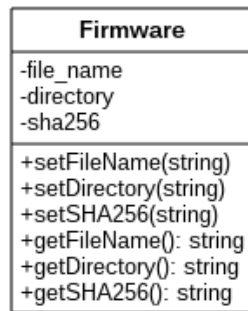


Figura 4.21: Diagramma delle classi della classe Firmware.

Inoltre la classe offre anche tre metodi setter e tre metodi getter per poter impostare e/o restituire i valori dei diversi campi.

Esplorando la cartella `/lib/firmware/` mi sono accorto che molte immagini venivano replicate ma in differenti cartelle, questo è dovuto al fatto che ogni versione del kernel Linux si porta con se un set di immagini firmware, molte delle quali risultano in comune tra le diverse versioni e, per ogni versione, all'interno della cartella sopracitata viene creata una subdirectory specifica. Per questo motivo ogni file è univocamente identificato dalla combinazione data

dal nome del file e dal percorso per raggiungerlo in modo da poter conto di tutte le immagini presenti. Inoltre ho deciso di utilizzare la funzione crittografica di hash sha256 perchè, come detto in 2.3.1, risulta essere molto resistente alle collisioni e, quindi, molto affidabile.

4.5.2 Modellazione e sviluppo FirmwareManager

Passiamo ora all'analisi della classe *FirmwareManager*. Questa classe si occupa della gestione di tutti firmware in possesso dal sistema e, come si può notare in Figura 4.22, offre un gran numero di metodi. Per comprendere al meglio il funzionamento di tutti i metodi e, quindi, dell'intera classe procederò con un'analisi centrata sulle funzionalità che vengono offerte dalla classe.

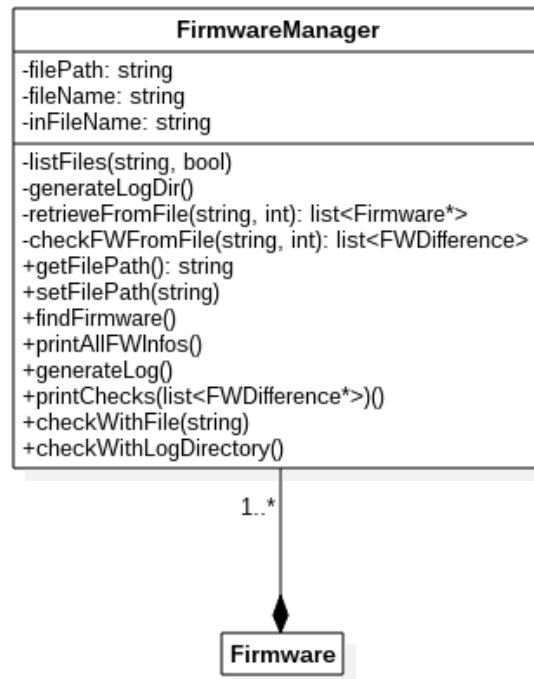


Figura 4.22: Diagramma delle classi della classe FirmwareManager.

Prima di tutto è meglio analizzare la fase di recupero delle informazioni relative alle immagini dei firmware attraverso la quale viene popolato il set di *Firmware* in possesso dal manager. Questa operazione viene svolta dal metodo pubblico *findFirmware* il quale non fa altro che richiamare la funzione *listFiles*. Quest'ultima funzione è in grado di ciclare ricorsivamente la cartella che le viene passata come parametro e ogni volta che identifica un file non fa

altro che creare un oggetto di tipo *Firmware* impostando le giuste proprietà e poi inserirlo all'interno del set di firmware del manager. Per effettuare l'hash di ogni file ho deciso di riutilizzare la classe *Command* in combinazione con il *CommandManager*, infatti i sistemi Linux offrono, tra i tanti comandi di default, anche una serie di comandi utilizzabili per calcolare i valori di hash di qualsiasi file dando la possibilità di scegliere tra diverse funzioni crittografiche, tra cui è presente anche la funzione sha256. Il comando in questione è *sha256sum* e, per essere eseguito, necessita solo del path completo del file sul quale si vuole effettuare il checksum.

La procedura di stampa delle informazioni avviene attraverso il metodo pubblico *printAllFWInfos*. Ciò che fa è ciclare tutti i firmware precedentemente popolati e stampare le relative informazioni a video.

La funzionalità di generazione dei log è stata implementata dal metodo *generateLog* il quale, inizialmente, non fa altro che generare la log directory (se non presente) basandosi sul path identificato dal campo *filePath* il quale, di default, è impostato al valore */var/log/hwfwinfo/FW* ma è anche impostabile attraverso opportuni metodi setter e getter in modo da poter dare la possibilità di generazione dei log anche in directory passate direttamente dall'utente. Dopo aver fatto ciò, la funzione in questione, esegue un ciclo su tutti i firmware in possesso dal manager e stampa su file le giuste informazioni.

L'ultima, ma anche più corposa, funzionalità offerta è il confronto dei firmware attualmente in possesso con quelli presenti in un file di log. Vediamo ora i componenti e le procedure principali utilizzate per offrire questa funzionalità:

1. **Recupero delle informazioni dei firmware da un file di log** - Naturalmente, per poter effettuare il suddetto confronto è necessario un modo per estrapolare il set di firmware contenuti all'interno di un file di log generato attraverso la funzione precedentemente descritta, il metodo incaricato per svolgere questo lavoro è *retrieveFromFile*. Quest'ultimo accetta come parametro il percorso contenente il file di log il quale viene completamente visitato dal metodo. Attraverso l'aiuto dello *StringManager* è stato possibile identificare tutti i firmware contenuti all'interno del file i quali, mano a mano che il file viene letto, vengono inseriti all'interno del set di firmware che, al termine della lettura del file, il metodo restituisce.
2. **Modellazione delle differenze tra due *Firmware*** - Come per il sottosistema hardware, anche in questa porzione di sistema ho dovuto modellare un'entità in grado di memorizzare le eventuali differenze di due *Firmware*. L'elemento in questione è stato modellato attraverso la classe *FWDifference*. Come si può notare il Figura 4.23 essa è composta da un'enumerazione interna la quale indica lo stato della differenza, ana-

lizziamo i diversi stati possibili di un istanza di questa classe in modo da comprendere al meglio il funzionamento della classe stessa:

- (a) **same** - Il *Firmware* sul quale si sta effettuando l'analisi era presente anche durante l'ultima scansione (quindi presente nel file di log) ed il suo contenuto non è cambiato, cioè il valore del suo digest è rimasto inalterato. In questo caso, le informazioni relative al firmware, sono contenute all'interno del campo *original* della classe.
- (b) **different** - Il *Firmware* in questione era presente anche durante l'ultima scansione ma il suo contenuto è cambiato, cioè il valore di hash restituito dalla funzione sha256 è differente. Le informazioni relative al *Firmware* attualmente in possesso sono contenute nel campo *original* mentre quelle relative al *Firmware* precedentemente in possesso sono nel campo *different*.
- (c) **added** - Questo stato indica che il *Firmware* in questione non era presente nel file di log ed è quindi stato aggiunto, tutte le informazioni riguardanti il suddetto firmware sono contenute nel campo *original*.
- (d) **removed** - Attraverso questo stato si vuole indicare che il *Firmware* in analisi è stato rimosso dal sistema, cioè è presente nel file di log ma non nel sistema e, tutte le relative informazioni, sono contenute dal campo *original*.

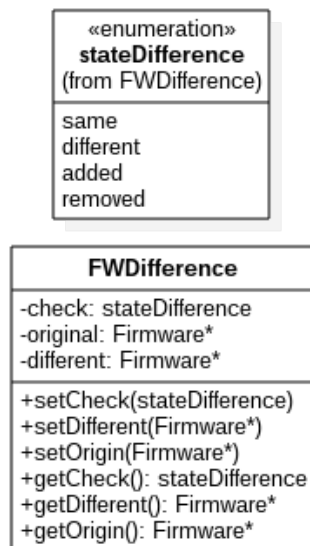


Figura 4.23: Diagramma delle classi della classe `FirmwareDifference`.

3. **Confronto dei firmware** - Il confronto dei *Firmware* in possesso con quelli contenuti all'interno di un file di log viene effettuato dal metodo *checkFWFromFile* il quale accetta un parametro indicante il percorso del file di log. Per prima cosa la funzione non fa altro che estrapolare tutti i *Firmware* contenuti all'interno del file di log attraverso la funzione esplicata al punto 1. Dopo aver fatto ciò, il metodo, è in possesso di entrambi i set di *Firmware* (quelli presenti sul sistema e quelli estrapolati dal file) e comincia a ciclare le due liste applicando le seguenti logiche:

- Se un file è presente nella stessa directory e con lo stesso nome ma con sha256 differente allora viene generato un *FWDifference* di tipo *different*.
- Se un file è presente nella stessa directory, con lo stesso nome e con lo stesso sha256 allora si genera un *FWDifference* di tipo *same*.
- Se un *Firmware* attualmente in possesso non viene trovato nel set di *Firmware* presenti nel file di log allora viene generato un *FWDifference* di tipo *added*.
- Se un *Firmware* è presente nel file di log ma non è attualmente sul sistema allora l'istanza di *FWDifference* che dovrà essere generata sarà di tipo *removed*.

Applicando le suddette logiche, l'ultimo metodo descritto, è in grado di generare un set di *FWDifference* il quale viene ciclato dal metodo *printChecks* che non fa altro che, per ogni *FWDifference*, stampare il relativo risultato, riuscendo ad avere così una visione complessiva del confronto effettuato.

4.5.3 Test del sottosistema

In questa sezione saranno mostrate le diverse funzionalità descritte in precedenza.

Il primo test mira alla visualizzazione di tutti i firmware in possesso. Come si può notare in Figura 4.24 l'esecuzione di questa funzionalità avviene attraverso il parametro `-fw` e, il risultato che restituisce, non è altro che la lista dei firmware contenuta nella directory `/lib/firmware/`.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -fw
---FIRMWARE---
Firmware file: spdif.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: bb1d5b2d4ec91bb77af31a36fdbf62f3e2525d7e75ac9814bdfbdf1f5b5759a5
---FIRMWARE---
Firmware file: bitstream.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: b15599a64ff4743b8c4b316767f61f2f74d9f86adb7e479cfa513babaebb9293
---FIRMWARE---
Firmware file: midi.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: d68a50f6b44549767e5db8df86e7bb1e2e56a13d1ec24e60ae51c0a63047c549
---FIRMWARE---
Firmware file: loader.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: fd67e700b77723649972bfd59c86d24e60f3a2cfce3d31e98a4fdbf05605a34
---FIRMWARE---
Firmware file: GPL-3
Firmware directory: /lib/firmware/
Firmware SHA256: fc82ca8b6fdb18d4e3e85cfd8ab58d1bcd3f1b29abe782895abd91d64763f8e7
---FIRMWARE---
Firmware file: boot.img
Firmware directory: /lib/firmware/RTL8192E/
Firmware SHA256: d57c36c69b8129bba44d5e5c6c7467e75cca354a8cad7d15482481209c969d59
```

Figura 4.24: Porzione di output restituito dal test di visualizzazione dei firmware in possesso.

La funzionalità di generazione dei log può essere espressa attraverso l'utilizzo di due combinazioni di parametri. Utilizzando come parametro `-fw -l` il tool genererà i file di log sotto la directory `/var/log/hwfwinfo/FW/` mentre utilizzando gli stessi parametri ma aggiungendo un path come ultimo parametro l'utente può indicare dove generare il file. Come si vede in Figura 4.25 il log generato contiene tutte le informazioni riguardanti i firmware in possesso.

```

Firmwares Infos.txt x
|---FIRMWARE---
Firmware file: spdif.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: bb1d5b2d4ec91bb77af31a36fdbf62f3e2525d7e75ac9814bdfbdf1f5b5759a5
|---FIRMWARE---
Firmware file: bitstream.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: b15599a64ff4743b8c4b316767f61f2f74d9f86adb7e479cfa513babaebb9293
|---FIRMWARE---
Firmware file: midi.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: d68a50f6b44549767e5db8df86e7bb1e2e56a13d1ec24e60ae51c0a63047c549
|---FIRMWARE---
Firmware file: loader.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: f1d67e700b77723649972bfd59c86d24e60f3a2cfce3d31e98a4fdbf05605a34
|---FIRMWARE---
Firmware file: GPL-3
Firmware directory: /lib/firmware/
Firmware SHA256: fc82ca8b6fdb18d4e3e85cfd8ab58d1bcd3f1b29abe782895abd91d64763f8e7
|---FIRMWARE---
Firmware file: boot.img
Firmware directory: /lib/firmware/RTL8192E/
Firmware SHA256: d57c36c69b8129bba44d5e5c6c7467e75cca354a8cad7d15482481209c969d59
|-----

```

Figura 4.25: Porzione di output restituito dal test di generazione dei log dei firmware in possesso.

L'ultimo test mira al confronto dei firmware presenti sul sistema con quelli presenti nel file di log. Anche in questo caso è possibile effettuare il confronto sia nella log directory principale `/var/log/hwfwinfo/FW/` utilizzando il parametro `-fw -c`, sia attraverso il path del file di log aggiungendo il percorso come ultimo parametro. Ho deciso di scomporre il test in quattro sotto-test, in modo da mostrare la completa copertura di questa funzionalità.

Il primo test prende di mira il caso in cui nessuna immagine sia variata rispetto all'ultima scansione, per fare ciò è bastato lanciare il comando appena dopo la generazione dei log e, come si può vedere in Figura 4.26, il tool non ha trovato alcuna differenza tra i firmware in possesso e quelli loggati.

```

doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -fw -c
No differences found! It seems to be all right!

```

Figura 4.26: Test del confronto con esito positivo.

Con il secondo test ho voluto cambiare il digest di una delle immagini presenti nel file di log in modo da testare il tool a fronte di un eventuale cambiamento del contenuto di una immagine. Come si può notare in Figura 4.27 il software è riuscito ad identificare il cambiamento, mostrando la directory contenente il file e l'alterazione del suo sha256.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -fw -c
---FW CHECKS---
Firmware name: spdif.fw In directory: /lib/firmware/emi62/ Seems to be changed!
His old SHA256 was: bb1d5b2d4ec91bb77af31a36fdbf62f3e2525d7e75ac9814bdfbdf1f5b5759a6
His new SHA256 is: bb1d5b2d4ec91bb77af31a36fdbf62f3e2525d7e75ac9814bdfbdf1f5b5759a5
```

Figura 4.27: Test del confronto con esito di alterazione.

Con gli ultimi due test ho voluto osservare il comportamento del tool a fronte di una aggiunta e una rimozione di un'immagine firmware dal sistema. Per fare ciò mi è bastato alterare il file di log inserendo prima, e rimuovendo poi, un'immagine e, come si può vedere il Figura 4.28, il software è stato in grado di rilevare queste modifiche.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -fw -c
---FW CHECKS---
Firmware name: bbnoxeon.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: b15599a64ff4743b8c4b316767f61f2f74d9f86wer7e479cfa511babwweb9293
Seems to be removed!
```

(a) Test di rimozione di un'immagine.

```
doomdiskday@doomdiskday:~/wspace3/hwfwinfo/Debug$ sudo ./hwfwinfo -fw -c
---FW CHECKS---
Firmware name: bitstream.fw
Firmware directory: /lib/firmware/emi62/
Firmware SHA256: b15599a64ff4743b8c4b316767f61f2f74d9f86adb7e479cfa513babaebb9293
Seems to be added!
```

(b) Test di aggiunta di un'immagine.

Figura 4.28: Test di rimozione e aggiunta di un'immagine.

Arrivati a questo punto posso affermare che tutte le funzionalità poste come obiettivo dal sottosistema appena analizzato sono state implementate e risultano funzionanti.

4.6 Modellazione e implementazione della classe ShellMenuParser

L'ultima entità del sistema non ancora descritta è lo *ShellMenuParser*, cioè la classe in grado di gestire l'esecuzione dell'intero sistema sulla base dei parametri passati come input dalla shell di comando. Come si può notare in Figura 4.29 la classe in questione offre il solo metodo pubblico *parseAndExecute* il quale prende in ingresso un intero rappresentante il numero di parametri passati al tool e un array di puntatori a caratteri rappresentanti i parametri stessi, inoltre esso è composto da un *HardwareManager* e un *FirmwareManager*. Il

metodo *parseAndExecute* non farà altro che, a fronte dei parametri in ingresso, scegliere il giusto manager da utilizzare e richiamare le opportune funzioni.

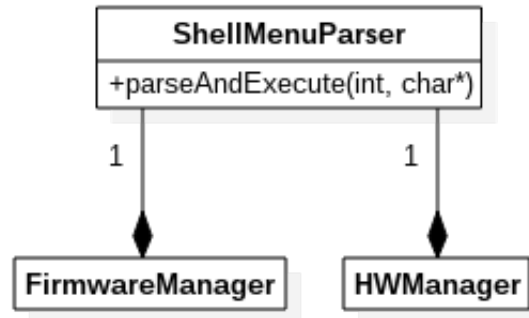


Figura 4.29: Diagramma delle classi della classe ShellMenuParser.

4.7 Sviluppi futuri

Dopo aver analizzato e descritto tutte le entità presenti nel sistema posso affermare che l'intero progetto è in grado di gestire tutte le funzionalità che erano state poste come obiettivo ed è quindi completo. Un eventuale sviluppo futuro potrebbe essere quello di supportare un numero più ampio di dispositivi hardware (e.g. processore, scheda madre, cache, ecc.), questo obiettivo potrebbe essere facilmente raggiungibile in quanto la progettazione del sottosistema hardware permette una facile estendibilità, offrendo la possibilità di aggiungere nuovi *HWDevice* e *DeviceManager* facilmente. Per quanto riguarda il sottosistema firmware sarebbe stato interessante implementare diverse funzionalità per riuscire ad effettuare il dump dei firmware da device come BIOS e dischi rigidi ma le conoscenze in mio possesso non mi hanno permesso di poterle implementare, come mostrato in Sezione 3.3 esistono comunque dei tool in grado di effettuare il dump dell'immagine del firmware del BIOS.

Capitolo 5

Conclusioni

In questa tesi è stato mostrato come implementare uno strumento software in grado di verificare che non ci siano state alterazioni hardware o firmware in una generica postazione utente. Tutta la fase di progettazione ed implementazione del sistema è stata preceduta da un'attenta analisi dello stato dell'arte, sono partito analizzando prima il mondo dei firmware estendendo la ricerca in tre diversi campi, i firmware nel mondo Linux (sezione 3.1), i firmware nei dischi rigidi (sezione 3.2) ed i firmware dei BIOS (sezione 3.3). Attraverso la prima di queste tre sezioni sono riuscito ad identificare il target del sottosistema firmware (la cartella */lib/firmware/*) e a comprendere come avvengono le richieste dei firmware nei sistemi Linux. Dopo l'analisi dei firmware utilizzati dai dischi rigidi e dai BIOS posso affermare che attacchi a queste componenti possono causare ingenti danni e, la loro rilevazione, non è per niente facile.

Infine ho analizzato il tracciamento hardware in un generico sistema Linux dal quale sono riuscito ad identificare i comandi che, successivamente, sono stati utilizzati nel sistema per effettuare il recupero delle informazioni riguardanti i dispositivi hardware installati.

Tutta la fase di ricerca è stata accompagnata da casi di attacchi, come quello dell'Equation Group (sezione 3.2.3) e il caso Lightteater (sezione 3.3.2), i quali mi hanno fatto capire come questo mondo sia tanto trascurato quanto pericoloso, fortunatamente ho scoperto molti progetti interessanti (FWUPD, Sistema VirusTotal e PSN Intel) i quali, con un opportuno appoggio dai produttori e dalla Comunità del software libero, potrebbero rendere molto più sicuro questo mondo.

Lo sviluppo dell'intero progetto mi ha permesso di migliorare ampiamente la mia conoscenza dei sistemi Linux e dei loro comandi, inoltre mi ha permesso di addentrarmi in una porzione del mondo della sicurezza informatica dalla quale sono sempre stato attratto, confermando l'idea che già avevo di questo

campo, cioè bello per quanto complicato. La parte di sviluppo del progetto, inoltre, mi ha permesso di approfondire la conoscenza del linguaggio C++ che avevo trattato poco nel corso di questi anni.

Complessivamente sono molto contento del lavoro svolto e dei risultati ottenuti.

Appendice A

Link GitHub al progetto

Il progetto è reperibile al link Github [42] <https://github.com/marcomancini94/hwfwinfo> coperto dalla licenza GNU General Public License V.3 [43].

Bibliografia

- [1] Sans,
Information Security,
<https://www.sans.org/information-security/>.
- [2] Kaspersky,
Equation Group: Questions and answers,
https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf
- [3] *Linux Kernel*,
<https://www.kernel.org/>
- [4] *Information Assurance*
<http://www.iac.iastate.edu/>
- [5] William Stallings, Lawrie Brown,
Computer Security: Principles and Practice (2nd Edition), Prentice Hall, Inc., 2012.
- [6] Ronald Rivest,
The MD5 Message Digest Algorithm, RFC1320, 1994.
- [7] Ronald Rivest,
The MD4 Message Digest Algorithm, RFC1320, 1992.
- [8] Bert den Boer, Antoon Bosselaers,
Collisions for the compression function of MD5, 1994.
- [9] Jie Liang, Xuejia Lai,
Improved Collision Attack on Hash Function MD5, 2005.
- [10] *Secure Hash Algorithms, RFC4634*, 2006.
- [11] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu
Collision Search Attacks on SHA1, 2005.

-
- [12] Linux Foundation,
Filesystem Hierarchy Standard (FHS),
<https://wiki.linuxfoundation.org/en/FHS>
- [13] Freedesktop.org
Udev manual,
<https://www.freedesktop.org/software/systemd/man/udev.html>
- [14] Manuel Estrada Sainz,
Request_firmware() hotplug interface,
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/firmware_class/README?id=HEAD , 2003.
- [15] Richard Hughes,
FWUPD,
<http://www.fwupd.org/> , 2015.
- [16] Freedesktop.org,
D-Bus Specification,
<https://dbus.freedesktop.org/doc/dbus-specification.html>,
2015.
- [17] ACE Laboratory,
PC3000,
<http://www.ancelaboratory.com/pc3000.Express.php>.
- [18] IEEE,
IEEE Standard for Test Access Port and Boundary-Scan Architecture,
<http://ieeexplore.ieee.org.ezproxy.unibo.it/xpl/articleDetails.jsp?arnumber=6515989&queryText=IEEE%201149.1&refinements=4294965216>
- [19] *Kaspersky*,
<https://www.kaspersky.com>.
- [20] *VirusTotal*,
<https://www.virustotal.com>.
- [21] Vladimir Bashun, Anton Sergeev, Victor Minchenkov, Alexandr Yakovlev,
Too Young to be Secure: Analysis of UEFI Threats and Vulnerabilities.
- [22] Shane Schick,
Trouble at the BIOS Level: LightEater Malware Proof of Concept Shows Major Security Risk, 2015.

- [23] *Tails*,
<https://tails.boum.org/>.
- [24] VirusTotal,
Putting the spotlight on firmware malware,
http://blog.virustotal.com/2016/01/putting-spotlight-on-firmware-malware_27.html?m=1, 2015.
- [25] *Flashrom*,
<https://www.flashrom.org/Flashrom>.
- [26] *Copernicus*,
<https://www.blackhat.com/docs/us-13/US-13-Butterworth-BIOS-Security-Code.zip>.
- [27] Intel,
Intel Processor Serial Number,
<http://www.intel.com/design/pentiumiii/applnots/24512501.pdf>,
1999.
- [28] Alan Cox, Jean Delvare,
dmidecode,
<http://www.nongnu.org/dmidecode/>.
- [29] *Distributed Management Task Force (DMTF). System Management Bios (SMBIOS) Reference Specification*,
http://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.0.0.pdf, 2015.
- [30] *lscpu*,
<http://manpages.courier-mta.org/htmlman1/lscpu.1.html>.
- [31] *lshw*,
<http://linux.die.net/man/1/lshw>
- [32] *hwinfo*,
<http://linux.fm4dd.com/en/man8/hwinfo.htm>.
- [33] Gamma, Helm, Johnson, Vlissides,
Design Patterns, elementi per il riuso di software a oggetti,
Visitor Pattern p.333, 2015.
- [34] Gamma, Helm, Johnson, Vlissides,
Design Patterns, elementi per il riuso di software a oggetti,
Factory Method p.107, 2015.

- [35] Gamma, Helm, Johnson, Vlissides,
Design Patterns, elementi per il riuso di software a oggetti,
Template Method p.327, 2015.
- [36] Techopedia,
<https://www.techopedia.com/definition/10284/integrity>.
- [37] [http://community.cadence.com/CSSharedFiles/blogs/sd/
VPLayers.jpg](http://community.cadence.com/CSSharedFiles/blogs/sd/VPLayers.jpg).
- [38] <http://www.databe.com/articles/drive-firmware.jpg>.
- [39] *dmidecode manual*,
<http://linux.die.net/man/8/dmidecode>.
- [40] *Definizione di ATA*,
[https://d2hcl8anv7xxg0.cloudfront.net/definition/2142/
advanced-technology-attachment-ata](https://d2hcl8anv7xxg0.cloudfront.net/definition/2142/advanced-technology-attachment-ata).
- [41] Eclipse Foundation,
Eclipse,
<https://eclipse.org>.
- [42] *GitHub*,
<https://github.com/>.
- [43] *GNU General Public License V.3*,
<https://www.gnu.org/licenses/gpl-3.0.html>.