Università degli Studi di Salerno
Dipartimento di Informatica
Corso di Laurea Magistrale in Informatica

Tesi di Laurea in Cybersecurity

Un sistema reputation-based in logica P2P per il blocco del malware a livello di Sistema Operativo

Relatore
Ch.mo Prof.
Francesco Palmieri

Candidato
Giovanni De Costanzo
Matr. 0522500347

Anno Accademico 2016-2017

Ringraziamenti

Vorrei ringraziare i miei genitori, e in particolare mia mamma, per i continui sacrifici ed il supporto che mi hanno dato durante questi due anni.

Ringrazio mia sorella, che non perde mai l'occasione per regalare un sorriso.

Ringrazio Luigi L., amico di una vita, che sin della triennale ha intrapreso con me questo percorso e mi ha dato i giusti stimoli quando ne avevo bisogno.

Infine ringrazio Rita, Luigi e Alessandro per aver condiviso insieme questi due anni intensi, e Mario, per la sua sempre divertente compagnia.

Sommario

Nel corso degli ultimi anni, parallelamente all'enorme diffusione tecnologica, è aumentata anche la quantità di pericoli correlati ai sistemi informatici. Una delle minacce principali è senz'altro rappresentata dai malware, ovvero software malevoli in grado di apportare danni di vario tipo all'interno di un sistema. Non sempre si è in grado di riparare i danni apportati da un malware, come potrebbe essere la cifratura dei dati su disco da parte di un ransomware. Si evince dunque la necessità di avere delle contromisure forti, in grado di bloccare sul nascere un'eventuale azione malevola. Nel presente lavoro di tesi viene proposto un meccanismo in grado di bloccare l'esecuzione del software che non è ritenuto affidabile. L'implementazione prevede l'utilizzo di un modulo kernel che intercetta le chiamate di sistema per l'esecuzione di un file, verificandone l'affidabilità attraverso un sistema di classificazione. I file eseguibili vengono identificati mediante il calcolo di un hash SHA-2, utilizzato come chiave all'interno di una distributed hash table Kademlia. Ad ogni eseguibile è associato un punteggio che determina la sua classificazione come affidabile o malevolo. Il punteggio è definito dalle segnalazioni effettuate da utenti registrati, i quali contribuiscono a mantenere aggiornato il sistema. Ogni utente ha una reputazione che varia in base al numero di segnalazioni corrette effettuate e che determina il peso dato a queste ultime. La scrittura sui nodi della DHT Kademlia può avvenire soltanto da parte di un'entità centrale, la quale raccoglie le segnalazioni degli utenti e successivamente invia i punteggi aggiornati, accompagnati da una firma digitale. Il sistema introduce quindi un approccio basato su whitelist che, a differenza dei meccanismi esistenti, definisce cosa è sicuro eseguire, bloccando tutto il resto. Si è visto infine che la latenza introdotta dal modulo kernel è minima, e mantiene dunque il sistema utilizzabile.

Indice

Elenco dei codici

Elenco delle figure

Capitolo 1

Introduzione

La diffusione tecnologica, ed in particolare dei sistemi informatici, è in enorme crescita negli ultimi anni. Secondo un report annuale pubblicato da Cisco [7], nel 2021 ogni persona possiederà in media più di 3 dispositivi connessi alla rete Internet. Tuttavia, questa crescita non è accompagnata di pari passo da un'adeguata consapevolezza, da parte degli utenti, riguardo al tema della sicurezza informatica. Al crescere del numero dei dispositivi in rete, infatti, è aumentato anche il quantitativo di dati sensibili, di servizi bancari online e di sistemi di messaggistica, che hanno attirato l'attenzione della criminalità informatica. Il rischio di frodi, furti digitali, compromissione della privacy, è sempre dietro l'angolo e tiene costantemente impegnati gli addetti ai lavori, i quali cercano di trovare ogni volta nuove contromisure.

Uno dei pericoli principali è senz'altro rappresentato dal malware, termine con il quale si identifica qualsiasi software creato con l'intenzione di arrecare danni più o meno gravi all'interno del sistema nel quale viene eseguito. I primi malware si sono diffusi già a partire dagli anni '80, per poi suddividersi nel corso degli anni in diverse categorie, quali: virus, worm, trojan, spyware, adware, rootkit, ed infine ransomware. Proprio quest'ultimo è riuscito a conquistare la scena negli ultimi anni. Un esempio recentissimo è quello del ransomware WannaCry [30], il quale ha rischiato di mettere in ginocchio diverse strutture in vari paesi, chiedendo il pagamento di un riscatto in Bitcoin. L'obiettivo del ransomware infatti, è quello di cifrare i dati presenti all'interno del sistema infettato, per poi richiedere il pagamento di un riscatto elettronico per consentire alla vittima di recuperarli.

Un altro malware di cui si è molto discusso negli ultimi anni è Stuxnet, sviluppato nel 2009 dagli americani in collaborazione con il governo israeliano [29]. Il malware venne creato per contrastare il programma nucleare iraniano, tuttavia se ne perse il controllo ed il worm iniziò a diffondersi anche su macchine esterne alla centrale.
Questo caso dovrebbe maggiormente farci capire che il malware non è più un pericolo per i semplici personal computer, ma è potenzialmente un'arma digitale che potrebbe scatenare una cyberguerra.
Si è quindi resa necessaria una incessante corsa alla ricerca di nuove contromisure per contrastare, e magari prevenire, l'azione di malware sempre più complessi e difficili da identificare.

1.1 Obiettivo

Gli attuali strumenti e meccanismi esistenti per il contrasto dei malware, non sempre sono efficaci nella loro identificazione o comunque capaci di arrestare tempestivamente la loro azione. L'obiettivo del presente lavoro di tesi è stato dunque quello di progettare un sistema in grado di arrestare qualsiasi software malevolo, prima che esso possa causare danni al sistema. Nel caso dei ransomware infatti, diventa di cruciale importanza riuscire ad impedire l'esecuzione del payload, in quanto comporterebbe la perdita dei dati sulla macchina vittima. L'idea è stata quella di realizzare un sistema collaborativo, basato su un meccanismo di reputazione degli utenti, che raccogliesse segnalazioni riguardo la genuinità del software per stabilire se bloccare o meno la sua esecuzione.

1.2 Struttura della tesi

Nel capitolo 2, verrà fornito il background per il lavoro svolto e saranno presentate le principali tecniche esistenti per la rilevazione del malware. Nel terzo capitolo sarà presentata, in termini di modello, la soluzione proposta nel presente lavoro di tesi. Nei capitoli 4 e 5, saranno invece proposti i punti cardine relativi all'implementazione dell'idea ed una prova di concetto, ovvero un'esempio di esecuzione del modello implementato. Infine, nel sesto capitolo saranno tratte delle conclusioni sul lavoro svolto e verranno indicati dei possibili sviluppi futuri.

Capitolo 2

Background

In questo capitolo sarà fornito il background di informazioni necessarie per la comprensione del lavoro di tesi presentato.

2.1 Malware

Il termine *malware* fu coniato da Yisrael Radai nel 1990 [9] ed è una abbreviazione inglese di *malicious software*, ovvero software malevolo. Solitamente un malware viene propagato inserendo parti di codice dannoso all'interno di uno software già esistente. Il frammento di codice potrebbe essere scritto all'interno di una applicazione già esistente direttamente in linguaggio macchina, oppure in uno script, in un programma di sistema, o anche all'interno del codice di boot del computer [31]. Può essere realizzato per diversi scopi, come ad esempio disturbare le operazioni di un sistema, cancellare file o directory, raccogliere informazioni sensibili, ottenere accessi non autorizzati a risorse di sistema, mostrare pubblicità invasiva, ecc.
In base al tipo di azione intrapresa dal malware, possiamo ad oggi riconoscere le seguenti principali categorie:

- **Adware**: sono programmi progettati per visualizzare annunci pubblicitari sul computer dell'utente, reindirizzando le richieste verso siti web pubblicitari e raccogliendo dati di marketing [15];
- **Backdoor**: letteralmente "porta sul retro", consiste di una tecnica per accedere ad un sistema informatico bypassando i suoi meccanismi di sicurezza. Uno sviluppatore potrebbe creare una backdoor in modo

che sia possibile accedere ad un'applicazione o al sistema operativo per la risoluzione di problemi o per altri scopi [1];

- **Botnet**: il termine deriva dalla composizione della parole inglesi *robot* e *network*. I cybercriminali sfruttano diverse tecniche per infettare e prendere il controllo di molte macchine ed organizzarle in una rete di "bots" amministrabili da remoto. Controllando migliaia di bots, è possibile lanciare ad esempio degli attacchi *DDoS* (Distributed Denial of Service) [14];

- **Ransomware**: è un tipo di malware progettato per estorcere denaro alle vittime (dall'inglese *ransom*, riscatto). Spesso richiede un pagamento per annullare le modifiche apportate al sistema della vittima, che consistono nel cifrare i dati presenti sul disco oppure bloccare il normale accesso al sistema [13];

- **Rootkit**: è un programma (o un insieme di programmi) che installa ed esegue codice in un sistema senza il consenso o la conoscenza dell'utente finale. Il rootkit si rende invisibile sulla macchina, fornendo un ambiente non rilevabile per l'esecuzione di codice dannoso. Viene installato sui sistemi attraverso l'ingegneria sociale, l'esecuzione di malware o semplicemente la navigazione su un sito dannoso. Una volta installato, un utente malintenzionato è in grado di eseguire qualsiasi funzione sul sistema, incluso l'accesso remoto ad esso e l'intercettazione del traffico [22];

- **Spyware**: software progettato per raccogliere dati da un sistema e inoltrarli a terzi, senza la consapevolezza dell'utente. Questo include spesso la raccolta di dati riservati come password, pin e numeri di carte di credito [16];

- **Trojan horse**: il nome prende spunto dal poema di Omero, dove si narra del famoso cavallo di Troia. Questo malware infatti, viene spesso mascherato come software legittimo. Gli utenti sono in genere ingannati da una qualche forma di ingegneria sociale che consente il caricamento e l'esecuzione di Trojan nei loro sistemi. Una volta attivati, i Trojan possono consentire ai cybercriminali di spiare, rubare dati sensibili e ottenere accesso remoto al sistema [17];

- **Worm**: è un agente software auto-replicante, ovvero che è in grado di diffondere copie funzionali di se stesso o dei suoi segmenti ad altri sistemi informatici. La propagazione avviene normalmente tramite connessioni di rete o allegati di posta elettronica [23];

- **Virus**: è un tipo di malware che si propaga inserendo una copia di se stesso all'interno di un altro software. Si diffonde da un computer all'altro piantando l'infezione durante il transito e può causare danni più o meno gravi all'interno di un sistema. Solitamente sono collegati ad un file eseguibile, dunque il virus può esistere in un sistema, ma sarà attivabile soltanto dall'azione di un utente [6].

2.1.1 Malware Analysis

La Malware Analysis è il processo di identificazione del comportamento di un malware, di quello che fa e di quali sono i suoi obiettivi [8]. La Malware Analysis prevede un lungo e complesso processo che include attività di forensic, reverse engineering, disassemblaggio e debugging. L'obiettivo è quello di capire il funzionamento del malware in modo da sviluppare adeguate contromisure per proteggerci dagli attacchi.
Esistono due diverse metodologie per la l'analisi del malware: **analisi statica** (analisi del codice) e **analisi dinamica** (analisi del comportamento). Queste due tecniche consentono di comprendere rapidamente e in maniera dettagliata, il rischio e le intenzioni di un malware.

Analisi Statica

L'analisi statica non prevede l'esecuzione del malware, tuttavia richiede una buona conoscenza di programmazione e del linguaggio assembly. Generalmente infatti, il codice sorgente del malware non è disponibile ed è quindi necessario disassemblarlo e decompilarlo, prima di poter procedere con la fase di reverse engineering e di analisi del codice assembly di basso livello. Gli analisti spesso preferiscono eseguire prima l'analisi statica in quanto più sicura rispetto all'analisi dinamica. Tuttavia malware sempre più sofisticati rendono difficile eseguire l'analisi statica, implementando tecniche anti-debugging per evitare che il codice venga analizzato.

Analisi Dinamica

L'analisi dinamica è un processo che consiste nell'esecuzione del malware per poterne studiare il comportamento e le modifiche che esso apporta al sistema. Infettare una macchina con un malware potrebbe essere molto pericoloso, dunque è necessario lavorare in un ambiente sicuro (una macchina virtuale o una sandbox) e che sia collegato ad una rete separata da quella di produzione. Con l'analisi dinamica è possibile monitorare i cambiamenti apportati al filesystem e ai registri, i processi e le comunicazioni di rete. Il vantaggio di questa tecnica risiede nel fatto che è possibile capire a pieno come funziona il software malevolo. Tuttavia, gli autori di malware utilizzano talvolta tecniche anti-virtual machine (anti-VM) per ostacolare i tentativi di analisi. Con queste tecniche, il malware tenta di rilevare se viene eseguito all'interno di una macchina virtuale e, in tal caso, può agire in modo diverso o semplicemente non eseguire, complicando il processo di analisi [21].

Un'altra tecnica con cui gli autori di malware cercano di ostacolarne il riconoscimento è il packing. I programmi di packing, conosciuti come *packers*, si sono molto diffusi in quanto aiutano un malware a nascondersi dai software antivirus, complicando la fase di analisi e riducendo le dimensioni dell'eseguibile malevolo. I malware impacchettati infatti, devono essere prima spacchettati per poter procedere, e ciò rende la fase di analisi più complicata.

2.1.2 Antivirus

Un *antivirus* (talvolta abbreviato come AV), è un programma utilizzato per prevenire, rilevare e rimuovere software malizioso. Oltre a fornire protezione contro la maggior parte dei malware esistenti, un antivirus può proteggere anche contro altri tipi di pericoli informatici, come URL malevoli, spam, phishing, ecc.
I software antivirus generalmente sfruttano diversi meccanismi per l'identificazione del malware:

- **rivelazione signature-based**: i software antivirus tradizionali si basano fortemente sull'utilizzo di firme per l'identificazione del malware. Quando un'azienda antivirus analizza un nuovo software e determina che esso è un malware, viene generata una firma del file ed aggiunta al database del software antivirus [4]. Il problema è che, sebbene questo approccio possa sembrare efficace, gli autori di malware hanno

cercato di rimanere un passo avanti rispetto a tali meccanismi, scrivendo malware polimorfi e metamorfi, ovvero che codificano parti di se stessi oppure si modificano per potersi mascherare, in modo da non corrispondere più alle firme del virus nel dizionario [32].

- **euristiche**: Molti virus iniziano come una singola infezione e, attraverso mutazioni o raffinamenti apportati da altri attaccanti, possono crescere in decine di ceppi leggermente diversi, chiamati varianti. Il rilevamento generico si riferisce alla rilevazione e alla rimozione di più minacce utilizzando una singola definizione di virus [18]. I ricercatori trovano aree comuni che tutti i virus di una famiglia condividono in modo univoco e quindi possono creare una sola firma generica. Queste firme contengono spesso un codice non contiguo, utilizzando caratteri jolly (*wildcard characters*) dove ci sono delle differenze. Questi caratteri jolly consentono allo scanner di rilevare i virus anche se sono imbottiti di codice extra e senza senso [10]. Tale metodo è chiamato "rilevamento euristico".

- **protezione real-time**: Questo metodo consente di monitorare attività sospette all'interno dei sistemi, analizzando i processi caricati nella memoria RAM del computer. Questo avviene quando si apre un'email, si naviga sul web, oppure viene aperto un file già presente sulla macchina.

2.1.3 Altri meccanismi di protezione

Flag di quarantena

A partire dalla versione Leopard 10.5 dei sistemi Mac OS X, Apple ha introdotto una funzionalità chiamata *quarantena dei file* [3]. Viene aggiunto un flag speciale nei metadati dei file scaricati da Internet tramite le applicazioni Safari, Messaggi, e Mail. Quando un utente prova ad aprire il file eseguibile (ad esempio un'applicazione o uno script Unix) che è stato contrassegnato come in quarantena, riceverà un avviso e gli verrà chiesto se è veramente intenzionato ad aprire il file (Fig. 2.1). Questo potrebbe essere un buon approccio per proteggere l'utente. Tuttavia, alcune persone infastidite dal messaggio, provano a rimuoverlo o comunque accettano di proseguire con l'apertura del file, senza pensare a quello che stanno approvando. Un'altra limitazione di questo sistema è che esso funziona soltanto con i file scaricati da Internet, e non con quelli copiati da dischi e memorie esterne, CD o

DVD, e volumi di rete condivisi. Inoltre, non tutte le applicazioni esistenti supportano questo meccanismo di quarantena quando effettuano il download di un file [20]. Apple nel tempo ha migliorato questo sistema, aggiungendo un meccanismo di protezione di base contro i malware, chiamato XProtect. Quest'ultimo esamina i file messi in quarantena quando sono aperti, ed avvisa l'utente se il file corrisponde ad un malware noto. La debolezza di questo meccanismo risiede nel fatto che esso funziona soltanto con malware noti, mentre è inutile contro quelli nuovi.

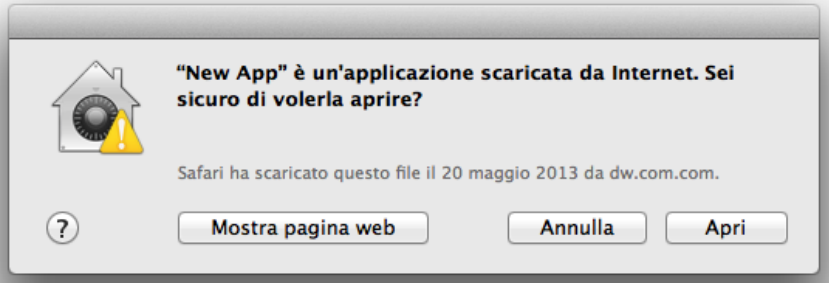

Figura 2.1: Messaggio di quarantena mostrato nei sistemi OS X

Gatekeeper

Dalla versione 10.8 di Mac OS X (Mountain Lion), Apple ha aggiunto un'ulteriore funzionalità di protezione, chiamata *Gatekeeper*. Tale sistema consente all'utente di eseguire solo certe classi di applicazioni, bloccando interamente le altre. Tali impostazioni possono essere modificate nelle preferenze di sistema (Fig. 2.2), e l'utente può scegliere di eseguire applicazioni provenienti da:

- Mac App Store: consente di aprire solo le app provenienti dal Mac App Store;
- Mac App Store e sviluppatori identificati: consente di aprire solo le app provenienti dal Mac App Store e da sviluppatori che utilizzano Gatekeeper;

- Dovunque: consente l'esecuzione di qualsiasi applicazione, senza considerarne l'origine su internet; Gatekeeper è disattivato.

Anche Gatekeeper soffre di limitazioni, a causa del fatto che si basa sul meccanismo di quarantena, ereditandone quindi i difetti. Gatekeeper può bloccare solo le applicazioni contrassegnate come in quarantena, mentre le altre saranno aperte senza restrizioni. Dunque esso non funzionerà se è stato completamente disattivato il meccanismo di quarantena. Questo significa che non tutti i malware possono essere bloccati. Inoltre, se ad esempio un malware sfruttasse delle vulnerabilità di Java per installarsi nel sistema, esso bypasserebbe l'intero meccanismo di protezione.

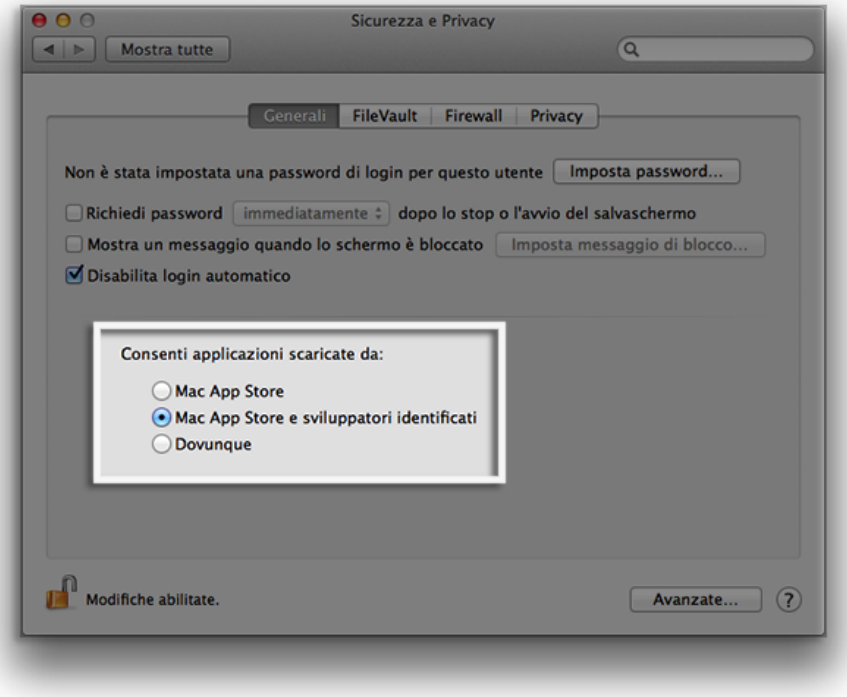

Figura 2.2: Opzioni disponibili per Gatekeeper

User Account Control

A partire da Windows Vista, Microsoft ha introdotto il sistema *UAC* (User Account Control), per impedire che i malware possano danneggiare il computer [24]. Con questa funzionalità, tutte le applicazioni vengono sempre eseguite in un contesto protetto, senza i privilegi di amministratore (a meno che non vengano specificamente assegnati). UAC può bloccare l'installazione automatica di applicazioni non autorizzate e impedire modifiche involontarie alle impostazioni di sistema. Sono stati inoltre introdotti degli avvisi di protezione, nei quali si chiede conferma all'utente quando si cerca di eseguire applicazioni con autore sconosciuto, oppure aprire file scaricati da Internet o copiati da altri computer (Fig. 2.3).

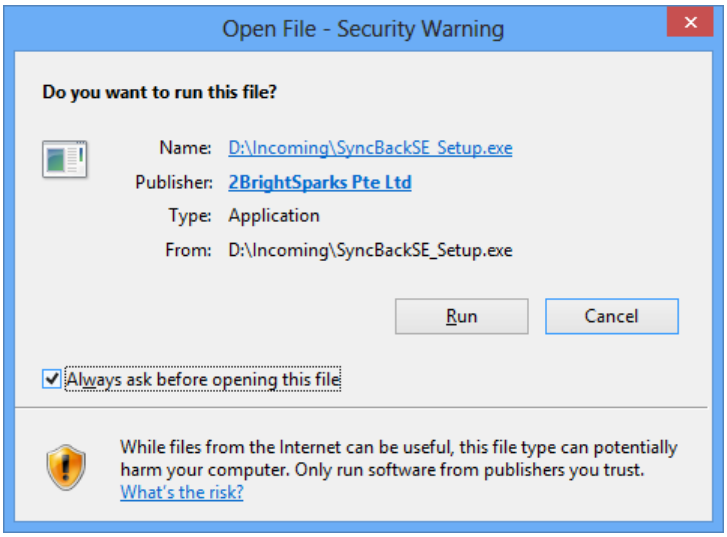

Figura 2.3: Security warning mostrato nei sistemi Windows

Trusted Application Mode

Kaspersky, la nota azienda specializzata in software per la sicurezza informatica, ha introdotto all'interno della propria suite antivirus per sistemi Windows, una modalità chiamata *Trusted Application* [12]. Quest'ultima blocca tutte le applicazioni che non sono considerate attendibili, come ad esempio quelle in cui non sono presenti informazioni nel database Kaspersky, o quelle ricevute da una fonte inaffidabile (Fig. 2.4).

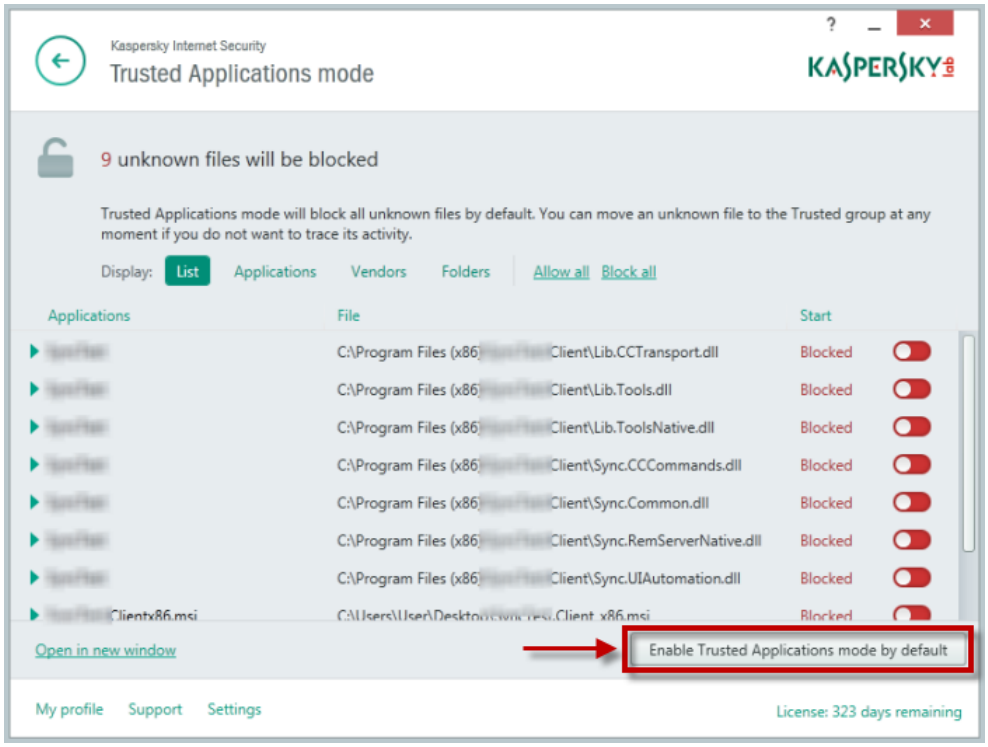

Figura 2.4: Trusted Application mode in Kaspersky Internet Security 2015

2.2 Sistemi di reputazione

Il concetto di *reputazione* concerne alla credibilità che un individuo ha all'interno di una comunità sociale. Avere una buona reputazione consente di apparire credibili e affidabili agli occhi della società. Nella rete Internet, una delle principali cause di cattiva reputazione sono la pubblicazione di contenuti falsi o imprecisi, e ciò rappresenta un grave problema. Infatti chiunque ha la possibilità di accedere al Web ed è in grado, senza particolari conoscenze tecniche, di pubblicare materiale falso o non attendibile, come ad esempio una notizia falsa.
Negli ultimi anni sono stati fatti vari tentativi nel cercare di realizzare dei sistemi che consentissero di ottenere una stima automatizzata della reputazione di un individuo nella rete. Alcuni usi comuni di questi sistemi possono essere trovati nei siti web di E-commerce come eBay o Amazon.com, oppure in comunità di consulenza online come Stack Exchange.
Un *reputation system* consente dunque agli utenti di valutarsi reciprocamente all'interno di una comunità online, al fine di costruire la fiducia attraverso la reputazione. L'idea fondamentale è quella di consentire alle parti di valutarsi l'un l'altra, ad esempio dopo la conclusione di una transazione, e di utilizzare i voti aggregati per ottenere un punteggio di fiducia o reputazione, che può aiutare le altre parti a decidere se effettuare o meno delle transazioni con quella parte in futuro. Una naturale conseguenza di questi sistemi è che forniscono un incentivo per un buon comportamento, e quindi tendono ad avere un effetto positivo sulla qualità del mercato [5].

Ulteriori ambiti in cui vengono adottati meccanismi di reputazione sono i seguenti:

- Motori di ricerca web (es. PageRank);
- Community di sviluppatori (es. StackOverflow);
- Internet Security (es. TrustedSource);
- Email (es. Vipul's Razor, utilizzato per il filtraggio dello spam);
- Mondo accademico (es. h-index di un ricercatore).

Paul Resnick, membro di ACM (Association of Computing Machinery), descrive tre proprietà necessarie affinché un sistema di reputazione funzioni in modo efficace:

- Le entità devono avere una lunga durata e creare aspettative attendibili sulle future interazioni.
- Devono catturare e distribuire feedback sulle interazioni precedenti.
- Devono utilizzare i feedback per guidare la fiducia.

Queste tre proprietà sono di fondamentale importanza per costruire reputazioni affidabili, e tutto ruota intorno ad un elemento fondamentale che è il feedback utente. Senza di essi non è pensabile un sistema che possa sostenere un meccanismo di fiducia.

Elicitare i feedback degli utenti può avere tre problemi correlati:

- Il primo problema è la disponibilità degli utenti a fornire un feedback quando questo non è obbligatorio. Se in una comunità online si verifica un grande flusso di interazioni, ma non vengono raccolti feedback, l'ambiente di fiducia e di reputazione non si può formare;
- Il secondo è ottenere feedback negativi da parte degli utenti, i quali sono condizionati da diversi fattori, come la paura di ritorsione. Quando i feedback non sono anonimi infatti, molti utenti hanno paura di riceverne uno negativo a loro volta;
- L'ultimo problema è quello di elicitare feedback onesti da parte degli utenti. Sebbene non esista un metodo concreto per stabilire la veridicità di feedback onesti in una comunità, nuovi utenti saranno più propensi a lasciare un feedback onesto.

Altre insidie nei sistemi di reputazione includono il cambio di identità e la discriminazione. Anche in questo caso è necessario regolare le azioni degli utenti al fine di ottenere feedback accurati e coerenti.

Attacchi ai sistemi di reputazione

I sistemi di reputazione sono in genere vulnerabili ad alcuni tipi di attacchi, i quali possono essere classificati in base agli obbiettivi dell'attaccante. In base

al contesto e al tipo di attacco, è tuttavia possibile applicare dei meccanismi di difesa.

- **Self-promoting Attack**: l'attaccante aumenta la propria reputazione in maniera falsata. Un esempio è il *Sybil attack*, dove l'attaccante sovverte il sistema di reputazione creando un gran numero di pseudonimi, usandoli per guadagnare un'influenza spropositata;
- **Whitewashing Attack**: viene sfruttata qualche vulnerabilità del sistema per aggiornare la propria reputazione. Si può anche combinare questo con altri tipi di attacchi per renderlo più efficace;
- **Slandering Attack**: l'attaccante riporta dati falsi per abbassare la reputazione di certi utenti;
- **Denial of Service Attack**: questo attacco impedisce la diffusione dei valori di reputazione nei sistemi utilizzando attacchi Denial of Service.
- **Man in the Middle**: un attaccante potrebbe decidere di intercettare il traffico e di manomettere i feedback inviati dagli altri utenti.

Capitolo 3

Soluzione proposta

Le attuali tecniche di analisi del malware non possono, per loro ovvia natura, contrastare l'azione di nuovi software malevoli non ancora analizzati e diffusi attraverso i vari canali di rete. Purtroppo, in alcuni casi più di altri, si evince la necessità di avere delle misure di difesa preventive, che impediscano l'iniziare di qualsiasi azione dannosa all'interno di un sistema, come potrebbe essere la cifratura dei dati da parte di un ransomware.

Il problema presente nella maggior parte dei meccanismi di protezione esistenti è che essi si basano su un modello a base negativa (*blacklist*), ovvero definiscono cosa è pericoloso, mentre lasciano implicitamente passare tutto il resto. L'idea alla base del presente lavoro, è stata dunque quella di bloccare a livello di sistema operativo, l'esecuzione di qualsiasi software non noto. Viene quindi adottato un modello a base positiva (*whitelist*), che definisce cosa è ammesso lasciar passare, bloccando tutto il resto.

Il passo successivo è stato quello di formulare un modello per la classificazione del software, in merito alla sua natura genuina o malevola. La classificazione consente al sistema operativo di stabilire se può o meno eseguire un software. Data l'enorme quantità di software esistenti, pensare che l'intera classificazione possa avvenire ad opera di un singolo individuo, o di un ente, è difatti impossibile. Per questo motivo, si è deciso di realizzare un modello di classificazione basato sulla collaborazione degli utenti. In questo modo potranno contribuire nel corso del tempo intere community, oltre alle varie aziende che già si occupano di analisi del malware.

3.1 Modello

Il modello realizzato prevede che per ogni file eseguibile venga calcolato un valore di hash, con il quale esso sarà identificato univocamente all'interno del sistema. Ogni nuovo eseguibile ha un punteggio iniziale di affidabilità pari a zero. Quando il punteggio raggiunge un valore maggiore o uguale ad uno, allora l'eseguibile è classificato come affidabile; se invece il suo valore diventa minore o uguale di meno uno, l'eseguibile è considerato malevolo. Il punteggio di ogni eseguile è determinato dagli utenti registrati al sistema, i quali contribuiscono inviando segnalazioni in merito alla sua affidabilità.

Occorre tuttavia fare alcune osservazioni sull'idea di base appena descritta. Infatti questa potrebbe essere vulnerabile ad alcuni tipi di attacchi, quali:

- **Bad Mouthing Attack**: un utente disonesto potrebbe effettuare delle segnalazioni errate, al fine di incrementare il punteggio di un eseguibile malevolo, oppure danneggiare quello di un eseguibile affidabile. Per arginare questa problematica, viene introdotto un modello di reputazione per gli utenti registrati. Le segnalazioni vengono quindi pesate in base alla corrispettiva reputazione di ogni utente, la quale potrà aumentare o diminuire in base al numero di segnalazioni corrette o errate.
  Un utente malizioso potrebbe tuttavia, decidere di effettuare un certo numero di segnalazioni corrette, al fine di accrescere la propria reputazione, prima di effettuare una segnalazione disonesta. Per scoraggiare questo tipo di comportamento, utenti che effettuano segnalazioni errate, vengono penalizzati con un decremento della loro reputazione proporzionato alla reputazione stessa. Questo significa che, più sarà alta la reputazione dell'utente, maggiore sarà il decremento in caso di una segnalazione errata.

- **Sybil attack**: utenti maliziosi potrebbero sovvertire il sistema registrandosi con un numero spropositato di pseudonimi. Infatti, numerosi utenti con una bassa reputazione, potrebbero comunque alterare il punteggio di un eseguibile. Per ovviare a questo problema, il processo di registrazione degli utenti prevede il rilascio di un certificato digitale da parte di una CA (Certification Authority), dopo una opportuna verifica dell'identità. Questo significa che il processo di registrazione al sistema sarà controllato, e nessun utente potrà registrarsi più volte utilizzando diversi pseudonimi.

- **Man in the Middle**: Un utente con cattive intenzioni, potrebbe decidere di intercettare e manomettere le segnalazioni di altri utenti, con gli scopi di alterare il punteggio di un eseguibile oppure danneggiare la reputazione di un utente onesto. Per prevenire questo tipo di attacchi, tutte le comunicazioni tra il server che raccoglie le segnalazioni e gli utenti vi effettuano l'accesso, avverranno utilizzando una connessione cifrata.

- **Denial of Services**: I sistemi centralizzati sono tipicamente vulnerabili ad attacchi Denial of Service (DoS); quelli distribuiti invece, sono solitamente meno vulnerabili se viene impiegata una sufficiente ridondanza, in modo che un comportamento scorretto, o la perdita di alcuni partecipanti, non influenzeranno il funzionamento dell'intero sistema. La soluzione proposta adotterà dunque una architettura decentralizzata, che sarà in grado di ridurre al minimo gli effetti di questo tipo di attacco.

Alla luce delle osservazioni fatte, è possibile aggiungere al modello anche un meccanismo di reputazione degli utenti.

Ogni nuovo utente registrato al sistema ha una reputazione iniziale pari a 0,5 e il cui valore potrà oscillare da un minimo di zero ad un massimo di uno. Un utente la cui reputazione scende al di sotto di 0,01 non sarà più in grado di effettuare segnalazioni. La reputazione di ogni utente è utilizzata per dare un peso alle segnalazioni effettuate: più segnalazioni corrette effettua un utente, più la sua reputazione cresce; viceversa, all'aumentare del numero di segnalazioni errate, la reputazione dell'utente decresce. Per determinare se la segnalazione relativa ad un eseguibile è corretta o meno, il punteggio complessivo associato ad un eseguibile, deve raggiungere un valore tale da poter essere classificato come affidabile o malevolo. Se ad esempio, il punteggio diventa affidabile, allora tutti gli utenti che hanno effettuato una segnalazione positiva per l'eseguibile, vedranno accrescere la propria reputazione. Al contrario, coloro che hanno inviato una segnalazione negativa, vedranno abbassarsi la propria reputazione. Una volta che un file eseguibile è stato classificato, le successive segnalazioni da parte di altri utenti, coerenti con la classificazione in atto, accresceranno (o diminuiranno) ulteriormente il punteggio, ma non andranno a modificare la reputazione dei relativi utenti. Questo perché un utente malizioso potrebbe inviare segnalazioni positive per

eseguibili già classificati, al fine di aumentare facilmente la propria reputazione.
Il punteggio di un eseguibile non viene aggiornato istantaneamente ad ogni segnalazione di un utente, ma allo scadere di una finestra temporale fissata. Il sistema quindi raccoglie le varie segnalazioni ricevute all'interno della finestra temporale e, al suo scadere, calcola una media pesata che sarà aggiunta al punteggio dell'eseguibile.

3.1.1 Formalizzazione del modello

Il modello appena descritto può essere formalizzato come segue.
Chiamiamo $\boldsymbol{e}$ un eseguibile ed $\boldsymbol{e_p}$ il punteggio ad esso associato. Ogni nuovo eseguibile e avrà un punteggio iniziale $e_p = 0$, e sarà classificato come **affidabile** o **malevolo**, rispettivamente per valori $e_p \geq 1$ ed $e_p \leq -1$.

Sia $\boldsymbol{u}$ un utente registrato al sistema e $\boldsymbol{rep_u{}^{(t)}}$ la sua reputazione al tempo $\boldsymbol{t}$. Considerando il tempo $t = 0$ come il momento in cui un utente si registra al sistema, avremo $rep_u{}^{(t)} = 0{,}5$ per ogni nuovo utente u. Il valore di rep_u potrà oscillare all'interno dell'intervallo [0,1].

Indichiamo con $\boldsymbol{T}$ una finestra temporale dalla durata fissata. Il sistema raccoglie tutte le segnalazioni pervenute per ogni eseguibile e allo scadere di T, calcola una media pesata delle segnalazioni per ogni eseguibile e, sommandola ad e_p.
Considerando $\boldsymbol{s_{e,u}}$ come la segnalazione relativa all'eseguibile e, inviata dall'utente u, nell'arco della finestra temporale T, $s_{e,u}$ potrà valere 1 oppure -1, rispettivamente nei casi di segnalazione positiva o negativa.
La media pesata delle segnalazioni, relative ad un eseguibile e, è calcolata allo scadere della finestra T come segue:

$$\boldsymbol{\mu} = \frac{\sum_{\forall u} s_{e,u} * rep_u{}^{(t)}}{n}$$

dove $\boldsymbol{n}$ è il numero di segnalazioni raccolte per l'eseguibile e, all'interno della finestra temporale T.

Dunque, il punteggio e_p aggiornato sarà:

$$e_p{}^{(t+1)} = e_p{}^{(t)} + \mu$$

Quando il punteggio e_p esce dall'intervallo $(-1, 1)$, l'eseguibile e viene classificato come affidabile o malevolo, e le segnalazioni precedentemente inviate dagli utenti contribuiranno ad aggiornare la loro reputazione. Se la segnalazione $s_{e,u}$ è coerente con la classificazione appena avvenuta, la reputazione dell'utente u cresce, in caso contrario diminuisce. In particolare, abbiamo che il valore rep_u aumenterà (1) o diminuirà (2) in base al suo valore al tempo t, secondo due costanti di proporzionalità $\boldsymbol{a}$ e $\boldsymbol{b}$, come segue:

$$(1) \qquad rep_u{}^{(t+1)} = rep_u{}^{(t)} + a * rep_u{}^{(t)}$$

$$(2) \qquad rep_u{}^{(t+1)} = rep_u{}^{(t)} - b * rep_u{}^{(t)}$$

Valori ragionevoli per le due costanti potrebbero essere $a = 0,1$ e $b = 0,5$. In questo modo, un utente che effettua una serie di segnalazioni corrette, vedrà gradualmente crescere la propria reputazione, fino a raggiungere il valore massimo. Invece, un utente che effettua una segnalazione errata, vedrà dimezzarsi la propria reputazione. Dunque questa scelta consente di dare un peso maggiore alle segnalazioni errate, in modo da scoraggiare eventuali attacchi al sistema. Infatti, quando $rep_u < 0{,}01$ l'utente u non sarà più in grado di effettuare segnalazioni.
Se e_p torna successivamente nell'intervallo $(-1, 1)$, tutti gli utenti ai quali era stata accresciuta (o diminuita) la reputazione, vedranno sottrarsi (o aggiungere) un valore pari all'incremento (o al decremento) precedentemente ricevuto.

3.2 Architettura del sistema

L'architettura proposta prevede l'utilizzo di una *distributed hash table* e di un *modulo kernel* da caricare nel sistema operativo, che verranno presentati nelle seguenti sezioni prima di procedere con la descrizione. Verrà inoltre introdotto il concetto di firma digitale.

3.2.1 Distributed Hash Table

Una distributed hash table, o *DHT*, è un sistema decentralizzato che fornisce la funzionalità di una tabella di hash, ovvero l'inserimento e il recupero di coppie chiave-valore [11]. Ogni nodo nel sistema memorizza una parte della tabella di hash. I nodi sono interconnessi in una overlay network (rete di sovrapposizione) strutturata, che consente una consegna efficiente delle richieste di una chiave e delle richieste di inserimento, dal nodo richiedente a quello che memorizza la chiave.

Ogni DHT definisce il proprio spazio di chiavi. Ad esempio, in molti casi le chiavi sono degli interi di 160 bit, ovvero l'output della funzione di hash SHA-1. Ogni nodo possiede una specifica locazione all'interno dello spazio delle chiavi e memorizza le coppie chiave-valore vicine alla propria locazione. I differenti sistemi DHT variano nel particolare algoritmo utilizzato per stabilire quale nodo memorizza quale chiave. La rete overlay strutturata fornisce una primitiva di instradamento che consente la ricerca e l'inserimento delle chiavi in modo scalabile ed affidabile, ed assicura l'efficienza del routing anche quando i nodi arrivano ed escono dal sistema (*node churn*). Anche la memorizzazione dei dati nella DHT deve gestire il node churn, e per questo le coppie chiave-valore vengono replicate su molti nodi. Per prevenire la perdita dei dati è quindi necessario mantenere un sufficiente numero di repliche. La strategia più comune prevede che, quando un nodo entra nella DHT, esso contatti i suoi vicini nello spazio di chiavi e replichi le coppie chiavi-valore memorizzate in essi.
Nella maggior parte delle implementazioni, il numero previsto di passi di routing scala in *O(log N)* con le dimensioni della rete, dove N è il numero di nodi.

Le caratteristiche di una DHT enfatizzano tre proprietà:

- **Decentralizzazione**: i nodi formano collettivamente il sistema senza alcun coordinamento centrale;

- **Scalabilità**: il sistema è strutturato in modo da garantire un funzionamento efficiente anche con milioni di nodi;
- **Tolleranza ai guasti**: il sistema risulta affidabile anche in presenza di nodi che entrano ed escono dalla rete, oppure sono frequentemente soggetti a malfunzionamenti.

Uno dei vantaggi di una DHT particolarmente interessante per il lavoro di tesi che sarà presentato, è dato appunto dalla struttura decentralizzata del sistema, che chiaramente aumenta la resistenza a possibili attacchi *DoS* (Denial of Service).

Kademlia

Kademlia è un sistema peer-to-peer basato su DHT, realizzato da Petar Maymounkov e David Mazières nel 2002 [27]. Kademlia calcola la distanza tra nodi utilizzando una metrica in *or esclusivo* (XOR) tra gli ID di due nodi, oppure tra una chiave e l'ID di un nodo: il risultato ottenuto, in binario, rappresenta la distanza. Il sistema prevede l'utilizzo del protocollo di trasporto UDP per lo scambio dei messaggi, che possono essere:

- PING: utilizzato per verificare che un nodo sia ancora attivo;
- STORE: per memorizzare una coppia chiave-valore in un nodo;
- FIND_NODE: per cercare nodi;
- FIND_VALUE: come FIND_NODE, ma se il ricevente del messaggio possiede la chiave, restituisce il valore.

Bootstrap

Per poter entrare nella rete un nodo deve prima di tutto iniziare un processo denominato *bootstrap*. In questa fase, è necessario conoscere l'indirizzo IP di un altro nodo che sia già connesso alla rete Kademlia (ricevuto da un utente oppure recuperato da una lista di nodi che dovrebbero essere sempre attivi). Se il nodo che deve iniziare il processo di bootstrap non ha mai partecipato alla rete, calcola un numero identificativo casuale non ancora assegnato ad un altro nodo e lo utilizza finché non lascia la rete.

3.2.2 Linux Kernel Module

Un modulo kernel è un pezzo di codice che può essere caricato o rimosso dal kernel su richiesta [25]. Un kernel module consente di aggiungere funzionalità al kernel senza rendere necessario il riavvio del sistema. Un esempio di modulo sono i *device driver*, che consentono al kernel di comunicare con l'hardware connesso al sistema. Senza moduli, dovremmo costruire kernel monolitici e aggiungere nuove funzionalità direttamente nell'immagine del kernel. Oltre alla dimensione, questo avrebbe lo svantaggio di dover ricompilare e riavviare il kernel ogni volta che vogliamo nuove funzionalità.

Oltre all'implementazione dei device driver, i moduli kernel possono essere utilizzati anche per altri scopi, come l'*hijacking* di una system call (chiamata di sistema). Questo tipo di operazione consente di modificare il comportamento del sistema, aggiungendo delle istruzioni che saranno eseguite ad ogni invocazione di una specifica system call. L'hijacking richiede particolare attenzione, in quanto anche un banalissimo errore nel modulo kernel, potrebbe rendere il sistema inutilizzabile e costringere ad un riavvio forzato della macchina. Per questo motivo è consigliabile effettuare tutte le prove del caso all'interno di una macchina virtuale.

User space e kernel space

Uno dei concetti fondamentali su cui si basa l'architettura dei sistemi Unix è quello della distinzione fra il cosiddetto *user space*, che contraddistingue l'ambiente in cui vengono eseguiti i programmi, e il *kernel space*, che è l'ambiente in cui viene eseguito il kernel [28]. Ogni programma in esecuzione lavora come se avesse la piena disponibilità della CPU e della memoria, ignaro del fatto che il kernel può far eseguire contemporaneamente altri programmi. Grazie a questa separazione un programma non è in grado di disturbare l'azione di un altro programma o del sistema; questo è uno dei principali motivi della stabilità di un sistema unix-like rispetto ad altri sistemi in cui non esistono queste limitazioni, o in cui i processi vengono eseguiti al livello kernel per qualche ragione. Dunque un programma in ambiente Unix non ha accesso diretto all'hardware, e questo può avvenire solo all'interno del kernel; al di fuori di quest'ultimo invece, il programmatore deve utilizzare le opportune interfacce fornite allo user space.

3.2.3 Firma digitale

La firma digitale è il risultato di una procedura informatica – detta validazione – che garantisce l'autenticità e l'integrità di documenti o messaggi digitali [26].
La firma digitale conferisce al documento informatico le seguenti caratteristiche:

- **autenticità**: la firma digitale garantisce l'identità del sottoscrittore del documento;
- **integrità**: la firma digitale assicura che il documento non sia stato modificato dopo la sottoscrizione;
- **non ripudio**: la firma digitale attribuisce piena validità legale al documento, pertanto il documento non può essere ripudiato dal sottoscrittore.

La generazione di una firma digitale necessita una coppia di chiavi digitali asimmetriche attribuite univocamente ad un soggetto, detto titolare.

La **chiave privata** è conosciuta soltanto dal titolare e viene usata per la generazione della firma digitale da apporre al messaggio. La **chiave pubblica** è invece nota a tutti, e viene usata per verificare l'autenticità della firma (e quindi del messaggio).

Tale metodo è conosciuto come crittografia asimmetrica e garantisce la piena sicurezza in quanto non è possibile ricostruire la chiave privata a partire da quella pubblica.

Se il titolare vuole creare una firma per un documento, procede in questo modo: utilizzando una funzione hash ricava l'impronta digitale del documento, detta anche *message digest*, ovvero un file dalle dimensioni ridotte (128, 160 o più bit) che contiene una sorta di codice di controllo relativo al documento stesso, quindi sfrutta la propria chiave privata per cifrare l'impronta digitale, ottenendo la firma.

Per verificare l'autenticità del documento chiunque, utilizzando la chiave pubblica, può decifrare la firma e ottenere l'impronta digitale, la quale viene

confrontata con quella ricalcolata sul documento ricevuto. Se le due impronte sono uguali, allora il documento è autentico ed integro.

RSA

Uno degli schemi di crittografia asimmetrica più diffusi è sicuramente RSA, presentato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman. Tale algoritmo si basa sull'elevata complessità computazionale della fattorizzazione in numeri primi. Possiamo definire il suo funzionamento di base come segue:

1. Si scelgono due numeri primi p e q sufficientemente grandi;
2. si calcolano il loro prodotto $n = pq$, chiamato *modulo* e il prodotto $\varphi(n) = (p-1)(q-1)$;
3. la fattorizzazione di n è considerata segreta e solo chi ha scelto p e q la conosce;
4. viene scelto un numero e (chiamato esponente pubblico), coprimo con $\varphi(n)$ e più piccolo di quest'ultimo;
5. infine si calcola d (detto esponente privato) tale che $e*d$ sia congruo ad 1 modulo $\varphi(n)$, cioè $ed \equiv 1 \ (mod \ \varphi(n))$

La chiave pubblica è (n, e) mentre la chiave privata è (n, d).

Il punto di forza dell'algoritmo risiede nel fatto che per calcolare d da e non è sufficiente conoscere n, ma è necessario $\varphi(n)$, il cui calcolo richiede tempi elevati.

X.509

X.509 è uno standard che definisce dei formati per i certificati a chiave pubblica, usati in molti protocolli Internet come *TLS/SSL*, che sta alla base di *HTTPS*. Un certificato X.509 contiene una chiave pubblica e un'identità (il nome di un host, un'organizzazione o un individuo) e può essere firmata da una certification authority (CA) oppure autonomamente. Quando un certificato è firmato da una CA attendibile, chi ne è in possesso può affidarsi alla chiave pubblica che esso contiene per stabilire connessioni protette con altre

parti, oppure validare documenti firmati digitalmente con la corrispondente chiave privata.

Sono inclusi, in X.509, anche gli standard per le implementazioni di *certificate revocation list* (CRL, liste di revoca di certificati), utilizzate per distribuire informazioni sui certificati che non sono più validi.

3.2.4 Descrizione dell'architettura

Il modello presentato prevede di classificare un vastissimo numero di file eseguibili esistenti, oltre a dover gestire le numerosissime segnalazioni da parte degli utenti. A questi numeri, ne va aggiunto uno ancora più grande, ovvero quello relativo alle richieste che arriveranno da parte dei sistemi degli utenti, riguardo al punteggio degli eseguibili che si appresteranno ad eseguire. Il numero di tali richieste è potenzialmente spropositato, in quanto il sistema potrebbe inviarne una per ogni file che prova ad eseguire. Per questo motivo, non è opportuno pensare di implementare tale modello con un'architettura centralizzata, in quanto si tradurrebbe in un single point of failure (*SPOF*), ovvero un singolo punto di fallimento, e quindi facilmente vulnerabile ad attacchi DoS.

La soluzione proposta consiste dunque di una architettura ibrida, composta da un server centrale e una distributed hash table basata su Kademlia, che costituisce la parte decentralizzata (Fig. 3.1). La DHT provvede alla memorizzazione delle coppie chiave-valore, composte dall'hash di un eseguibile e dal suo punteggio associato. Nel nostro caso, i nodi della rete P2P consentiranno la scrittura solamente ad una entità autoritativa, mentre chiunque sarà in grado di leggere e dunque richiedere il punteggio relativo ad un eseguibile. La limitazione in scrittura è garantita dall'utilizzo di una firma digitale, utilizzata ogni qualvolta venga inviata una coppia chiave-valore per la memorizzazione sulla DHT.
Tuttavia, firmare soltanto tale coppia (hash, punteggio), comporterebbe una vulnerabilità di tipo *Replay attack*. Infatti, qualsiasi attaccante è in grado di catturare un messaggio inviato sulla rete P2P, e quindi leggere la coppia chiave-valore e la relativa firma digitale. L'attaccante potrebbe decidere di conservare tale messaggio e di re-inviarlo alla rete successivamente, riuscendo così a ripristinare il vecchio punteggio associato ad un eseguibile. Per ovviare a questo problema, viene utilizzato un timestamp (marca temporale)

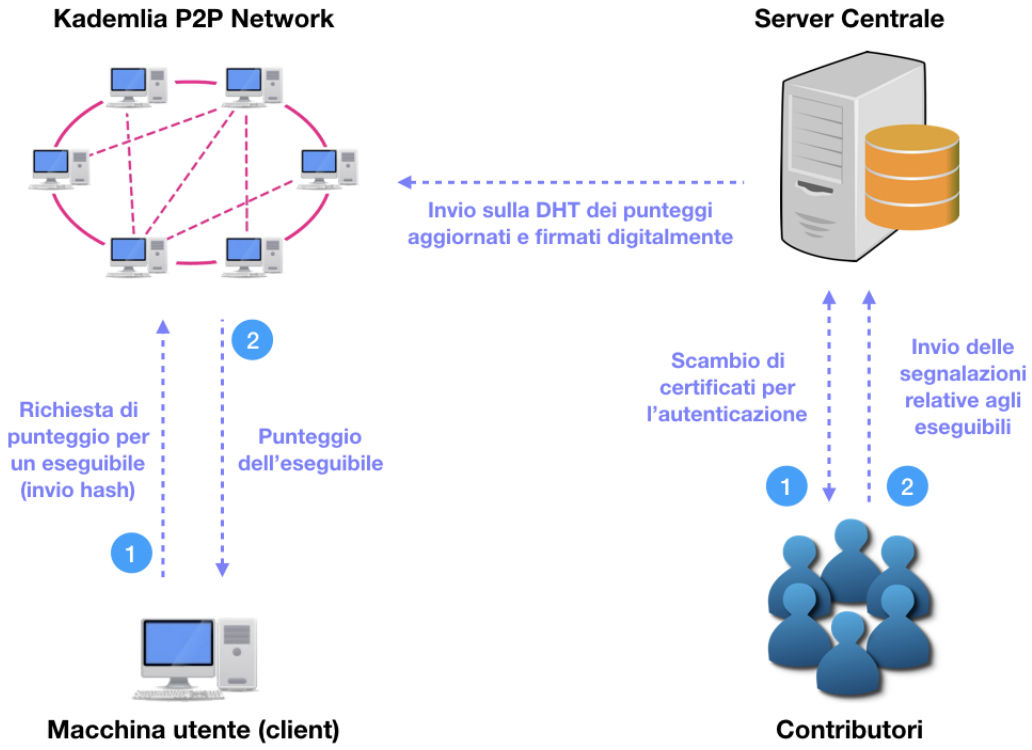

Figura 3.1: Architettura del sistema

indicante la scadenza per uno specifico messaggio. La firma digitale è quindi calcolata aggiungendo anche una data di fine validità, divenendo così inutilizzabile dopo un certo periodo di tempo stabilito.
Il server centrale funge dunque da authority, ed è l'unico in possesso della chiave privata per la firma digitale dei messaggi. La sua funzione è quella di raccogliere le segnalazioni degli utenti, previa autenticazione, e di aggiornare i punteggi degli eseguibili alla chiusura di una finestra temporale, inviando ai nodi della rete un messaggio firmato contenente il nuovo valore.

L'autenticazione degli utenti avviene mediante l'utilizzo di un certificato client (Fig. 3.2), assegnato all'utente da una CA alla fine del processo di registrazione, che prevede un'opportuna verifica della sua identità.

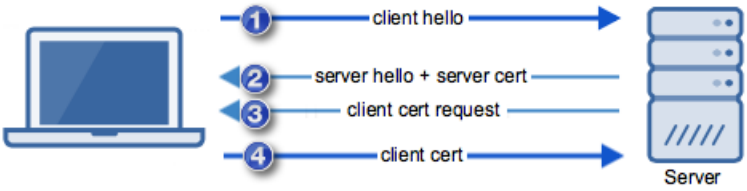

Figura 3.2: Schema di autenticazione mediante un certificato client

Le reputazioni degli utenti sono memorizzate sul server centrale, che provvede ad aggiornarle e stabilisce a chi è consentito inviare segnalazioni.

Questa separazione dei ruoli tra il server centrale e la rete Kademlia ci garantisce che, anche in caso di failure del server (a causa di un attacco o per problemi tecnici), i sistemi operativi degli utenti sono sempre in grado di stabilire se un eseguibile è affidabile o meno, in quanto tale informazione è presente sulla DHT, che per sua natura è più affidabile e robusta, e dunque garantisce in qualsiasi momento l'accesso ai dati.

Infine, la strato software sul sistema utente, adibito al controllo dell'eseguibile prima della sua esecuzione, è stato realizzato attraverso un modulo kernel. A tale scopo, si è effettuato l'hijacking della system call *execve*. Dunque, prima di qualsiasi esecuzione, il modulo kernel calcola un hash SHA-2 del file eseguibile di interesse e verifica il suo punteggio inviando una richiesta sulla DHT. Se il file è classificato come affidabile viene eseguito regolarmente, altrimenti, se il file è classificato come malevolo oppure non ha ancora ricevuto segnalazioni, il modulo kernel ne impedirà l'esecuzione.

Tale progettazione presenta tuttavia alcuni inconvenienti, ovvero:

- **Latenza**: ogni chiamata di sistema *execve* necessita di effettuare una richiesta sulla rete P2P, prima di poter procedere con l'esecuzione. Questo comporta ovviamente un degrado prestazionale del sistema.
- **Connessione obbligata alla rete**: per poter stabilire se eseguire o meno un file, è necessaria una verifica sulla rete Kademlia, e ciò richiede una connessione Internet. Un sistema non connesso alla rete risulterebbe quindi inutilizzabile.

Per ovviare a questi due problemi è stato introdotto l'utilizzo di una cache locale, nella quale vengono memorizzati i punteggi relativi ad ogni eseguibile. Tale cache contiene gli hash di tutti gli eseguibili presenti con l'installazione iniziale del sistema operativo, ai quali è associato un permesso permanente di esecuzione. In questo modo è possibile un normale utilizzo del sistema anche senza connessione ad Internet. Inoltre, per ogni file eseguibile per il quale viene effettuata una verifica sulla rete Kademlia, viene memorizzato temporaneamente il punteggio in cache. Si riesce così ad evitare di effettuare una richiesta ad ogni singola esecuzione di un file.
Seppure tale soluzione non risolve completamente il problema della latenza, in quanto ci sarà comunque una ricerca in cache ad ogni esecuzione, possiamo sicuramente affermare che i tempi vengono notevolmente ridotti.
In figura 3.3 è riportato uno schema completo di funzionamento del modulo kernel all'interno di un sistema.

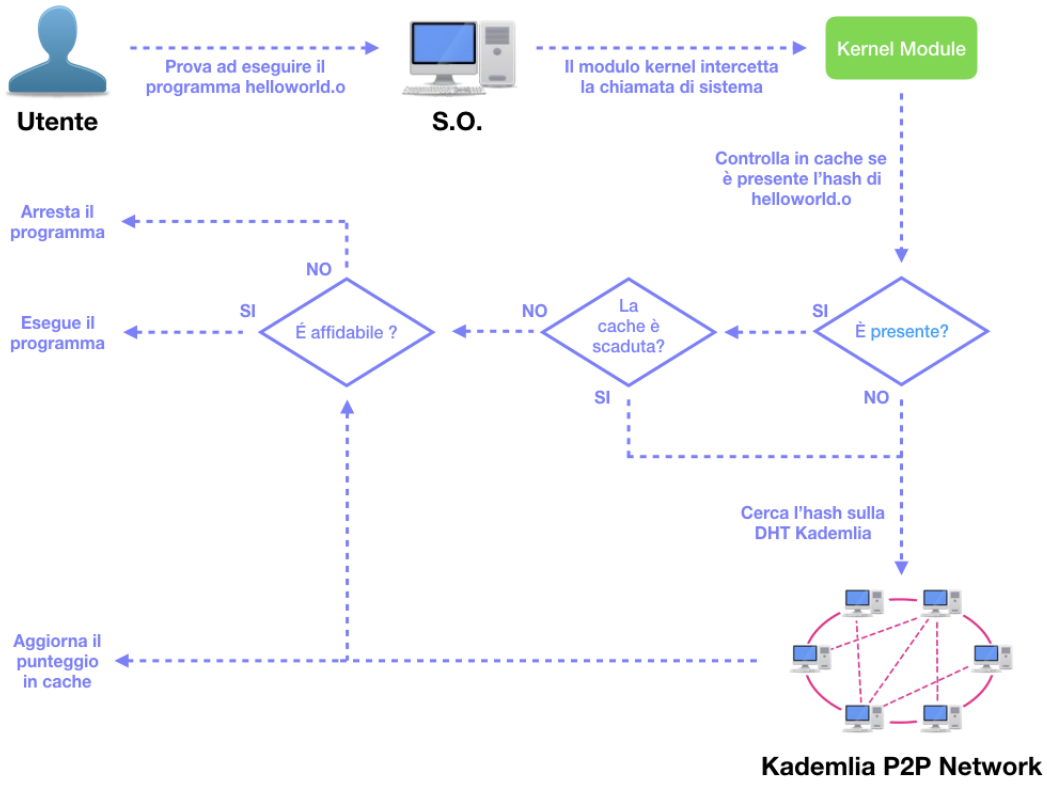

Figura 3.3: Schema di funzionamento del modulo kernel

Capitolo 4

Implementazione

In questo capitolo sarà descritta l'implementazione dell'architettura presentata in precedenza.

Lo sviluppo è avvenuto su una macchina virtuale con sistema operativo Linux Ubuntu in versione 12.04.1 64 bit e kernel 3.2.0-23-generic. L'immagine iso della versione utilizzata è disponibile all'indirizzo `http://old-releases.ubuntu.com/releases/12.04.1/`. Il lavoro si è focalizzato sull'implementazione del modulo kernel per il blocco degli eseguibili malevoli sul sistema e della rete P2P Kademlia. La parte server invece non è stata implementata, in quanto si è ritenuto che non ci fossero particolari configurazioni da effettuare. Il codice è disponibile su GitHub al seguente indirizzo: `https://github.com/gdecostanzo/MalwareKernelModule`.

4.1 Modulo Kernel

Il kernel module implementato si suddivide in tre funzionalità principali che sono:

- **Hijacking** della system call *execve*, attraverso l'*hooking* (aggancio) della tabella delle chiamate di sistema;
- **Hashing** SHA-256 del file eseguibile;
- **Verifica** del punteggio dell'eseguibile, tramite connessione alla rete Kademlia.

In appendice A vengono fornite le nozioni di base per lo sviluppo di un kernel module.

4.1.1 Hijacking

Come anticipato, per poter effettuare l'hijacking della sysem call è stata utilizzata una tecnica abbastanza diffusa, ovvero l'hooking della system call table [19]. I processi nello user space chiamano una serie di interrupt verso il kernel space, i quali sono memorizzati in una tabella predefinita durante il processo di inizializzazione del sistema Linux. Sono quindi memorizzati i puntatori a diverse funzioni per la gestione degli interrupt. L'idea è quella di manipolare questi puntatori all'interno della tabella per poter inserire delle azioni desiderate. In coda alle azioni aggiunte, viene poi ripristinata l'azione della system call originale per evitare crash o anomalie nel sistema operativo.

L'indirizzo della system call table può essere recuperato da *System.map*, utilizzando il seguente comando da terminale:

```
$ sudo cat /boot/System.map-‘uname -r‘ | grep sys_call_table
ffffffff81801300 R sys_call_table
```

In questo modo si riesce a recuperare l'indirizzo della system call table che tuttavia, come si può notare dalla "R", è in sola lettura. Il kernel infatti posiziona alcune strutture in una zona di memoria "read-only", in modo da proteggerle da eventuali cambiamenti, intenzionali o meno, che potrebbero portare all'instabilità del sistema. Per poter apportare modifiche alla tabella è necessario abilitare la scrittura. Viene quindi utilizzata, all'interno del kernel module, la funzione di sistema *write_cr0()* per rendere scrivibile la tabella, direttamente abilitando e disattivando la protezione del registro di controllo CR0 della CPU. Di default il bit WP (Write Protect) del registro è settato a *1*, dunque impostando il valore a *0* è possibile disabilitare la protezione in scrittura.

```
write_cr0(read_cr0() | 0x10000) // CR0 protection Enabled
write_cr0(read_cr0() & (~ 0x10000)) // CR0 protection Disabled
```

Codice 4.1: Funzione C per la scrittura del registro CR0

Abilitata la scrittura, si può precedere con l'hijacking della system call desiderata, ovvero execve. A tale scopo viene salvato il puntatore alla funzione originale; in questo modo è possibile aggiungere all'esecuzione le linee di codice volute e poi ripristinare il comportamento originale della chiamata di sistema, per far procedere l'esecuzione.

```
original_execve = (void *)syscall_table[__NR_execve];
syscall_table[__NR_execve] = new_execve;
```

Codice 4.2: Hijacking della system call execve

4.1.2 Hashing

Una volta effettuato l'hijacking della system call, si procede aggiungendo la parte di codice che si vuol fare eseguire. Nel nostro caso sarà aggiunta all'interno della funzione *new_execve()*. Si vuole che prima di eseguire un file sia controllata la sua affidabilità, dunque è necessario calcolare un hash SHA-256 del file per verificare il suo punteggio associato. Se l'eseguibile è affidabile, allora viene restituita una chiamata alla system call originale, garantendo il normale comportamento del sistema. In caso contrario invece, la system call non sarà eseguita.
Di seguito è riportato il codice della funzione *new_execve()*.

```
asmlinkage int new_execve(const char *filename, const char *argv
    [], const char *envp[]) {

  printk(KERN_ALERT "Executing: %s", filename);
  if(check_SHA256_hash(filename) == 0)
    return (*original_execve)(filename, argv, envp);
  else
    return NULL;
}
```

Codice 4.3: Nuova funzione execve

All'interno della funzione *check_SHA256_hash()* viene calcolato l'hash del file che si sta provando ad eseguire, il quale sarà usato per la verifica. A tale scopo viene utilizzata la libreria *crypto/internal/hash.h* fornita dal sistema, come descritto nelle più recenti versioni di *The Linux Kernel Programming Guide [2]*. In appendice A è fornito un esempio.

4.1.3 Verifica

Il passo finale è quello di verificare l'affidabilità dell'eseguibile utilizzando l'hash calcolato. Dobbiamo quindi collegarci alla rete Kademlia oppure verificare all'interno della cache, nel caso in cui non sia la prima esecuzione del file. Tuttavia, nello sviluppo di un modulo kernel non sono disponibili tutte le librerie standard che normalmente sono usate nella programmazione C in user space. Dunque si è reso necessario trovare una soluzione alternativa per ovviare al limite incontrato. L'idea adottata è quella di utilizzare una connessione socket locale che metta in comunicazione il modulo kernel con una applicazione eseguita in user space. A tal proposito è stata instaurata una connessione di tipo *AF_UNIX*, con la quale è possibile mettere in comunicazione un client e un server in esecuzione sulla stessa macchina, senza la necessità di avere un indirizzo IP. Per l'operazione di *bind* viene infatti utilizzato un file descriptor che punta ad un file all'interno del file system locale. Nel nostro caso il modulo kernel agisce da client, collegandosi all'applicazione eseguita in user space, la quale funge da server e resta in attesa di connessioni. Quando effettua una richiesta alla controparte server, il modulo kernel riceve un valore booleano con il quale andrà a stabilire se può procedere o meno all'esecuzione.

```
#define SOCK_PATH "/tmp/usocket"

int check_DHT(const char *hash)
{
  // other code here
  struct socket *sock = NULL;
  struct sockaddr_un addr;

  // create socket and connect
  retval = sock_create(AF_UNIX, SOCK_STREAM, 0, &sock);
  memset(&addr, 0, sizeof(addr));
  addr.sun_family = AF_UNIX;
  strcpy(addr.sun_path, SOCK_PATH);
  retval = sock->ops->connect(sock, (struct sockaddr *)&addr,
      sizeof(addr), 0);

  // other code here

  //send hash
  retval = sock_sendmsg(sock, &msg, strlen(hash)+1);

  // other code here

  //receive_msg
  retval = sock_recvmsg(sock, &msg, MAX, 0);

  // release socket
  sock_release(sock);
  return (return_msg[0]=='1');
}
```

Codice 4.4: Funzione C per la connessione del modulo kernel al server

Il client socket sopra riportato si collega quindi ad un server in ascolto, il quale è eseguito nello user space. L'implementazione del server è stata realizzata in Python e provvede ad accettare connessioni in arrivo dal modulo kernel. Ad ogni connessione, il server riceve la stringa contenente l'hash dell'eseguibile e verifica la sua presenza prima nella cache locale e poi, in caso negativo, provvede alla ricerca sulla rete Kademlia. Per la realizzazione della cache

è stata utilizzata la libreria Python *anydbm*. Quest'ultima è un front-end per database in stile DBM che usa semplici valori stringa come chiavi per accedere ai record. Nella presente implementazione è stato utilizzato un database *dbhash*.

```
# import statements
# Create socket
server_address = '/tmp/usocket'
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
sock.bind(server_address)
sock.listen(1)

while True: # Wait for a connection
  connection, client_address = sock.accept()
  try:
    hash_key = connection.recv(64) # Receive the hash key
    if hash_key and len(hash_key) == 64:
      cache_outdated = False
      try:
        db = anydbm.open("/dht_cache/cache", "r")
        cache_value = db[hash_key]
        #if == 1 then hash_key is in the initial cache
  o    if (cache_value == "1"):
          connection.send("1")
        else: # check that cache score is not expired
          last_update = float(cache_value.split(";")[1])
          if( (time.time() - last_update) < 3600): #3600 = 1 hour
            connection.send(cache_value.split(";")[0])
          else:
            cache_outdated = True
      except KeyError, e:
        cache_outdated = True
      finally:
        db.close()
      if (cache_outdated):
        #Search in DHT
  finally:
    connection.close()
```

Codice 4.5: Server Python per l'interfacciamento con il modulo kernel

La cache è stata creata in una directory dedicata, con permessi di scrittura per il solo utente *root*. I passi seguiti per la creazione sono descritti di seguito.

Come prima cosa è necessario loggarsi come root e creare la directory:

```
$ sudo su
$ mkdir /dht_cache
$ cd /
$ nano create_cache.py
```

all'interno dell'editor *nano*, viene definito lo script Python che sarà utilizzato per la creazione della cache:

```
# create_cache.py
#!/usr/bin/python

import sys
import anydbm

db = anydbm.open("/dht_cache/cache", "c")
for line in sys.stdin:
  hash = line.strip()
  db[hash] = "1"
```

Codice 4.6: Script Python per la creazione della cache

si salva lo script premendo *ctrl+x* e poi si assegnano i permessi di esecuzione. Infine si aggiungono alla cache tutti gli eseguibili presenti sul sistema appena installato.

```
$ chmod +x create_cache.py
$
$ find . -executable -type f -exec sha256sum {} \; | grep -o
    '^[^ ]*' | ./create_cache.py
```

Nel caso si volesse aggiungere in cache un nuovo file eseguibile, come ad esempio potrebbe essere l'output di un processo di compilazione, è possibile utilizzare il seguente script Python:

```
# addHashInCache.py
#!/usr/bin/python

import sys
import hashlib
import anydbm

BUF_SIZE = 65536 # lets read stuff in 64kb chunks!

sha256 = hashlib.sha256()

with open(sys.argv[1], 'rb') as f:
  while True:
    data = f.read(BUF_SIZE)
    if not data:
      break
    sha256.update(data)

hash_key = sha256.hexdigest()

db = anydbm.open("/dht_cache/cache", "w")
db[hash_key] = "1"
db.close()

print "Added in cache!"
```

Codice 4.7: Script Python per l'aggiunta in cache di un nuovo eseguibile

Dopo aver assegnato i permessi di esecuzione allo script, è possibile aggiungere un nuovo file eseguibile alla cache semplicemente lanciando questo comando:

```
$ sudo ./addHashInCache.py PATH_FILE
```

4.2 Kademlia

Per lo realizzazione della DHT si è partiti da un'implementazione già esistente, scritta in Python da Isaac Zafuta, e reperibile su GitHub all'indirizzo `https://github.com/isaaczafuta/pydht`. In appendice B viene fornito un esempio di utilizzo della libreria.

La descrizione si focalizzerà dunque sulle modifiche apportate al codice esistente. Come anticipato nel terzo capitolo, la caratteristica della nostra implementazione Kademlia, è quella di limitare la scrittura sui nodi ad una sola entità autoritativa. Ciò viene realizzato mediante l'utilizzo di una firma digitale. Tutti i nodi sono a conoscenza della chiave pubblica con la quale possono verificare la firma che accompagna il messaggio in arrivo. Poiché soltanto il server centrale è in possesso della chiave privata, sarà l'unico a poter scrivere sui nodi della DHT.
Le chiavi possono essere generate da riga di comando utilizzando la libreria OpenSSL. Nell'esempio di seguito è generato un certificato che utilizza l'algoritmo RSA a 4096 bit e un digest SHA-256, e che segue lo standard X.509.

```
$ openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout "
    private.key" -out "public.crt" -days 365 -subj "C=IT/ST=
    Salerno/L=Fisciano/O=Universita degli Studi di Salerno/OU=
    Dipartimento di Informatica/CN=unisa.it/emailAddress=g.
    decostanzo@studenti.unisa.it"
```

Le modifiche avvenute sull'implementazione Kademlia sono state effettuate all'interno della funzione di *STORE* dove ogni nodo, prima di memorizzare il contenuto, verifica la firma ed il timestamp, accertandosi quindi sull'autenticità del messaggio e che non sia oggetto di un Replay attack.
Il messaggio è quindi composto da una coppia chiave valore. Quest'ultimo, oltre a contenere il punteggio relativo all'eseguibile, conterrà una data di scadenza (*expiringDate*) e una firma (*signature*) calcolata sulla concatenazione delle stringhe della chiave, del punteggio e della data di scadenza. Per poter verificare la firma, il nodo deve ricalcolare la stringa concatenata, quindi recupera dal *value* sia la firma che la data di scadenza e poi le rimuove, in quanto non saranno più necessarie una volta memorizzato il nuovo punteggio.

```
def handle_store(self, message):
  key = message["id"]
  value = message["value"]

  public_key = sv.get_public_key("test.cert")
  score = value[0]
  signature = value[1]
  expiringDate = value[2]
  data = str(key) + score + expiringDate
  value.remove(signature)
  value.remove(expiringDate)

  if sv.verify(public_key, signature, data, digest='sha256'):
    if (time.time() < float(expiringDate)):
      self.server.dht.data[key] = value
```

Codice 4.8: Modifica alla funzione di store in Kademlia

Per la firma e la verifica in Python si è utilizzato il pacchetto pyOpenSSL, il quale fornisce un'interfaccia di alto livello alle funzioni di libreria di OpenSSL. All'interno del file *sign_verify.py* sono state quindi definite le seguenti funzioni per il caricamento delle chiavi, la firma e la verifica, utilizzate anche nella porzione di codice precedente.

```
import OpenSSL
import base64

def get_private_key(path):
  with open(path, 'r') as f:
    private_key = OpenSSL.crypto.load_privatekey(OpenSSL.crypto.
        FILETYPE_PEM, f.read())
  return private_key

def get_public_key(path):
  with open(path, 'r') as f:
    private_key = OpenSSL.crypto.load_certificate(OpenSSL.crypto
        .FILETYPE_PEM, f.read())
  return private_key
```

```
def sign(private_key, data, digest='sha256'):
  signature = OpenSSL.crypto.sign(private_key, data, digest)
  signature_base64 = base64.encodestring(signature)
  return signature_base64

def verify(public_key, signature, data, digest='sha256'):
  signature = base64.b64decode(signature)
  try:
    is_valid = OpenSSL.crypto.verify(public_key, signature, data
        , digest)
    return is_valid is None
  except OpenSSL.crypto.Error:
    return False
```

Codice 4.9: Funzioni Python per la firma e la verifica dei messaggi

Capitolo 5

Prova di concetto

In questo capitolo è riportata una prova concreta di esecuzione dell'implementazione realizzata. Sebbene non sia stato possibile testare le reali prestazioni del sistema, in quanto non si dispone di una rete Kademlia in funzione, il test di seguito mostra l'effettiva solidità di funzionamento del modello realizzato.

L'implementazione completa è composta dai seguenti file:

- **user_server.py**: script che fa da server ed accetta connessioni socket dal modulo kernel;
- **cacheUpdater.py**: viene eseguito con permessi di root e si interfaccia con *user_server.py*, provvedendo ad aggiornare la cache quando è necessario;
- **start_dht.py**: script che avvia il primo nodo della DHT Kademlia; sarà usato per il processo di *bootstrap*;
- **addHashInCache.py**: utilizzato per aggiungere alla cache un nuovo eseguibile (necessita dei permessi di root);
- **test.key**: chiave privata utilizzata per la firma digitale;
- **test.cert**: chiave pubblica utilizzata per la validazione della firma;
- **pydht**: cartella contenente l'implementazione Python di Kademlia;
- KernelModule/**blocker.c**: codice del modulo kernel;

- KernelModule/**Makefile**: Makefile usato per la compilazione;
- KernelModule/**blocker.ko**: kernel module compilato.

5.1 Test di esecuzione

Prima di caricare il modulo nel kernel, viene creato un semplice programma in C e poi compilato. Viene quindi mostrata una normale esecuzione del programma nel sistema.

```c
/* helloworld.c */
#include <stdio.h>
int main() {
  printf ("Hello World! \n");
  return 0;
}
```

Codice 5.1: Codice Helloworld in C

```
$ gcc helloworld.c -o helloworld.o
$ ./helloworld.o
Hello World!
$
```

L'applicazione è stata appena creata e, ovviamente, il suo hash non è ancora presente ne in cache locale ne sulla DHT. Dunque non sarà eseguita dal sistema una volta caricato il modulo kernel.

Si procede avviando il primo nodo di Kademlia ed il server socket, e poi caricando il modulo nel kernel.

```
$ ./start_dht.py &
[1] 3421
$ Kademlia DHT started

$ ./user_server.py &
[2] 3439
$ User Server for Kernel Module started

$ sudo bash -c './cacheUpdater.py &'
[sudo] password for user:

$ Cache Updater Started
$
$ sudo insmod KernelModule/blocker.ko
$
```

Provando ad eseguire nuovamente il programma C compilato in precedenza, si può notare come questo non venga eseguito.

```
$ ./helloworld.o
bash: ./helloworld.o: Success
```

Utilizzando lo script *addHashInCache.py* è possibile aggiungere il programma alla cache locale e quindi riprovare ad eseguire il file. Il software viene correttamente eseguito.

```
$ sudo ./addHashInCache.py helloworld.o
Added in cache!
$
$ ./helloworld.o
Hello World!
$
```

5.2 Prestazioni del sistema

Sono stati misurati i tempi di esecuzione del software prima e dopo aver caricato il modulo nel kernel. La macchina virtuale sulla quale sono stati effettuati i test è stata eseguita da un disco rigido esterno a 5400 rpm collegato al sistema host tramite USB 3.0. Le risorse assegnate alla macchina virtuale sono 1 GB di Ram e un singolo core del processore Intel Core i7 a 2,2GHz.

Per misurare il tempo di esecuzione è stato utilizzato il comando *time* da terminale. Viene prima mostrato il tempo impiegato senza il modulo kernel:

```
$ time ./helloworld.o
Hello World!

real 0m0.324s
user 0m0.000s
sys  0m0.004s
```

Infine il tempo con il modulo kernel in funzione:

```
$ time ./helloworld.o
Hello World!

real 0m0.355s
user 0m0.000s
sys  0m0.000s
```

Il test è stato effettuato utilizzando anche altri comandi. La tabella 5.1 e il grafico in figura 5.1 riassumono i risultati ottenuti riportando i tempi *real* restituiti dal comando *time*.

Tabella 5.1: Tempi di esecuzione

Comando	Senza modulo kernel	Con modulo kernel
./helloworld.o	0m0.324s	0m0.355s
ls /usr/bin	0m0.045s	0m0.092s
uname -a	0m0.001s	0m0.007s

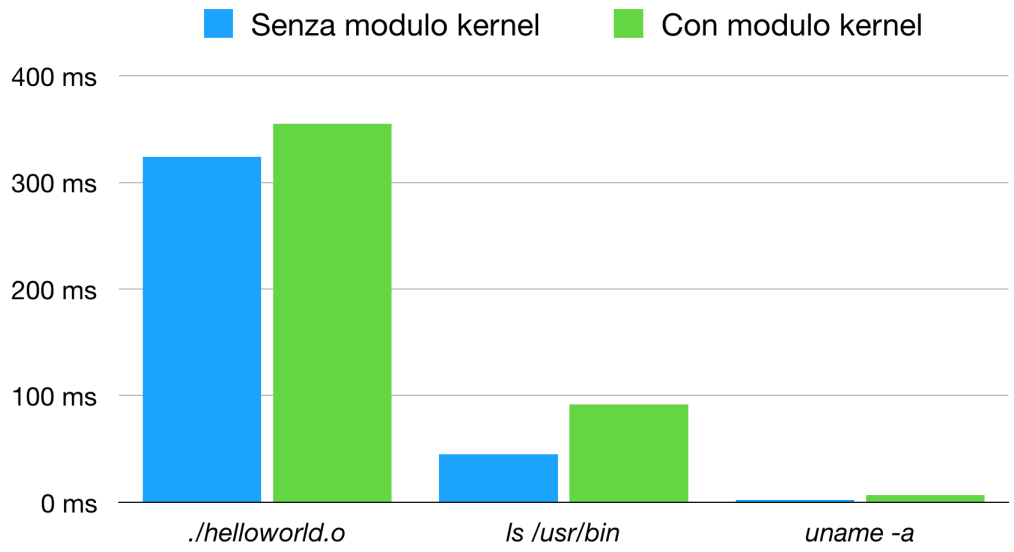

Figura 5.1: Grafico dei tempi di esecuzione

Dalla tabella e dal grafico riportati si evince che, sul sistema in uso, la latenza media introdotta dal modulo kernel all'esecuzione di un programma è di circa $28ms$. Possiamo dunque affermare che successivamente alla prima esecuzione di un file, la quale richiede il collegamento alle rete di nodi Kademlia, e quindi un tempo maggiore, la latenza introdotta per la verifica dell'hash in cache locale è accettabile, soprattutto in virtù del compromesso fatto con la sicurezza del sistema.

5.3 Limiti dell'implementazione

Il modulo kernel così realizzato presenta un limite di funzionamento, che tuttavia può essere risolto.

Quando si prova ad eseguire applicazioni scritte in linguaggi interpretati come Python, oppure eseguite in macchine virtuali come la JVM di Java, l'eseguibile intercettato dal modulo kernel sarà quello dell'interprete (o della macchina virtuale) e non quello dell'applicazione stessa. Questo significa che, avendo in cache l'interprete segnato come affidabile, ogni applicazione scritta in quel linguaggio potrà essere eseguita bypassando il controllo implementato.

Controllando con il comando *strace* le chiamate di sistema effettuate quando si eseguono le applicazioni, si può osservare quanto segue:

```
$ strace ./helloworld.o
execve("./helloworld.o", ["./helloworld.o"], [/*37 vars*/]) = 0
```

```
$ strace python helloworld.py
execve("/usr/bin/python", ["python", "helloworld.py"], [/*36
    vars*/]) = 0
```

```
$ strace ./helloworld.py
execve("./helloworld.py", ["./helloworld.py"], [/*36 vars*/]) =
    0
```

```
$ strace java -jar helloworld.jar
execve("/usr/bin/java", ["java", "-jar", "helloworld.py"], [/*48
     vars*/]) = 0
```

L'applicazione *helloworld.py* è stata eseguita due volte:

- la prima volta passandola all'interprete Python;
- la seconda eseguendola in modo diretto (assegnando i permessi di esecuzione e aggiungendo la riga "*#!/usr/bin/python*" all'inizio dello script).

Si può osservare la differenza nei parametri della chiamata *execve*:

- nel primo caso, così come nell'esecuzione Java, il primo parametro passato alla system call è il binario dell'interprete (*/usr/bin/python* e */usr/bin/java*, rispettivamente);
- nel secondo invece, viene passato come primo parametro il file dell'applicazione.

Poiché il modulo kernel effettua il controllo sul primo parametro della chiamata, ne segue che nel primo caso non viene eseguito il controllo sul file dell'applicazione.

Questo inconveniente è risolvibile in due modi:

- delegando il controllo dell'applicazione all'interprete, il quale dovrà interfacciarsi con la rete Kademlia e con la cache locale;
- istruendo il modulo kernel a verificare il path del file passato nel secondo parametro della chiamata *execve*, quando il primo corrisponde al binario dell'interprete di un linguaggio.

Capitolo 6

Conclusioni e sviluppi futuri

6.1 Conclusioni

Ad oggi i malware costituiscono una grave minaccia per la sicurezza informatica, sia per sistemi end-user che infrastrutture aziendali. Basti pensare infatti ai disagi apportati da malware come WannaCry o Stuxnet. Si evince quindi la necessità di trovare delle contromisure forti, in grado di bloccare sul nascere l'azione malevola di software ostili, in quanto non sempre è sufficiente riparare i danni.

Il lavoro di tesi presentato introduce un nuovo sistema per il contrasto del malware. A differenza della maggior parte dei meccanismi di protezione esistenti, i quali si basano sul concetto di *blacklist*, questo lavoro presenta un approccio basato su *whitelist*, e quindi più conservativo, il quale può essere riassunto dall'aforisma "prevenire è meglio che curare". Il sistema di segnalazione del software e di reputazione degli utenti che vi contribuiscono, realizzato in logica P2P, garantisce la persistenza dei dati, e quindi la disponibilità delle informazioni anche in caso di malfunzionamenti o attacchi. Inoltre, è stato mostrato come è possibile prevenire diversi tipi di attacchi attuabili al fine di sovvertire il sistema di reputazione. Si è visto infine che la latenza introdotta dal sistema per l'esecuzione di un file, può essere ridotta e quindi resa accettabile grazie all'utilizzo di una cache locale.

6.2 Sviluppi futuri

Partendo dall'idea di hijacking di una system call, utilizzando un modulo kernel si potrebbe realizzare un ulteriore sistema per il riconoscimento del payload di cifratura di un ransomware, analizzando l'entropia dei dati scritti. L'entropia può essere definita come la misura della quantità di disordine in un sistema fisico, o della relativa assenza di informazione in un sistema di comunicazione. Ad esempio, in un linguaggio naturale l'entropia è bassa, in quanto vi è ridondanza e una certa regolarità statistica nei termini e nei caratteri, mentre in un testo cifrato il valore dell'entropia sarà molto alto. Intercettando le system call di tipo *write*, si potrebbe analizzare il livello di entropia dei dati che un processo sta scrivendo, e stabilire una soglia oltre la quale generare un allarme e arrestare la scrittura. Altra idea, da affiancare eventualmente all'analisi dell'entropia, potrebbe essere quella di misurare la quantità di dati letti e scritti da un processo. Infatti, se le quantità dovessero corrispondere, ci sarebbero buone probabilità che tale processo stia effettuando delle trasformazioni ai dati sul disco.

Appendice A

Kernel Module: comandi di base

A.1 Configurazione dell'ambiente

Prima di iniziare a sviluppare un kernel module, è necessario configurare l'ambiente di sistema installando i pacchetti necessari ed i file headers per il proprio kernel.
Su sistemi Debian sono sufficienti i seguenti comandi da terminale:

```
$ sudo apt-get update
$ sudo apt-get install build-essential module-init-tools
$ apt-cache search linux-headers-$(uname -r)
```

Il pacchetto *module-init-tools* è stato sostituito da *kmod* nelle più recenti versioni del sistema. L'ultimo comando fornisce informazioni su quali file headers sono disponibili per il kernel in uso. Si procede quindi all'installazione in questo modo:

```
$ sudo apt-get install linux-headers-3.2.0-23-generic
```

A.2 Funzioni e comandi di base

Un modulo kernel deve avere almeno due funzioni:

- **init_module()**: funzione di inizializzazione chiamata quando il modulo è inserito nel kernel;
- **cleanup_module()**: funzione chiamata quando si vuole rimuovere il modulo dal kernel.

Di seguito è riportato un semplice esempio HelloWorld di un modulo kernel:

```
/*
* helloworld.c - The simplest kernel module.
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
  printk(KERN_INFO "Hello world.\n");

  /*
  * A non 0 return means init_module failed; module can't be
      loaded.
  */
  return 0;
}

void cleanup_module(void)
{
  printk(KERN_INFO "Goodbye world.\n");
}
```

Codice A.1: Kernel Module HelloWorld

Per poter compilare il modulo kernel è necessario un Makefile:

```
obj-m += helloworld.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Codice A.2: Makefile per il modulo kernel

Infine, si lancia il comando *make* per poter eseguire la compilazione utilizzando il Makefile creato:

```
$ make
```

L'output della compilazione sarà il modulo **helloworld.ko**. Si possono ottenere informazioni su di esso utilizzando il seguente comando:

```
$ sudo modinfo helloworld.ko
```

si carica quindi il modulo kernel nel sistema:

```
$ sudo insmod helloworld.ko
```

e si può verificare l'effettivo caricamento del modulo all'interno del kernel.

```
$ sudo lsmod | grep helloworld
```

Infine, per rimuovere il modulo viene usato il seguente comando:

```
$ sudo rmmod helloworld
```

Il funzionamento può essere riassunto nell'immagine di seguito (Fig. A.1).

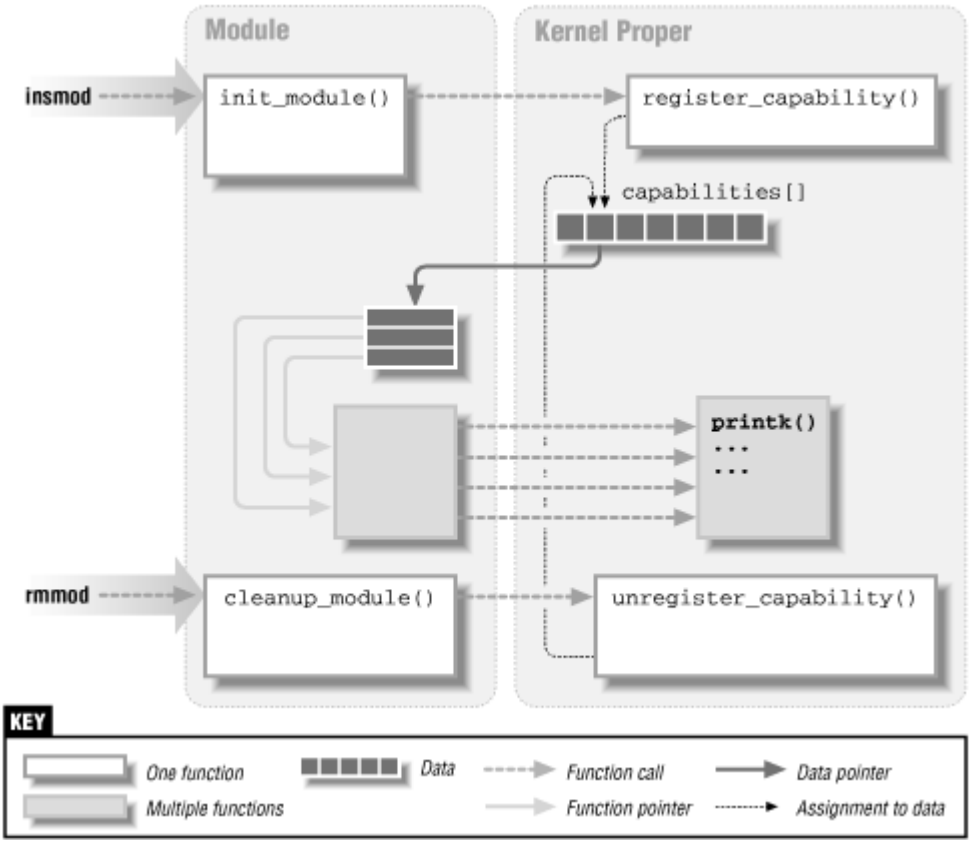

Figura A.1: Caricamento di un modulo nel kernel

Printk

A differenza di quello che si potrebbe pensare, la funzione *printk()* non è intesa per comunicare informazioni all'utente. Il suo scopo è fornire un meccanismo di logging per il kernel, ed è usata per riportare informazioni o warning. Ci sono otto diverse priorità con cui può essere chiamata la funzione printk(), ed è possibile trovarle in *linux/kernel.h*. Se non viene specificato un livello, viene usata la priorità di default, ovvero *DEFAULT_MESSAGE_LOGLEVEL*. Per visualizzare i log del kernel è possibile usare il seguente comando:

```
$ tail -f /var/log/kern.log
```

A.3 Funzione hash

Utilizzando la libreria di sistema *crypto/internal/hash.h* è possibile calcolare un hash all'interno di un modulo kernel. Di seguito è riportato l'esempio fornito in [2], in cui viene calcolato un hash SHA-256 di una stringa all'interno di un modulo kernel.

```
#include <linux/module.h>
#include <crypto/internal/hash.h>

#define SHA256_LENGTH (256/8)

static void show_hash_result(char * plaintext, char *
    hash_sha256)
{
  int i;
  char str[SHA256_LENGTH*2 + 1];

  printk("sha256 test for string: \"%s\"\n", plaintext);
  for (i = 0; i < SHA256_LENGTH ; i++)
    sprintf(&str[i*2],"%02x", (unsigned char)hash_sha256[i]);
  str[i*2] = 0;
  printk("%s\n", str);
}
```

```
int cryptosha256_init(void)
{
  char * plaintext = "This is a test";
  char hash_sha256[SHA256_LENGTH];
  struct crypto_shash *sha256;
  struct shash_desc *shash;

  sha256 = crypto_alloc_shash("sha256", 0, 0);
  if (IS_ERR(sha256))
    return -1;

  shash =
  kmalloc(sizeof(struct shash_desc) + crypto_shash_descsize(
      sha256),
  GFP_KERNEL);
  if (!shash)
    return -ENOMEM;

  shash->tfm = sha256;
  shash->flags = 0;

  if (crypto_shash_init(shash))
    return -1;

  if (crypto_shash_update(shash, plaintext, strlen(plaintext)))
    return -1;

  if (crypto_shash_final(shash, hash_sha256))
    return -1;

  kfree(shash);
  crypto_free_shash(sha256);

  show_hash_result(plaintext, hash_sha256);

  return 0;
}

void cryptosha256_exit(void)
```

```
{
  test_skcipher_finish(&sk);
}

module_init(cryptosha256_init);
module_exit(cryptosha256_exit);

MODULE_AUTHOR("Bob Mottram");
MODULE_DESCRIPTION("sha256 hash test");
MODULE_LICENSE("GPL");
```

Codice A.3: Calcolo di un hash SHA-256

Appendice B

Pydht: esempio di utilizzo

Per realizzare una nuova rete DHT utilizzando la libreria *pydht*, è necessario creare il nodo principale utilizzando il comando *DHT()* con l'indirizzo e la porta sulla quale sarà l'ascolto. Per entrare in una DHT, è necessario conoscere l'indirizzo e la porta in uso dal nodo esistente, in modo da avviare il processo di bootstrap. Di seguito è fornito un esempio:

```
>>> from pydht import DHT
>>> host1, port1 = 'localhost', 3000
>>> dht1 = DHT(host1, port1)
>>>
>>> host2, port2 = 'localhost', 3001
>>> dht2 = DHT(host2, port2, boot_host=host1, boot_port=port1)
>>>
>>> dht1["my_key"] = [u"My", u"json-serializable", u"Object"]
>>>
>>> print dht2["my_key"]
[u'My', u'json-serializable', u'Object']
```

Codice B.1: Esempio di utilizzo della libreria pydht

B.1 Scrittura sulla DHT

La versione modificata nel presente lavoro di tesi, prevede l'utilizzo di una firma digitale per poter scrivere delle coppie chiave-valore sulla DHT. Oltre al punteggio quindi, la parte *value* conterrà anche un timestamp indicante la scadenza e la firma del messaggio. Viene fornito un esempio, dove l'hash e il relativo punteggio da inserire sono già noti:

```
>>> from pydht import DHT
>>> from pydht import sign_verify as sv
>>> from pydht.hashing import hash_function
>>> import time
>>>
>>> host1, port1 = 'localhost', 3000
>>> dht1 = DHT(host1, port1)
>>>
>>> private_key = sv.get_private_key("test.key")
>>>
>>> hash = "56
    cea3186a73c5217583ee3d4049a673b906d6d48dcf77775a0b79805b67f03f
    "
>>> score = "1.1"
>>>
>>> expiringDate = str( time.time()+1800 ) #1800 = 30 minutes
>>> data = str(hash_function(hash)) + score + expiringDate
>>>
>>> signature = sv.sign(private_key, data, digest='sha256')
>>>
>>> dht1[hash]=[unicode(score),signature,unicode(expiringDate)]
```

Codice B.2: Esempio di firma di un messaggio

Bibliografia

[1] Backdoor definition. https://www.techopedia.com/definition/3743/backdoor.

[2] The Linux Kernel Module Programming Guide 4.9.11. https://github.com/bashrc/LKMPG/blob/master/older_versions/4.9.11/LKMPG-4.9.11.org.

[3] Apple. Informazioni sull'avviso "Sei sicuro di volerlo aprire?" (quarantena dei file/rilevamento di malware noto) in OS X. https://support.apple.com/it-it/HT201940.

[4] Karin Ask. Automatic Malware Signature Generation. http://www.gecode.org/~schulte/teaching/theses/ICT-ECS-2006-122.pdf, 2006.

[5] Roslan Ismail Audun Jøsang. *Decision Support Systems*, volume 43, chapter A survey of trust and reputation systems for online service provision, pages 618–644. 2007.

[6] Cisco. Viruses, Worms, Trojans, and Bots. https://www.cisco.com/c/en/us/about/security-center/virus-differences.html#3.

[7] Cisco. VNI Global Fixed and Mobile Internet Traffic Forecasts. https://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html.

[8] Iqbal Muhardianto Digit Oktavianto. *Cuckoo Malware Analysis*. Packt Publishing, 2013.

[9] Christopher C. Elisan. *Malware, Rootkits & Botnets A Beginner's Guide*. McGraw Hill Professional, 2012.

[10] ExtremeTech. Antivirus Research and Detection Techniques. https://web.archive.org/web/20090227002351/http://www.extremetech.com/article2/0%2C2845%2C1154648%2C00.asp, 2009.

[11] Wojciech Galuba and Sarunas Girdzijauskas. *Encyclopedia of Database Systems*, pages 903–904. Springer US, 2009.

[12] Kaspersky. How to enable or disable the Trusted Applications mode in Kaspersky Internet Security 2015. https://support.kaspersky.com/11158.

[13] Kaspersky. Ransomware & cyber blackmail. https://usa.kaspersky.com/resource-center/threats/ransomware.

[14] Kaspersky. What is a Botnet? https://usa.kaspersky.com/resource-center/threats/botnet-attacksr.

[15] Kaspersky. What is Adware? - Definition. https://usa.kaspersky.com/resource-center/threats/adware.

[16] Kaspersky. What is Spyware? - Definition. https://usa.kaspersky.com/resource-center/threats/spyware.

[17] Kaspersky. What is Trojan virus? - Definition. https://usa.kaspersky.com/resource-center/threats/trojans.

[18] Kaspersky. Generic detection. https://securelist.com/threats/generic-detection-glossary/, 2013.

[19] Kevin Koo. Rootkit module in Linux Kernel 3.2. http://dandylife.net/blog/archives/304.

[20] Malwarebytes. What is Gatekeeper? http://www.thesafemac.com/what-is-gatekeeper/.

[21] Andrew Honig Michael Sikorski. *Practical Malware Analysis*. No Starch Press, 2012.

[22] Trend Micro. Rootkit. https://www.trendmicro.com/vinfo/us/security/definition/rootkit.

[23] Trend Micro. Worm. https://www.trendmicro.com/vinfo/us/security/definition/worm.

[24] Microsoft. User Account Control. https://docs.microsoft.com/en-us/windows/access-protection/user-account-control/user-account-control-overview.

[25] Peter Jay Salzman Ori Pomerantz, Michael Burian. The Linux Module Kernel Programming Guide. http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf, 2007.

[26] Agenzia per l'Italia Digitale. Firma digitale. http://www.agid.gov.it/firma-digitale.

[27] David Mazières Petar Maymounkov. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. http://www.scs.stanford.edu/~dm/home/papers/kpos.pdf, 2002.

[28] Simone Piccardi. GaPiL. Guida alla Programmazione in Linux. http://gapil.gnulinux.it/files/2011/12/gapil.pdf, 2011.

[29] Repubblica.it. Stuxnet, Israele e Usa dietro al virus. "Creato da noi, ci è sfuggito di mano". http://www.repubblica.it/tecnologia/2012/06/01/news/stuxnet_israele_e_usa_ammettono_creato_da_noi_ci_sfuggito_di_mano-36353500/, Giugno 2012.

[30] Repubblica.it. Attacco hacker mondiale: virus "Wannacry" chiede il riscatto, ospedali britannici in tilt. "Usato codice Nsa". http://www.repubblica.it/tecnologia/sicurezza/2017/05/12/news/maxi_attacco_hacker_mondiale_virus_chiede_riscatto_colpita_anche_l_italia_-165285797/, Maggio 2017.

[31] W. Stallings, L. Brown, M.D. Bauer, and M. Howard. *Computer Security: Principles and Practice*. Prentice Hall, 2008.

[32] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.