UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

UNIVERSITÀ DI PISA

Corso di Laurea Triennale in Informatica

Analisi di contaminazione di file

Stefano Paganucci

Tutore accademico Fabrizio Baiardi

Indice

Capitolo 1

Introduzione

La digitalizzazione delle informazioni ha cambiato il modo in cui è possibile archiviare grandi quantità di dati e, di conseguenza, ha fatto sorgere nuovi problemi che riguardano le modalità per garantire che informazioni confidenziali non vengano distribuite impropriamente. Le intrusioni e gli attacchi ai computer aumentano ogni anno e vengono effettuati con tecniche automatizzate sempre più sofisticate. Analizzare intrusioni di tipo informatico è tutt'oggi un'operazione che consuma molto tempo e molte risorse. Dopo aver scoperto un'intrusione, l'amministratore di un sistema deve riuscire a capire come l'intruso abbia ottenuto l'accesso al sistema, deve riuscire a identificare i danni che questa contaminazione ha causato, e infine, deve riportare il sistema nello stato in cui si trovava prima dell'attacco [3]. Dal momento che i costi delle risorse umane dominano ormai i costi delle risorse macchina, è auspicabile che i prossimi sistemi vengano costruiti con meccanismi automatici di ripristino [4].

Le informazioni contenute in un sistema automatizzato devono essere protette:

1. dalla propagazione non autorizzata (*confidenzialità*).

2. dalla modifica non autorizzata (*integrità*).

3. dalla negazione di servizio (*accessibilità*).

Il raggiungimento di questi obiettivi si complica nel momento in cui certi sistemi vengono affacciati su una rete, favorendo l'accesso di utenti remoti, virus o worm. Sono quindi necessari: una classificazione dell'insieme delle informazioni contenute in un computer sulla base del loro grado di segretezza e meccanismi in grado di garantire che gli utenti autorizzati ad avere accesso ad informazioni confidenziali non le distribuiscano impropriamente [1]. I meccanismi di sicurezza di molti sistemi non sono in grado ancora oggi di garantire un flusso sicuro di informazioni. D'altra parte esistono modelli matematici in grado di negare determinati flussi di informazione non autorizzati causati da intrusioni di virus o worm che abbiano inserito malware in un sistema [5].

In questa tesi viene descritto e valutato lo strumento che è stato sviluppato durante il lavoro di tirocinio. Lo strumento fornisce ad un sistema automatizzato un meccanismo di sicurezza in grado di ridurre la vulnerabilità di un filesystem rispetto ad attacchi di tipo informatico e in grado di agevolare il ripristino di dati danneggiati da tali contaminazioni. È stato sviluppato un prototipo che estende il protocollo NFS (Network File System) di Linux con le suddette funzionalità e che presenta prestazioni vicine a quelle del NFS originale. Lo strumento realizzato si divide in due componenti fondamentali:

1. la componente on-line, chiamata Tainter, processa le singole richieste di operazioni su file che gli utenti effettuano al filesystem implementando un modello di flusso sicuro di informazioni e applicando un algoritmo di *tainting*. Tainter è in grado di negare l'esecuzione di operazioni considerate illecite in modo da ridurre i danni che un intrusione potrebbe causare.

2. la componente off-line, chiamata Analyzer, permette la costruzione e l'analisi del grafo dei flussi di informazioni che hanno avuto origine dalle richieste. Tale analisi fornisce ad un amministratore le informazioni necessarie per poter ripristinare il sitema in modo veloce e preciso.

Gli argomenti trattati vanno dalla sicurezza informatica all'analisi di infezioni da parte di virus o worm che abbiano inserito malware in uno o più file. Sono state acquisite conoscenze sulla teoria dei modelli di flusso sicuro di informazioni e sulle tecniche per l'automatizzazione dei processi di ripristino di sistemi compromessi. È stato inoltre effettuato uno studio approfondito del Network File System versione 3 di Linux in modo da effettuare le giuste scelte architetturali e implementative per la realizzazione del prototipo.

Durante la fase di implementazione e di testing del prototipo è stato utilizzato Xen, un Virtual Machine Monitor (VMM) open-source per architetture x86. Xen permette la creazione e la configurazione di macchine virtuali che hanno prestazioni molto vicine a quelle dell'hardware della macchina fisica ospitante, permette l'esecuzione simultanea di macchine virtuali con diversi sistemi operativi e rende possibile (in modo veloce e senza il rischio di danni permanenti al sistema operativo) il test e il debugging di modifiche al kernel. Il prototipo realizzato è indipendente dal supporto a macchine virtuali e può essere inserito in qualunque sistema Linux che faccia uso di NFS. La familiarità con lo strumento è stata acquisita durante il tirocinio attraverso documentazioni e manuali [6] [7]. I test per la valutazione delle prestazioni del prototipo sono stati effettuati tramite un benchmark per filesystems chiamato IOzone [8]. IOzone effettua una serie di operazioni su file misurando la velocità di trasferimento. I dati ottenuti vengono poi salvati in un foglio di calcolo permettendone la rappresentazione madiante grafici. Per quanto riguarda la visualizzazione dei grafi che rappresentano i flussi di informazioni,

è stato scelto uno strumento automatico chiamato Graphviz [9]. Questo software open source è in grado di disegnare qualsiasi tipo di grafo scegliendo fra diversi layout. L'applicazione prende in input un file con estensione .dot e produce in output un file immagine con un formato specificato (ad esempio .ps, .jpeg, etc.). Un file .dot contiene informazioni su nodi, archi e loro attributi: tali informazioni vengono specificate attraverso una grammatica ben definita. Questo strumento è stato scelto per testare la componente off-line del prototipo e per visualizzare i flussi di informazione calcolati nella fase di analisi.

Il tirocinio si è svolto, in maniera principalmente autonoma, presso il Laboratorio di Sicurezza del Dipartimento di Informatica di Pisa su macchine con processore AMD Athlon(tm), CPU a 651.512 MHz, memoria RAM variabile tra 128 Mb e 256 Mb, directory home montate tramite NFS, rete Ethernet 100 Mb, sistema operativo Linux Fedora Core 5 e kernel 2.6.18-1.2257.fc5xen0.

Il Capitolo 2 introduce il concetto di flusso sicuro di informazioni e descrive alcuni modelli matematici in grado di garantirlo. Presenta inoltre i problemi legati al ripristino di sistemi compromessi e le tecniche di tainting che vengono utilizzate per risolverli. Il capitolo successivo descrive il modello astratto dello strumento e la specifica implementazione del prototipo che è stato realizzato. Nel capitolo 4 viene presentata la verifica e la valutazione del lavoro svolto attraverso alcuni esempi e i grafici dei dati raccolti mentre nelle Conclusioni verranno forniti spunti per eventuali sviluppi futuri.

Capitolo 2

Lattici e Algoritmi di *tainting*

In questo capitolo vengono presentati, attraverso una formalizzazione matematica che fa uso di lattici, alcuni modelli che garantiscono un "flusso sicuro di informazioni" . Vengono quindi descritti gli algoritmi di tainting che permettono la ricostruzione del flusso di informazioni a partire dall'insieme di attività svolte da un sistema. Attraverso tali ricostruzioni è possibile attuare tecniche di ripristino mirate e veloci.

2.1 Modelli di flusso sicuro di informazioni

Garantire un flusso sicuro di informazioni all'interno di un computer non significa solo prevenire che un utente trasferisca dati da un file con alto livello di confidenzialità ad uno con un livello più basso, significa anche negare l'accesso indiretto a questo tipo di informazioni attraverso l'interazione con un altro utente che possiede le credenziali per farlo. Questo flusso implicito di informazioni viene gestito attraverso i *modelli di flusso sicuro di informazioni* e attraverso l'introduzione di *classi di sicurezza*. Nei prossimi paragrafi ven-

gono descritti alcuni modelli adottati in sistemi esistenti e viene fornita la loro rappresentazione formale attraverso delle strutture chiamate *lattici*.

2.1.1 Classi di sicurezza

Una classe di sicurezza identifica un insieme di oggetti di un sistema (files, utenti, processi, sockets, etc.) sulla base delle informazioni da essi contenute. Il livello di confidenzialità delle informazioni contenute in un oggetto determina la classe di sicurezza a cui l'oggetto è associato. L'assegnamento di oggetti a classi di sicurezza può avvenire in due modi:

- In maniera statica, quando la classe dell'oggetto rimane costante.
- In maniera dinamica, quando la classe dell'oggetto varia al variare del suo contenuto.

Come vedremo nei prossimi paragrafi, gli algoritmi di tainting fanno uso di classi di sicurezza dinamiche e definiscono il modo in cui le associazioni fra oggetti e classi possono cambiare. Esistono modelli di flusso sicuro di informazioni che definiscono classi di sicurezza statiche e altri che invece definiscono classi di sicurezza dinamiche. Vedremo i dettagli di tali modelli nelle sezioni che seguono.

2.1.2 Lattici

In questa sezione verrà data la definizione matematica di lattice che fa uso del concetto di classe di sicurezza. Per maggiori dettagli su questo argomento è possibile consultare [5]. Consideriamo un insieme disgiunto di

classi di sicurezza e lo denotiamo con $SC = A, B, ...$, dove $A, B, ...$ rappresentano le classi di sicurezza. Ogni oggetto del sistema viene associato ad una classe di sicurezza in relazione al tipo di informazioni contenute. Se queste appartengono ad una determinata classe di sicurezza, allora anche l'oggetto apparterrà alla stessa classe. L'operatore di giunzione "$\oplus$" viene applicato a due classi di sicurezza e specifica la classe risultante dall'applicazione di una funzione binaria ai valori degli operandi. Quest'operatore è sia commutativo che associativo. Se diciamo che $A \oplus B = C$ significa che C è una classe i cui oggetti contengono informazioni appartenenti agli oggetti della classe A e informazioni appartenenti a quelli della classe B. Per ultimo introduciamo il concetto di *relazione di flusso*, denotato col simbolo $\rightarrow$, per cui $A \rightarrow B$ significa che l'informazione contenuta in A può fluire in B in contrapposizione a $A \nrightarrow B$ che invece nega la possibilità di un flusso di informazioni tra la classe A e la classe B. Un lattice è una struttura rappresentata da una tripla $< SC, \rightarrow, \oplus >$ che rispetta i seguenti vincoli:

1. $< SC, \rightarrow >$ è un insieme parzialmente ordinato di classi di sicurezza.
2. SC è finito.
3. SC ha un limite inferiore tale che $L \rightarrow A$ vale per ogni $A \in SC$.
4. $\oplus$ è un operatore su SC.

Definendo SC, le relazioni di flusso e l'operatore $\oplus$ è possibile rappresentare formalmente un modello di flusso sicuro di informazioni. Il prossimo paragrafo presenta alcuni modelli che sono stati realmente implementati e per alcuni di essi viene mostrato il lattice che li formalizza.

2.1.3 Modello a classi isolate

Il modello a classi isolate è un modello in cui nessun flusso di informazioni è consentito eccetto quello tra informazioni contenute in oggetti appartenenti alla stessa classe. Questo è il modello più semplice in quanto non è possibile prevenire un flusso fra una classe e sé stessa. Il modello a classi isolate non è rappresentabile formalmente attraverso un lattice perchè non esiste nessun limite inferiore L tale che $L \rightarrow A$ vale per ogni $A \in SC$. $SC = A_1, ..., A_n$:

per ogni $i = 1, ..., n$ abbiamo che $A_i \rightarrow A_i$ e $A_i \oplus A_i = A_i$ e per $i, j = 1, ..., n$ $i \neq j$ $A_i \nrightarrow A_j$ e $A_i \oplus A_j$ non è definita.

2.1.4 Modello High-Low

Nel modello high-low sono presenti due sole classi di sicurezza (Figura 2.1 a) che, per semplicità, chiameremo H (high) e L (low). Tutti i flussi di informazioni sono consentiti tranne quelli dalla classe H alla classe L. $SC = H, L$ e $\rightarrow = (H, H), (L, L), (L, H)$ che si può esprimere attraverso le

seguenti regole di flusso $H \rightarrow H$, $L \rightarrow L$, $L \rightarrow H$. L'operatore di giunzione è definito come $H \oplus H = H, L \oplus H = H, H \oplus L = H, L \oplus L = L$.

2.1.5 Modello a classi isolate delimitate

Il primo modello può essere esteso introducendo L e H in modo da renderlo rappresentabile attraverso un lattice (Figura 2.1 b). $SC = A_i, ..., A_n, H, L$

e $L \rightarrow L, L \rightarrow H, H \rightarrow H$ e per $i = 1, ..., n$, $L \rightarrow A_i, A_i \rightarrow A_i, A_i \rightarrow H$. E

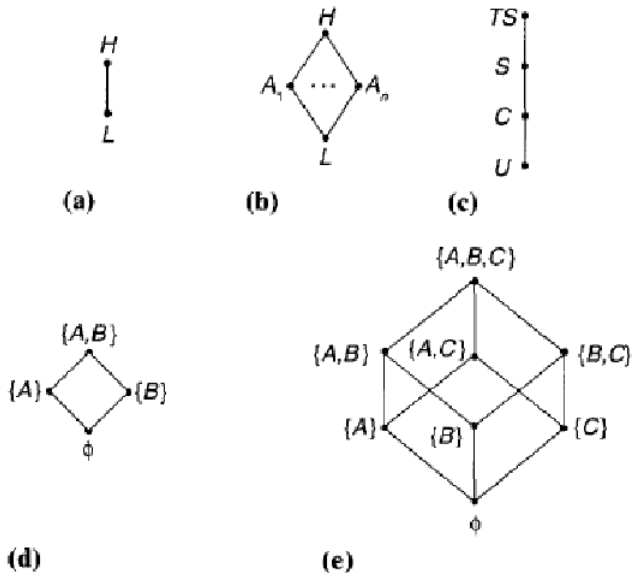

Figura 2.1: Alcuni esempi di lattici che rappresentano modelli di flusso sicuro di informazioni. a) Modello High-Low b) Modello a classi isolate delimitate c) Modello militare d-e) Modelli ad ordinamento parziale.

per finire $A_i \oplus A_i = A_i, A_i \oplus H = H, A_i \oplus L = A_i$ e per ogni $i, j = 1, ..., n$, $A_i \oplus A_j = H$.

2.1.6 Modello militare

Un esempio di ordinamento totale delle classi di sicurezza viene fornito dai settori militari che usano la classe TS (che sta per *top secret*), la classe S (*secret*), la classe C (*confidential*) e la classe U (*unclassified*), come mostrato in Figura 2.1 c. Si dice che A domina B se vale $B \rightarrow A$ ovvero se è possibile un flusso di informazioni da oggetti appartenenti a B a oggetti appartententi ad A. Il risultato di ogni operazione $A \oplus B$ è uguale al massimo tra A e B nella relazione di dominanza.

2.1.7 Modello ad ordinamento parziale

Consideriamo un ordinamento parziale di classi ottenute dall'insieme potenza di un insieme di classi (Figura 2.1 d-e). In questo caso la relazione di flusso può essere identificata con la relazione di sottoinsieme mentre l'operatore $\oplus$ può essere invece definito come l'operazione di unione fra insiemi:

- $A \rightarrow B$ equivale a $A \subseteq B$.
- $A \oplus B$ equivale a $A \cup B$.

2.1.8 Modello con matrice di accesso

A differenza dei modelli trattati precedentemente, il modello con matrice di accesso si basa sulle strutture dati e sulle operazioni di un sistema invece

che sulle classi di sicurezza. Le componenti principali di questo modello sono: un insieme di oggetti, un insieme di soggetti e un insieme di regole di accesso. Gli oggetti sono i file, i devices etc. mentre i soggetti sono i processi. Ogni processo è associato all'utente che sta eseguendo il programma di cui il processo è l'istanza. La matrice di accesso è un array bidimensionale in cui le righe identificano i soggetti e le colonne identificano gli oggetti. Un elemento della matrice contiene le regole di accesso, ovvero le operazioni che il soggetto corrispondente alla riga può effettuare sull'oggetto corrispondente alla colonna. Le modalità di accesso dipendono dal sistema in questione ma generalmente vengono considerate la lettura, la scrittura, la scrittura in append e l'esecuzione. Il *reference monitor* è il meccanismo che controlla se esiste la regola corrispondente nella matrice di accesso per ogni operazione da effettuare su un oggetto del sistema. Se la regola esiste la richiesta viene completata, altrimenti viene negata [1]. Ogni configurazione della matrice di accesso rappresenta la politica di sicurezza adottata dal sistema in un determinato istante. Il difetto di questo modello è la possibilità per un soggetto di cambiare la politica di sicurezza del sistema modificando le regole di accesso relative agli oggetti di cui è proprietario o eliminando tali oggetti dal sistema. Quando un oggetto viene rimosso viene cancellata la colonna corrispondente nella matrice. Esistono molti modelli che possono essere considerati una specializzazione del modello con matrice di accesso il quale, pur prevedendo l'esistenza di regole di accesso, non definisce quali queste siano in dettaglio. Ogni modello può quindi specificare un insieme di oggetti, soggetti e regole di accesso in modo da definire la propria politica di sicurezza in base al sistema su cui deve essere applicato.

2.1.9 Modello Bell-LaPadula

Il modello Bell-LaPadula rappresenta una specializzazione del modello con matrice di accesso e del modello militare. In aggiunta ai concetti di soggetto e di oggetto, propri del modello con matrice di accesso, il modello Bell-LaPadula introduce il concetto di classi di sicurezza, proprio del modello militare. Ogni oggetto e soggetto appartiene ad una classe di sicurezza e le regole di accesso corrispondono alla lettura, alla scrittura in append, all'esecuzione e alla lettura-scrittura. Questo modello utilizza macchine a stati finiti definendo cosa significa per uno stato essere sicuro e fornendo una serie di transizioni che da uno stato sicuro portano ad un altro stato sicuro. Uno stato è considerato sicuro se rispetta due proprietà:

1. la *simple security property*: un soggetto può accedere in lettura ad un oggetto solo se la classificazione del primo è maggiore di quella del secondo (No read up).

2. la **-property*: un soggetto può scrivere su un oggetto solo se la classificazione dell'oggetto è maggiore di quella del soggetto; un soggetto può accedere in lettura-scrittura un oggetto solo se le due classificazioni sono uguali (No write down).

2.1.10 Modello Biba

il modello Biba utilizza il concetto di integrità delle informazioni. Il principio alla base del modello di Biba è che nessuna informazione con integrità bassa fluisca in un oggetto con integrità alta, mentre il contrario è possibile. Secondo questo modello le informazioni possono fluire solo dalle classi di

sicurezza più alte a quelle più basse, esattamente il contrario del flusso permesso nel modello Bell-LaPadula. Le due proprietà definite nel precedente paragrafo vengono ridefinite come di seguito:

1. la *simple security property*: un soggetto può accedere in lettura un oggetto solo se la classificazione di quest'ultimo è maggiore di quella del soggetto (No read down).

2. la **-property*: un soggetto può scrivere su un oggetto solo se la classificazione dell'oggetto è minore di quella del soggetto (No write up).

In alcuni sistemi reali, i modelli Bell-LaPadula e Biba vengono combinati per garantire maggiore sicurezza. In questo caso le proprietà che garantiscono la transazione verso uno stato sicuro vengono fuse e vengono utilizzati due lattici differenti per rappresentare rispettivamente le integrità e le classi di sicurezza di oggetti e soggetti [2].

2.2 Tainting

Nei prossimi paragrafi vengono presentate le problematiche relative al rispristino di un sistema automatizzato che è stato compromesso e in che modo gli algoritmi di tainting aiutano a risolvere questi problemi.

2.2.1 Problematiche legate al ripristino di sistemi compromessi

Ripristinare un sistema compromesso da attacchi di virus, worm o errori umani, è tutt'oggi una procedura manuale che consuma molto tempo e molte

risorse. In realtà il processo di ricostruzione diventa sempre più complicato col passare del tempo se si considera la crescente quantità di operazioni che un sistema può compiere durante un ciclo di attività. Esistono oggi *intrusion detection systems* che avvertono l'amministratore di una avvenuta contaminazione, ma nella maggior parte dei casi questi strumenti non forniscono nessuna informazione sugli eventi del sistema collegati alla contaminazione stessa. Gli algoritmi di tainting vengono utilizzati principalmente per fornire in input ai processi di recovery una maggiore quantità di informazioni, in modo tale da renderli il più automatizzati possibile. Per raggiungere questo obiettivo è necessario, non solo tenere traccia di tutte le funzioni svolte dal sistema in un apposito file di log, ma anche ridurre la quantità di registrazioni da analizzare. Pur conoscendo o ipotizzando il momento (*break-in point* o *detection point*) in cui si è manifestata la contaminazione, lo sforzo di ricostruire manualmente i danni causati o l'origine dell'intrusione può risultare particolarmente oneroso. Un altro aspetto importante è che la rilevazione manuale del detection point avviene di solito molto tempo dopo il momento in cui è iniziata la contaminazione con la possibilità che i danni causati al sistema siano ormai irreversibili.

2.2.2 Registrare le operazioni

La costruzione del file di log prevede la possibilità di registrare le operazioni effettuate dal sistema. Il progetto Forensix [10] [11] [12] prevede la presenza di due macchine: la prima, potenzialmente insicura, viene chiamata "front-line machine" ; la seconda, progettata sicura, viene invece chiamata "back-end machine" . Mentre "front-line machine" è la macchina monitora-

ta, la "back-end machine" è un database MySQL che immagazzina tutte le informazioni relative alle systemcalls eseguite sulla prima.

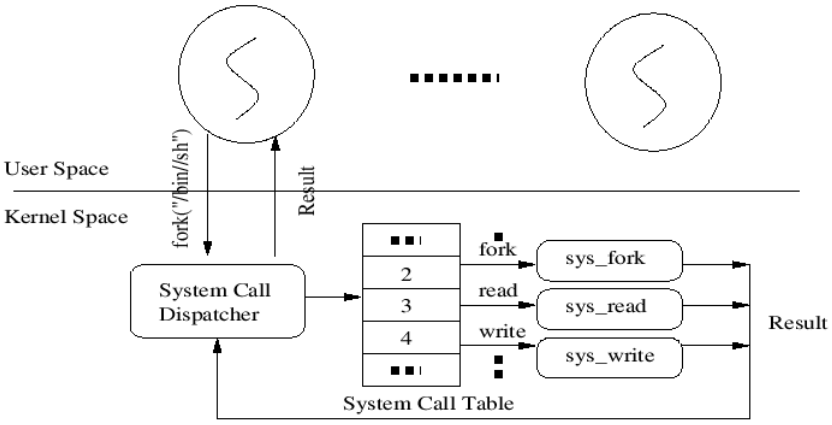

Figura 2.2: Meccanismo originale delle system calls

È possibile osservare gli eventi di un sistema a diversi livelli. Il livello delle applicazioni è semanticamente molto ricco di informazioni ma sarebbe relativamente semplice, per un attaccante, disattivare la funzionalità di log ottenendo un accesso privilegiato. Registrare eventi al livello di rete può essere molto utile perchè fornisce maggiori informazioni su attacchi remoti. Tali informazioni vengono spesso offuscate o criptate rendendo l'analisi del file di log molto difficile o a volte impossibile. Le istruzioni macchina danno una completa informazione delle attività del sistema ma risultano difficili da comprendere in fase di analisi perchè semanticamente povere [3]. Il livello del sistema operativo rappresenta un buon compromesso tra la sicurezza offerta dal livello macchina e la completezza di informazioni fornita dal livello applicativo. Esistono diversi metodi per registrare operazioni del sistema operativo tra cui due dei più utilizzati prevedono rispettivamente il *wrapping* e l'intercettazione delle system calls. Nel primo caso viene modificata la

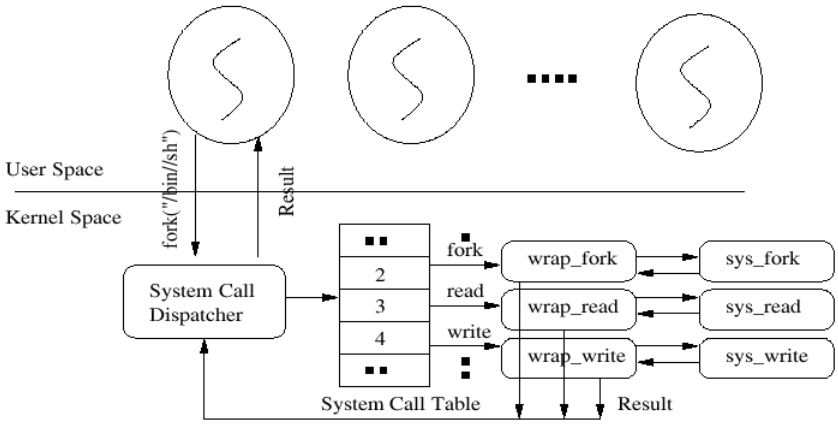

Figura 2.3: La tabella delle system calls viene modificata in modo che ogni riga punti ad una nuova funzione

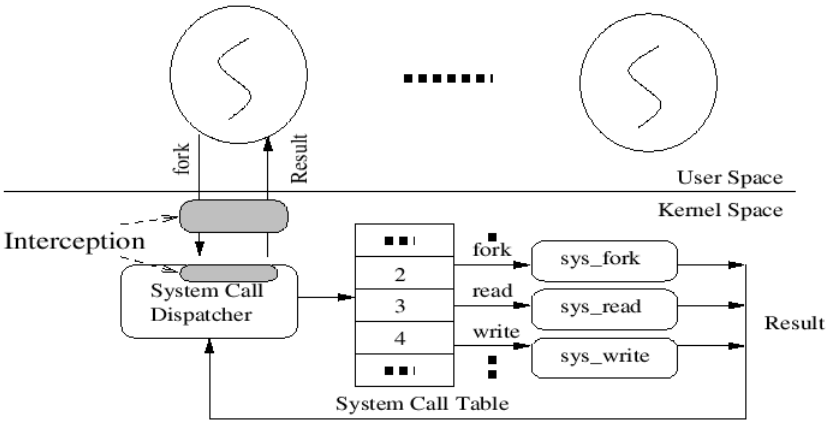

Figura 2.4: Le system calls vengono intercettate

tabella delle chiamate di sistema in modo che ogni elemento punti a una nuova funzione (Figura 2.3); questo modello risulta però facilmente vulnerabile ad attacchi che vanno anch'essi a modificare la tabella delle system calls per riportare la situazione allo stato originale. Il secondo metodo, più sicuro da questo punto di vista, intercetta la richiesta di esecuzione di una system call prima che ne venga fatto il dispatch verso l'effettiva funzione (vedi Figura 2.4).

2.2.3 Descrizione degli algoritmi di tainting

Gli algoritmi di *tainting* cercano di risolvere i problemi legati al ripristino di sistemi compromessi assegnando ad ogni oggetto del sistema un "colore" (da cui il termine *taint*). Il colore è tipicamente rappresentato da un identificatore univoco e viene "diffuso" tra i singoli oggetti attraverso le operazioni effettuate dal sistema. Oltre alle informazioni di interesse, ogni entry del file di log conterrà, quindi, anche le informazioni realtive al colore di ogni singolo oggetto. Questo processo rende più agevole la ricostruzione dell'insieme di operazioni collegate ad una contaminazione perchè permette di cosiderare solo le operazioni che hanno un colore uguale a quello della entry che rappresenta il *detection point*. Gli algoritmi di tainting possono però far sorgere alcuni problemi legati alle false dipendenze perchè in generale non è facile distinguere tra un'operazione legale e una non legale. Un esempio può essere la modifica legittima di un file che è stato creato illegalmente. Se per ripristinare il sistema il file viene rimosso, sarà persa per sempre la modifica legale che era stata apportata. Le problematiche legate al ripristino di un sistema corrotto vengono trattate in maggior dettaglio in [3] e in [4].

Regole di dipendenza

Le regole di dipendenza definiscono le modalità di diffusione dei colori in base alle operazioni che vengono effettuate. Una dipendenza nasce ogni volta che avviene un flusso di informazioni fra due oggetti attraverso l'esecuzione di un'operazione. Inizialmente il colore di ogni oggetto e processo viene inizializzato dall'amministratore sulla base, rispettivamente, delle informazioni contenute e dei privilegi posseduti. Nella seconda fase il colore viene propagato attraverso delle *regole di dipendenza* che variano a seconda dell'implementazione e del modello che si decide di adottare. Una regola di dipendenza può essere denotata come $O_s \xrightarrow{op} O_d$ dove O_s è l'oggetto che da origine al flusso, O_d è l'oggetto dipendente in cui il flusso termina e *op* è l'operazione in esame. Si consideri un processo che scrive su un file: l'oggetto O_s in questo caso è il processo, l'oggetto O_d è il file, e l'operazione è l'operazione di scrittura. In questo caso si dice che il file è dipendente dal processo. In termini di *tainting*, possiamo dire che il colore viene propagato dal processo al file su cui ha scritto [4]. Un esempio di regole di dipendenza adottate in alcuni sistemi reali, prevede che, quando si forma una dipendenza tra due oggetti, il colore dell'oggetto dipendente diventa quello che aveva prima più il colore dell'oggetto responsabile dell'operazione. Il nuovo colore viene registrato nel file di log per non perdere l'informazione così ottenuta. La Tabella 2.1 mostra un esempio di un modello di regole di dipendenza adottato da uno strumento reale [13] in cui le operazioni vengono divise in quattro macro categorie. Sono possibili altre tipologie di regole di dipendenza classificate in base agli oggetti invece che al tipo di operazione. Un esempio è il modello utilizzato da Taser [4] la cui parte fondamentale è riportata nella Tabella 2.2.

Abstract Operation	Color Diffusion	Description
create <s1 , o>	color(o) = color(s1)	Subject s1 creates a new object o
create <s1 , s2>	color(s2) = color(s1)	Subject s1 creates a new subject s2
read <s1 , o>	color(o)U = color(s1)	Subject s1 reads a new object o
read <s1 , s2>	color(s2)U = color(s1)	Subject s1 reads a new subject s2
write <s1 , o>	color(o)U = color(s1)	Subject s1 writes a new object o
write <s1 , s2>	color(s2)U = color(s1)	Subject s1 writes a new subject s2
destroy <s1 , o>	-	Subject s1 destroy a new object o
destroy <s1 , s2>	-	Subject s1 destroy a new subject s2

Tabella 2.1: Semplice modello di diffusione dei colori. Un subject è un processo mentre un object è una risorsa condivisa.

Dependency Rules	Type of Operation
Process $\longrightarrow$ Process	Fork, IPC, signals
Process $\longrightarrow$ File	Write file content Write file name Write file attributes
File $\longrightarrow$ Process	Execute Read file content Read file name Read file attributes

Tabella 2.2: Regole di dipendenza fra processi e file.

Capitolo 3

Descrizione dello strumento

In questo capitolo verrà descritto ad alto livello lo strumento realizzato e le scelte architetturali più rilevanti ai fini della comprensione del lavoro svolto. Nelle sezioni successive sarà invece data una descrizione dettagliata del prototipo implementato presentando gli aspetti di interesse del protocollo NFS e del Virtual File System di Linux.

3.1 Modello astratto

Lo strumento definisce un modello di flusso sicuro di informazioni in grado di rendere un filesystem meno vulnerabile rispetto ad attacchi esterni e ricostruisce la sequenza degli eventi di un sistema per permettere un'analisi degli effettivi danni causati da un'infezione. Il modello astratto adottato per la realizzazione dello strumento si propone di:

- costruire un meccanismo in grado di garantire un flusso sicuro di informazioni all'interno di un sistema utilizzando il concetto di classi di sicurezza.

- permettere la registrazione di informazioni di interesse relative alle attività del sistema.
- costruire un grafo delle dipendenze che rappresenta i flussi di informazioni tra file ed utenti.
- analizzare tale grafo per ricostruire i danni causati da intrusioni e contaminazioni.

È stato scelto di registrare le operazioni al livello del sistema operativo sulla base delle considerazioni che sono state fatte nel capitolo precedente. Le prossime sezioni definiscono quali tipi di oggetti di un sistema e quali tipi di dipendenze fra di essi sono stati presi in considerazione per l'analisi degli eventi.

3.1.1 Oggetti

Gli eventi che vengono tracciati dallo strumento sono quelli che causano dipendenze tra oggetti appartenenti al sistema. In questo modello gli oggetti sono rappresentati dai file e dagli utenti. In un sistema Linux un file può essere un device, una directory, un file vero e proprio etc. Un utente è identificato univocamente dal suo *user id* mentre un file dall'*inode number* e dal *generative number*. Come conseguenza del fatto che i file non vengono identificati attraverso il loro nome, le operazioni su di essi vengono tracciate senza considerare eventuali rinominazioni o creazione di link simbolici.

3.1.2 Regole di dipendenza

Lo strumento considera le dipendenze utente/file: ogni volta che un utente legge un file si crea una dipendenza file→utente mentre ogni volta che un utente scrive su file si crea una dipendenza utente→file. Nel primo caso l'informazione passa dal file all'utente, nel secondo caso il flusso è inverso.

3.2 Architettura del sistema

Lo strumento è diviso in due parti fondamentali: una parte on-line, chiamata Tainter, che realizza i primi due punti della sezione 3.1, e una parte off-line, chiamata Analyzer, che invece realizza i successivi due. La Figura 3.1 rappresenta uno schema dell'architettura complessiva del sistema. Il mod-

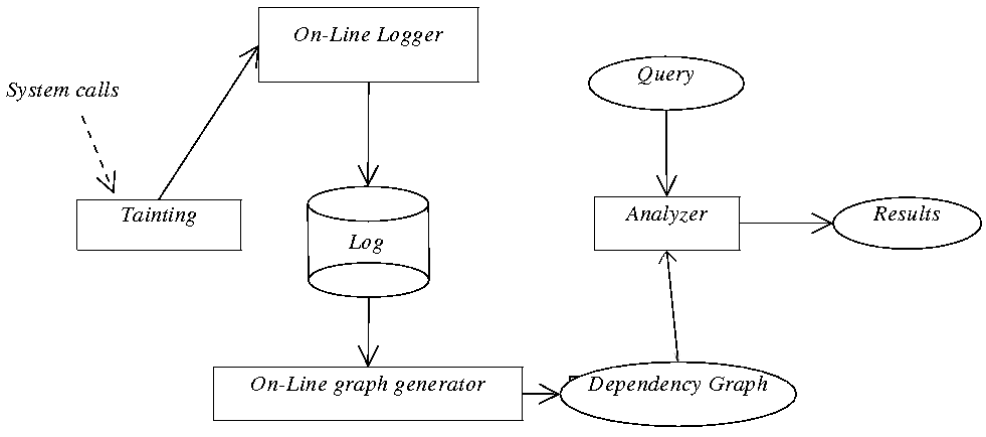

Figura 3.1: Architettura del sistema

ello di flusso di informazioni viene specificato dall'amministratore attraverso due file di configurazione. Il primo file contiene un ordinamento delle classi di sicurezza mentre il secondo una serie di regole che, in base a tale ordinamento, definiscono le operazioni lecite e quelle illecite. Ad ogni richiesta di esecuzione di una system call lo strumento utilizza il modello di flusso di informazioni per stabilire se la richiesta è lecita o meno. Se la richiesta viene considerata illecita la system call non viene eseguita e viene sollevato un errore di "permission denied" . Al contrario, se la richiesta è lecita, viene applicato un algoritmo di tainting, vengono registrate tutte le informazioni d'interesse in un apposito file di log e la system call viene eseguita. Le funzionalità descritte sopra vengono eseguite on-line ovvero, per ogni evento o operazione che ha luogo nel sistema. Analyzer è invece una componente off-line perchè opera solo su esplicita richiesta dell'amministratore. Questa componente analizza il file di log costruito da Tainter e, per ogni entry, aggiunge incrementalmente nodi e archi al grafo delle dipendenze. Il grafo così ottenuto rappresenta i flussi di informazioni tra file e utenti che hanno avuto origine dall'avvio del sistema all'inizio della fase di analisi. Come mostra la Figura 3.1, nella fase di analisi vera e propria, Analyzer prende in input il grafo delle dipendenze e una query richiesta dall'amministratore per produrre in output uno o più flussi che possono rappresentare un'intrusione.

3.3 Prototipo

Il prototipo che è stato realizzato estende il filesystem Linux NFS versione 3 e ricostruisce i flussi di informazioni tra i file e gli utenti che utilizzano questo protocollo. Di seguito è riportata una breve descrizione del funziona-

mento del Network File System versione 3 e del Virtual File System (VFS) di Linux.

3.3.1 NFS versione 3

NFS è un protocollo sviluppato dalla Sun Microsystems nel 1984 e definito nelle RFC 1094, 1813 e 3530. Il protocollo NFS permette di avere accesso, in maniera del tutto trasparente a filesystems condivisi attraverso la rete. NFS nasconde la reale locazione di file e directory in modo che il filesytem condiviso sembri essere fisicamente residente sul filesystem locale. Un aspetto importante di questo protocollo sta nel fatto che è stato progettato in modo da essere indipendente dall'architettura di rete, dal protocollo di trasporto, dal sistema operativo e dall'architettura hardware su cui risiede.

Remote Procedure Call

L' indipendenza dai livelli sottostanti è stata raggiunta con l'utilizzo del protocollo RPC (Remote Procedure Call) [14]. Il protocollo RPC prevede un'architettura client/server in cui il client invia un datagramma contentente i parametri della richiesta al server, il quale, dopo aver esaminato ed eventualmente eseguito la richiesta, spedisce un messaggio di risposta al client (vedi Figura 3.2). Ogni server NFS mette a disposizione un programma composto da un insieme di procedure (*service*), ognuna di queste procedure è univocamente identificata da una quadrupla: l'indirizzo dell'host, il *program number*, il *version number* e il *procedure number*. Lo scambio di strutture dati tra il client e il server è reso possibile attraverso un'opportuna rappresentazione astratta dei dati chiamata XDR (eXternal Data Rapresentation)[15].

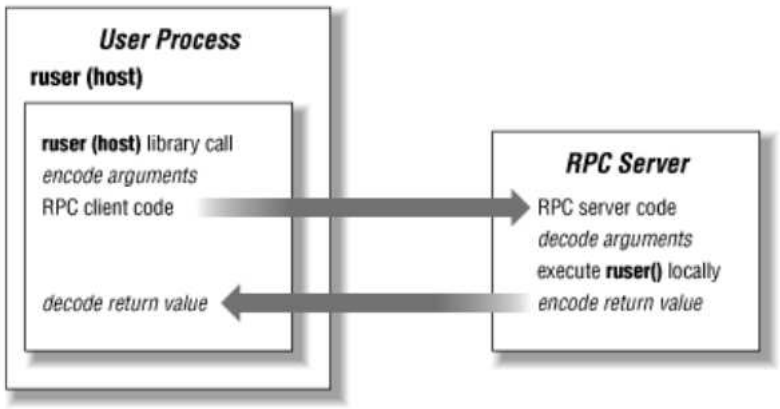

Figura 3.2: Paradigma RPC

Gestione delle richieste

Le richieste RPC vengono attese dai threads del server che sono stati inizializzati dal demone *nfsd*. Esiste poi un altro demone residente sul server chiamato *mountd* che serve a gestire le richieste di mount del filesystem remoto. L'utilizzo del protocollo mount permette ad un client di "attaccare" un sottoalbero di directory fisicamente residente sul server remoto in un ramo dell'albero del filesystem locale. Ogni volta che un client richiede l'esecuzione di un'operazione su un oggetto del filesystem condiviso viene inviata una richiesta RPC al server su cui l'oggetto risiede. Ogni richiesta RPC porta anche informazioni necessarie all'autenticazione dell'utente: nel caso di UNIX per esempio, ogni richiesta conterrà informazioni sull'effettivo user id e grup id del richiedente. Una volta ricevuta la richiesta, il server controlla se l'utente ha i permessi per effettuarla tramite una tabella associativa fra utenti e user id e fra gruppi e group id e attraverso una serie di regole che definiscono i permessi. A questo punto viene chiamata la procedura *lookup*,

la quale cerca la directory per il file name specificato nella richiesta e ritorna il file handle corrispondente [1]. Nel caso di una *open()* di un file la *lookup* viene chiamata per ogni directory presente nel pathname del file. Esistono poi una serie di procedure simili alle system call dei sistemi UNIX per le richieste di esecuzione delle operazioni corrispondenti: *read*, *write*, *getattr*, *setattr*, *mkdir* etc.

3.3.2 Virtual File System

Il Vitual File System permette l'accesso di un sistema client a molti differenti tipi di filesystems come se questi fossero attaccati localmente. VFS nasconde le differenze di implementazione attraverso un'interfaccia consistente. Su un client NFS UNIX, l'interfaccia del VFS fa sembrare tutti i filesystem NFS come se fossero dei filesystem UNIX, anche se questi sono stati esportati da un IBM MVS o da un Windows NT server. Le operazioni che vengono effettuate su interi filesystem, come la richiesta dello spazio libero utilizzabile, sono chiamate *VFS operations*; operazioni che invece vengono effettuate su file o directory vengono chiamate *vnode operations*. Le *vnode operations* vengono eseguite sui *vnodes* che rappresentano un'astrazione degli oggetti che fanno parte di tutti i filesystem verso cui il VFS si interfaccia. Nel caso di NFS, ad esempio, una system call effettuata da un client viene trasformata dal VFS in una virtual node operation, la quale, nel lato server, viene nuovamente trasformata nella corrispondente system call sul filesystem locale [15].

[1] Un file handle è l'equivalente del file descriptor del livello applicativo e viene utilizzato dal protocollo NFS per consentire al client di accedere alle risorse condivise.

3.3.3 Tainter

La componente off-line è stata implementata all'interno del modulo *nfsd* senza aggiungere ulteriori moduli (Figura 3.3). La funzionalità che permette il controllo della legittimità di una richiesta è stata implementata estendendo la funzione *nfsd_permission* che è dedicata al controllo originale dei permessi (l'ordine delle chiamate per l'esecuzione di una scrittura è illustrato in Figura 3.4).

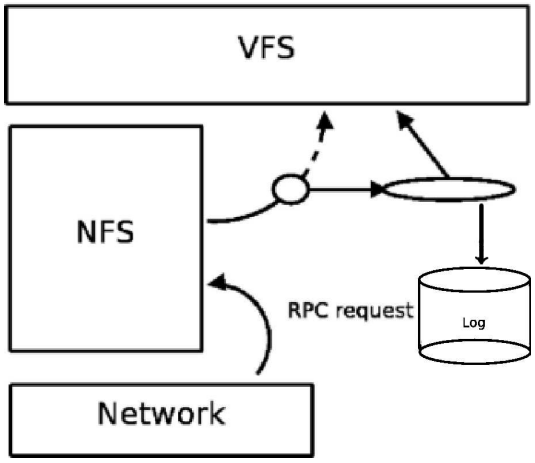

Figura 3.3: Tainter processa le richieste prima che vengano passate al VFS

Classi di sicurezza e loro rappresentazione

Per implementare un modello di flusso sicuro di informazioni e un algoritmo di tainting è stato previsto l'assegnamento di una etichetta ad ogni file e ad ogni utente che utilizza NFS. Le etichette rappresentano le classi di sicurezza a cui appartengono gli oggetti e corrispondono al colore nell'algoritmo di tainting. Le etichette sono state implementate attraverso delle

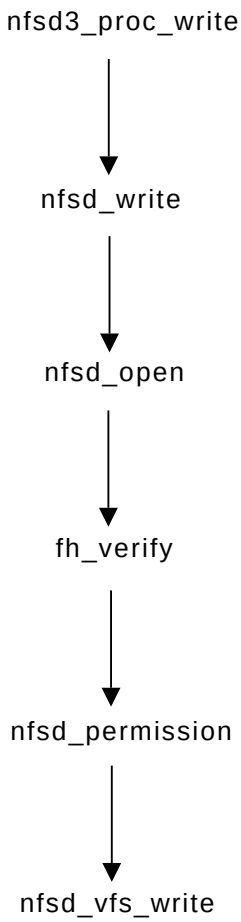

Figura 3.4: All'arrivo di una richiesta di scrittura il server nfsd esegue la sequenza di funzioni riportata in figura.

bitmap a 128 bit la cui configurazione di 1 e di 0 identifica una specifica classe. Inizialmente, quando nessuna operazione è stata eseguita dal sistema, le etichette di file e utenti si assume che abbiano una configurazione con un solo bit uguale a "1" e i restanti uguali "0" .

Operazioni trattate

Le operazioni che sono state estese sono le operazioni di lettura e quelle di scrittura. Come si è visto nel Capitolo 2 queste sono le operazioni che creano flussi di informazioni perchè spostano effettivamente dati da un file ad un'altro. Le altre operazioni, tra cui anche le operazioni di apertura e di chiusura di un file, non sono state trattate nello sviluppo del prototipo perchè non considerate essenziali al fine di ricostruire un flusso di informazioni tra file e utenti. Le funzioni che sono state modificate sono la *nfsd_vfs_read()* e la *nfsd_vfs_write()* rispettivamente per l'operazione di lettura e quella di scrittura.

Inizializzazione

Tainter prevede una fase di inizializzazione che coincide con la prima chiamata di *nfsd_permission* dal momento in cui è stato lanciato il demone nfsd. In questa fase vengono letti tre file di configurazione: *ulabels*, *sort.conf* e *rules.conf* che devono trovarsi in */etc/taint/*. Il primo file contiene associazioni tra userid e classi di sicurezza e viene utilizzato per associare ad un utente la classe di sicurezza a cui appartiene. Il file *sort.conf* specifica un ordinamento parziale tra le classi di sicurezza definendo, per ogni coppia, quale classe è maggiore dell'altra e viceversa. Il terzo file descrive le regole

di sicurezza che si intendono adottare, ovvero specifica, per ogni relazione d'ordine, le operazioni consentite. Le informazioni contenute nei tre file vengono mappate in memoria in apposite strutture dati. In particolare è stata aggiunta una hash table contenente gli user id e le relative classi di sicurezza consentendo inserimenti e ricerche in tempo $O(1)$. Tainter utilizza il meccanismo degli attributi estesi di Linux per assegnare e manipolare le classi di sicurezza associate ad un file. Il meccanismo degli attributi estesi permette di:

- Aggiungere un nuovo attributo ad un file.
- Rimuovere un attributo esteso già esistente.
- Assegnare un valore ad un attributo esteso.
- Leggere il valore corrente di un attributo esteso.

Anche in questo caso possiamo dire che la complessità nella gestione delle etichette è $O(1)$ perchè, data la struttura dati che rappresenta un inode, possiamo accedere in tempo costante al membro che rappresenta l'attributo esteso.

Applicazione delle regole di sicurezza

Ogni richiesta di lettura e scrittura su file deve essere processata in modo da valutarne la legittimità o meno. L'algoritmo determina la relazione d'ordine tra la classe di sicurezza del file e quella dell'utente e controlla se esiste una regola che permette l'operazione richiesta. Nel caso la bitmap del file o quella dell'utente presenti più occorrenze di 1, vengono controllati i permessi per ogni sottoinsieme di bitmap con un 1 solo e l'operazione viene concessa solo se tutti i controlli hanno esito positivo.

Implementazione dell'algoritmo di tainting

Una volta che una richiesta di lettura o di scrittura è stata considerata lecita, prima di essere effettivamente eseguita, vengono aggiornate le etichette del file e dell'utente coinvolti. L'algoritmo di tainting si comporta come specificato di seguito.

- In caso di lettura il flusso di informazioni va dal file all'utente. La classe di appartenenza di quest'ultimo diventa $C(u) \cup = C(f)$.
- In caso di scrittura le informazioni passano dall'utente al file. La classe del file diventa $C(f) \cup = C(u)$.

Il primo caso è stato implementato calcolando l'or bit a bit fra la bitmap dell'utente e la bitmap del file e assegnando il risultato alla nuova bitmap dell'utente. Il secondo caso, invece, è stato implementato calcolando l'or bit a bit fra la bitmap del file e la bitmap dell'utente e assegnando il risultato alla nuova bitmap del file. Una formalizzazione delle regole di dipendenza appena descritte è rappresentata in Tabella 3.1

Abstract Operation	**Color Diffusion**
read <s1 , o>	color(o)U = color(s1)
read <s1 , s2>	color(s2)U = color(s1)
write <s1 , o>	color(o)U = color(s1)
write <s1 , s2>	color(s2)U = color(s1)

Tabella 3.1: Regole di dipendenza implementate dal prototipo.

Logging delle richieste

Per ogni operazione eseguita vengono registrate informazioni relative al tipo, alla data e all'ora in cui è avvenuta, all'utente che l'ha richiesta, al file su cui è stata effettuata, l'offset di lettura o di scrittura e le etichette aggiornate. Il file così costruito verrà utilizzato in fase di analisi per la ricostruzione dei flussi di informazioni.

3.3.4 Analyzer

Analyzer è la componente off-line del prototipo che permette l'analisi dei flussi di informazioni a partire da un insieme di richieste di operazioni su file. Questa componente consiste in un'applicazione user-space scritta in linguaggio C che realizza le seguenti funzionalità:

1. Costruzione del grafo delle dipendenze.
2. Analisi del grafo.
3. Generazione dei risultati.

I prossimi paragrafi descrivono in dettaglio le singole fasi.

Costruzione del grafo delle dipendenze

A partire dal file di log che contiene le richieste viene costruito il grafo delle dipendenze che rappresenta i flussi di informazioni tra file e utenti. Un nodo del grafo può essere un file o un utente mentre un arco rappresenta un'operazione tra di essi. Un nodo del grafo è identificato da un valore e

contiene la bitmap associata al file o all'utente. Nel caso di un nodo utente il valore è lo user id, nel caso invece di un nodo file il valore è rappresentato dalla coppia inode number - generative number. Un arco del grafo contiene tutte le informazioni ralative alla richiesta, come la data in cui è avvenuta e l'offset di lettura o di scrittura espresso in byte. Il grafo che viene generato è un grafo orientato in cui l'orientamento degli archi identifica il flusso di informazioni. Analyzer legge ogni entry del file di log e verifica se l'utente e il file associati alla richiesta sono già presenti nel grafo. Se il file è già presente non viene fatto niente altrimenti viene creato e aggiunto un nuovo nodo. La stessa cosa viene fatta per l'utente. Se la richiesta è una lettura viene aggiunto un arco che va dal nodo file al nodo utente, se invece è una scrittura l'arco viene orientato in senso opposto.

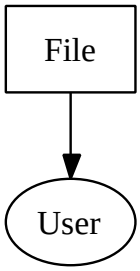

Figura 3.5: Grafo generato a partire da una richiesta di lettura

Il grafo viene implementato mediante due distinte tabelle hash che contengono rispettivamente i nodi utente e i nodi file: questa rappresentazione garantisce un costo di inserimento e di ricerca di un nodo pari a $O(1)$. Le due tabelle hash utilizzano un indirizzamento aperto con scansione lineare e una funzione hash molto semplice che fa uso dell'operatore modulo:

key%prime number

Il valore restituito dalla funzione hash è il modulo fra la chiave di ricerca e un

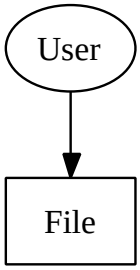

Figura 3.6: Grafo generato a partire da una richiesta di scrittura

numero primo. L'utilizzo di un numero primo riduce il numero di collisioni e quindi aumenta le prestazioni delle operazioni di inserimento e di ricerca. Esiste poi un'altra tabella hash che mantiene l'insieme delle coppie di nodi strettamente connessi in modo da riuscire a trovare in tempo $O(1)$ se un arco è già presente o meno. In caso sia già presente Analyzer aggiorna le informazioni relative all'ora in cui è avvenuta l'operazione e l'offset di lettura o di scrittura. Ogni volta che viene aggiunto un nuovo arco che va da un nodo *node1* ad un nodo *node2* viene anche aggiunto un arco "fittizio" che va da *node2* a *node1* per poter percorrere il grafo in entrambi i sensi nella successiva fase di analisi.

Analisi del grafo

L'algoritmo di analisi prende in input il grafo delle dipendenze e una query. I tipi di query effettuabili sono sei e rappresentano la tipologia di ispezione che si desidera effettuare:

- Effettuando una ricerca all'indietro (back) partendo da un file che rappresenta il *detection point* è possibile risalire al/ai file che possono essere all'origine della contaminazione.

- Effettuando una ricerca in avanti (forward) partendo da un file che rappresenta il *detection point* è possibile raggiungere i/il file che possono essere stati contaminati.

- Effettuando una ricerca all'indietro (back) partendo da un utente che rappresenta il *detection point* è possibile risalire all'/agli utenti che possono essere all'origine della contaminazione.

- Effettuando una ricerca in avanti (forward) partendo da un utente che rappresenta il *detection point* è possibile raggiungere l'/gli utenti che possono essere stati contaminati.

- È possibile controllare se sia avvenuto un passaggio di informazioni tra due specifici file.

- È possibile controllare se sia avvenuto un passaggio di informazioni tra due specifici utenti.

Tutti questi casi si possono essere ricondotti al problema di trovare un cammino fra due nodi del grafo. L'algoritmo implementato è un algoritmo ricorsivo di ricerca in ampiezza (BFS Breadth-First Search) in grado di visitare il grafo nel senso orientato dagli archi e in quello contrario. L'algoritmo marca i nodi visitati per evitare cicli infinti e appena trova una soluzione si ferma. La ricerca non è ottima e i cammini minimi non sono garantiti ma per quanto riguarda il problema della ricostruzione dei flussi di informazione questo aspetto può essere trascurato.

Generazione dei risultati e output grafico

Analyzer produce in output gli eventi che rappresentano i flussi di informazione sulla base della query richiesta. In base a questa lista di eventi l'amministratore del sistema è in grado di ricostruire il ciclo di vita di un'infezione in modo da rendere la fase di ripristino il più efficiente possibile. Viene eseguita, infine, un'ultima visita dell' intero grafo per produrre un file con estensione .dot. Questo file viene passato al programma Graphviz che visualizza il grafo delle dipendenze evidenziando i cammini soluzione.

Capitolo 4

Valutazione delle prestazioni

Questo capitolo fornisce una valutazione del prototipo realizzato, presenta alcune situazioni interessanti per quanto riguarda il comportamento di Tainter e di Analyzer e mostra i risultati dei test effettuati con IOzone. Per effettuare una verifica del funzionamento di Tainter ne è stato confrontato il comportamento con quello di NFS originale in due scenari semplici ma significativi. Nel primo scenario è stato creato un flusso di informazioni diretto o esplicito e rappresenta il caso più semplice che possa essere trattato. La Figura 4.1 mostra la situazione in cui un utente desideri leggere il contenuto del file *c*. La directory corrente è stata montata attraverso il protocollo NFS e risiede su un server esteso con la componente Tainter. Come si può notare dalla figura l'utente avrebbe i permessi necessari ad eseguire l'operazione richiesta la quale invece viene negata. Il motivo per cui l'utente non vede completata la sua richiesta è che non esiste una regola nel file */etc/taint/rules.conf* che permetta un flusso di informazioni tra la classe di sicurezza a cui appartiene il file e la classe di sicurezza a cui appartiene l'utente. Tale politica di sicurezza è stata decisa dall'amministratore del server che ha configurato in modo appropriato il meccanismo di tainting. Vediamo adesso una situazione

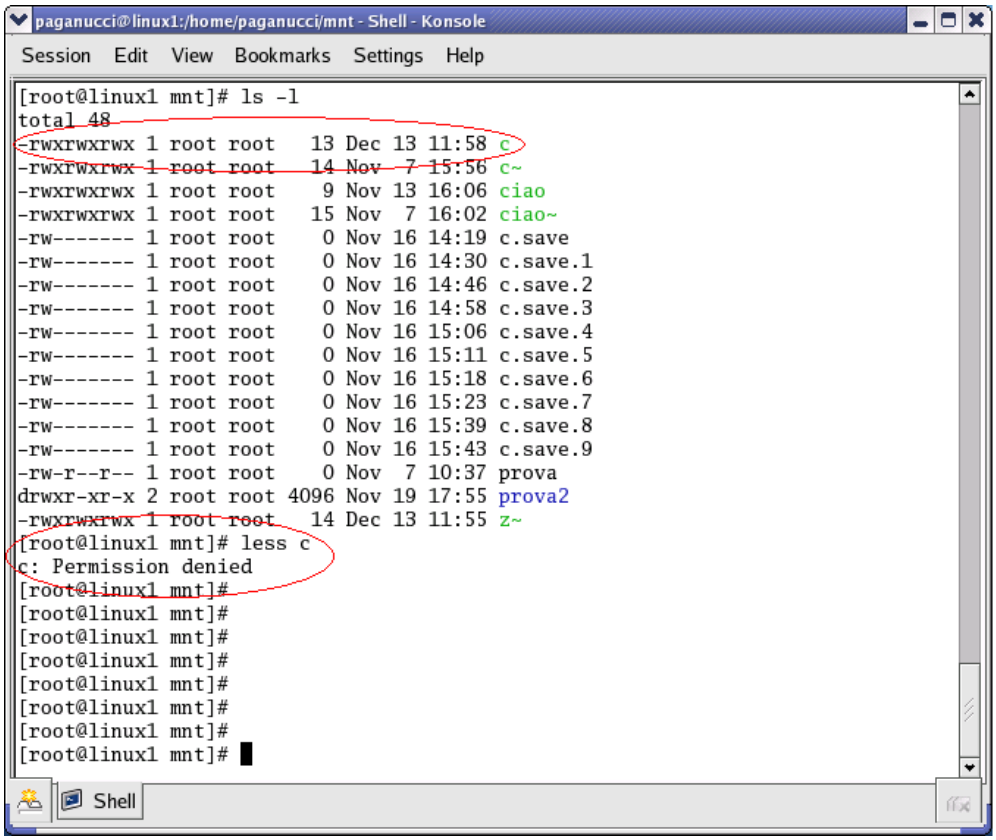

Figura 4.1: Il flusso diretto di informazione fra l'utente e il file viene negato da Tainter.

in cui Tainter sia in grado di prevenire un flusso di informazioni indiretto, ovvero un flusso che ha luogo attraverso l'interazione fra più file e utenti. Il caso più semplice da analizzare è quello in cui abbiamo due file e due utenti. Per rendere più comprensibile l'esempio, consideriamo le etichette dei file e

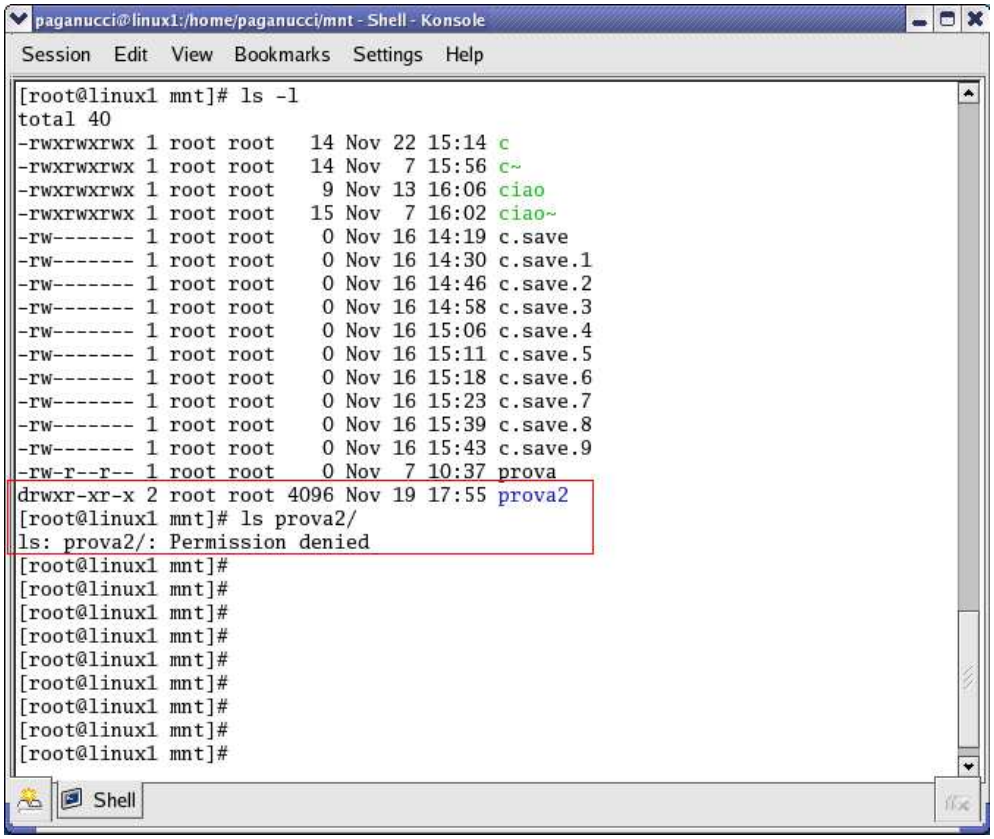

Figura 4.2: Tentativo di lettura della directory *prova2/*. Il contenuto della directory *mnt/* è stato montato via NFS da una macchina estesa con Tainter.

degli utenti come bitmap a 4 bit invece che a 128 bit (le stesse considerazioni posso essere riportate al caso più complesso). Ipotizziamo che inizialmente i file e gli utenti siano associati alle classi di sicurezza come in Tabella 4.1, che il file *sort.conf* sia quello rappresentato in Tabella 4.2 e che il file *rules.conf* sia quello rappresentato in Tabella 4.3. Supponiamo inoltre che vengano

utente1	1000
file1	0100
utente2	0010
file2	0001

Tabella 4.1: Associazione dei file e degli utenti a classi di sicurezza.

$$1000 > 0100$$
$$1000 < 0001$$

Tabella 4.2: Relazioni di ordinamento.

effettuate le richieste elencate in Tabella 4.4. Alla richiesta di op1, Tainter ricava le etichette di utente1 e file1 e trova che la classe di sicurezza del primo è maggiore di quella del secondo. La prima regola del file *rules.conf* permette l'esecuzione dell'operazione e la bitmap dell'utente viene aggiornata facendone l'or bit a bit con quella del file. La seconda richiesta viene anch'essa concessa secondo lo stesso procedimento e la bitmap di file2 diventa 1101. Per quanto riguarda op3, Tainter deve controllare che esistano delle regole che permettano la lettura da una classe di sicurezza uguale a 0010 ad ogni sottoinsieme della classe di sicurezza uguale a 1101 rappresentato da una bitmap con un solo 1 e tutti 0. In questo caso non esiste nessuna regola di questo tipo e, di conseguenza, op3 viene negata. Il controllo originale dei permessi di NFS permetterebbe tutte le richieste provocando un flusso di informazioni dal file1 all'utente2. La Figura 4.2 mostra l'esempio in cui file1

$$> \text{read}$$
$$< \text{write}$$

Tabella 4.3: Regole di sicurezza.

op1	utente1	read	file1
op2	utente1	write	file2
op3	utente2	read	file2

Tabella 4.4: Sequenza delle richieste da effettuare.

sia rappresentato dalla directory *prova2*. In questo caso potremmo supporre che utente1 abbia letto il contenuto di *prova2* e che abbia creato al suo interno un nuovo file il cui nome contiene informazioni confidenziali. Come visto sopra utente2 non è adesso in grado di leggere il contenuto della directory e di accedere quindi a tali informazioni confidenziali. Il funzionamento di Tainter in scenari più complessi può essere dimostrato per induzione riconducendoci ai due casi base descritti in questo paragrafo.

La valutazione delle prestazioni di Tainter è stata effettuata tramite un benchmark per filesystems chiamato IOzone. Le operazioni esaminate durante i test sono l'operazione di lettura e quella di scrittura perchè sono le system calls che sono state modificate per la realizzazione del prototipo. Per prima cosa è stato eseguito un test sul filesystem NFS versione 3 senza Tainter per fornire un confronto fra le due performance. I grafici rappresentati nelle Figure 4.3, 4.5, 4.7 e 4.9 rappresentano i dati ottenuti dall'esecuzione di questo primo test. I test sono stati poi effettuati sul filesystem esteso con Tainter e i dati ottenuti sono quelli rappresentati nelle Figure 4.4, 4.6, 4.8 e 4.10. Com'è possibile osservare, l'overhead sulla velocità di trasferimento introdotto da Tainter è in generale di modeste dimensioni e, per alcuni test, perfino inesistente. Tale costo aggiuntivo può essere ragionevolmente pagato per usufruire delle funzionalità offerte da Tainter e del livello di sicurezza che esso garantisce.

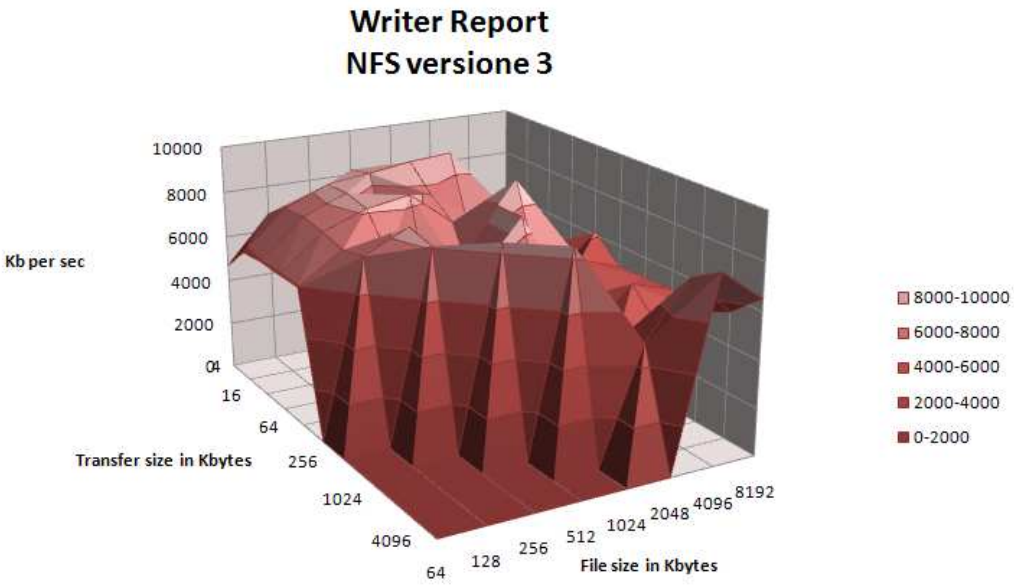

Figura 4.3: Test di scrittura effettuati su NFS versione 3.

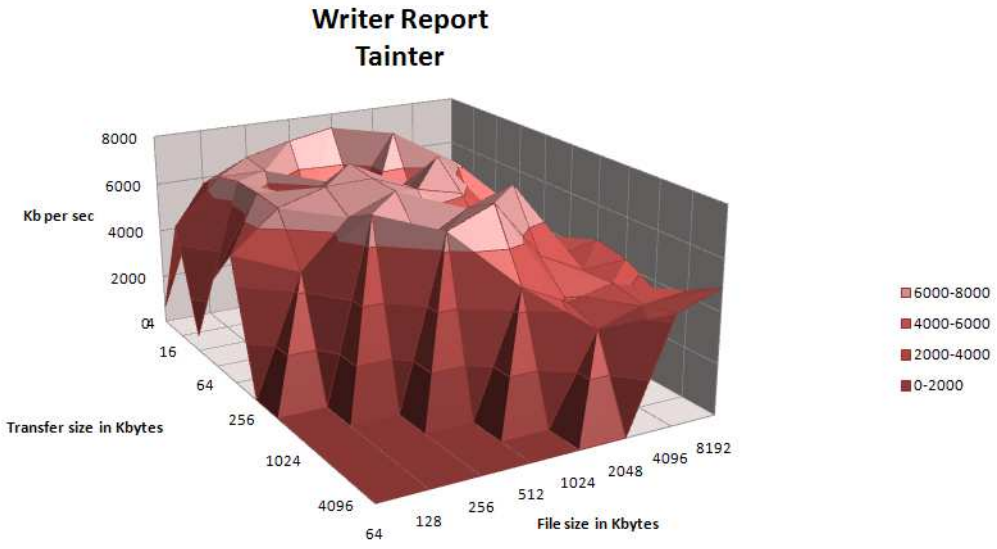

Figura 4.4: Test di scrittura effettuati su Tainter.

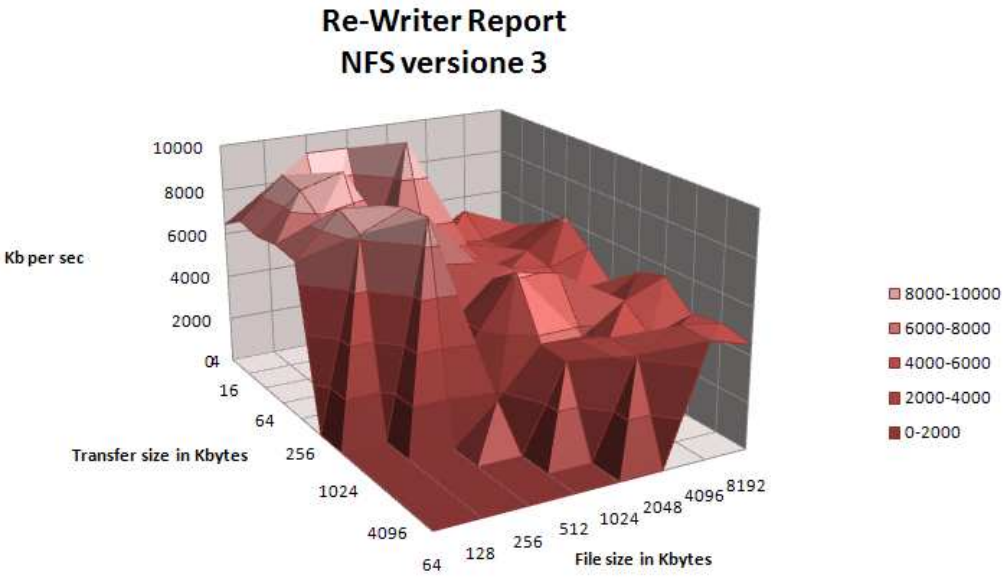

Figura 4.5: Test di riscrittura effettuati su NFS versione 3.

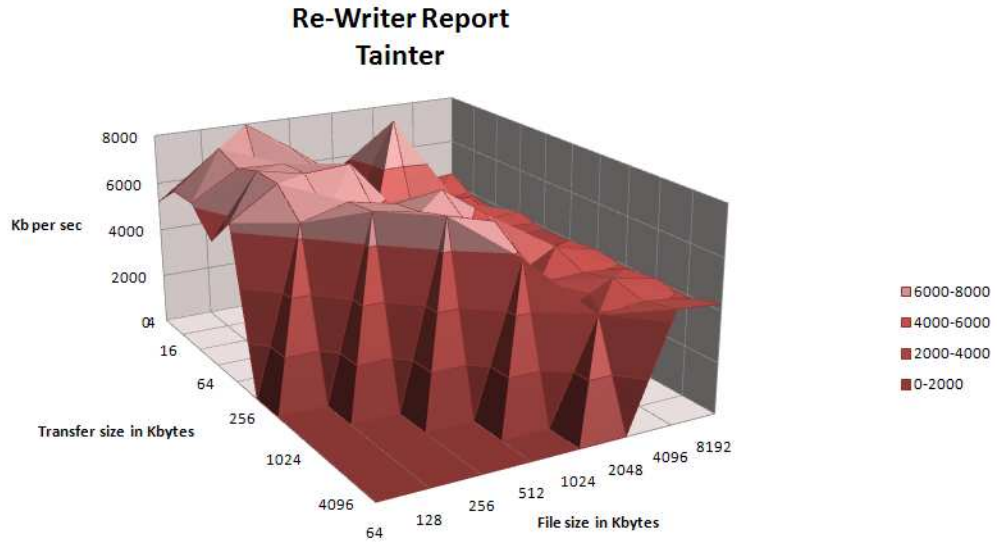

Figura 4.6: Test di riscrittura effettuati su Tainter.

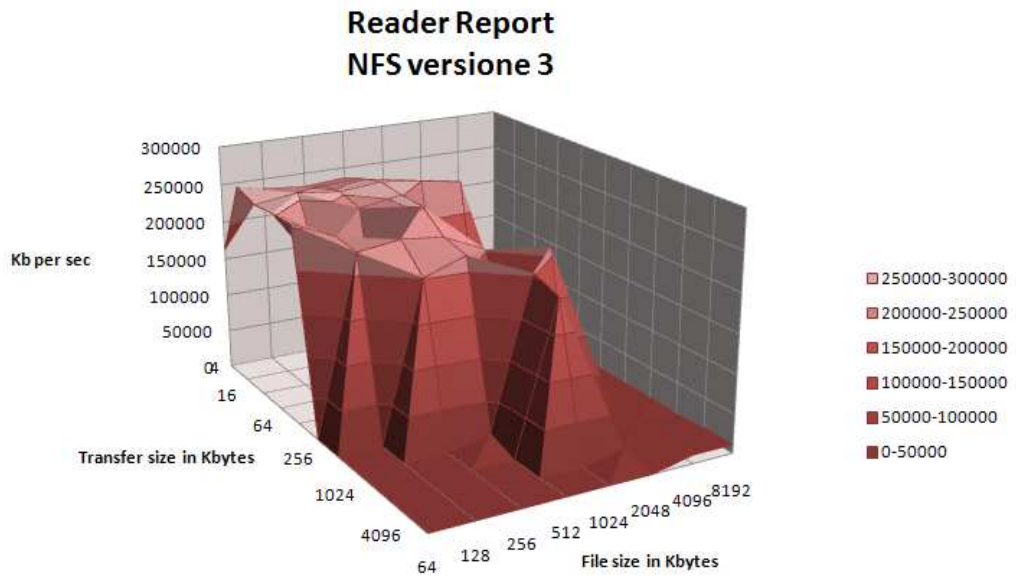

Figura 4.7: Test di lettura effettuati su NFS versione 3.

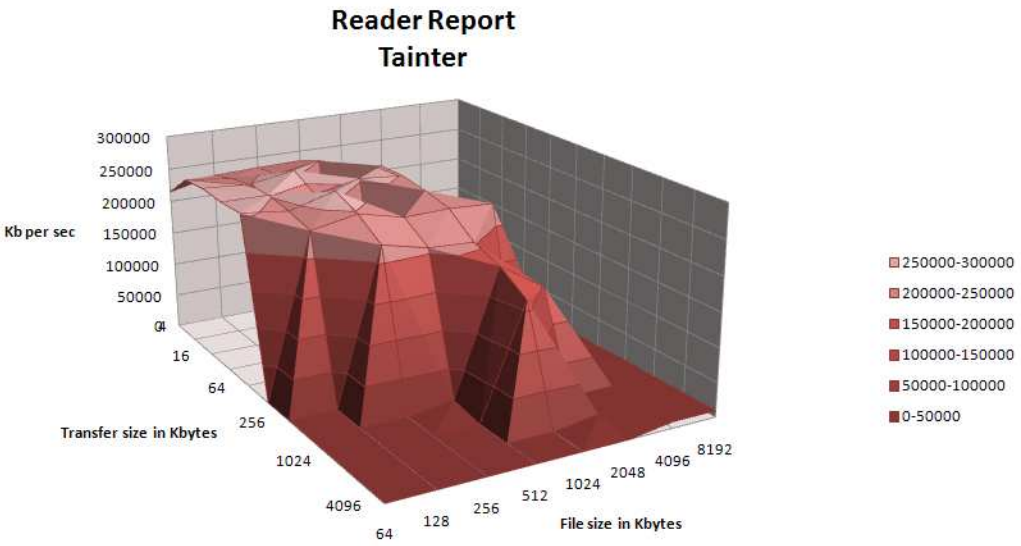

Figura 4.8: Test di lettura effettuati su Tainter.

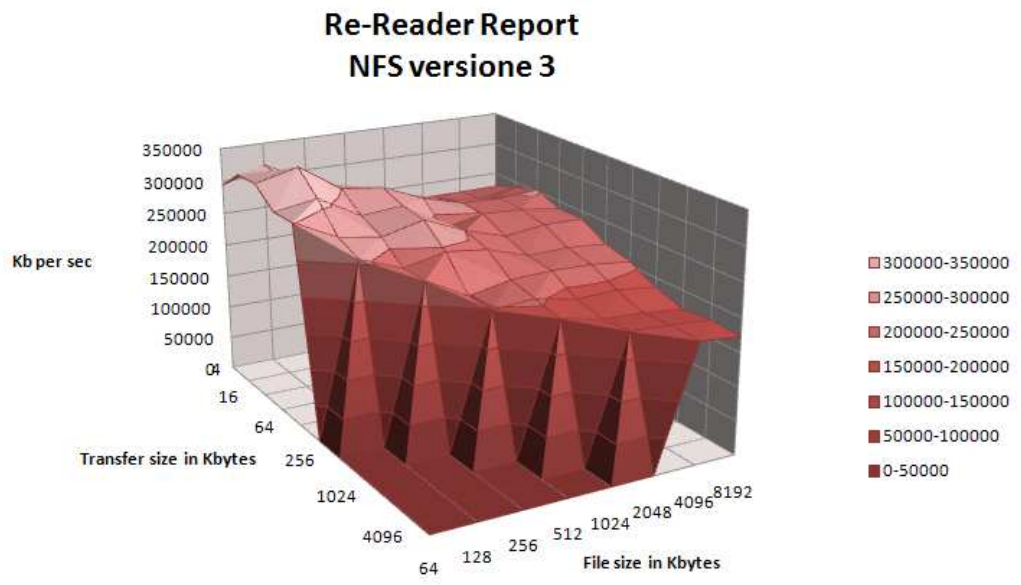

Figura 4.9: Test di rilettura effettuati su NFS versione 3.

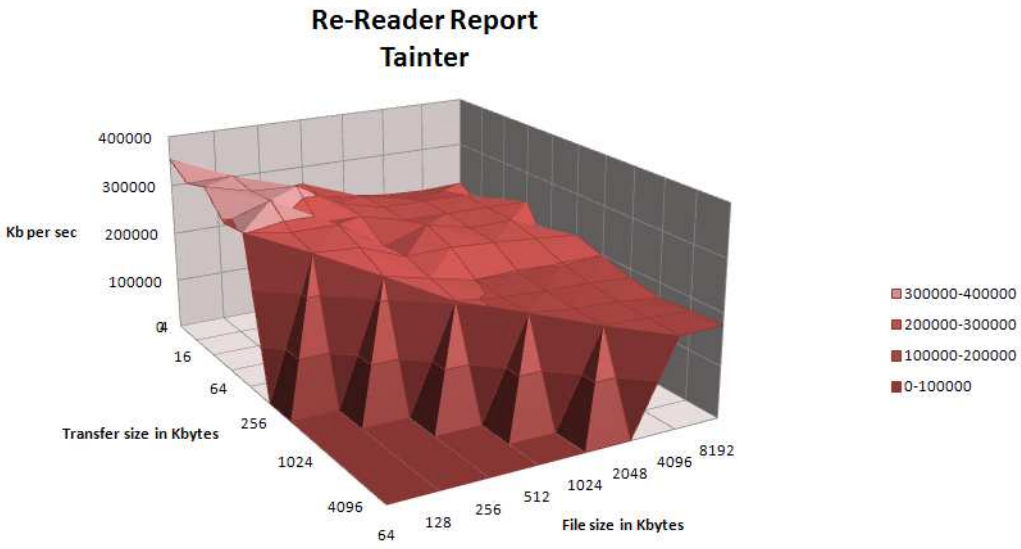

Figura 4.10: Test di rilettura effettuati su Tainter.

Il corretto funzionamento di Analyzer è stato verificato costruendo alcuni file di log contenenti un numero arbitrario di richieste casuali. Come è stato discusso nel Capitolo 3, la costruzione del grafo dei flussi di informazioni ha una complessità lineare nel numero di richieste che compongono il file di log. Se n è il numero di nodi ed m il numero degli archi il suddetto algoritmo ha una complessità pari a $O(n+m)$. L'algoritmo di analisi ha invece una complessità quadratica nel numero di nodi che compongono il grafo perchè per ognuno viene effettuata una visita per trovare se esiste un cammino tra lui e il nodo che rappresenta il *detection point*. L'algoritmo che genera il file .dot effettua un ulteriore visita dell'intero grafo, per questo motivo ha anch'esso complessità $O(n+m)$. Siamo ora in grado di definire la complessità di Analyzer come la somma delle complessità dei tre algoritmi che lo compongono visto che questi ultimi vengono eseguiti in maniera sequenziale ottenendo quindi $O(n^2+2n+2m)$ che corrisponde ad una complessità quadratica. La Figura 4.11 rappresenta il risultato dell'analisi di un grafo ottenuto attraverso una serie di richieste fittizie. In questo esempio il *detection point* coincide con l'utente 10 e i nodi evidenzianti fanno parte dei cammini soluzione della query "Quali file ed utenti sono stati contaminati dall'utente 10?" .

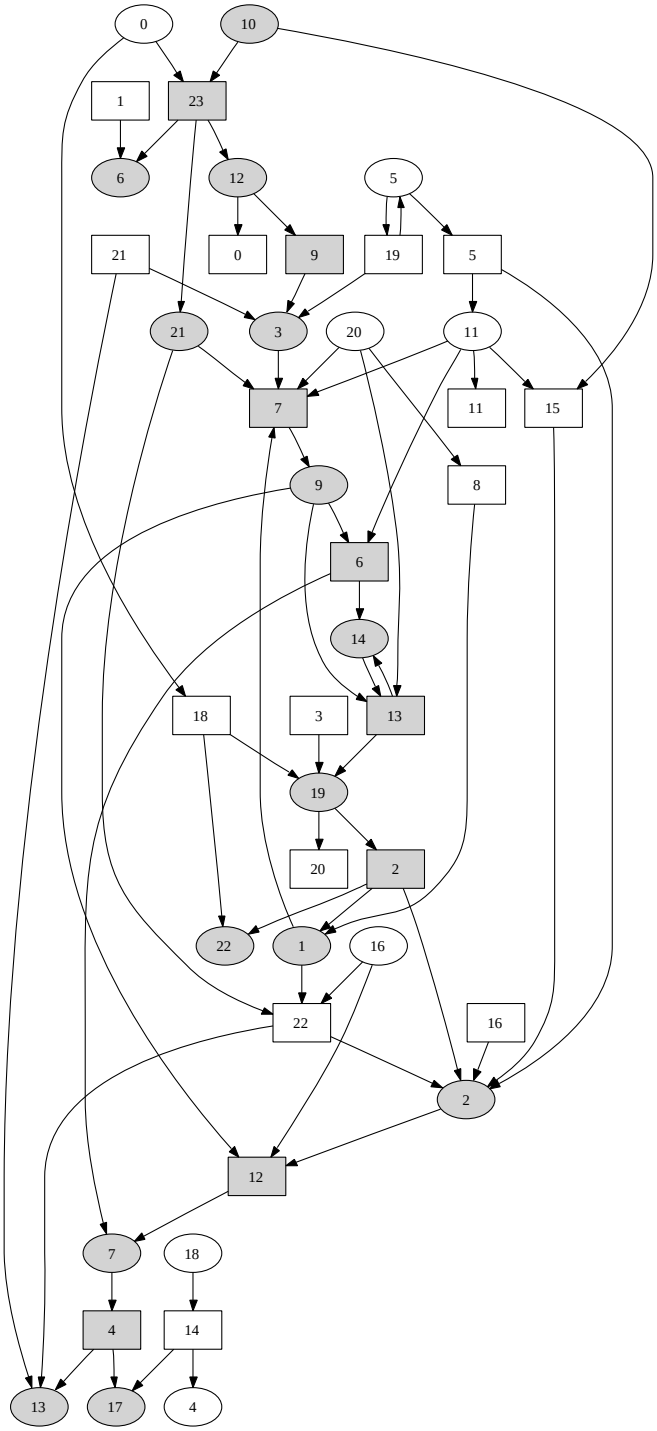

Figura 4.11: Grafo ottenuto da un file di log generato in modo casuale. I rettangoli rappresentano i file mentre le ellissi rappresentano gli utenti.

Capitolo 5

Conclusioni

Lo strumento realizzato durante il tirocinio fornisce ad un sistema automatizzato un meccanismo di sicurezza capace di contrastare intrusioni di tipo informatico attraverso la ricostruzione dei flussi di informazioni tra file e utenti. A partire dalle richieste dei singoli utenti è possibile negare l'esecuzione di determinate operazioni che sono considerate illecite o che potrebbero causare danni al sistema. Lo strumento permette inoltre ad un amministratore di ricostruire la sequenza di eventi che rappresenta un'intrusione e, di conseguenza, ripristinare le parti del sistema che sono state corrotte o danneggiate. Il prototipo che è stato realizzato estende il filesystem Linux/NFS ed i test effettuati dimostrano che la diminuizione delle performance può essere sostenuta per utilizzare lo strumento in un sistema reale ed usufruire delle funzionalità che esso offre. In successivi lavori il prototipo potrebbe essere esteso in modo da considerare anche altri tipi di eventi che causano dipendenze tra file e utenti. Durante il tirocinio non è stato possibile realizzare tali funzionalità per mancanza di tempo ma fornirebbe sicuramente una maggiore completezza. Può inoltre essere realizzata una nuova componente in grado di effettuare il ripristino di un sistema compromesso in maniera totalmente

automatica utilizzando i risultati forniti da Tainter e da Analyzer. La realizzazione di questo meccanismo non è semplice perchè deve essere in grado di riconoscere una serie di situazioni in cui sono sorte delle false dipendenze tra oggetti del sistema e considerare gli istanti temporali in cui sono state eseguite le operazioni.

Bibliografia

[1] Carl E. Landwehr. Formal models for computer security. *Computing Surveys, Volume 13, Number 3*, September 1981.

[2] Ravi S. Sandhu. Lattice-based access control models. November 1993.

[3] Samuel T. King and Peter M. Chen. Backtracking intrusions. 2003.

[4] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. 2005.

[5] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM, Volume 19, Number 5*, May 1976.

[6] *Xen User's Manual.*

[7] *http://wiki.xensource.com/.*

[8] *http://www.iozone.org/.*

[9] *http://www.graphviz.org/.*

[10] Ashvin Goel, Wu chang Feng, Wu chi Feng, and David Maier. Automatic high-performance reconstruction and recovery. *Computer Networks*, April 2007.

[11] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, Mike Shea, Sourabh Ahuja, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. *in 19th Symposium on Operating Systems Principles (SOSP)*, October 2003.

[12] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. *in International Conference on Distributed Computing Systems Security Workshop*, June 2005.

[13] Xuxian Jiang, Aaron Walters, Florian Buchholz, Dongyan Xu, Yi-Min Wang, and Eugene H. Spafford. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach.

[14] IETF. *http://www.ietf.org/rfc/rfc1057.txt*. RPC RFC 1057.

[15] Hal Stern, Mike Eisler, and Ricardo Labiaga. *Managing NFS and NIS*. O'Really, second edition.