UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA

STUDIO E REALIZZAZIONE DI UN ANOMALY BASED NETWORK INTRUSION DETECTION SYSTEM

Relatore: Prof. Danilo BRUSCHI
Correlatore: Prof. Francesco TISATO

Tesi di Laurea di
Giampaolo FRESI ROGLIA
matr. 568394

Anno Accademico 2005/2006

Ai miei genitori.

RINGRAZIAMENTI

Ringrazio tutte le persone che mi sono state vicine in questi anni di studio, ringrazio anzitutto i miei genitori che mi hanno permesso di intraprendere questa strada, ringrazio i miei coinquilini tutti, che mi hanno sopportato in questi anni ed i miei compagni di università con cui ho condiviso la mia passione per l'informatica.

Ringrazio il professor Danilo Bruschi che mi ha consentito di studiare liberamente l'argomento degli Intrusion Detection System ed altri temi della sicurezza informatica che fanno parte delle mie passioni.

Indice

Elenco delle figure

List of Algorithms

Introduzione

In questa tesi viene proposto lo studio e l'analisi delle proprietà di sicurezza di PAYL, un modello di Network Intrusion Detection System (NIDS) anomaly based. Un NIDS è un meccanismo utilizzato per rilevare intrusioni all'interno di una rete informatica. Esistono principalmente due tipologie di tali sistemi: misuse detection e anomaly detection. La prima basa l'efficacia del rilevamento degli attacchi sulle signature, firme caratteristiche delle violazioni conosciute che si trovano all'interno di codice maligno dell'attacco; la seconda basa l'efficacia del rilevamento degli attacchi sulla capacità di misurare le deviazioni statistiche del traffico illecito dal traffico lecito ed è costituita da una fase di learning in cui vengono raccolte e registrate le informazioni del comportamento normale del traffico di rete, e da una fase di detection in cui ogni deviazione che va oltre una soglia predefinita viene identificata come un attacco.

Diversi modelli di anomaly based NIDS sono stati proposti in letteratura: da quelli che basano le fasi di learning e di detection su determinati parametri appartenenti ai livelli più bassi dello stack TCP/IP, in grado così di rilevare anomalie relative a quei livelli (Datalink, Network, Transport), a quelli payload based come PAYL che basa le due fasi sull'analisi dei dati relativi al livello più alto dello stack TCP/IP, il livello applicazione, permettendo così di rilevare una maggior quantità di intrusioni dal momento che la maggior parte degli attacchi avviene a quel livello.

Tale modello si basa sulla costruzione in fase di learning di diversi profili statistici dei dati leciti in transito per ogni servizio e per ogni dimensione di payload osservata. In una successiva fase di detection si deciderà in real-time se ogni pacchetto in transito

è da considerarsi valido misurandone la distanza di Mahalanobis dal relativo profilo generato in fase di learning.

Durante il lavoro di tesi sono state individuate delle criticità su alcune assunzioni alla base della realizzazione di PAYL e legate ad ottimizzazioni del calcolo della distanza di Mahalanobis ed alla classificazione dei payload. In particolare si critica l'assunzione di indipendenza stocastica dei byte costituenti i payload da analizzare e la classificazione dei payload realizzata unicamente in base alla loro dimensione.

Sulla base del modello originale di PAYL e delle criticità individuate, sono stati realizzati un clone di PAYL stesso ed una versione modificata. Sono stati inoltre effettuati test sull'efficacia dei modelli con lo stesso dataset utilizzato dagli autori di PAYL (DARPA dataset 1999). I test hanno evidenziato una maggior precisione in termini di minori falsi positivi e maggior detection rate da parte del modello modificato.

Nel primo capitolo verranno introdotti i modelli di architetture di rete ISO/OSI e TCP/IP, i concetti principali della sicurezza informatica e verrà inoltre fornita una classificazione delle minacce informatiche per poi introdurre il concetto di Intrusion Detection System.

Nel secondo capitolo verrà presentato un modello di Network Intrusion Detection System, PAYL, descrivendone le funzionalità e le assunzioni effettuate dai suoi autori alla base del suo funzionamento.

Nel terzo capitolo verranno descritte le criticità individuate nel modello di payl relative ad alcune assunzioni effettuate per semplificare il calcolo della distanza di mahalanobis e relative alla classificazione dei payload.

Nel quarto capitolo verranno descritti i componenti principali costituenti il modello di PAYL e verrà descritta l'implementazione del clone di PAYL e delle versioni modificate PAYL-B e PAYL-C in base alle criticità individuate nel terzo capitolo.

Nel quinto capitolo si fornirà una descrizione dell'ambiente di test utilizzato e delle metodologie adottate per la verifica dell'efficacia relativamente al clone di PAYL e alle versioni modificate PAYL-B e PAYL-C. Verranno inoltre mostrati i risultati ottenuti nei test.

Capitolo 1

Modello architetture di rete e classificazione attacchi

In questo capitolo introdurremo i modelli di architettura di rete ISO/OSI e TCP/IP, stabilendo da quelli un modello ibrido a cui faremo riferimento nel resto della tesi. Verranno introdotti i concetti principali della sicurezza informatica e definite le varie tipologie di attacco attualmente conosciute e rilevate nell'ambito dell'Information Technology, dandone una collocazione nei corrispondenti livelli dell'architettura di rete. Infine introdurremo il concetto di Intrusion Detection System.

1.1 Modello ISO/OSI e TCP/IP

Per ridurre la complessità nella loro realizzazione, le reti sono organizzate in una struttura definita a layer, ognuno dei quali fornisce nuove funzionalità al livello immediatamente superiore ed allo stesso tempo basa il suo funzionamento sulle funzionalità fornite dal livello immediatamente inferiore.

Tra i modelli più significativi proposti in letteratura si collocano il modello ISO/OSI ed il modello TCP/IP.

Il modello ISO/OSI è suddiviso in sette layer: *Application, Presentation, Session, Transport, Network, Datalink e Physical*.

Il modello TCP/IP invece presenta una suddivisione in quattro layer: *Application, Transport, Internet e Host-to-network.*

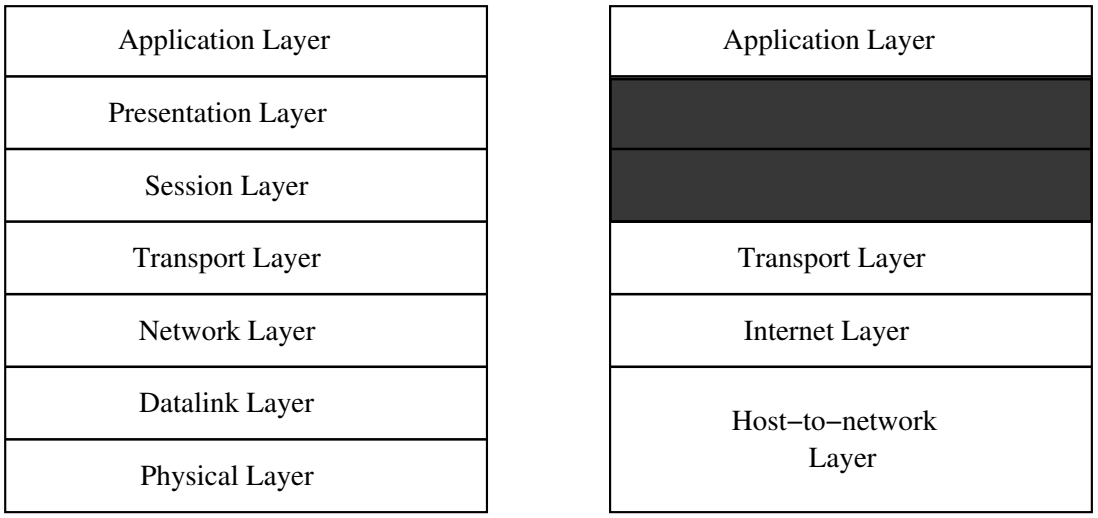

Figura 1.1: Modelli ISO/OSI e TCP/IP

Entrambi i modelli sono stati ampiamente criticati per la loro struttura. Il modello ISO/OSI è uno standard ben definito in tutti i suoi livelli, ma sia per la carenza di implementazioni di buona qualità che per la rivelata inutilità di alcuni suoi livelli come il livello Session ed il livello Presentation, nella pratica non ha preso piede.

Il modello TCP/IP al contrario rappresenta l'architettura di rete più diffusa su vasta scala. Tale modello però non ha la granularità presentata dal modello ISO/OSI, in particolare il layer Host-to-network non è ben definito. Quando ci si riferisce ad esso vien descritto, come unico requisito di tale livello, la capacità di far comunicare due host tra loro. Nel modello TCP/IP risultano totalmente assenti i livelli Session e Presentation. I livelli Application e Transport corrispondono ai livelli con lo stesso nome nel modello ISO/OSI, mentre il livello Internet può essere assimilato al livello Network.

In figura 1.1 sono rappresentati i modelli dello stack ISO/OSI e TCP/IP.

Per quanto riguarda il proseguimento di questa tesi, faremo riferimento ai livelli del modello ISO/OSI, facendo corrispondere al livello Host-to-network del modello

TCP/IP i due livelli Physical e Datalink. Di fatto faremo riferimento al modello ibrido descritto da Tanenbaum in [9] e rappresentato in figura 1.2.

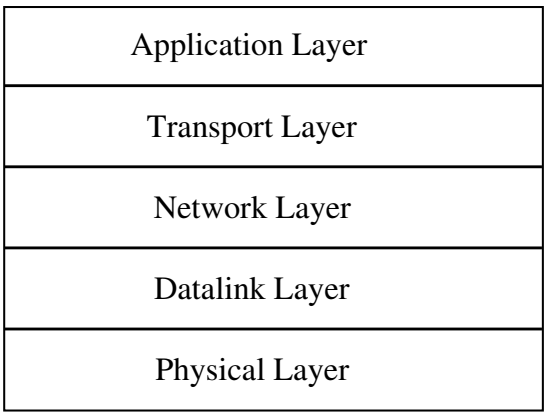

Figura 1.2: Modello di riferimento di questa tesi

1.1.1 Physical Layer

Il Physical Layer è il livello più basso del modello ISO/OSI ed è quello che si occupa di definire le specifiche fisiche dei canali di comunicazione come le caratteristiche del materiale di cui sono costituiti i cavi di comunicazione, come debbano essere realizzati i connettori fisici dei cavi e come debbano essere codificati i segnali. Il livello fisico fornisce in pratica un'astrazione che permetta di inviare sequenze di bit nel canale di comunicazione.

1.1.2 Datalink Layer

Il Datalink Layer è il livello che si appoggia al livello fisico. Questo è il livello che si occupa di fornire un'astrazione del canale fisico, suddividendo i segnali in frame di dati ripuliti da eventuali errori di trasmissione, si occupa anche delle modalità di trasmissione in canali di tipo *broadcast* in cui due o più host possono accedere contem-

poraneamente al mezzo fisico causando così delle collisioni che comprometterebbero i dati trasmessi.

Fa parte del livello Datalink il protocollo Ethernet. Collochiamo però per convenzione nel livello Datalink anche protocolli come ARP[1] e RARP[2], che anche se non strettamente legati alla risoluzione dei problemi di cui si occupa il livello Datalink, non troverebbero una collocazione all'interno dei livelli superiori. I due protocolli sono di supporto al livello Network e troverebbero una collocazione più corretta a cavallo dei livelli Datalink e Network.

1.1.3 Network Layer

Il livello Network si occupa del controllo delle funzionalità della rete. Si occupa in particolare di stabilire un canale di comunicazione tra due macchine definendo il percorso che i messaggi devono seguire per raggiungere la loro destinazione. Controlla inoltre il traffico lungo la rete qualora vi siano presenti troppi messaggi nello stesso istante, al fine di evitare congestioni. Il livello Network rende inoltre possibile la comunicazione tra sottoreti di tipo diverso. Fa parte di questo livello il protocollo IP[3], oltre ai relativi protocolli di controllo e di gestione come ICMP[4] ed SNMP[5]. Fanno inoltre parte del livello Network i protocolli di routing come OSPF[6] e BGB[7].

1.1.4 Transport Layer

Il livello Transport ha il compito di gestire i messaggi in ingresso dai livelli superiori e, se necessario, dividerli in parti più piccole per inviarli al livello Network, occupandosi di farli arrivare correttamente a destinazione privi di errori. Il livello di trasporto è il primo livello *end-to-end*, ovvero il processo sulla macchina sorgente comunica direttamente col processo sulla macchina destinazione. Nei livelli inferiori invece la

[1] Address Resolution Protocol
[2] Reverse ARP
[3] Internet Protocol
[4] Internet Control Message Protocol
[5] Simple Network Management Protocol
[6] Open Shortest Path First
[7] Border Gateway Protocol

comunicazione avviene tra una macchina ed una sua vicina e non fra i due estremi della comunicazione, che possono essere separati da diverse altre macchine.

Fanno parte del livello di trasporto protocolli come TCP[8] ed UDP[9]. In particolare il protocollo TCP fornisce ai livelli più alti il concetto di connessione, in cui i messaggi arrivano tutti a destinazione nell'ordine in cui sono stati spediti.

1.1.5 Application Layer

Il livello applicazione contiene una varietà di protocolli che sono comunemente richiesti. Per esempio possiamo considerare relativamente al servizio di trasferimento file la quantità di file system presenti in rete ed incompatibili tra loro. Una funzionalità che deve offrire il livello applicazione consiste nel gestire le incompatibilità dovute per esempio ad una convenzione diversa sull'assegnazione dei nomi dei file. Il trasferimento di file attraverso due macchine deve essere in grado di gestire queste incompatibilità. Il livello applicazione si basa sulle funzionalità offerte dal livello di trasporto. Esso contiene tutti i protocolli di livello più alto come i servizi di terminale virtuale cifrati e non cifrati (SSH e TELNET), il servizio di trasferimento file (FTP), il servizio di posta elettronica (SMTP), il web (HTTP), risoluzione dei nomi internet (DNS) ed altri ancora.

1.2 Sicurezza Informatica

La sicurezza informatica negli anni passati veniva spesso considerata più un'arte che una vera e propria disciplina. L'apprendimento di essa era basato sull'esperienza più che sullo studio. Pochi studi pionieristici hanno posto le basi per i metodi formali sviluppati negli ultimi anni.

Il continuo espandersi delle reti e l'aumentare dei servizi da rendere disponibili al pubblico, hanno reso necessaria una formalizzazione rigorosa dei concetti relativi alla sicurezza informatica.

[8] Transmission Control Protocol
[9] User Datagram Protocol

1.2.1 Il paradigma C.I.D.

Gli obiettivi della sicurezza informatica vengono spesso descritti con i termini che costituiscono il paradigma C.I.D.(*Confidenzialità, Integrità, Disponibilità*).

- **Confidenzialità:** Per Confidenzialità si intende la capacità del sistema informatico, di fornire l'accessibilità alle proprie risorse solamente alle persone che hanno il diritto ad accedervi.

- **Integrità:** Per Integrità si intende la capacità del sistema informatico, di consentire la modifica dei dati presenti nel sistema stesso alle sole persone autorizzate.

- **Disponibilità:** Per Disponibilità invece si intende la capacità del sistema informatico di garantire alle persone l'accesso ai dati per cui essi sono autorizzati. Il concetto di disponibilità serve a completare i due concetti precedenti del paradigma C.I.D. Senza il concetto di disponibilità un sistema inaccessibile a chiunque sarebbe conforme al paradigma.

Le varie realizzazioni del paradigma C.I.D. si possono suddividere in due gruppi principali: il paradigma *D.A.C* (discretionary access control) ed il paradigma *M.A.C.* (mandadory access control). Entrambi i paradigmi condividono il concetto di identificazione di un utente e di cosa l'utente stesso sia autorizzato a fare.

Il paradigma D.A.C. è quello a cui siamo più abituati, esso è il meccanismo su cui si basa il controllo degli accessi tramite i permessi nei sistemi UNIX e tramite le ACL nei sistemi Microsoft. In pratica ogni oggetto ha un proprietario, ed ogni proprietario può disporre dell'oggetto come crede, concedendo i permessi ad altri utenti fino a poter cedere la proprietà dell'oggetto ad un'altro utente. Tipicamente esiste un utente speciale, l'amministratore, che detiene la proprietà degli oggetti del sistema ed è in grado di violare tutte le restrizioni d'accesso.

Il paradigma M.A.C. invece si basa sui livelli di segretezza e sui livelli di accesso. Gli oggetti del sistema hanno un loro livello di segretezza (confidential, secret, top-secret), a cui è associato un livello di accesso. Gli utenti sono associati ad un livello di accesso. Sono previste regole restrittive per cui un utente non possa scrivere verso i

livelli inferiori al suo e non possa leggere dai livelli superiori. Per il paradigma M.A.C. è prevista la figura del *Security Officer* che si occupa della gestione dei livelli di accesso e segretezza.

1.2.2 Difetti del paradigma C.I.D.

Il paradigma C.I.D. nelle sue incarnazioni sembrerebbe dare tutte le garanzie necessarie per la sicurezza di un sistema informatico. La realtà però dimostra l'impossibilità di garantire la conformità di un sistema informatico a tale paradigma.

I sistemi informatici sono spesso costituiti da una varietà sempre maggiore di software. Per motivi legati alle risorse economiche disponibili, le aziende tendono ad utilizzare software di terze parti, quasi mai utilizzano software su cui hanno il completo controllo. Le necessità di interoperabilità tra i diversi software si scontrano spesso con le limitazioni imposte da una precisa politica di sicurezza, per cui si rendono spesso necessari dei compromessi che tendono ad allentare tali politiche.

Un'altro problema è costituito dall'impossibilità oggettiva di valutare completamente la conformità di un software al paradigma C.I.D. I test sul software sono sempre rivolti a rivelare la presenza di errori ma questo non ci garantisce nulla riguardo la loro assenza.

Ultimo problema, ma non come importanza, è costituito dal comportamento degli utenti nei confronti delle politiche di sicurezza. Questo problema è considerato spesso come l'anello più debole della catena della sicurezza informatica. Le politiche di sicurezza si basano sostanzialmente sull'identificazione dell'utente, questo però non impedisce ad un utente di condividere la password con altri utenti. Un'altro fattore influente sulla sicurezza è costituito dalla scelta delle password. Gli utenti tendono a scegliere password facili da ricordare e spesso facili da ricavare per un utente malevolo. L'utilizzo forzato di password difficili da ricavare porta spesso a comportamenti dannosi da parte dell'utente esemplificati dall'abitudine di segnare la propria password nei post-it e di attaccarli al proprio monitor.

1.3 Tipologie di attacco

Nell'ambito dell' *Information Security* possono essere definite attacchi informatici tutte le azioni che violano in qualche modo le restrizioni di confidenzialità, integrità e disponibilità imposte dal paradigma C.I.D. Gli attacchi informatici possono essere suddivisi in due categorie principali:

- **Attacchi Locali:** Gli attacchi locali sono costituiti dai tentativi di intrusione il cui presupposto è l'accesso diretto al sistema informatico. Possiamo portare come esempio un utente malevolo posto fisicamente davanti alla tastiera del sistema "vittima" ma anche un utente remoto e con la possibilità di accedere al sistema "vittima" tramite servizio di login.
- **Attacchi Remoti:** Sono considerati invece attacchi remoti i tentativi di intrusione portati da utenti malevoli che non abbiano inizialmente un'accesso diretto alla macchina vittima.

Possiamo effettuare una classificazione degli attacchi informatici sia per gli attacchi locali che per gli attacchi remoti. Per gli scopi di questa tesi forniamo una classificazione orientata in particolar modo agli attacchi di tipo remoto. Le tipologie di attacco remoto possono essere suddivise in cinque categorie principali:

- Denial of Service
- Information Gathering (raccolta informazioni)
- Accessi utente non autorizzati
- Accessi con privilegi di amministratore non autorizzati
- Accessi a dati sensibili ed interazione con essi

Gli attacchi di tipo Denial of Service (D.o.S) sono tipologie di attacco mirate all'induzione di malfunzionamenti nei servizi erogati verso il pubblico. Possono essere fini a se stessi oppure possono far parte di attacchi più estesi, in cui il D.o.S. può costituire un mezzo per l'attaccante per poter portare a segno attacchi di altro tipo. Questo

tipo di attacchi si può collocare nella pila ISO/OSI nei livelli *Datalink*, *Rete*, *Trasporto* e *Applicazione*.

Gli attacchi di tipo Information Gathering costituiscono spesso una fase di preparazione da parte dell'attaccante ad attacchi ben più gravi, in cui cerca di ottenere il maggior numero di informazioni dalle vittime, come la topologia della rete, i servizi forniti al pubblico, tipologia e versione del software in esecuzione sugli apparati della vittima. Come gli attacchi di tipo D.o.S. anche questi attacchi possono essere collocati su vari livelli della pila ISO/OSI: *Rete*, *Trasporto* e *Applicazione*.

Gli Accessi utente non autorizzati sono costituiti da tentativi portati a termine per accedere a servizi il cui uso è destinato ad un numero limitato di utenti come servizi di login (telnet, ssh, rlogin), ma anche accessi a database. Spesso questi accessi non autorizzati possono costituire l'inizio di attacchi più gravi, essendoci da parte dell'attaccante la possibilità di accedere a privilegi più elevati di quelli garantiti ad un utente non amministratore. Questo tipo di attacchi viene generalmente portato sfuttando attacchi di forza bruta al fine di indovinare le password scelte dagli utenti, ma accessi di questo tipo si possono ottenere anche sfruttando vulnerabilità di programmazione presenti nei servizi remoti o nei protocolli di rete. Questa tipologia di attacchi si colloca nella pila ISO/OSI sui livelli di *Rete* e *Applicazione*.

Gli accessi come amministratore costituiscono la minaccia più pericolosa, poichè generalmente l'amministratore può gestire i servizi a suo piacimento; accessi di questo tipo possono essere effettuati in via remota sfruttando direttamente vulnerabilità dei servizi, effettuando attacchi di forza bruta sulle password, oppure in via locale, avendo a disposizione (lecitamente o meno) un'accesso diretto al servizio come utente non privilegiato, sfruttando vulnerabilità locali del sistema operativo oppure ottenendo informazioni sensibili poco protette grazie all'accesso diretto al servizio. Questa tipologia di attacchi si colloca sulla pila ISO/OSI sui livelli di *Rete* e *Applicazione*.

Gli attacchi ai dati sono costituiti da tentativi di accesso portati a termine, di rivelare dettagli interni del servizio attaccato allo scopo di interagire con i dati memorizzati dal servizio stesso. Questa tipologia di attacchi si colloca sulla pila ISO/OSI sul livello *Applicazione*.

1.4 Collocazione attacchi nel modello ISO/OSI

In questa sezione cercheremo di collocare le varie tipologie di attacchi precedentemente descritte, sui vari livelli del modello di architetture di rete di riferimento mostrato in figura 1.2.

Gli attacchi remoti possono essere portati ai diversi livelli della pila ISO/OSI:

- Livello Datalink
- Livello Rete
- Livello Trasporto
- Livello Applicazione

1.4.1 Attacchi a livello Datalink

Gli attacchi al livello datalink, in una tipica rete ethernet consistono in tentativi di sfruttare le vulnerabilità intrinseche dei protocolli di supporto come il protocollo ARP, a sfruttare le capacità fisiche limitate di alcuni apparati di rete oppure a sfruttare errori di configurazione degli apparati stessi. Il protocollo ARP si occupa di effettuare una traduzione degli indirizzi IP del livello rete in indirizzi relativi al livello datalink. Tale protocollo è intrinsecamente insicuro basandosi sull'assunzione di fiducia riposta nelle risposte date dalle macchine presenti in rete. In sostanza un'attaccante puo' indurre alcune macchine ad inviare il traffico destinato ad altre macchine verso la propria, rendendo così possibili altri attacchi ad un livello più alto dello stack TCP/IP come gli attacchi di tipo MITM[10]. Questa tecnica è conosciuta come *ARP Poisoning*.

Un'altro tipo di attacchi possibili a livello datalink consiste nell'indurre gli apparati di rete a comportarsi in un modo diverso da quello per cui sono stati realizzati; un'esempio può essere dato dagli attacchi volti all'intasamento delle tabelle ARP di risoluzione degli indirizzi. Tali tabelle sono infatti limitate in dimensione ed una quantità eccessiva di indirizzi da memorizzare puo' portare alla cessazione dell'attività degli apparati coinvolti oppure a modifiche nel loro funzionamento, rendendo così

[10]Man In The Middle

uno switch di rete equiparabile come funzionalità ad uno HUB. In pratica in uno switch il traffico tra due macchine collegate tra loro attraverso di esso viene visto solamente dalle due macchine interessate, mentre con un HUB il traffico tra le due macchine viene visto da tutte le macchine collegate all'apparato. Questo attacco particolare può tradursi, secondo la classificazione precedentemente descritta, in un attacco di tipo D.o.S. nel caso gli apparati reagiscano a quest'evento impedendo il transito di ulteriore traffico. Potrebbero invece rendere possibili altre tipologie di attacco come il già citato MITM nel caso contrario.

1.4.2 Attacchi a livello Rete

Gli attacchi a livello rete coinvolgono i protocolli di livello rete come IP ed i relativi protocolli di supporto come ICMP.

Tra gli attacchi possibili a livello rete possiamo segnalare l'*IP Spoofing*, che consiste nel falsificare l'indirizzo IP sorgente dei messaggi inviati in rete per poter così sfruttare debolezze di protocolli a livello più alto che in alcuni casi basano il riconoscimento di un'utente proprio sul quell'indirizzo, come accade con protocolli di accesso remoto tuttora utilizzati come *rlogin*.

Un'altro tipo di attacco diffuso in rete è conosciuto come *Smurf attack*. Lo Smurf attack si basa sull'IP Spoofing e consiste nell'invio di uno o più messaggi di controllo, appartenenti al protocollo ICMP. Il messaggio di controllo utilizzato consiste in un *ICMP echo-request*, a cui, da protocollo una macchina dovrebbe rispondere con un messaggio di tipo *ICMP echo-reply*. Un messaggio di quel tipo inviato ad un indirizzo di broadcast in una particolare sottorete, viene ricevuto e processato da tutte le macchine appartenenti alla sottorete destinazione generando cosi' tanti messaggi ICMP echo-reply in direzione della macchina specificata nell'indirizzo sorgente della richiesta echo. La conseguenza di un'attacco di questo tipo può essere un Denial Of Service dovuto all'eccessiva quantità di traffico. Un'attaccante potrebbe inoltre falsificare l'indirizzo IP sorgente indicandone uno appartenente ad un'altra macchina in rete che in questo caso subirà le conseguenze dell'attacco. Questo tipo di attacco fa parte della tipologia D.o.S.

Un'altro attacco collocabile a questo livello del modello ISO/OSI è costituito dall'*IP Sweeping* e consiste in pratica in una verifica da parte dell'attaccante che ad un determinato indirizzo IP corrisponda una macchina accesa. Questo non è propriamente un'attacco ma rappresentando una parte della fase di preparazione da parte di un'attaccante, può essere considerato un campanello d'allarme, solitamente preludio di attacchi più gravi. Questo tipo di attacchi fa parte della tipologia Information Gathering.

1.4.3 Attacchi a livello Trasporto

Gli attacchi a livello Trasporto coinvolgono i protocolli di trasporto come il TCP e l'UDP. A questo livello si collocano attacchi conosciuti come *Port Scanning*, in cui un'attaccante cerca di raccogliere il maggior numero di informazioni sulla presenza o meno di servizi su macchine rispondenti a determinati indirizzi IP. Un'altro attacco possibile a questo livello è l'attacco conosciuto come *OS Fingerprinting*, strettamente legato al Port Scanning, in cui l'attaccante può, in determinati casi, essere in grado di riconoscere il tipo di sistema operativo remoto in virtù delle differenti implementazioni del TCP/IP dovute ad un'interpretazione differente dello standard stesso da parte delle case produttrici. Port scanning ed OS fingerprinting fanno parte della tipologia di Information Gathering.

Un'altro tipo di attacchi possibile a livello trasporto, consiste nel tentativo di saturare le tabelle che un sistema operativo remoto deve mantenere per tener traccia dello stato delle connessioni. Queste tabelle sono limitate nelle dimensioni. Con l'invio di una quantità eccessiva di messaggi in rete costituenti per esempio la parte iniziale di una connessione, tali tabelle possono raggiungere presto il loro limite massimo, rendendo quindi impossibile lo stabilirsi di ulteriori connessioni. Questo attacco è meglio conosciuto come *SYN Flood* e si traduce, secondo la classificazione delle tipologie di attacco precedentemente presentata, in un attacco di tipo D.o.S.

1.4.4 Attacchi a livello Applicazione

Gli attacchi a livello Applicazione invece coinvolgono tutti i protocolli di alto livello come i servizi di login remota Telnet, Ssh, rlogin, servizi di trasferimento file come FTP, servizi web come HTTP ed HTTPS, servizi di posta elettronica come SMTP, SMTPS, IMAP, IMAPS, POP, POP3, il servizio di risoluzione nomi o DNS e tante altre tipologie di servizi. Il livello di applicazione è quello soggetto alla maggior varietà di attacchi remoti, dovuta alla gran varietà di servizi offerti ed alla quantità di diverse implementazioni disponibili di tali servizi. Si possono ritrovare nel livello applicazione la maggior parte delle tipologie di attacco elencate. Dall'information gathering, effettuato andando a leggere il contenuto dei cosittetti *banner* con cui si annunciano spesso i servizi remoti, ai D.o.S. mirati al singolo servizio come l'attacco *crashIIS*, in cui una richiesta http particolare causa la cessazione dell'erogazione del servizio web. Sono presenti in questo livello anche altri tipi di D.o.S. mirati al degrado delle prestazioni delle macchine remote, inducendo un servizio all'esecuzione di tante istanze dei processi server e causando un riempimento eccessivo della tabella dei processi del sistema operativo. Gli attacchi a questo livello dello stack TCP/IP sono spesso mirati allo sfruttamento di errori di programmazione nelle varie implementazioni disponibili dei servizi come controlli errati o mancanti sulle dimensioni delle variabili che possono essere sfruttati attraverso tecniche di exploiting[11] conosciute come *buffer overflow* oppure a errori nell'utilizzo delle stringhe di formato relative alla famiglia di funzioni '*printf' conosciuti come *format bug*. In pratica questo tipo di attacchi consiste nell'invio da parte di un'attaccante di un *payload* contenente un flusso di dati appositamente costruito per indurre il servizio remoto a comportarsi diversamente dalle sue funzionalità originarie. Solitamente il payload contiene codice binario costruito ad hoc per eseguire una shell interattiva, fornendo spesso all'attaccante la possibilità di ottenere il completo controllo della macchina.

[11] sfruttamento

1.5 Intrusion Detection Systems

A fronte delle possibilità di violazione delle politiche di sicurezza, si rende necessario trovare dei meccanismi complementari di difesa. Tra i meccanismi di difesa necessari, ci sono quelli il cui scopo consiste nell'individuazione delle intrusioni, il contenimento del danno e la riparazione delle falle della sicurezza. Gli Intrusion Detection System (IDS) giocano un ruolo fondamentale relativamente al rilevamento delle intrusioni.

Ciò che un IDS dovrebbe fare consiste nell'individuazione delle azioni che cercano di far agire il sistema in un modo diverso da quello per cui è stato progettato.

Gli IDS possono essere suddivisi in due grandi categorie in base all'approccio utilizzato per il rilevamento delle intrusioni:

- **Signature (o misuse) based IDS:** questa categoria di IDS comprende tutti i sistemi che tentano di rilevare i comportamenti sospetti utilizzando una base di conoscenza contenente un catalogo di attacchi noti. Tale base è costituita dalle cosiddette "signature".

- **Anomaly based IDS:** questa categoria di IDS invece comprende i sistemi che tentano di rilevare le intrusioni studiando il comportamento normale degli utenti, tipicamente generandone un profilo statistico, per poi segnalare eventuali deviazioni da tale profilo come anomalie.

Gli IDS basati sulle signature, richiedono alla base del loro funzionamento, uno studio estensivo delle forme di attacco per poter generare le "signature" necessarie al loro rilevamento. L'efficacia del sistema dipende fortemente dalla qualità delle signature e dal loro continuo aggiornamento per tenere il passo della crescente quantità di attacchi nuovi da rilevare.

Gli IDS basati sul rilevamento delle anomalie invece non richiedono una *knowledge base* come nel caso dei signature based ma richiedono una fase più o meno lunga di training, in cui il sistema viene addestrato per delineare un profilo del traffico lecito a cui far riferimento nella fase di detection.

Gli IDS signature based e anomaly based, possono a loro volta essere suddivisi in altre due classi differenziate in base alla loro collocazione in un sistema informatico:

- **Host based IDS:** più brevemente HIDS si occupano di controllare una singola macchina, a volte una singola applicazione e la loro implementazione è strettamente legata al sistema operativo per tracciare chiamate di sistema, utilizzo di risorse e cambio di privilegi.
- **Network based IDS:** più brevemente NIDS si occupano di monitorare il comportamento di una rete cercando di controllare tutto il traffico in transito cercando all'interno dei flussi di dati indicazioni di possibili attacchi.

Ognuna delle due classi di IDS presenta vantaggi e svantaggi. I Network based IDS possono controllare anche reti di grandi dimensioni ma non sono in grado di individuare per esempio attacchi che sfruttano come mezzo di trasmissione un canale cifrato. Gli Host based d'altro canto presentano la necessità di essere installati nelle singole macchine da monitorare e le loro implementazioni sono dipendenti dal sistema operativo.

1.5.1 Proglemi degli IDS

Tra i problemi tipici che un IDS si trova a dover affrontare, possiamo indicare il problema dell'individuazione degli attacchi nuovi ed anche degli attacchi modificati.

Gli IDS signature based presentano come unica soluzione lo sviluppo di nuove signature per gli attacchi nuovi, e di signature più generiche per gli attacchi modificati. Questo tipo di soluzione però rende gli IDS signature based inutili per questo tipo di attacchi nella durata di una finestra di tempo più o meno ampia in cui gli esperti del settore devono sviluppare e diffondere tali signature. Il problema si evidenzia maggiormente in presenza di polimorfismo negli attacchi.

Gli IDS anomaly based d'altro canto si pongono l'obiettivo di identificare non un attacco in particolare, ma una deviazione del comportamento del sistema da quello che è considerato un comportamento normale, per cui sono meno sensibili alle mutazioni degli attacchi e dovrebbero almeno in linea teorica riconoscere sia gli attacchi normali che quelli mutati.

Gli IDS anomaly based però soffrono potenzialmente di inefficacia nei confronti di tecniche di evasione mirate a questo tipo di IDS. Una tecnica particolarmente in-

teressante è costituita dagli attacchi denominati *Mimicry*. In breve un'attaccante cerca di nascondere le proprie tracce e quindi passare inosservato all'IDS tentando il più possibile di dare al proprio attacco l'aspetto di un comportamento normale.

Gli IDS nella loro funzione, sono soggetti a due categorie di errori: i *falsi positivi* e i *falsi negativi*.

- **Falsi positivi:** sono costituiti da segnalazioni di allarme effettuate a fronte di azioni lecite, possono essere considerati dei veri e propri falsi allarmi. In pratica viene segnalato un tentativo di intrusione quando esso non è avvenuto. I falsi positivi possono sembrare innocui ma l'eccessiva quantità porterebbe inevitabilmente ad ignorare la totalità degli allarmi da parte delle persone assegnate al loro monitoraggio. Spesso sono considerati falsi positivi anche istanze di attacchi veri e propri portati a segno fuori dal loro contesto. Per esempio un attacco mirato a sfruttare una vulnerabilità del server web IIS ma portato a segno nei confronti del server web Apache non avrebbe successo, ma potrebbe benissimo essere segnalato da parte di un IDS.

- **Falsi negativi:** sono costituiti da mancate segnalazioni di allarme a fronte di un attacco portato a termine. Nei sistemi signature based, i falsi negativi sono associati solitamente alla mancanza delle signature relative agli attacchi non riconosciuti. Nei sistemi anomaly based invece, i falsi negativi possono essere riscontrati anche a fronte di un attacco precedentemente riconosciuto, a causa della natura statistica di tali IDS.

Capitolo 2

PAYL

In questo capitolo descriveremo un modello di Networ Intrusion Detection System: PAYL. Descriveremo i suoi obiettivi ed il suo funzionamento introducendo la definizione del profilo del traffico lecito e le metriche utilizzate per il rilevamento delle anomalie.

2.1 Descrizione di PAYL

PAYL si colloca nella categoria dei Network Intrusion Detection System Anomaly based. PAYL è un NIDS proposto da S.J.Stolfo e K.Wang[10] e si pone come obiettivo il rilevamento delle intrusioni esaminando il contenuto (*Payload*) del livello più alto della pila ISO/OSI, il livello applicazione. Si differenzia dagli altri NIDS anomaly based come SPADE[2], PHAD[7], ALAD[6] e NIDES[1] che modellano i profili del traffico lecito esaminando le intestazioni dei livelli di rete e di trasporto del modello ISO/OSI. Tali NIDS sono quindi in grado di rilevare anomalie riferite solamente a quei livelli.

Vi sono alcuni NIDS come NATE che costruiscono un profilo del traffico lecito in modo "ibrido" basandosi sulle intestazioni dei livelli rete e trasporto ed anche su una piccola parte (otto byte) del livello applicazione.

PAYL è il primo a proporsi di esaminare l'intero contenuto del traffico a livello applicazione.

In PAYL, come nella maggior parte dei NIDS anomaly based, si possono distinguere due fasi nel suo funzionamento:

- fase di training
- fase di detection

Nella fase di training viene generato un profilo del traffico lecito basandosi sul contenuto dei payload del livello applicazione. Nella fase di detection viene calcolata la distanza di *Mahalanobis* per verificare la conformità dei dati in transito col profilo costruito e genera un allarme nel caso tale distanza superi una soglia determinata in base alla capacità di detection voluta ed alla quantità di falsi positivi che si ritiene accettabile.

2.2 Profilo dei payload

I payload del livello applicazione non hanno un formato predefinito, contrariamente a quanto avviene con le intestazioni relative ai livelli rete e trasporto, i valori dei byte sono generalmente indipendenti dalla loro posizione nel flusso di dati. Per costruire un profilo dei payload è necessaria una suddivisione del traffico in diversi insiemi, questo viene fatto in base ad alcuni criteri come:

- Protocollo di trasporto (TCP, UDP)
- Porta del servizio in ascolto
- Dimensione del payload

La suddivisione tra i protocolli e tra le porte è giustificata dal fatto che tipicamente un servizio ha associata una coppia protocollo-porta ed ogni servizio avrà un suo payload caratteristico a distinguerlo dagli altri. Per esempio, il protocollo HTTP ha un linguaggio diverso rispetto al protocollo FTP o al protocollo SMTP. Questi protocolli daranno origine presumibilmente a profili di payload diversi tra loro.

L'ulteriore suddivisione basata sulla dimensione del payload è dovuta al fatto che tipicamente pacchetti di dati brevi conterranno istruzioni proprietarie del protocollo,

mentre pacchetti che si avvicinano alla dimensione massima (che per una rete ethernet è di 1460 byte) presumibilmente conterranno dati multimediali. Un esempio efficace può essere dato dal protocollo HTTP in cui le descrizioni di una pagina html sono costituite da solo testo e sono solitamente di breve lunghezza, mentre le immagini che con la descrizione in formato testo della pagina html la vanno a completare sono spesso superiori alla dimensione massima.

Per esigenze di complessità computazionale e di ragionevole occupazione di risorse il profilo viene generato basandosi sull'analisi degli n-grammi, in particolare degli 1-grammi, ossia della distribuzione dei byte. Un n-gramma è una sequenza di n byte adiacenti all'interno del payload, in pratica viene fatta scorrere una finestra di n byte nel payload, vengono contate le occorrenze di ogni n-gramma e successivamente vengono divise per la quantità di n-grammi contenuti nel payload. Per n = 1 il risultato è un vettore contenente la quantità media di occorrenze di ogni singolo byte nel payload. Per ogni payload di uguale dimensione viene calcolato il vettore delle quantità medie di occorrenze dei byte. Tale vettore viene definito *Feature Vector*. Dall'insieme di Feature Vector relativi a payload della stessa lunghezza, viene calcolato il vettore medio. Dal momento che diversi insiemi di Feature Vector molto variabili tra loro, possono avere un vettore medio uguale a quello di altri insiemi di Feature Vector molto meno variabili, si rende necessario tener conto anche della varianza di tali vettori al fine della costruzione di ogni modello.

Più formalmente, per una data quantità di traffico in fase di training, PAYL genera un profilo $P_{i,j}$ costituito da una coppia $(\mu_{i,j}, \sigma_{i,j})$ dove i e j sono rispettivamente la lunghezza del payload e la porta del servizio di cui si sta generando il profilo, mentre $\mu_{i,j}$ e $\sigma_{i,j}$ sono rispettivamente il vettore medio e il vettore delle deviazioni standard dei Feature Vector osservati. In fase di detection, per ogni pacchetto osservato, viene generato il Feature Vector relativo per poi venir confrontato col profilo corrispondente e, nel caso in cui il vettore sia particolarmente diverso dalla distribuzione di riferimento, viene generato un'allarme.

2.2.1 Costruzione del profilo

Consideriamo un insieme $\mathbf{P} = \{\mathbf{p}_1, \mathbf{p}_2 \ldots, \mathbf{p}_n\}$ di n payload osservati di uguale lunghezza m. Per ogni $\mathbf{p}_i$ viene costruito il Feature Vector $\mathbf{x}_i = \{\frac{x_{i,0}}{m}, \frac{x_{i,1}}{m}, \ldots, \frac{x_{i,255}}{m}\}$ dove i valori $x_{i,0}, x_{i,1}, \ldots, x_{i,255}$ sono le occorrenze dei valori *ASCII* osservate nel payload $\mathbf{p}_i$. Si ottiene quindi un insieme di vettori $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ da cui si calcola il vettore dei valori medi:

$$\overline{\mu} = \frac{\sum_{i=1}^{n} \mathbf{x}_i}{n} \tag{2.1}$$

ed il vettore delle deviazioni standard:

$$\overline{\sigma} = \sqrt{\frac{\sum_{i=1}^{n} (\mathbf{X}_i - \overline{\mu})^2}{n}} \tag{2.2}$$

I due vettori così ottenuti andranno a costituire il profilo di riferimento per i payload di lunghezza n. Il procedimento viene seguito per ogni lunghezza di payload osservata e per ogni porta di cui si è interessati a costruire il profilo $P_{i,j}$.

2.3 Distanza di Mahalanobis

La distanza di *Mahalanobis* è una misura di distanza utilizzata in statistica per determinare il gradi di similitudine tra due variabili aleatorie multidimensionali. Si differenzia dalla distanza euclidea in quanto tiene conto anche delle correlazioni tra le variabili aleatorie considerate. Tale distanza è utilizzata da PAYL come metro di confronto in fase di detection tra i payload nuovi osservati e i profili di riferimento generati in fase di training.

La formula della distanza di Mahalanobis è la seguente:

$$d^2(x, y) = (x - \overline{y})^T \Sigma^{-1} (x - \overline{y}) \tag{2.3}$$

dove x, in PAYL è il Feature Vector relativo al payload appena osservato, mentre $\overline{y}$ è il valore atteso dei Feature Vector relativi ai payload osservati in fase di training. Σ^{-1} invece è l'inversa della matrice di varianza-covarianza dei Feature Vector calcolata sulla

base dei dati osservati in fase di training. Gli autori di PAYL fanno una semplificazione della distanza di Mahalanobis basandosi sull'assunzione che i byte presenti nel traffico da analizzare siano stocasticamente indipendenti tra loro. Questo porterebbe ad avere una matrice di varianza-covarianza diagonale, per cui la formula della distanza diverrebbe cosi:

$$d(x,\overline{y}) = \sqrt{\sum_{i=0}^{n-1} \frac{(x_i - \overline{y}_i)^2}{\overline{\sigma}_i^2}} \tag{2.4}$$

Infine, per evitare di dover calcolare radici quadrate e limitare il numero di moltiplicazioni necessarie al calcolo, la formula effettivamente utilizzata è la seguente:

$$d(x,\overline{y}) = \sum_{i=0}^{n-1} \frac{|x_i - \overline{y}_i|}{\overline{\sigma}_i} \tag{2.5}$$

2.4 Distanza di Manhattan

La distanza di *Manhattan*, conosciuta anche col nome di *Taxicab distance*, è una misura che rappresenta intuitivamente la distanza che dovrebbe coprire un taxi per muoversi tra due punti nella città di Manhattan. Consideriamo due punti $X = (x_1, x_2)$ e $Y = (y_1, y_2)$, nella geometria euclidea la distanza tra X ed Y viene rappresentata come $d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$, mentre nella *Taxicab Geometry* la distanza è definita come $d = |x_1 - y_1| + |x_2 - y_2|$. Tale definizione di distanza può essere estesa a spazi n-dimensionali, in tal caso la distanza di Manhattan tra due punti $X = (x_1, x_2, \ldots, x_n)$ e $Y = (y_1, y_2, \ldots, y_n)$ in uno spazio ad n dimensioni, risulta definita come $d = \sum_{i=1}^{n} |x_i - y_i|$.

In figura 2.1 è rappresentato un'esempio in due dimensioni di distanza di manhattan confrontato con la distanza euclidea.

La distanza di manhattan verrà utilizzata in payl come metrica per stabilire quali profili sono più simili tra loro ed effettuare o meno la fusione dei profili nella fase di clustering che descriveremo più avanti.

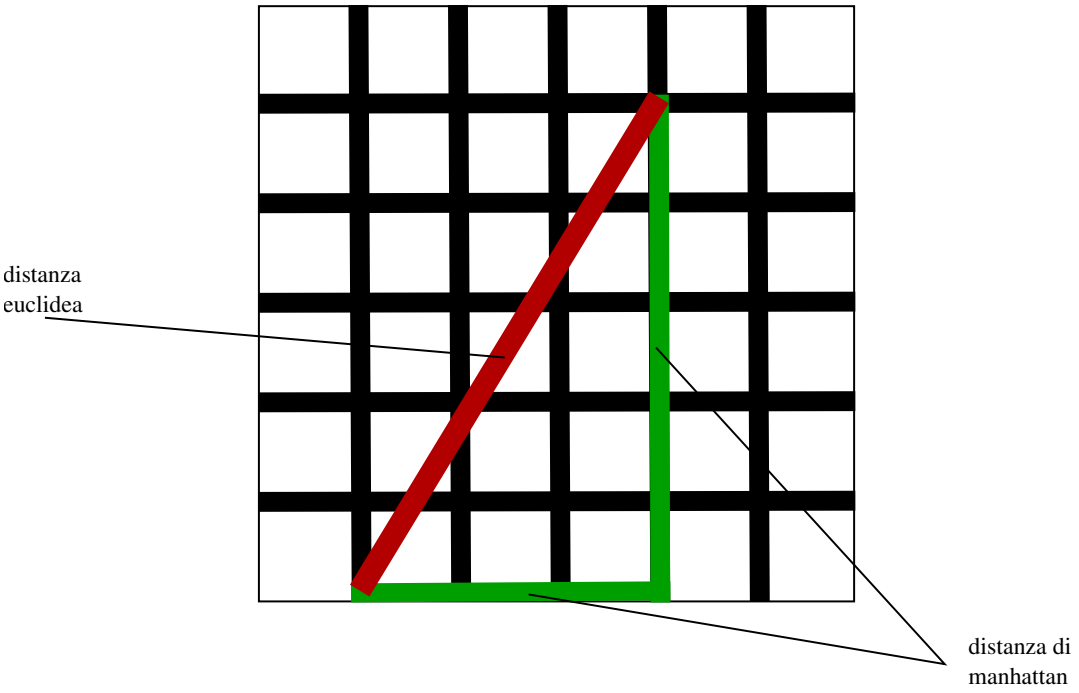

Figura 2.1: Manhattan distance

2.5 Costruzione incrementale del Profilo

Nella fase di training, i profili possono essere aggiornati in modo incrementale, utilizzando una tecnica descritta da Knuth[4] in cui le medie $\bar{x}$ dei singoli caratteri possono essere aggiornate con il nuovo valore x_{n+1} se si tiene in memoria il valore n che rappresenta il numero di campioni analizzati, per cui conoscendo la media $\bar{x}_n$ ed il numero di valori n che la determinano, la nuova media $\bar{x}_{n+1}$ può essere calcolata così:

$$\bar{x}_{n+1} = \bar{x}_n + \frac{x_{n+1} - \bar{x}}{n+1}$$

Per l'aggiornamento della varianza, data la seguente proprietà:

$$Var(X) = E[X - E[X]]^2 = E[X^2] - (E[X])^2$$

è sufficiente tenere in memoria ed aggiornare insieme al valore $E[X]$ anche il valore

$E[X^2]$, per poi calcolare la varianza o la deviazione standard una volta terminata la fase di training.

2.6 Clustering

In termini di occupazione di memoria, considerando che ogni profilo $P_{i,j}$ deve memorizzare due vettori di 256 valori in virgola mobile oltre al numero di campioni analizzati e che le possibili dimensioni di un pacchetto contenente dati in una tipica rete ethernet sono comprese tra 1 e 1460, per ogni porta j, avremmo un'occupazione di memoria pari a $1460 \cdot (2 \cdot 256 + 1) \cdot 4 = 2,9 Mbyte$. Data la richiesta di risorse del profilo, anche limitando la quantità di servizi di cui si vuol controllare l'integrità, una tale richiesta di memoria può rappresentare un problema. PAYL risolve il problema introducendo il *clustering*. In breve si tratta di fondere assieme ogni coppia di profili $P_{i,j}$ con una lunghezza di payload simile, quando la distanza di *Manhattan* tra due profili adiacenti è al di sotto di una predeterminata soglia. In pratica si procede come segue: per ogni coppia di profili $P_{i,j}, P_{i+1,j}$ e per ogni i nell'intervallo $(1 \ldots 1459)$ viene calcolata la distanza di manhattan tra i rispettivi vettori di valori medi; se la distanza di manhattan tra i due profili è inferiore alla soglia, i due profili vengono fusi assieme. Il ciclo di fusione viene ripetuto finchè non ci sono più profili da fondere. Il clustering, oltre a ridurre sensibilmente la quantità di memoria utilizzata dai profili, tende a ridurre i problemi relativi alla scarsa quantità di campioni che si potrebbe avere per alcuni valori di i; ciò ha come conseguenza anche una riduzione generale della quantità di falsi positivi, portando ad avere profili più significativi in termini statistici del traffico osservato.

2.7 Smoothing factor

In base alle caratteristiche del traffico osservato ed alla quantità di campioni utilizzati per costituire i profili del traffico lecito in fase di training, è possibile che il calcolo della distanza mediante la formula (2.5) possa dar luogo a divisioni per zero. Per risolvere questo problema viene introdotto nel calcolo delle deviazioni standard uno

Smoothing factor, ossia un fattore di arrotondamento. In pratica ai valori dell'intero vettore delle deviazioni standard viene sommata una quantità abbastanza grande da risolvere il problema delle divisioni per zero ed abbastanza piccola da non generare eccessive perturbazioni nel calcolo finale della distanza stessa.

La necessità di utilizzare lo smoothing factor diminuisce in presenza di maggiori quantità di campioni analizzati, e viene inoltre ridotta dall'utilizzo del clustering, che di fatto non fa altro che fondere tra loro due profili, col risultato di ottenere un profilo con una quantità di campioni maggiore rispetto agli originali.

Capitolo 3

Considerazioni sul modello di PAYL

In questo capitolo descriveremo le criticità rilevate nel modello di payl relativamente alle assunzioni alla base di alcune semplificazioni effettuate per la realizzazione del modello originale. Verrà inoltre effettuata una ridefinizione del profilo del traffico lecito ed evidenziata la necessità di aggiungere nel modello il componente classificatore.

3.1 Indipendenza stocastica dei byte e ridefinizione del profilo

PAYL, durante la fase di detection, basa il controllo del grado di similitudine dei dati in transito rispetto ai profili del traffico generati in fase di training, non tanto sulla distanza di Mahalanobis (2.3), ma sulla distanza euclidea normalizzata (2.4), a cui si riduce la distanza di Mahalanobis in presenza di campioni stocasticamente indipendenti tra loro.

Gli autori di PAYL giustificano la scelta in base al fatto che la presenza di un valore di un byte o meno all'interno di un flusso di dati non condizioni in alcun modo la presenza di altri valori di byte all'interno dello stesso flusso.

L'assunzione di indipendenza stocastica dei byte può valere in alcuni casi e non in altri. In particolare, tale assunzione si è rivelata vera per flussi di dati distribuiti uniformemente, come per esempio in una connessione cifrata o nel flusso relativo al

trasferimento di un file compresso, mentre si è rivelata generalmente falsa per flussi contenenti dati in cui è possibile rilevare una struttura.

Un esempio pratico può essere dato dal codice HTML che costituisce il linguaggio utilizzato per definire la struttura che costituisce le pagine web, sulla quale si basano i vari Browser per darne una rappresentazione grafica. Tale linguaggio è costituito da 'tag' che identificano varie proprietà del testo da visualizare. In particolare i tag, sono costituiti da parole chiave racchiuse tra parentesi angolari, come per esempio i tag di inizio pagina: <html>, ed il tag di fine pagina: </html>.

Ora è lecito aspettarsi da questo tipo di flusso di dati una certa correlazione tra il valore del byte rappresentante la parentesi angolare aperta $<$ ed il valore del byte relativo alla parentesi angolare chiusa $>$ che si traduce in un valore della covarianza tra i due caratteri elevato. Questo discorso vale oltre che per l'esempio dell'html anche per altri tipi di flusso che contengano una struttura non così nettamente facile da rilevare come nel caso delle parentesi angolari, ma anche per il trasferimento di file contenenti testo in lingua Italiana, o in un'altra lingua, come capita spesso nel contenuto delle e-mail oppure per le sessioni di login interattiva non cifrata, come le sessioni telnet o le sessioni ftp. Forse appare poco intuitivo ma vale anche relativamente al trasferimento di file eseguibili in formato binario o a file di documenti come il formato .doc od il formato .xls e ppt, rispettivamente proprietari delle applicazioni come Word, Excel e Power Point.

Si può vedere dalla figura 3.1, la rappresentazione grafica della distribuzione dei byte all'interno di un flusso cifrato relativo al traffico contenuto in una sessione SSH[1]. Le ascisse rappresentano il valore dei byte, le ordinate rappresentano la probabilità che un dato valore si presenti o meno. Il traffico di questo tipo di servizio è compresso e cifrato.

In figura 3.2 invece abbiamo una rappresentazione tridimensionale della matrice di varianza-covarianza relativo al traffico osservato in una sessione SSH, la cui distribuzione di frequenza è rappresentata nella figura 3.1. In questo caso l'approssimazione effettuata in PAYL relativamente all'indipendenza stocastica dei byte risulta essere buona.

[1] Secure SHell

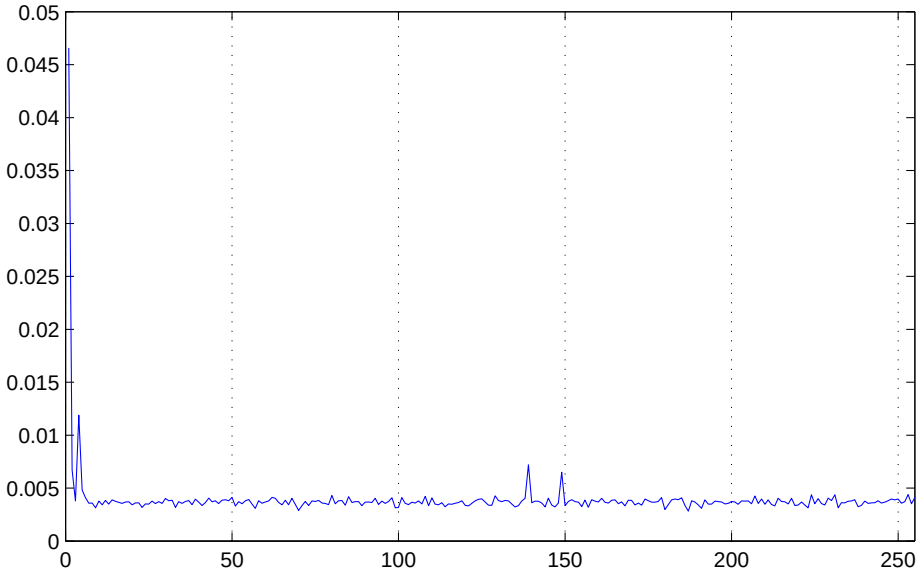

Figura 3.1: Distribuzione di frequenza del traffico SSH

Nella figura 3.3 si può vedere la rappresentazione grafica della distribuzione di frequenza dei byte all'interno di una sessione HTTP. Si può notare come i byte siano concentrati nello 'spettro' appartenente ai caratteri ASCII, e la distribuzione non lasci pensare ad una distribuzione uniforme.

In figura 3.4 invece abbiamo la rappresentazione tridimensionale di un'estratto della matrice di varianza-covarianza relativamente al traffico osservato nella sessione HTTP la cui rappresentazione grafica della distribuzione di frequenza è mostrata nella figura 3.3.

Per poter tener conto anche dei valori della covarianza tra i byte dei payload, si rende necessaria una ridefinizione del profilo definito in 2.2.1. Per cui ridefiniremo il profilo come una coppia (μ, Σ), dove μ è il valore atteso dei Feature Vector, e Σ è la matrice di varianza-covarianza dei Feature Vector.

Similmente a quanto può accadere con PAYL in presenza di valori di varianze nulle, che darebbero origine ad errori di divisione per zero nel calcolo delle distanze, ci troviamo costretti ad introdurre una perturbazione nella matrice di varianza-covarianza

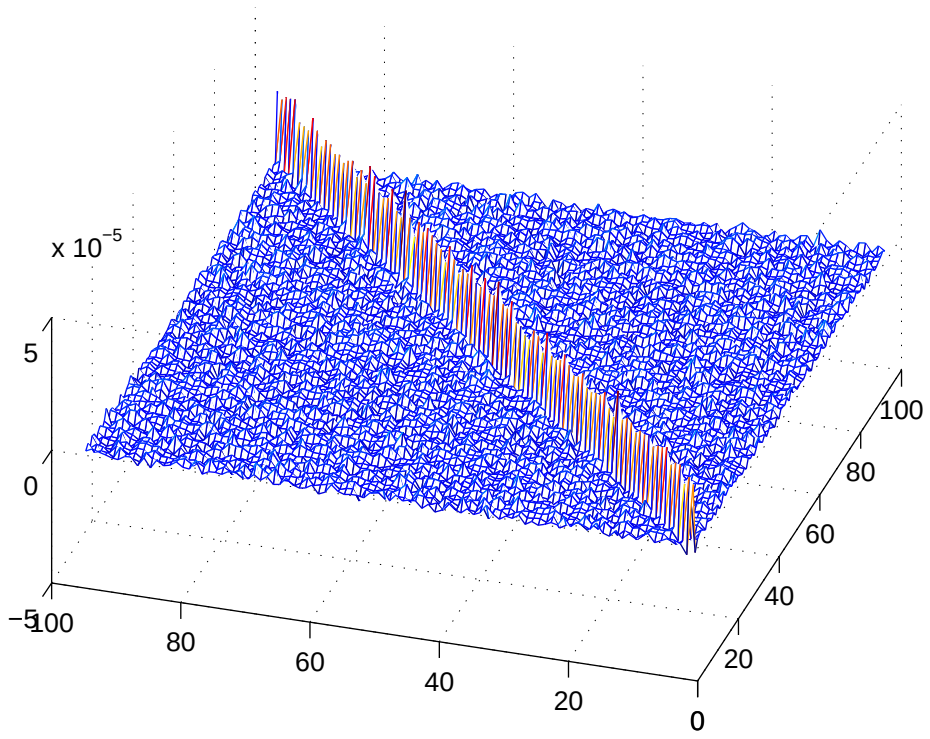

Figura 3.2: Matrice di varianza covarianza traffico SSH

per scongiurare la singolarità della matrice stessa e per permetterne l'inversione fondamentale per il calcolo della distanza di Mahalanobis.

Tale perturbazione avrà la forma di una matrice diagonale, i cui elementi possono essere considerati alla pari dello smoothing factor in PAYL. Similmente allo smoothing factor, la necessità di perturbare la matrice di varianza-covarianza diminuisce con l'aumentare dei campioni utilizzati per definire il profilo del traffico lecito.

Per ottenere il valore da utilizzare come fattore di arrotondamento si rende necessaria una calibrazione per stabilire il rapporto tra falsi positivi e detection rate.

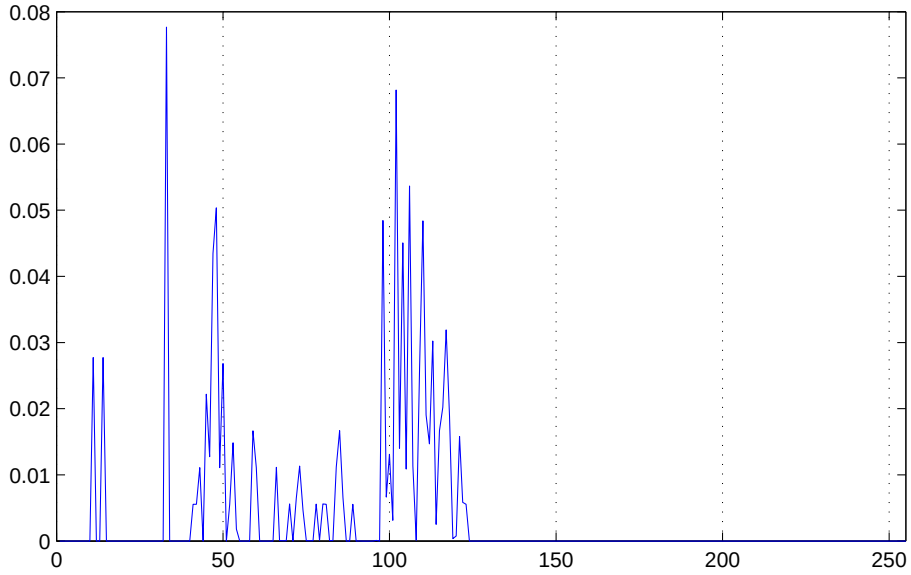

Figura 3.3: Distribuzione di frequenza del traffico HTTP

3.2 Considerazioni sulla lunghezza del payload

PAYL, sebbene basi la sua capacità di rilevamento delle anomalie sul contenuto del payload delle connessioni, collocandosi così al livello applicazione della pila ISO/OSI, per quanto riguarda la classificazione dei campioni osservati è Packet based. Si occupa infatti di recuperare i dati dei campioni prelevandoli dai livelli più bassi della pila ISO/OSI, in cui non è presente il concetto di sessione o di connessione. PAYL provvede da sé a decodificare i dati dei vari livelli della pila estraendo così il contenuto delle connessioni o delle sessioni separate in diversi pacchetti dati. Non si occupa di ricostruire le connessioni TCP e le sessioni UDP. Per questi motivi vede il traffico suddiviso in pacchetti la cui dimensione massima è imposta dai livelli più bassi della pila ISO/OSI.

La natura packet based di PAYL può in alcuni casi manifestare un certo grado di corruzione dei profili del traffico generati. Ricordiamo che l'assunzione alla base del funzionamento di PAYL è costituita dal fatto che i pacchetti di piccole dimensioni

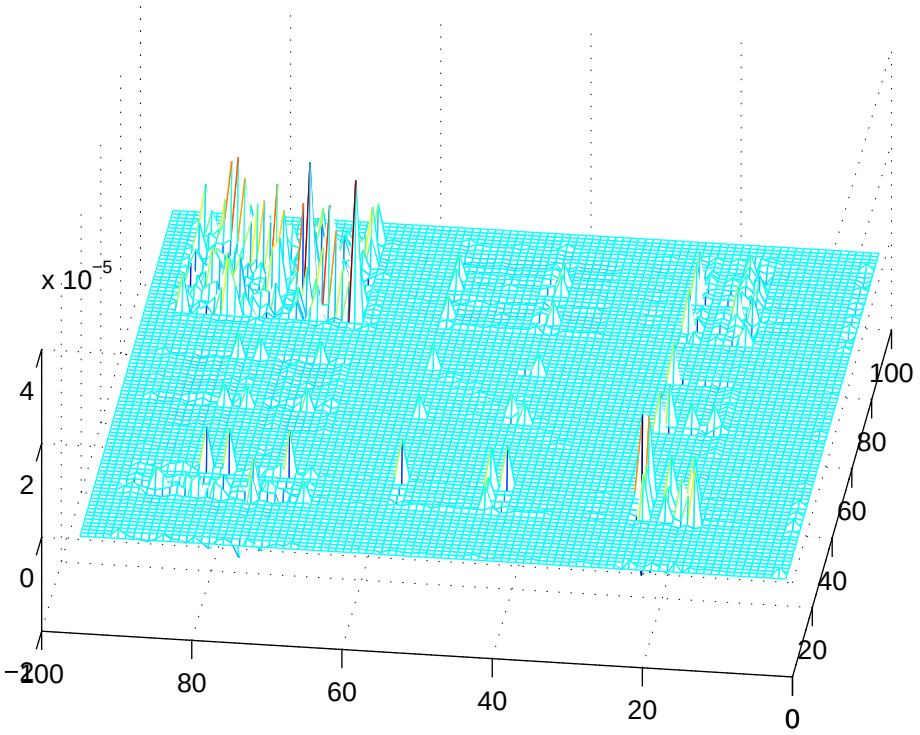

Figura 3.4: Matrice di varianza covarianza traffico HTTP

rappresentino presumibilmente un'insieme di dati di caratteristiche diverse, in termini di distribuzione di frequenza e varianza, da pacchetti di dimensione che si avvicini al limite massimo, che per una rete ethernet è di 1460.

Quest'assunzione, è generalmente falsa, dal momento che si possono presentare pacchetti di piccole dimensioni anche se appartenenti a livello applicazione ad un flusso di dati superiore al limite di 1460 byte.

Un po' più formalmente: considerando a livello applicazione un flusso dati di lunghezza n, e considerando la lunghezza massima m di un payload imposta dai livelli inferiori della pila ISO/OSI, quello che si ottiene è una quantità $i = \lfloor \frac{n}{m} \rfloor$ di campioni di lunghezza m che andranno a contribuire al profilo del traffico della stessa lunghezza, in più si avrà un campione di lunghezza $j = n \bmod m$ con le stesse caratteristiche in

termini di distribuzione di frequenza, degli *i* pacchetti assegnati al profilo relativo alla lunghezza *m*. Quest'ultimo campione andrà a contribuire nella costruzione del profilo relativo alla lunghezza *j* dei payload. Il difetto di classificazione è esemplificato nella figura 3.5.

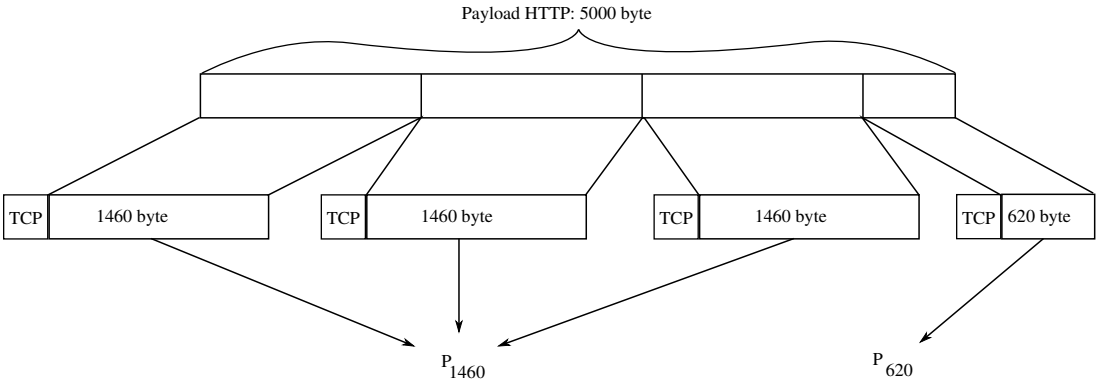

Figura 3.5: Difetto di classificazione

Questo difetto di classificazione si riperquote sui profili del traffico lecito generati, dando luogo in generale a profili più imprecisi nella fase di training, mentre nella fase di detection, essendo questo tipo di pacchetti conformi ad un profilo diverso da quello di cui si va a calcolare la distanza di Mahalanobis, possono causare un numero maggiore di falsi positivi.

Capitolo 4

Implementazione di PAYL e varianti

In questo capitolo vengono descritti i componenti principali rilevati nell'architettura di PAYL, vengono descritte le proprietà di tali componenti e le loro funzioni. In particolare si analizzerà nel dettaglio la struttura dei componenti critici di cui si fornirà un'implementazione modificata del prototipo originale.

4.1 Componenti di PAYL

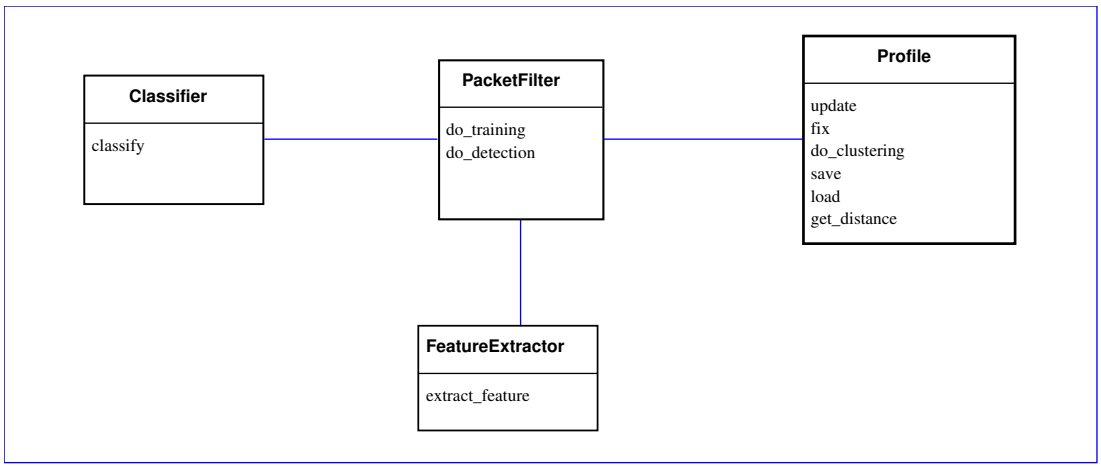

Figura 4.1: Componenti principali di PAYL

In PAYL si sono individuati quattro componenti principali:

- **PacketFilter:** Il PacketFilter è il componente che si occupa di interfacciarsi coi livelli più bassi dello stack TCP/IP, occupandosi delle impostazioni delle interfacce di rete per poter leggere i dati in transito nella rete stessa. Il packet filter è in grado di decodificare gli header dei livelli datalink, rete e trasporto estraendone dati relativi agli indirizzi IP ed alle porte TCP e UDP sorgente e destinazione oltre ad essere capace di isolare il payload del livello applicazione. Il PacketFilter inoltre è il componente che si occupa di controllare la fase di training e la fase di detection interfacciandosi con gli altri componenti.

- **Classifier:** Il classifier è il componente che si occupa della classificazione dei payload in ingresso in base alla loro dimensione ed alla loro collocazione nel flusso dati a livello applicazione. Per quanto riguarda il clone di PAYL il classifier ha come unico fattore discriminante la lunghezza del payload osservato, e per questo motivo può essere omesso. Per quanto riguarda invece le modifiche proposte in questa tesi, il classificatore si rende necessario allo scopo di identificare i pacchetti dati di piccole dimensioni appartenenti a flussi al livello applicazione di dimensioni maggiori.

- **FeatureExtractor:** Il feature extractor è il componente che si occupa di estrarre dai payload osservati, i dati costituenti i feature vector, necessari alla costruzione dei profili del traffico lecito relativamente alla fase di training, e di prepararli al confronto con i profili del traffico lecito nella fase di detection.

- **Profile:** Il componente Profile si occupa della gestione dei profili del traffico. Esso deve essere in grado di effettuare operazioni come il learning incrementale, il clustering ed il calcolo della distanza di Mahalanobis.

Nella figura 4.1 sono rappresentati i componenti principali di PAYL.

4.2 PacketFilter

Per la realizzazione del componente PacketFilter sono state utilizzate le librerie Pcap[5], in grado di fornire un'astrazione semplice da utilizzare e multipiattaforma delle interfacce del sistema operativo altrimenti necessarie per la cattura dei pacchetti in ingresso, oltre ad essere in grado di interpretare il formato dei file costituenti il dataset di riferimento su cui sono stati effettuati i test.

4.2.1 Fase di Training

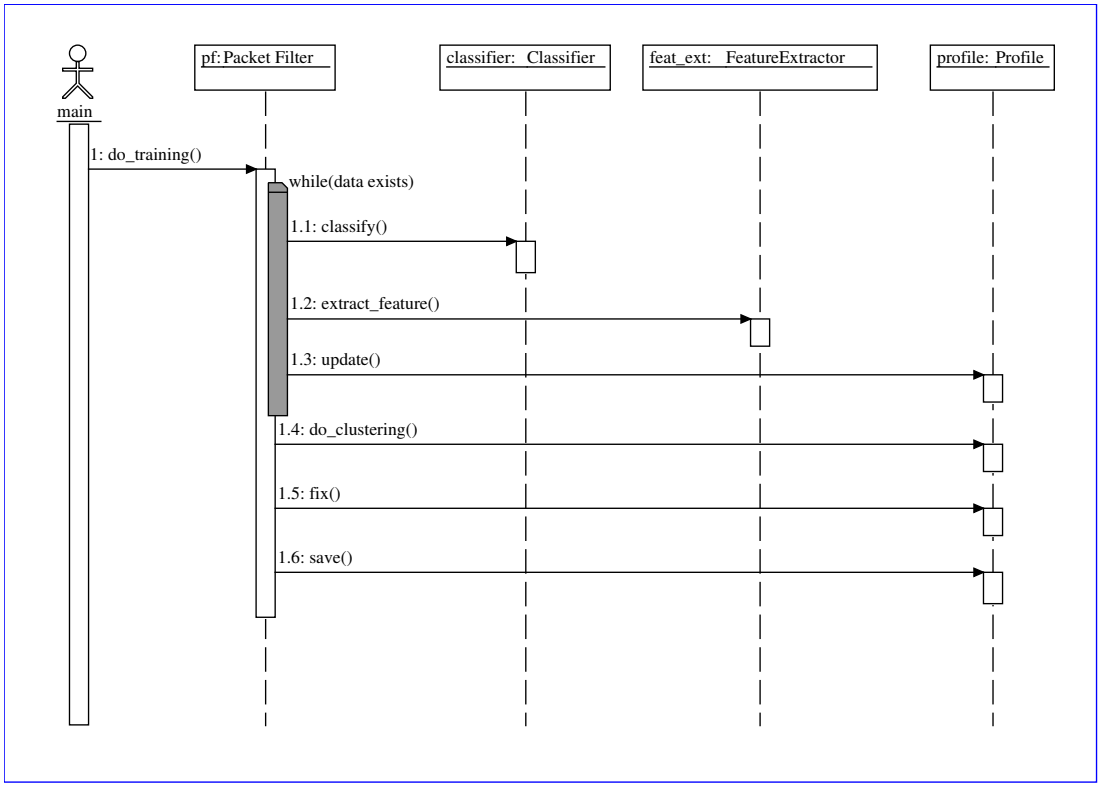

Figura 4.2: Diagramma di sequenza della fase di training

Durante la fase di training, vengono catturati dal PacketFilter i pacchetti in transito nella rete completi di header dei protocolli dei livelli datalink, network e transport. Per ogni pacchetto catturato, il PacketFilter provvede a togliere l'intestazione del

protocollo di livello datalink dal pacchetto ed a passarlo così ripulito al componente Classifier tramite la funzione classify. Il PacketFilter riceve dal Classifier l'indice del profilo di traffico a cui i dati relativi al pacchetto osservato devono essere assegnati. A questo punto il PacketFilter ripulisce il pacchetto ulteriormente privandolo dell'intestazione relativa al livello network ed al livello trasporto ottenendo il payload del livello applicazione. Il Packet filter passa il payload così ottenuto al componente FeatureExtractor che si occupa di calcolarne il feature vector relativo tramite la funzione extract_feature. Il PacketFilter ottiene il feature vector dal FeatureExtractor come risultato della funzione extract_feature. Il PacketFilter a questo punto è in grado di comunicare al componente Profile il feature vector e la relativa 'classe' di appartenenza attraverso la funzione update, che si occupa di aggiornare il profilo con i nuovi dati in ingresso.

Terminati i pacchetti da catturare, il PacketFilter comunica al componente Profile mediante la funzione do_clustering di effettuare il clustering, e tramite la funzione fix di effettuare le operazioni conclusive per ottenere il completamento dei profili.

Terminate queste due ultime fasi, il PacketFilter comunica al componente Profile di salvare il profilo del traffico lecito su file, per poterlo utilizzare nella successiva fase di detection.

Il funzionamento della fase di training è rappresentato nel diagramma di sequenza in figura 4.2.

4.2.2 Fase di Detection

Durante la fase di detection, il PacketFilter comunica al componente Profile di caricare il profilo del traffico lecito da file. Il PacketFilter inizia a catturare i pacchetti in transito nella rete completi di header dei protocolli dei livelli datalink, network e transport. Per ogni pacchetto catturato, il PacketFilter provvede a togliere l'intestazione del protocollo di livello datalink dal pacchetto ed a passarlo così ripulito al componente Classifier tramite la funzione classify. Il PacketFilter riceve dal Classifier l'indice del profilo di traffico a cui i dati relativi al pacchetto osservato devono essere assegnati. A questo punto il PacketFilter ripulisce il pacchetto ulteriormente

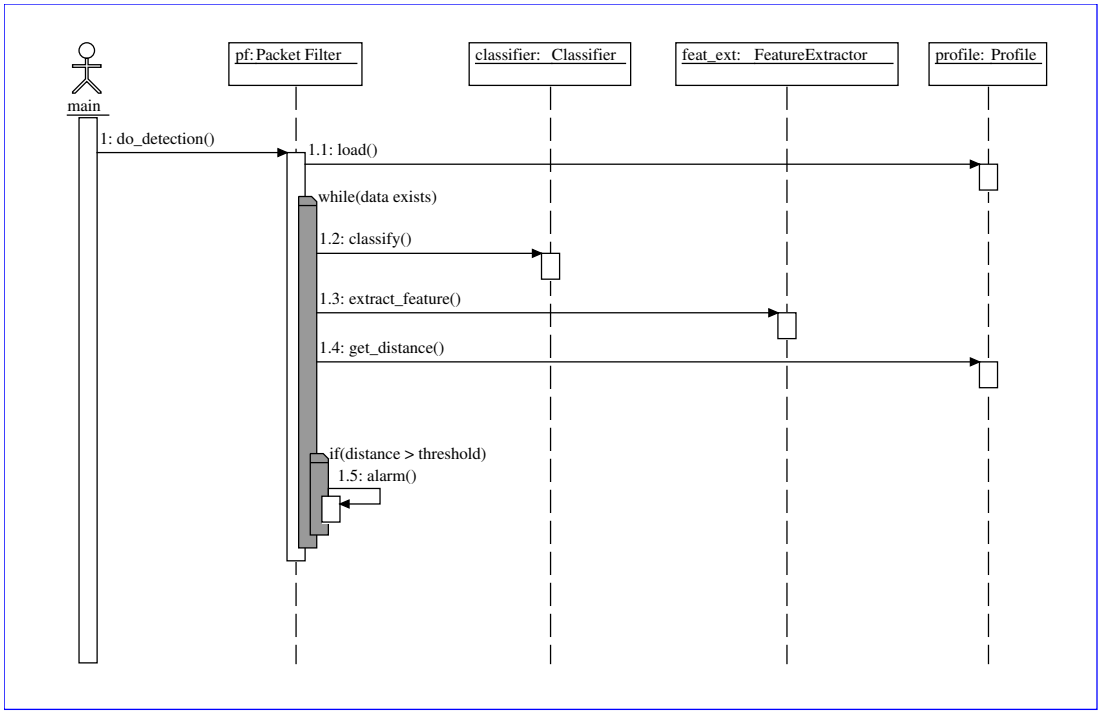

Figura 4.3: Diagramma di sequenza della fase di detection

privandolo dell'intestazione relativa al livello network ed al livello trasporto ottenendo il payload del livello applicazione. Il Packet filter passa il payload così ottenuto al componente FeatureExtractor che si occupa di calcolarne il feature vector relativo tramite la funzione extract_feature. Il PacketFilter ottiene il feature vector dal FeatureExtractor come risultato della funzione extract_feature. Il PacketFilter a questo punto è in grado di comunicare al componente Profile il feature vector e la relativa 'classe' di appartenenza attraverso la funzione get_distance, che si occupa di calcolare la distanza di Mahalanobis del feature vector dal relativo profilo di appartenenza. Il PacketFilter confronta la distanza di Mahalanobis così ottenuta con la soglia di allarme, e nel caso tale distanza superi la soglia, esegue la procedura di allarme.

Il funzionamento della fase di detection è rappresentato dal diagramma di sequenza in figura 4.3.

4.3 Classifier

Il Classifier si occupa della classificazione dei payload in transito. Data una lunghezza n del payload da classificare, un valore *MAX* rappresentante la dimensione massima di un payload, e data una struttura che chiamiamo *sockpair* che ci permette di identificare in modo univoco una connessione ed una sua precisa direzione:

```
typedef struct
{
   u_int32_t s_addr; /* IP sorgente        */
   u_int32_t d_addr; /* IP destinazione    */
   u_int16_t s_port; /* Porta sorgente     */
   u_int16_t d_port; /* Porta destinazione */
} sockpair;
```

possiamo schematizzare il funzionamento del Classifier come mostrato nell'algoritmo 1.

Algorithm 1 COMMON:classify

```
if n = MAX then
   salva sockpair nella hash table
   return n
else
   if sockpair presente nella hash table then
      rimuovi sockpair dalla hash table
      return MAX
   else
      return n
   end if
end if
```

Il Classifier per tener traccia delle varie connessioni, memorizza le variabili di tipo sockpair in una hash table.

4.4 FeatureExtractor

Il feature extractor si occupa di estrarre il feature vector dal payload in ingresso. Il payload è un vettore di n byte mentre il feature vector è un vettore di 2^k byte dove k è

la quantità di bit meno significativi di cui si vuole tener conto. Per il clone di PAYL, il valore k è fisso ad 8. Per le versioni modificate il valore varia da 4 ad 8. MASK invece è la maschera di bit necessaria per poter tener conto dei k bit meno significativi di ogni byte del payload. Il feature vector è inizializzato a 0. Fatte queste premesse, il funzionamento del FeatureExtractor può essere schematizzato come descritto nell'algoritmo 2.

Algorithm 2 COMMON:extract_feature

```
for i = 1 to n do
    Feature[Payload[i]&MASK]++
end for
for i = 1 to 2^k do
    Feature[i] /= n
end for
return Feature
```

4.5 Profile

Descriviamo brevemente gli algoritmi per generare i profili del traffico lecito. Da ora in avanti, per semplificare le notazioni faremo riferimento ad un profilo relativo ad un solo servizio e ad una generica lunghezza di payload i. Denomineremo questo profilo $P[i]$ per sottolineare la natura della variabile P costituita nella nostra implementazione da un vettore di puntatori a strutture contenenti i profili relativi ai payload di lunghezza i. Il profilo del traffico in PAYL, come abbiamo specificato nella sezione 2.2, è costituito da una coppia (μ, σ), mentre in PAYL-B (la nostra versione modificata), come specificato nella sezione 3.1, è costituito da una coppia (μ, Σ). Pertanto nella descrizione degli algoritmi faremo riferimento a μ, σ e Σ per definire le variabili che alla fine del processo conterranno i valori costituenti i profili per le rispettive implementazioni. Nella nostra implementazione in C il profilo del traffico lecito assumerà la forma seguente:

```
tupedef struct profile *PROFILE[1460];

struct profile
```

```
{
   float mu[DIM];
#ifdef PAYL
   float Sigma[DIM];
#else
   float Sigma[DIM*DIM];
#endif
   ...
};
```

Dove DIM è la dimensione del vettore delle varianze o della matrice di varianza-covarianza a seconda dei casi, mentre PROFILE rappresenta il tipo di variabile che conterrà i singoli profili definiti nella struttura "struct profile".

Sia per il clone di PAYL che per la versione modificata PAYL-B, si giunge alla costruzione dei profili del traffico con la sequenza di funzioni: update, do_clustering, fix.

Procediamo ora a descrivere tale sequenza per entrambe le versioni.

4.5.1 Clone di PAYL

Nella funzione update, le variabili μ e σ contengono inizialmente i valori $E[X]$ ed $E[X^2]$ rappresentanti rispettivamente il valore atteso ed il momento centrale di ordine due degli n FeatureVector finora osservati. La variabile F contiene invece il nuovo FeatureVector appena osservato. Fatte queste premesse possiamo schematizzare l'algoritmo di update relativo al generico profilo $P[i]$ come nell'algoritmo 3.

Algorithm 3 PAYL:update

$\mu \Leftarrow \mu + \frac{F-\mu}{n+1}$
$\sigma \Leftarrow \sigma + \frac{F^2-\sigma}{n+1}$
$n \Leftarrow n+1$

La seconda fase è il clustering. In questa fase si effettua la fusione dei profili $P[i]$ e $P[i+1]$ se la distanza di manhattan tra $P[i].\mu$ e $P[i+1].\mu$ è al di sotto della soglia *Threshold*. Per effettuare la fusione ricordiamo che se s_i ed s_{i+1} sono le numerosità di X_i ed X_{i+1}, i cui valori attesi sono rispettivamente $E[X_i]$ ed $E[X_{i+1}]$, il valore atteso dell'unione degli insiemi X_i ed X_{i+1} si può ricavare così: $E[X_i \cup X_{i+1}] =$

$\frac{s_{i+1}E[X_i]+s_iE[X_{i+1}]}{s_i+s_{i+1}}$. Fatte queste premesse possiamo descrivere l'algoritmo di clustering come rappresentato nell'algoritmo 4.

Algorithm 4 PAYL:do_clustering

```
repeat
  cycle ⇐ 0
  for i ⇐ 1 to 1459 do
    if ManhattanDistance(P[i].μ, P[i+1].μ) < Threshold then
      P[i].μ ⇐ (s_{i+1}P[i].μ + s_iP[i+1].μ) / (s_i + s_{i+1})
      P[i].σ ⇐ (s_{i+1}P[i].σ + s_iP[i+1].σ) / (s_i + s_{i+1})
      s_i ⇐ s_i + s_{i+1}
      free(P[i+1])
      P[i+1] ⇐ P[i]
      cycle ⇐ 1
    end if
  end for
until cycle = 0
```

L'ultima fase per la costruzione del profilo è costituita dalla funzione fix, in cui si effettua il calcolo finale della deviazione standard basandosi sulla proprietà della varianza: $Var[X] = E[X^2] - E[X]^2$. Al termine della fase di update e di clustering, le variabili μ e σ contengono i valori $E[X]$ ed $E[X^2]$, per cui la fase di fix si riduce a quella descritta nell'algoritmo 5.

Algorithm 5 PAYL:fix

```
for i ⇐ 1 to 1460 do
  if P[i].fixed ≠ 1 then
    P[i].σ ⇐ P[i].σ − P[i].μ^2
    P[i].σ ⇐ √P[i].σ
    P[i].fixed ⇐ 1
  end if
end for
```

4.5.2 PAYL-B

Nella funzione update, le variabili μ e Σ contengono inizialmente i valori $E[X]$ ed $E[XX^T]$ rappresentanti rispettivamente i valori attesi degli n FeatureVector finora os-

servati F_n e delle n matrici $F_n F_n^T$. La variabile F contiene invece il nuovo FeatureVector appena osservato. Fatte queste premesse possiamo schematizzare l'algoritmo di update come nell'algoritmo 6.

Per l'aggiornamento del valore $E[XX^T]$ in Σ è stata usata la funzione *cblas_sgemm* delle librerie *Atlas*[8].

Algorithm 6 PAYL-B:update

$\mu \Leftarrow \mu + \frac{F-\mu}{n+1}$
$\Sigma \Leftarrow \Sigma + \frac{FF^T-\Sigma}{n+1}$
$n \Leftarrow n+1$

La seconda fase è il clustering. In questa fase si effettua la fusione dei profili $P[i]$ e $P[i+1]$ se la distanza di manhattan tra $P[i].\mu$ e $P[i+1].\mu$ è al di sotto della soglia *Threshold*. Per effettuare la fusione ricordiamo che se s_i ed s_{i+1} sono le numerosità di X_i ed X_{i+1}, i cui valori attesi sono rispettivamente $E[X_i]$ ed $E[X_{i+1}]$, il valore atteso dell'unione degli insiemi X_i ed X_{i+1} si può ricavare così: $E[X_i \cup X_{i+1}] = \frac{s_{i+1}E[X_i]+s_iE[X_{i+1}]}{s_i+s_{i+1}}$. Fatte queste premesse possiamo descrivere l'algoritmo di clustering come rappresentato nell'algoritmo 7.

Algorithm 7 PAYL-B:do_clustering

repeat
  $cycle \Leftarrow 0$
  for $i \Leftarrow 1$ to 1459 **do**
    if $ManhattanDistance(P[i].\mu, P[i+1].\mu) < Threshold$ **then**
      $P[i].\mu \Leftarrow \frac{s_{i+1}P[i].\mu+s_iP[i+1].\mu}{s_i+s_{i+1}}$
      $P[i].\Sigma \Leftarrow \frac{s_{i+1}P[i].\Sigma+s_iP[i+1].\Sigma}{s_i+s_{i+1}}$
      $s_i \Leftarrow s_i + s_{i+1}$
      free($P[i+1]$)
      $P[i+1] \Leftarrow P[i]$
      $cycle \Leftarrow 1$
    end if
  end for
until $cycle = 0$

L'ultima fase per la costruzione del profilo è costituita dalla funzione fix, in cui si effettua il calcolo finale della matrice di varianza-covarianza basandosi sulla proprietà:

$\Sigma = E[XX^T] - \mu\mu^T$. Al termine della fase di update e di clustering, le variabili μ e Σ contengono i valori $E[X]$ ed $E[XX^T]$, per cui la fase di fix si riduce a quella descritta nell'algoritmo 8.

Calcolata la matrice di varianza-covarianza il profilo del traffico risulta completo, tuttavia per ridurre il carico di lavoro nella successiva fase di detection, nella funzione fix viene effettuata l'inversione della matrice Σ.

Per il calcolo finale della matrice di varianza-covarianza è stata utilizzata la funzione *cblas_sgemm*, mentre per il calcolo della sua inversa, sono state utilizzate in sequenza le funzioni *clapack_sgetrf* e *clapack_sgetri* delle librerie *Atlas*[8] che effettuano rispettivamente la fattorizzazione *LU* della matrice e l'inversione della matrice *LU* risultante.

Algorithm 8 PAYL-B:fix

for $i \Leftarrow 1$ to 1460 **do**
  if $P[i].fixed \neq 1$ **then**
    $P[i].\Sigma \Leftarrow P[i].\Sigma - P[i].\mu P[i].\mu^T$
    $P[i].\Sigma \Leftarrow P[i].\Sigma^{-1}$
    $P[i].fixed \Leftarrow 1$
  end if
end for

4.5.3 Calcolo distanza

Per il calcolo della distanza con PAYL di un FeatureVector relativo ad un payload appena osservato faremo uso dell'equazione (2.5). La funzione get_distance può essere schematizzata come nell'algoritmo 9.

Algorithm 9 PAYL:get_distance

$d \Leftarrow 0$
for $i \Leftarrow 0$ to 255 **do**
  $d \Leftarrow d + \frac{F[i]-\mu[i]}{\sigma[i]}$
end for
return d

Per calcolare la distanza in PAYL-B utilizzeremo la formula della distanza di Mahalanobis definita dall'equazione (2.3). Ricordiamo che al termine della sequenza di funzioni necessaria per la generazione del profili si ottiene la coppia μ e Σ che contengono rispettivamente il valore atteso dei FeatureVector e l'**inversa** della matrice di varianza covarianza. Fatte queste premesse possiamo schematizzare l'algoritmo del calcolo della distanza come nell'algoritmo 10.

Algorithm 10 PAYL-B:get_distance

for $i \Leftarrow 0$ to 255 **do**
  $X[i] \Leftarrow F[i] - \mu[i]$
end for
$d \Leftarrow X^T \Sigma X$
$d \Leftarrow \sqrt{d}$
return d

Capitolo 5

Ambiente di test e risultati sperimentali

In questo capitolo descriveremo l'ambiente in cui sono stati effettuati i test su PAYL e sulle sue varianti, verranno inoltre forniti i dettagli relativi allo svolgimento dei test e le motivazioni sulle scelte effettuate. Infine verranno mostrati i risultati dei test con le relative considerazioni sul loro esito.

5.1 DARPA dataset 1999

Il DARPA IDS dataset del 1999 è stato realizzato ai laboratori MIT Lincoln Labs per la valutazione degli Intrusion Detection System. Si tratta in sostanza di una raccolta di dati sotto forma di dump del traffico di rete e dei file system delle macchine 'vittima' coinvolte, oltre a vari logfile nel formato di *syslog* e in formato *BSM*[1]. Il dataset è stato realizzato sulla base di una rete simulata, composta da diciotto computer reali e da altri computer virtuali realizzati tramite emulazione. La topologia della rete di test è rappresentata in figura 5.1.

Il DARPA dataset 1999 è stato criticato pesantemente per le modalità con cui sono state realizzate le simulazioni e per non essere propriamente rappresentativo di una rete informatica reale. In [3] viene fatta un'analisi dettagliata dei problemi del dataset. In particolare è stata evidenziata la possibilità di progettare ad-hoc un NIDS in gra-

[1] Basic Security Module

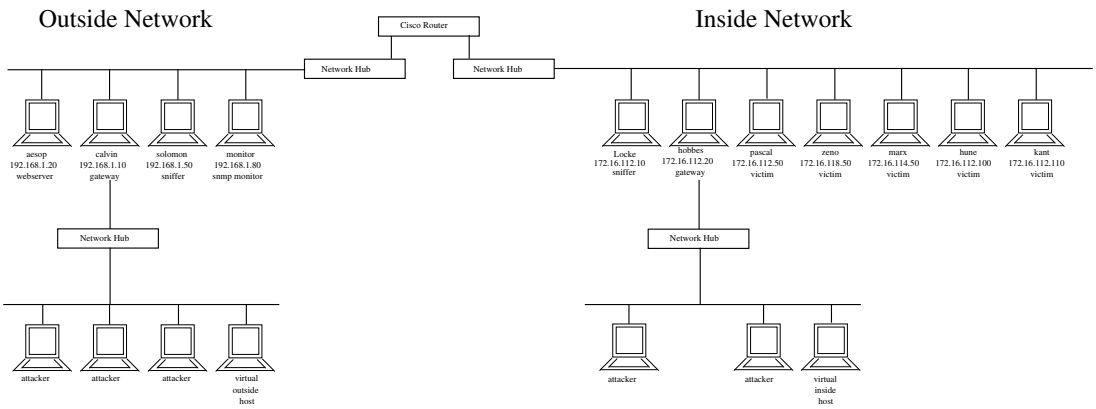

Figura 5.1: Topologia della rete di test

do di ottenere un elevata detection rate con un false positive rate irrisorio andando ad osservare un byte particolare dell'indirizzo IP sorgente di ogni connessione. Il dataset inoltre presenta alcuni campi degli header IP come il TTL[2] limitati a nove valori sui 256 possibili in una rete reale. Simili considerazioni sono state fatte anche per la quantità limitata di opzioni contenute negli header TCP e per la totale assenza di frammentazione del traffico. Sebbene presenti molti problemi, il DARPA dataset 1999 rappresenta ad oggi la raccolta di dati più completa e disponibile in rete per effettuare i test sulle capacità di rilevamento degli IDS.

Questo dataset è stato usato in diversi lavori di ricerca, in particolare è il dataset di riferimento utilizzato dai realizzatori di PAYL per la valutazione del loro progetto. PAYL non basa la sua capacità di detection sui campi degli header IP e TCP, pertanto non risente delle particolarità del dataset DARPA che sono state criticate, eccezion fatta per una particolarità riguardante la dimensione massima dei payload che definiremo al termine del capitolo.

5.1.1 Contenuto del dataset

Il dataset è costituito dal dump del traffico di cinque settimane. Le prime tre settimane, di cui due prive di attacchi, contengono il traffico destinato alla fase di training dei

[2]time to live

vari IDS da analizzare, mentre le ultime due settimane contengono diverse istanze di attacchi di vario tipo miste a traffico normale. Gli attacchi sono catalogati in base al tipo di vulnerabilità sfruttata ed ai vari servizi coinvolti. Il dump del traffico è stato realizzato in formato tcpdump e sono stati messi a disposizione, in aggiunta al dump del traffico, anche diversi log in vari formati come BSM (syscall di solaris) e dump dei filesystem su base giornaliera. Per facilitare la verifica dei vari NIDS, sono stati forniti alcuni file contenenti i dati relativi ai vari attacchi. In particolare il *master identification file* consente di localizzare le diverse istanze di ogni attacco all'interno dei dump.

Il master identification file contiene al suo interno diversi dati relativamente ad ogni attacco, come l'orario di inizio e la durata, la categoria di appartenenza, i servizi coinvolti e gli indirizzi IP delle macchine utilizzate nella simulazione.

Un'estratto del master identification file:

```
ID: 41.084031
Date: 03/29/1999
Name: ps
Category: u2r
Start_Time: 08:18:35
Duration: 00:46:05
Attacker: 209.154.098.104
Victim: 172.016.112.050
Username: haraldl
Ports:
   At_Attacker: 80{1}, 6000{2}
   At_Victim: 23{3}
```

In questo estratto del master identification file, viene indicato un identificativo dell'istanza dell'attacco composto da otto cifre. Le prime due cifre indicano la settimana ed il giorno in cui tale istanza è presente, le altre sei invece indicano l'orario approssimato in cui si ha un'evidenza dell'attacco. L'ID 41.084031 indica un'istanza presente nei dump del primo giorno della quarta settimana e si ha un'evidenza dell'attacco alle 8:40:31. L'attacco però ha inizio alle 8:18:35 e dura 45 minuti e 5 secondi. Relativamente ad ogni singola istanza di ogni attacco, vengono indicati anche gli indirizzi IP coinvolti, quello dell'attaccante e quelli delle vittime. Vengono indicate inoltre le porte TCP o UDP coinvolte nell'attacco e la quantità di connessioni per ogni porta.

Nel nostro esempio possiamo notare che l'attaccante ha effettuato 3 connessioni verso la porta 23 (telnet) e da lì ha effettuato una connessione verso la porta 80 (http) e due connessioni verso la porta 6000 (x11) dell'attaccante. Vengono inoltre indicate altre informazioni come il nome login utilizzato dall'attaccante per collegarsi al servizio remoto. In questo caso le tre connessioni alla porta telnet sono state fatte usando l'utente haraldl.

5.1.2 Filtraggio preliminare

Sulla base dell'articolo originale di PAYL, è stato effettuato un filtraggio del traffico contenuto nel dataset per riprodurre il più fedelmente possibile i risultati da loro ottenuti e per poter dare quindi una misura più attendibile dei miglioramenti ottenuti con le modifiche al modello proposte.

Per la sperimentazione di PAYL e delle sue varianti si è tenuto conto del traffico interno del dataset, ossia del traffico osservato tra il router e le vittime appartenenti all'*Inside Network* rappresentata in figura 5.1. Oltre ad aver considerato il solo traffico interno, è stato fatto un ulteriore filtraggio dei dati limitandolo alla sottorete 172.16.0.0/16 contenente la maggior parte delle vittime ed all'intervallo di porte 0-1023, contenente i servizi maggiormente diffusi in Internet.

Sulla base dei filtraggi preliminari e sul master identification file, abbiamo preparato una lista delle istanze di attacchi presenti nelle ultime due settimane di traffico limitatamente alla sottorete ed ai servizi coinvolti.

Un'estratto della lista:

```
...
43.084000_netcat 172.16.112.100 25
43.100000_ppmacro 172.16.112.100 25 80
43.101313_processtable 172.16.113.50 25
43.111111_warezmaster 172.16.112.50 21
...
```

In seguito ci riferiremo a questa lista nominandola *datafile*.

Il datafile contiene in ogni riga l'ID di ogni attacco seguito dal nome, l'IP della vittima e le porte TCP coinvolte nell'attacco. L'ID concatenato col nome servirà in fase di detection ad indicare il file contenente l'istanza dell'attacco, mentre le coppie

IP-porta, per ogni porta elencata, serviranno ad identificare il file contenente il traffico lecito per quel determinato IP e quella specifica porta sia in fase di training che nella verifica dei falsi positivi. I dettagli relativi a queste fasi verranno discussi nelle sezioni successive.

5.2 Fase di Training

Per la fase di training sono stati usati i dump del traffico privo di attacchi nel dataset relativo alla prima ed alla terza settimana.

Sulla base del datafile generato in precedenza, per velocizzare questa fase, il dump è stato suddiviso in più file ognuno contenente il dump del traffico relativamente a:

- direzione del traffico relativa alla macchina vittima.
- indirizzo IP della macchina vittima.
- porta TCP del servizio coinvolto nell'attacco.

Portiamo come esempio l'istanza di un attacco:

```
43.100000_ppmacro 172.16.112.100 25 80
```

Per effettuare il training necessario al rilevamento dell'attacco *ppmacro*, verranno separati dal dump privo di attacchi, quattro flussi di traffico e messi nei seguenti file:

- training_src_172.16.112.100_25
- training_dst_172.16.112.100_25
- training_src_172.16.112.100_80
- training_dst_172.16.112.100_80

ottenendo nel primo file il traffico uscente dalla porta 25 e dall' IP 172.16.112.100, nel secondo il traffico diretto a quell'IP e quella porta, nel terzo e nel quarto otterremo la stessa cosa ma per la porta 80.

Questa separazione in file diversi del traffico lecito ha permesso di evidenziare una totale assenza di dati relativamente a 32 file sui 70 ottenuti. In particolare nella fase di training sono assenti i dati relativi ai servizi 53/TCP (DNS), 143/TCP (IMAP) e 513/TCP (login). Ciò rende impossibile effettuare la generazione dei profili del traffico lecito relativamente a tali servizi.

I restanti 38 file sono costituiti in media da 17 megabyte di traffico e tra questi, la quasi totalità dei file relativi al servizio 21/TCP (FTP), si trova al di sotto di 200 kilobyte il che si traduce in una quantità insufficiente di campioni necessaria alla generazione dei profili del traffico lecito.

Ottenuta la separazione dei file in questo modo, per ogni file, per PAYL e le sue varianti, è stato generato un profilo del traffico in esso contenuto. Ogni profilo è stato salvato in un file separato.

Facendo riferimento al primo dei quattro file ottenuti per il training relativo al rilevamento dell'attacco *ppmacro*, al termine della fase di training otterremo i seguenti file:

- payl_src_172.16.112.100_25
- payl-b_src_172.16.112.100_25
- payl-c_src_172.16.112.100_25

questi file conterranno rispettivamente i modelli del clone di PAYL, della versione modificata col classificatore disattivato (PAYL-B) e della versione modificata col classificatore attivo (PAYL-C).

5.3 Fase di Detection

Per la fase di detection, sono stati usati i dump della quarta e quinta settimana del dataset, contenenti diverse istanze di attacchi elencate nel datafile. Le istanze stesse di ogni attacco sono state separate dal resto del dump in cui erano contenute e sono state salvate in file il cui nome corrisponde all'identificativo dell'istanza contenuto nel datafile.

Riportando l'esempio dell'attacco *ppmacro*:

```
43.100000_ppmacro 172.16.112.100 25 80
```

abbiamo separato dal dump del terzo giorno della quarta settimana l'istanza dell'attacco ed abbiamo provveduto a salvarla nel file 43.100000_ppmacro.

L'operazione di separazione degli attacchi è stata effettuata con l'ausilio di *tcpdump* ed *ethereal* ed è stata ripetuta per ogni istanza di attacco presente nel datafile.

Al termine di questo filtraggio, sono stati ottenuti un'insieme di file contenenti tutte le istanze degli attacchi presenti nel dump della quarta e della quinta settimana, inoltre, relativamente a tali settimane di traffico è stato ottenuto un file contenente unicamente il traffico lecito.

Il dump della quarta e quinta settimana, privato degli attacchi, è stato separato similmente a quanto è stato effettuato nella preparazione della fase di training. In base al datafile, tale dump è stato separato in diversi file.

Portiamo come esempio ancora una volta l'istanza dell'attacco *ppmacro*, in base ai dati relativi a quell'attacco, verranno separati dal dump privato degli attacchi i seguenti file:

- testing_src_172.16.112.100_25
- testing_dst_172.16.112.100_25
- testing_src_172.16.112.100_80
- testing_dst_172.16.112.100_80

ottenendo nel primo file il traffico uscente dalla porta 25 e dall' IP 172.16.112.100, nel secondo il traffico diretto a quell'IP e quella porta, nel terzo e nel quarto otterremo la stessa cosa ma per la porta 80.

Questi dump verranno utilizzati per il conteggio dei falsi positivi relativamente all'attacco *ppmacro*.

5.3.1 Calibrazione dei parametri

Sia per il clone di PAYL che per le versioni modificate, è stata effettuata la calibrazione dei parametri come lo *smoothing factor* e la *manhattan distance*. Ricordiamo che lo smoothing factor è in PAYL il valore che viene aggiunto alle varianze memorizzate nei profili al fine di evitare divisioni per zero, mentre nelle versioni modificate tale valore viene aggiunto alla diagonale della matrice di varianza-covarianza al fine di scongiurare la singolarità della matrice stessa permettendone l'inversione. La manhattan distance invece è il valore di distanza che guida l'operazione di clustering nella fase di training.

Lo smoothing factor inoltre, incide sulla quantità di falsi positivi e sulla capacità di detection ed il suo valore andrebbe diminuito con l'aumentare della quantità di campioni utilizzati per la fase di training.

Per il dataset DARPA 1999 sono stati ottenuti i risultati migliori utilizzando in PAYL un valore dello smoothing factor pari a 0.001, valore indicato anche nell'articolo originale di PAYL, mentre nelle versioni modificate tale valore è stato stabilito a 0.00003. Per questo dataset inoltre, la distanza di manhattan che dà i risultati migliori è 0.5.

5.3.2 Avvio dei test

Terminata la fase di preparazione in cui sono stati effettuati tutti i filtraggi dei dump e la generazione dei profili del traffico separatamente per ogni tripla di indirizzo IP, porta TCP e direzione del flusso, abbiamo ottenuto tre insiemi di file costituiti rispettivamente da profili del traffico lecito per PAYL e le sue varianti, file contenenti le singole istanze di ogni attacco e file contenenti il traffico della quarta e quinta settimana del dataset privato degli attacchi.

Per ogni istanza degli attacchi elencata nel datafile abbiamo calcolato la soglia massima necessaria alla sua rilevazione per ogni porta TCP e per ogni direzione del flusso. Abbiamo in seguito utilizzato le soglie così ottenute per contare i falsi positivi avviando il detector utilizzando come traffico di test il traffico della quarta e quinta settimana relativo agli indirizzi IP e le porte coinvolte nell'attacco.

Portiamo come esempio il calcolo dei falsi positivi effettuato tramite il clone di PAYL sull'istanza dell'attacco *ppmacro*:

```
43.100000_ppmacro 172.16.112.100 25 80
```

Riepilogando, l'istanza 43.100000 dell'attacco ppmacro coinvolge l'indirizzo IP 172.16.112.100 e le porte 25 e 80; limitiamoci a considerare solamente il traffico proveniente dalla porta 25. Nella fase di training è stato ottenuto per PAYL il file contenente il profilo del traffico lecito 'payl_src_172.16.112.100_25'.

Il file testing_src_172.16.112.100_25 contiene il traffico privo di attacchi proveniente dall'IP 172.16.112.100 e dalla porta 25. Il file 43.100000_ppmacro contiene l'istanza 43.100000 dell'attacco ppmacro.

```
$ payl -d 0 -f 43.100000_ppmacro -r payl_src_172.16.112.100_25 \
      src host 172.16.112.100 and src port 25
$ 61.729393
```

Il valore 61.729393 rappresenta la distanza massima utilizzabile per rilevare l'attacco nel file 43.100000_ppmacro.

Ora utilizzeremo la soglia ottenuta per ottenere la quantità di falsi positivi.

```
$ payl -d 61.729393 -f testing_src_172.16.112.100_25 -r payl_src_172.16.112.100_25 \
      src host 172.16.112.100 and src port 25
$ 120 / 45866
```

Il risultato 120 / 45866 sta ad indicare che nel file testing_src_172.16.112.100_25 sono stati individuati 120 payload su 45866 con distanza superiore o uguale alla soglia stabilita di 61.729393, il che si traduce in una percentuale di falsi positivi pari a 0.2%.

L'operazione viene ripetuta anche per la direzione opposta del traffico ottenendo in generale un valore diverso dei falsi positivi. In tal caso abbiamo tenuto conto unicamente del risultato migliore.

Il test viene ripetuto per ogni istanza di attacco contenuta nel datafile ed i risultati sono stati separati per tipologia di servizio.

5.4 Risultati

Al termine del test abbiamo ottenuto i dati relativamente alla capacità di detection di PAYL, PAYL-B e PAYL-C per i servizi corrispondenti alle porte 21,22,23,25 e 80. Per ottenere i grafici che mostreremo nei prossimi paragrafi, abbiamo provveduto all'ordinamento delle quantità di falsi positivi rilevate per ogni istanza per mostrare più chiaramente l'andamento crescente della quantità di attacchi rilevati in funzione della quantità di falsi positivi.

In ogni grafico abbiamo rappresentato la percentuale di falsi positivi nelle ascisse, mentre nelle ordinate abbiamo rappresentato la quantità di istanze di attacco rilevate.

Ftp Per il servizio ftp abbiamo selezionato gli attacchi in base alla quantità di dati presenti in fase di training ed in fase di testing. In particolare sono presenti nel dataset 24 istanze di 12 diversi attacchi. Delle 24 istanze solo 6 presentano dati sufficienti alla generazione dei profili. Per queste 6 istanze abbiamo a disposizione 4,5 megabyte di dati per generare il profilo, per un totale di 50k payload circa.

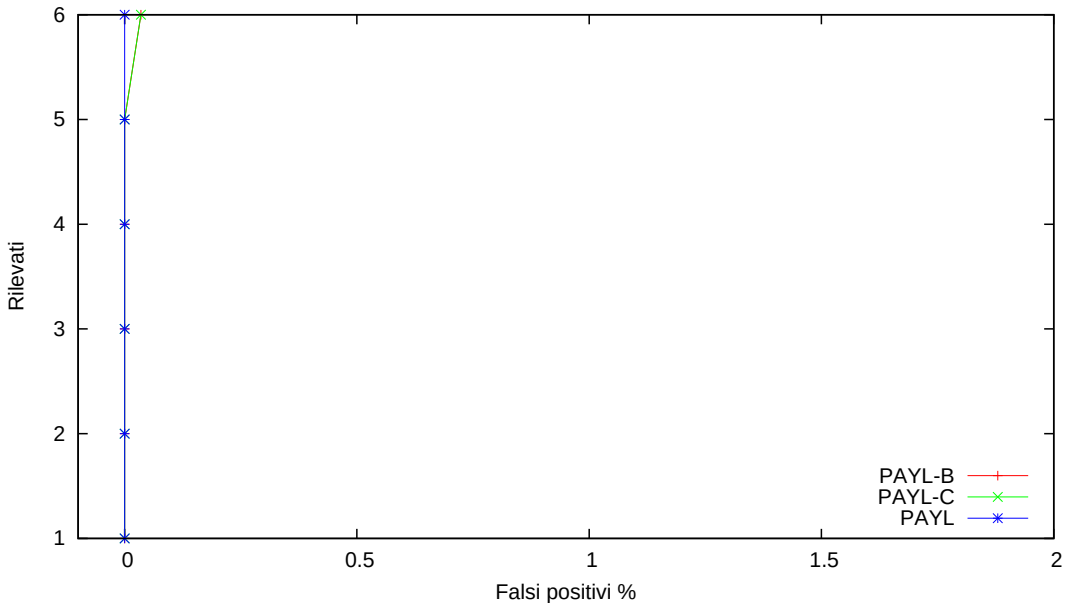

Figura 5.2: Risultati Ftp (porta 21/TCP)

Sulle sei istanze considerate, cinque presentano gli stessi risultati in termini di falsi positivi, mentre in un caso abbiamo un risultato migliore con il clone di payl rispetto alle versioni modificate. In questo caso abbiamo ottenuto lo 0.035% di falsi positivi con PAYL-B e PAYL-C, mentre con il clone di PAYL abbiamo ottenuto il rilevamento di tutte le istanze degli attacchi con lo 0% di falsi positivi. In figura 5.2 è rappresentato il grafico relativo ai test.

SSh Nel caso dell'SSh, abbiamo 3 istanze di 2 diversi attacchi su cui effettuare i test. La quantità di dati utilizzabili per la fase di training varia da 11MB a 32MB.

SSh è un servizio di login cifrato. Questo farebbe pensare all'impossibilità di rilevare attacchi all'interno di tale servizio. In realtà gli attacchi che coinvolgono il servizio SSh, non sono veicolati nel payload cifrato, ma coinvolgono la parte iniziale della connessione SSh, in particolare dopo lo stabilirsi di una connessione tra client e server, il server presenta al client la propria chiave pubblica. Gli attacchi coinvolti nel servizio SSh e rilevati nei test sono costituiti dalla sostituzione da parte di un utente malevolo del server SSh con uno diverso contenente una backdoor. La sostituzione del server ha causato anche il cambio della chiave pubblica che il server comunica ai client all'inizio della connessione. In questo caso l'anomalia consiste nella rilevazione della nuova chiave pubblica, il cui scambio tra server e client avviene in un canale non cifrato.

In questo caso, come si può notare nel grafico in figura 5.3, abbiamo ottenuto una quantità di falsi positivi inferiore allo 0.107% con PAYL-B ed inferiore allo 0.098% con PAYL-C per rilevare tutti le istanze di tali attacchi, mentre con il clone di PAYL abbiamo ottenuto lo stesso risultato con una quantità di falsi positivi di poco inferiore allo 1.488%.

Telnet Per il servizio telnet abbiamo 43 istanze di 20 diversi attacchi. Di queste 43 istanze 39 avevano dati sufficienti per la generazione dei profili.

Il servizio telnet è coinvolto nella maggior parte degli attacchi nel dataset utilizzato. Il motivo di questa maggior quantità rispetto agli altri servizi è dovuto alla presenza nel traffico di attacchi considerati come locali. Il telnet è infatti un servizio di login

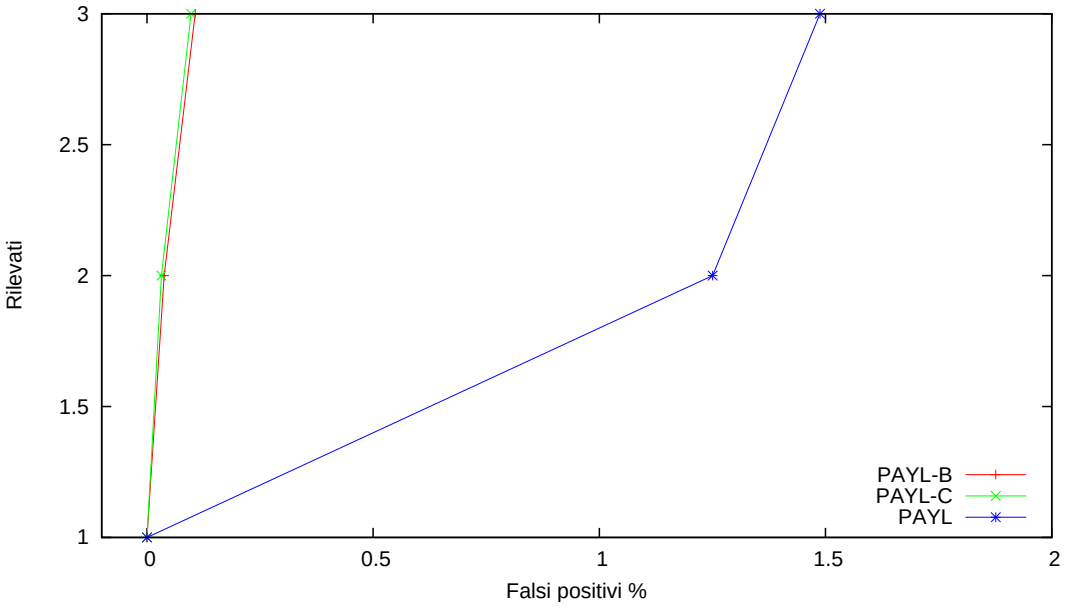

Figura 5.3: Risultati Ssh (porta 22/TCP)

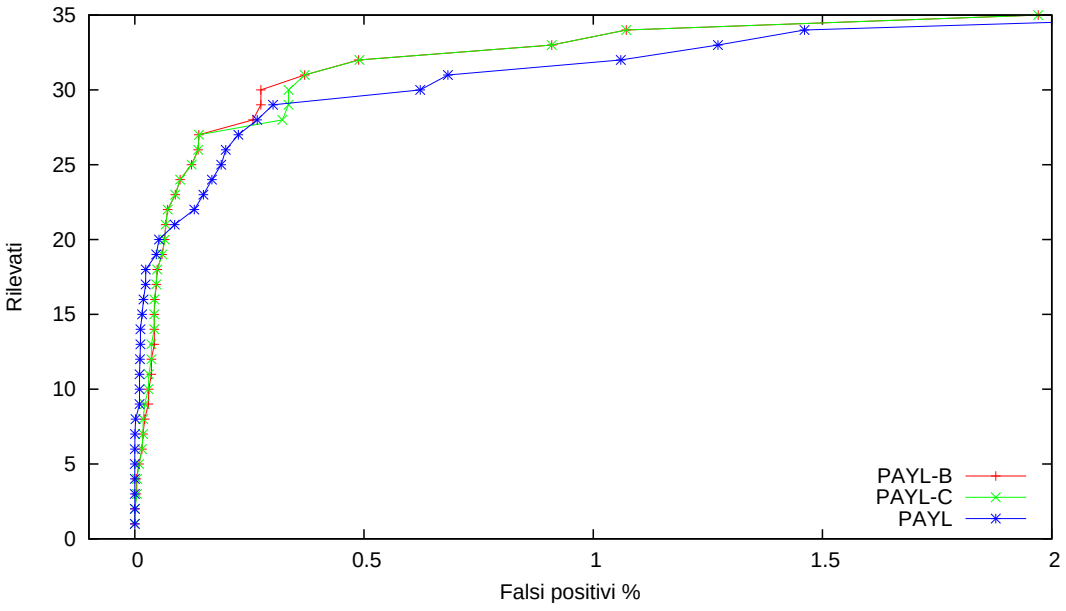

Figura 5.4: Risultati Telnet (porta 23/TCP)

remota e gli attacchi portati attraverso telnet lasciano tracce nel traffico di rete per cui è possibile per un IDS Network based rilevare questo tipo di intrusioni.

In figura 5.4, sono mostrati i risultati ottenuti. In particolare possiamo notare come PAYL-B e PAYL-C diano risultati peggiori fino allo 0.065% di falsi positivi per poi comportarsi meglio del clone di PAYL.

Da notare inoltre ad un certo punto del grafico il peggioramento ottenuto da PAYL-C ossia dalla versione modificata col classificatore attivo.

SMTP Per il servizio di posta elettronica (SMTP) abbiamo un totale di 16 istanze di 8 attacchi diversi con una quantità sufficiente di dati per la fase di training.

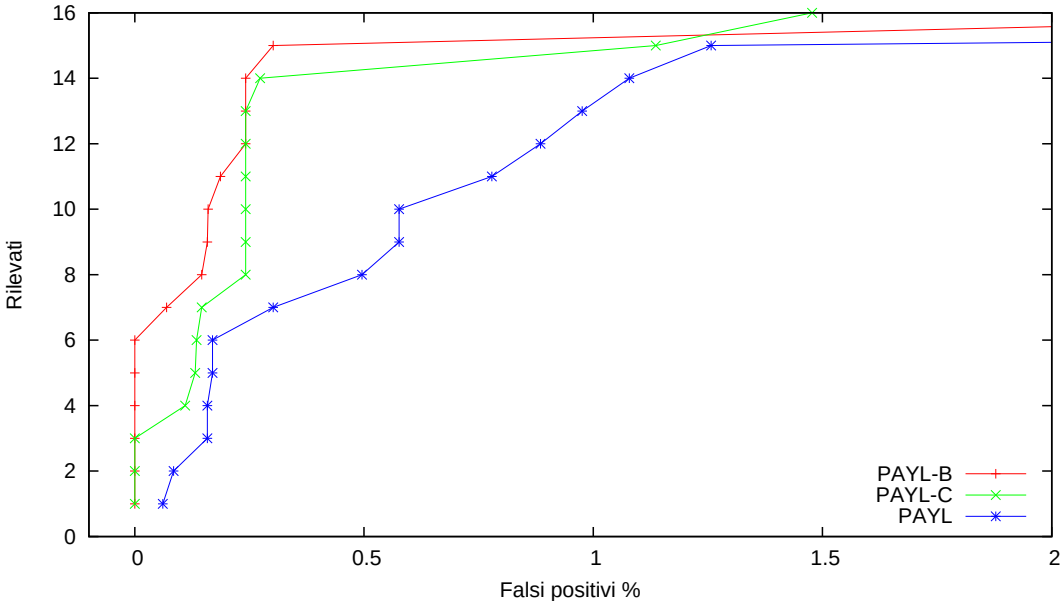

Figura 5.5: Risultati SMTP (porta 25/TCP)

In figura 5.5 sono mostrati i risultati dei test. In questo caso abbiamo ottenuto i risultati migliori con entrambe le versioni modificate. Anche nel caso del servizio di posta possiamo notare una discordanza in peggio dei risultati ottenuti con la versione col classificatore attivo.

HTTP Nel caso del servizio web (HTTP) abbiamo 15 istanze di 8 attacchi diversi con una quantità di dati sufficiente per la generazione dei profili in fase di training.

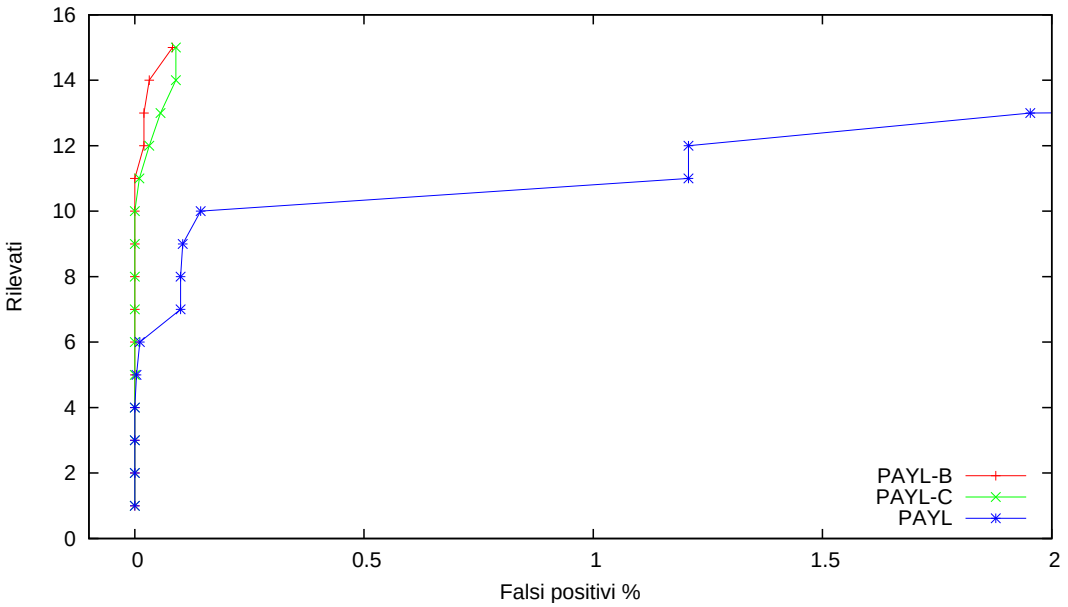

Figura 5.6: Risultati HTTP (porta 80/TCP)

In figura 5.6 sono mostrati i risultati ottenuti per il servizio web. Anche in questo caso, come nel servizio di posta elettronica possiamo notare una riduzione dei falsi positivi in PAYL-B e PAYL-C rispetto al clone di PAYL. Troviamo ancora, anche in questi risultati una discordanza tra PAYL-B e PAYL-C a sfavore di quest'ultimo.

Visione complessiva dei risultati La figura 5.7 rappresenta i risultati ottenuti su tutte le istanze degli attacchi relativamente ad ogni servizio considerato nei test.

Per riassumere, su 79 istanze di attacco considerate nei test abbiamo ottenuto in PAYL-B il riconoscimento di 72 anomalie con una quantità di falsi positivi inferiore all'1%, con PAYL-C la quantità di anomalie rilevate scende a 71 a parità di falsi positivi. Nel clone di PAYL abbiamo ottenuto il riconoscimento di 61 anomalie su 79 con una quantità di falsi positivi inferiore all'1%.

Analogamente, per ottenere il riconoscimento di 61 anomalie su 79, abbiamo ot-

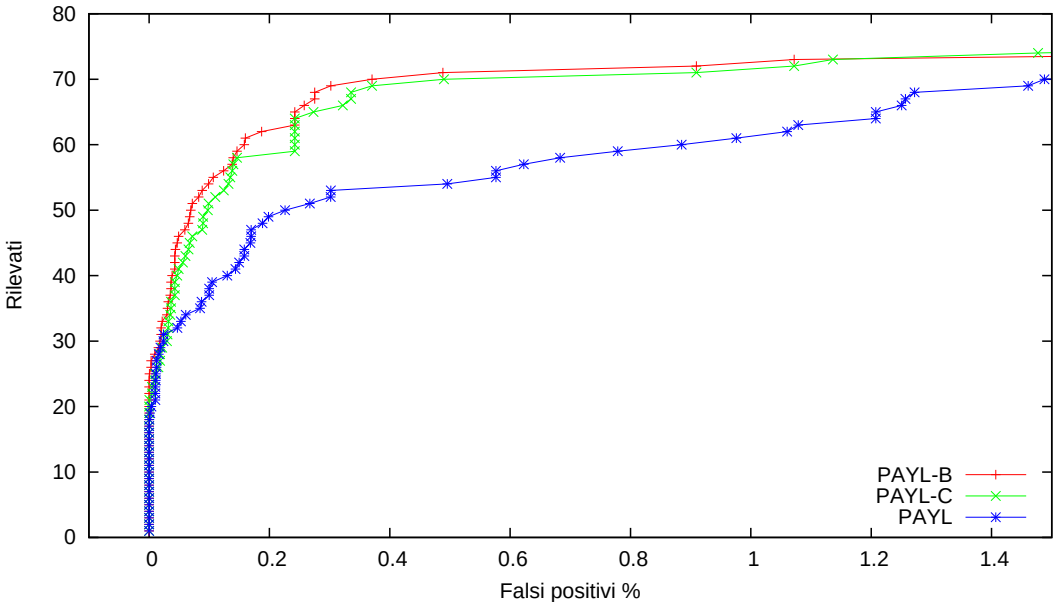

Figura 5.7: Risultati complessivi

tenuto in PAYL-B e PAYL-C una quantità di falsi positivi di poco inferiore allo 0.3% mentre nel clone di PAYL abbiamo ottenuto lo 0.97% di falsi positivi.

Nella tabella 5.4 sono mostrati i risultati per ogni singolo servizio con un false posirive rate inferiore all'1%.

Anche nella visione complessiva dei risultati possiamo notare però una quantità di falsi positivi generalmente maggiore in PAYL-C rispetto a PAYL-B, contrariamente a quanto ci si aspettava, l'introduzione nel modello del classificatore porta ad un peggioramento nei risultati. Nella prossima sezione tenteremo di dare una spiegazione a questo comportamento "anomalo".

5.5 Considerazioni sull'incidenza del classificatore

Abbiamo accennato nella sezione 3.2 ad un problema di classificazione dei payload. Riassumendo brevemente, il modello di PAYL basa il suo funzionamento sulla classificazione dei payload in funzione della loro dimensione. Il problema da noi rilevato si

NIDS		PAYL	PAYL-B	PAYL-C
HTTP	Detection Rate	10/15 (67%)	15/15 (100%)	15/15 (100%)
	False positive Rate	0,14%	0,082%	0,089%
FTP	Detection Rate	6/6 (100%)	6/6 (100%)	6/6 (100%)
	False positive Rate	0%	0,035%	0,035%
SMTP	Detection Rate	13/16 (81%)	15/16 (93%)	14/16 (87%)
	False positive Rate	0,97%	0,30%	0,27%
TELNET	Detection Rate	31/39 (79%)	33/39 (84%)	33/39 (84%)
	False positive Rate	0,68%	0,90%	0,90%
SSH	Detection Rate	1/3 (33%)	3/3 (100%)	3/3 (100%)
	False positive Rate	0%	0,10%	0,09%
Totale	Detection Rate	61/69 (77%)	72/79 (91%)	71/79 (89%)
	False positive Rate	0,97%	0,90%	0,90%

Tabella 5.1: Confronto tra il clone di PAYL, PAYL-B e PAYL-C con false positive rate inferiore all'1%.

riferisce alla diversa rappresentazione che si ha dei payload nei vari livelli dello stack TCP/IP. Un payload di dimensione 5000 byte, viene visto, nei livelli più bassi dello stack TCP/IP, suddiviso in più payload di dimensione non superiore ai limiti imposti dall'infrastruttura di rete. In una tipica rete ethernet tale limite è di 1460 byte, per cui il payload originale di 5000 byte verrebbe rappresentato nei livelli inferiori dello stack TCP/IP, in 3 segmenti da 1460 byte ed uno da 620 byte.

L'assunzione degli autori di PAYL consiste nella distinzione tra payload di piccole dimensioni, contenenti presumibilmente dei comandi come possono essere i comandi HTTP, e payload di grandi dimensioni come possono essere le immagini .gif o .jpg di una pagina web. Tale distinzione consentirebbe di ottenere profili separati tra comandi e file come .gif e .jpg.

I segmenti "terminali" dei payload di dimensioni superiori al limite di 1460, secondo il modello di PAYL, verrebbero associati a profili relativi a dimensioni piccole. Nel nostro esempio il payload terminale è di 620 byte e pertanto verrebbe associato al profilo relativo alla dimensione 620. Questi casi li abbiamo considerati come difetti di classificazione.

Con l'introduzione del classificatore nel modello, riducendo i "disturbi" introdotti

dal difetto di classificazione, ci si aspettava ragionevolmente un miglioramento sia in termini di maggior detection rate che in termini di minori falsi positivi, ma i risultati hanno dimostrato il contrario nella maggior parte dei test.

I risultati ottenuti ci hanno indotto ad esaminare più nel dettaglio questa particolare "anomalia" rilevando alcune caratteristiche a nostro avviso peculiari del dataset su cui sono stati effettuati i test.

Per identificare l'origine dei risultati peggiori ottenuti con l'introduzione del classificatore abbiamo provato a quantificare l'incidenza dei payload "terminali" sul rilevamento delle istanze degli attacchi rilevando la presenza di una quantità relativamente ridotta di tali payload sia nei dump del traffico relativi alla fase di training che nei dump relativi al conteggio dei falsi positivi in fase di detection.

Nei dump utilizzati in fase di training abbiamo rilevato la presenza di 34905 payload che presentano difetti di classificazione 2931989 totali, pari all'1% del totale rappresentanti comunque una quantità non trascurabile in termini di corruzione dei profili.

Nei dump utilizzati in fase di controllo dei falsi positivi tale rapporto scende a 28049 payload che presentano difetti di classificazione su 3125865 totali pari allo 0,9% del totale.

Il risultato più interessante però è stato ottenuto analizzando le istanze dei singoli attacchi. Verificando la presenza di tali payload all'interno delle singole istanze, abbiamo notato che 34 istanze di attacchi su 79 considerate presentano payload con tali caratteristiche.

Abbiamo verificato inoltre che i payload con tali caratteristiche presenti nelle 34 istanze di attacchi in virtù della loro errata classificazione rivelano in PAYL e PAYL-B avere una distanza di mahalanobis elevata rispetto ai payload "normali" mentre in PAYL-C col classificatore attivo tali pacchetti, venendo confrontati con i profili da noi ritenuti corretti presentano delle distanze di mahalanobis sensibilmente inferiori, andando così ad incidere sulla scelta delle soglie di allarme da utilizare per il loro rilevamento. Dovendo utilizzare, in PAYL e PAYL-B, nella rilevazione di quelle istanze di attacco una soglia più elevata, otteniamo in fase di conteggio dei falsi positivi un valore in gererale più basso.

Presentate queste caratteristiche peculiari del dataset, si potrebbe provocatoriamente realizzare un IDS in grado di rilevare il 43% degli attacchi considerati (34 su 79) con una quantità di falsi positivi pari allo 0,9% ricavata dal rapporto tra la quantità di payload classificati male e la quantità totale di payload presenti nel dump utilizzato per il conteggio dei falsi positivi.

5.6 Prestazioni dei modelli

Per quanto riguarda i tempi di esecuzione del clone di PAYL e di PAYL-B, dobbiamo considerare il metodo di calcolo della distanza di Mahalanobis. In PAYL, l'assunzione di indipendenza stocastica dei byte porta a semplificare il calcolo della distanza considerando solamente la diagonale della matrice di varianza-covarianza, mentre in PAYL-B si è deciso di effettuare tale calcolo sulla base della matrice completa.

Le misure effettuate sui due modelli relativamente alla fase di training, hanno mostrato un incremento notevole nei tempi di calcolo relativi all'aggiornamento dei profili. Il risultato ottenuto è da imputare alla necessità di aggiornare in PAYL-B per ogni payload la matrice quadrata di dimensione 256 contenente il valore $E[XX^T]$, mentre in PAYL al suo posto viene aggiornato il vettore $E[X^2]$ di 256 elementi. In PAYL-B viene inoltre effettuata l'inversione delle matrici di varianza-covarianza, ma questa operazione non incide sensibilmente nei tempi di calcolo, dovendo essere effettuata una sola volta per ogni profilo al termine della fase di training. In definitiva, le misurazioni effettuate hanno indicato un tempo di aggiornamento dei profili per 100 payload alla volta considerati in PAYL di 10 micro secondi, mentre in PAYL-B il tempo utilizzato per il calcolo sale a 680 microsecondi per 100 payload.

Anche per la fase di detection l'utilizzo della matrice completa incide pesantemente nei tempi di calcolo in PAYL-B. Abbiamo effettuato la misurazione dei tempi di calcolo per ogni singolo payload, includendo anche l'overhead dovuto agli accessi in memoria ed alle letture dei dati in transito. Il risultato ottenuto è di 10 microsecondi per payload in PAYL e di 200 microsecondi per PAYL-B.

In definitiva PAYL è in grado di reggere, nella fase di detection, nel caso migliore in cui i payload fossero della lunghezza massima di 1460, una quantità di traffico pari

a 1.2 Gigabit al secondo, mentre PAYL-B nelle stesse condizioni arriverebbe a reggere un traffico pari a 60 Megabit al secondo presentando un peggioramento nei tempi di calcolo rispetto a PAYL di circa 20 volte.

Per quanto riguarda la fase di training PAYL è in grado di processare 10 milioni di payload al secondo pari a 120 Gigabit nel caso migliore, mentre PAYL-B è in grado di processare 1,7 Gigabit.

Per quanto riguarda l'utilizzo della memoria, nel modello di PAYL vengono memorizzati per ogni profilo 2 vettori di 256 elementi in virgola mobile pari a $2 \cdot 256 \cdot 4 = 2048$ byte. Nel caso peggiore in cui dovessero venir memorizzati tutti i 1460 profili relativi ad ogni dimensione di payload possibile per una rete ethernet, l'occupazione di memoria arriverebbe a 4 Megabyte per servizio. L'operazione di clustering, in cui vengono fusi assieme i profili più simili tra loro, ha portato nei nostri test, ad avere una media di 6 profili memorizzati per servizio ottenendo un'occupazione in media di 12 Kilobyte per servizio monitorato.

In PAYL-B e PAYL-C abbiamo deciso di memorizzare l'intera matrice di varianza-covarianza al posto del solo vettore delle deviazioni standard, per cui per ogni profilo vengono memorizzati un vettore da 256 elementi in virgola mobile ed una matrice quadrata di dimensione 256 composta da elementi in virgola mobile. Riassumendo, per PAYL-B e PAYL-C avremmo nel caso peggiore in cui dovessero venir memorizzati tutti i 1460 profili, un'occupazione di memoria pari a $1460 \cdot 4 \cdot (256 + 256^2) = 366$ Megabyte di ram per servizio. L'operazione di clustering nei nostri test, riducendo a 6 la quantità media di profili da memorizzare, ha portato ad avere un'occupazione di memoria pari a 1,5 Megabyte per servizio. I prototipi di PAYL-B e PAYL-C utilizzano l'intera matrice quadrata di dimensione 256 contenente la matrice di varianza-covarianza. Tale matrice è simmetrica, per cui memorizzando $(256 \cdot 257)/2 = 32896$ elementi potremmo portare l'occupazione di memoria a 0,76 Megabyte per servizio.

I test sono stati effettuati utilizzando un computer portatile BENQ con processore Pentium M ad 1,73 GHz e 512MB di ram.

Appendice A

Codice Sorgente

A.1 main.c

```
#include "common.h"


int main(int argc, char **argv)
{
   PARAMS conf;

   if(parse_options(&argc, &argv, &conf)==-1)
   {
      exit(EXIT_FAILURE);
   }

   if(conf.mode == TRAINING)
   {
      PACKET_FILTER_do_training(conf);
   }
   else if(conf.mode == DETECTION)
   {
      PACKET_FILTER_do_detection(conf);
   }
   exit(EXIT_SUCCESS);
}
```

A.2 Packet Filter

A.2.1 packet_filter.h

```
#ifndef PACKET_FILTER_H
#define PACKET_FILTER_H

int PACKET_FILTER_do_training(PARAMS);

int PACKET_FILTER_do_detection(PARAMS);

#endif
```

A.2.2 packet_filter.c

```
#include "common.h"

static PROFILE p;

static PARAMS params;

static int packets = 0;

static int malicious = 0;

static float maximum = 0;

static pcap_t *open_stream(char *, char *);

static void training(u_char *, const struct pcap_pkthdr *, const u_char *);

static void detection(u_char *, const struct pcap_pkthdr *, const u_char *);

int PACKET_FILTER_do_training(PARAMS conf)
{
   pcap_t *stream;

   params = conf;

   stream = open_stream(params.streamfile, params.filter);
   if(stream)
   {
      p = PROFILE_init();
```

```
      pcap_loop(stream, -1, training, NULL);

      PROFILE_do_clustering(p, params.manhattan);

      PROFILE_fix(p);

      PROFILE_save(p, params.outfile);
   }
   return EXIT_SUCCESS;
}

int PACKET_FILTER_do_detection(PARAMS conf)
{
   pcap_t *stream;

   params = conf;

   stream = open_stream(params.streamfile, params.filter);
   if(stream)
   {
      p = PROFILE_init();

      PROFILE_load(p, params.infile);

      pcap_loop(stream, -1, detection, NULL);
   }
   return EXIT_SUCCESS;
}

static pcap_t *open_stream(char *file, char *filter)
{
   struct bpf_program fp;
   char errbuf[PCAP_ERRBUF_SIZE];
   pcap_t *stream;

   stream = pcap_open_offline(file, errbuf);

   if(stream == NULL)
   {
      printf("%s\n", errbuf);
      return NULL;
   }

   if(pcap_compile(stream, &fp, filter, 0, 0xffffffff) == -1)
   {
      printf("%s\n", pcap_geterr(stream));
      return NULL;
```

```
   }

   if(pcap_setfilter(stream, &fp)==-1)
   {
      printf("%s\n", pcap_geterr(stream));
      return NULL;
   }
   return stream;
}

static void training(u_char *user, const struct pcap_pkthdr *pkthdr, const u_char *data)
{
   struct ether_header *ethh;
   struct ip *iph;
   struct tcphdr *tcph;
   unsigned char *payload;
   int pay_caplen;
   int pay_len;
   int class;
   float FV[DIM];
   sockpair s;

   payload = NULL;
   pay_len = 0;

   ethh = (struct ether_header *) data;
   if(ethh->ether_type == htons(ETHERTYPE_IP))
   {
      iph = (struct ip *)(ethh + 1);
      switch(iph->ip_p)
      {
         case IPPROTO_TCP:
            tcph = (struct tcphdr *)(((char *)iph) + (iph->ip_hl<<2));

            payload = ((unsigned char *)tcph) + (tcph->th_off<<2);
            pay_len = ntohs(iph->ip_len) - (iph->ip_hl<<2) - (tcph->th_off<<2);
            pay_caplen = (int) (pkthdr->caplen) - ((int)payload - (int)data);
            if(pay_len > pay_caplen)
            {
               printf("truncated packet\n");
               printf("%d - %d - %d\n", ntohs(iph->ip_len), (iph->ip_hl<<2), (tcph->th_off<<2));
               printf("pay_len: %d pay_caplen: %d\n", pay_len, pay_caplen);
            }
            else if((tcph->th_flags&TH_SYN) != TH_SYN)
            {
               if(pay_len > 0)
               {
```

```
                    s.s_addr = iph->ip_src.s_addr;
                    s.d_addr = iph->ip_dst.s_addr;
                    s.s_port = tcph->th_sport;
                    s.d_port = tcph->th_dport;

                    class = CLASSIFIER_classify(s, pay_len);

                    FEATURE_EXTRACTOR_extract_feature(payload, FV, pay_len);

                    PROFILE_update(p, FV, class);

                    packets++;
                 }
              }
              break;
           default:
              break;
        }
     }
}

static void detection(u_char *user, const struct pcap_pkthdr *pkthdr, const u_char *data)
{
   struct ether_header *ethh;
   struct ip *iph;
   struct tcphdr *tcph;
   unsigned char *payload;
   int pay_caplen;
   int pay_len;
   int class;
   float FV[DIM];
   float d;
   sockpair s;

   payload = NULL;
   pay_len = 0;

   ethh = (struct ether_header *) data;
   if(ethh->ether_type == htons(ETHERTYPE_IP))
   {
      iph = (struct ip *)(ethh + 1);
      switch(iph->ip_p)
      {
         case IPPROTO_TCP:
            tcph = (struct tcphdr *)(((char *)iph) + (iph->ip_hl<<2));

            payload = ((unsigned char *)tcph) + (tcph->th_off<<2);
```

```
            pay_len = ntohs(iph->ip_len) - (iph->ip_hl<<2) - (tcph->th_off<<2);
            pay_caplen = (int) (pkthdr->caplen) - ((int)payload - (int)data);
            if(pay_len > pay_caplen)
            {
               printf("truncated packet\n");
               printf("%d - %d - %d\n", ntohs(iph->ip_len), (iph->ip_hl<<2), (tcph->th_off<<2));
               printf("pay_len: %d pay_caplen: %d\n", pay_len, pay_caplen);
            }
            else if((tcph->th_flags&TH_SYN) != TH_SYN)
            {
               if(pay_len > 0)
               {
                  s.s_addr = iph->ip_src.s_addr;
                  s.d_addr = iph->ip_dst.s_addr;
                  s.s_port = tcph->th_sport;
                  s.d_port = tcph->th_dport;

                  class = CLASSIFIER_classify(s, pay_len);

                  FEATURE_EXTRACTOR_extract_feature(payload, FV, pay_len);

                  d = PROFILE_get_distance(p, FV, class);

                  if(d>maximum)
                     maximum = d;
                  if(d > params.mahalanobis)
                  {
                     malicious++;
                  }
                  packets++;
               }
            }
            break;
         default:
            break;
      }
   }
}
```

A.3 Classifier

A.3.1 classifier.h

```
#ifndef CLASSIFIER_H
#define CLASSIFIER_H

typedef struct
{
   u_int32_t s_addr;
   u_int32_t d_addr;
   u_int16_t s_port;
   u_int16_t d_port;
} sockpair;

int CLASSIFIER_classify(sockpair, int);

void CLASSIFIER_init(void);

#endif
```

A.3.2 classifier.c

```
#include "common.h"
#include "classifier.h"

#ifndef HT_DIM
#define HT_DIM 1024
#endif

struct elem
{
   sockpair s;
   struct elem *next;
};

static struct elem *ht[HT_DIM];

static int is_dirt(sockpair);

static void set_dirt(sockpair);

static int hash(sockpair s)
{
   int key;
```

```
   key = s.s_port ^ s.d_port;
   key *= HT_DIM;
   key /= (u_int16_t)(-1);
   return key;
}

void CLASSIFIER_init(void)
{
   int i;

   for(i=0;i<HT_DIM;i++)
   {
      ht[i]=NULL;
   }
}

int CLASSIFIER_classify(sockpair s, int paylen)
{
   int class;

   if(paylen == MAXPAYLEN)
   {
      set_dirt(s);
      class = MAXPAYLEN;
   }
   else
   {
      if(is_dirt(s))
      {
         class = MAXPAYLEN;
      }
      else
      {
         class = paylen;
      }
   }
   return class;
}

static int is_dirt(sockpair s)
{
   int key;
   struct elem *list;
   struct elem *prev;
```

```
   prev = NULL;
   key = hash(s);
   list = ht[key];
   while(list)
   {
      if(memcmp(&s,&list->s, sizeof(sockpair))==0)
      {
         if(prev==NULL)
         {
            ht[key]=list->next;
            free(list);
         }
         else
         {
            prev->next = list->next;
            free(list);
         }
         return 1;
      }
      prev = list;
      list = list->next;
   }
   return 0;
}

static void set_dirt(sockpair s)
{
   int key;
   struct elem *list;
   struct elem *prev;

   key = hash(s);
   list = ht[key];
   prev = NULL;

   if(list == NULL)
   {
      list = malloc(sizeof(struct elem));
      list->s = s;
      list->next = NULL;
      ht[key] = list;
   }
   else
   {
      while(list)
      {
         if(memcmp(&s, &list->s, sizeof(sockpair))==0)
```

```
         {
            return;
         }
         prev = list;
         list = list->next;
      }
      list = malloc(sizeof(struct elem));
      list->s = s;
      list->next = NULL;
      prev->next = list;
   }

   return;
}
```

A.4 FeatureExtractor

A.4.1 feature_extractor.h

```
#ifndef FEATURE_EXTRACTOR_H
#define FEATURE_EXTRACTOR_H

#include "common.h"

int FEATURE_EXTRACTOR_extract_feature(unsigned char *, float *, int);

#endif
```

A.4.2 feature_extractor.c

```
#include "common.h"

int FEATURE_EXTRACTOR_extract_feature(unsigned char *payload, float *FV, int paylen)
{
   int i;
   if(payload&&FV)
   {
      if(paylen>0)
      {
         for(i=0;i<paylen;i++)
         {
            FV[payload[i]&MASK]++;
         }
```

```
         for(i=0;i<DIM;i++)
         {
            FV[i] /= paylen;
         }
      }
      return paylen;
   }
   else
   {
      return 0;
   }
}
```

A.5 Profile

A.5.1 profile.h

```
#ifndef PROFILE_H
#define PROFILE_H

#include "common.h"

typedef struct profile **PROFILE;

PROFILE PROFILE_init(void);

int PROFILE_update(PROFILE, float *, int);

int PROFILE_fix(PROFILE);

int PROFILE_do_clustering(PROFILE, float);

float PROFILE_get_distance(PROFILE, float *, int);

int PROFILE_load(PROFILE, char *);

int PROFILE_save(PROFILE, char *);

#endif
```

A.5.2 profile.c

```
#include "common.h"
#include <cblas.h>
```

```
#include <clapack.h>

#define MAX_PROFILES 1460
#define SMOOTH 0.00003
#define MAX 10000

struct profile
{
   float mu[DIM];
#ifdef PAYL
   float Sigma[DIM];
#else
   float Sigma[DIM*DIM];
#endif
   int n_samples;
   int fixed;
   int ref_no;
   int refs[MAX_PROFILES+1];
};

#ifndef PAYL
static int PROFILE_correct(struct profile *);
#endif

static float manhattan(float *, float *, int);

static int merge(PROFILE, int, int);

PROFILE PROFILE_init(void)
{
   PROFILE p;

   p = malloc(sizeof(PROFILE)*(MAX_PROFILES+1));
   if(p)
   {
      bzero(p, sizeof(PROFILE)*(MAX_PROFILES+1));
   }
   else
   {
      fprintf(stderr, "malloc: file:%s, line:%d\n", __FILE__, __LINE__);
   }
   return p;
}

int PROFILE_update(PROFILE p, float *FV, int c)
{
   int i;
```

```
   float FV_t[DIM];
   float *mu;
   float *Sigma;

   if(p[c]==NULL)
   {
      p[c] = malloc(sizeof(struct profile));
      if(p[c]==NULL)
      {
         fprintf(stderr, "malloc: file:%s, line:%d\n", __FILE__, __LINE__);
         return 0;
      }
      else
      {
         bzero(p[c], sizeof(struct profile));
         p[c]->refs[p[c]->ref_no++]=c;
      }
   }
   bzero(FV_t, sizeof(float)*DIM);
   if(FV)
   {
      mu = p[c]->mu;
      Sigma = p[c]->Sigma;
      for(i=0;i<DIM;i++)
      {
         mu[i] = mu[i] + (FV[i]-mu[i]) / (p[c]->n_samples + 1);
#ifdef PAYL
         Sigma[i] = Sigma[i] + ((FV[i] * FV[i]) - Sigma[i]) / (p[c]->n_samples + 1);
#else
         FV_t[i] = FV[i];
#endif
      }
      p[c]->n_samples += 1;
#ifndef PAYL
      cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, DIM, DIM, 1, 1, FV, DIM, FV_t, 1, 1, Sigm
#endif
   }
   else
   {
      return 0;
   }
   return 1;
}

int PROFILE_fix(PROFILE p)
{
   int i, j;
```

```
   float *mu;
   float *Sigma;
#ifndef PAYL
   int info;
   int ipiv[DIM];
   float beta;
   float mu_t[DIM];
   float *tmp;
#endif

   struct profile *cur;

   if(p)
   {
      for(i=0;i<MAX_PROFILES+1;i++)
      {
         cur = p[i];

         if(cur)
         {
            if(cur->fixed == 0)
            {
#ifndef PAYL
               if(cur->n_samples >0)
                  beta = 1.0/cur->n_samples;
               else
                  return 0;
#endif
               mu = cur->mu;
               Sigma = cur->Sigma;
               for(j=0;j<DIM;j++)
               {
#ifdef PAYL
                  Sigma[j] = sqrt(Sigma[j] - mu[j] * mu[j] + SMOOTH);
#else
                  mu_t[j] = mu[j];
#endif
               }
#ifndef PAYL
               cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, DIM, DIM, 1, -1, mu, DIM, mu_t,
               tmp = malloc(sizeof(float)*DIM*DIM);
               PROFILE_correct(cur);
               memcpy(tmp, Sigma, sizeof(float)*DIM*DIM);
               bzero(ipiv, sizeof(int)*DIM);
               info = clapack_sgetrf(CblasColMajor, DIM, DIM, tmp, DIM, ipiv);
               while(info!=0)
               {
```

```
                PROFILE_correct(cur);
                memcpy(tmp, Sigma, sizeof(float)*DIM*DIM);
                bzero(ipiv, sizeof(int)*DIM);
                info = clapack_sgetrf(CblasColMajor, DIM, DIM, tmp, DIM, ipiv);
             }
             memcpy(Sigma, tmp, sizeof(float)*DIM*DIM);
             free(tmp);
             info = clapack_sgetri(CblasColMajor, DIM, Sigma, DIM, ipiv);
             if(info!=0)
             {
                printf("sgetri\n");
                return -1;
             }
#endif
             cur->fixed = 1;
          }
       }
    }
    return 1;
 }
 else
 {
    return 0;
 }
}

int PROFILE_do_clustering(PROFILE p, float d)
{
 int i,loop;

 loop = 0;

 do
 {
    loop = 0;
    for(i=0;i<MAXPAYLEN;i++)
    {
       if(p[i]==NULL)
       {
          if(p[i+1] != NULL)
          {
             p[i]=p[i+1];
             p[i]->refs[p[i]->ref_no++] = i;
             loop = 1;
          }
       }
    }
```

```
      for(i=1460;i>0;i--)
      {
         if(p[i]==NULL)
         {
            if(p[i-1] != NULL)
            {
               p[i]=p[i-1];
               p[i]->refs[p[i]->ref_no++] = i;
               loop = 1;
            }
         }
      }
   } while(loop);
   do
   {
      loop=0;
      for(i=0;i<MAX_PROFILES;i++)
      {
         if((p[i]!=NULL) && (p[i+1]!=NULL) && (p[i]!=p[i+1]))
         {
            if(manhattan(p[i]->mu, p[i+1]->mu, DIM)<d)
            {
               merge(p, i, i+1);
               loop=1;
            }
         }
      }
   }
   while(loop);
   return 0;
}

float PROFILE_get_distance(PROFILE p, float *FV, int c)
{
   int i;
   float d;
   float *mu,*Sigma;
#ifdef PAYL
   float tmp;
#else
   float X[DIM];
   float T[DIM];
#endif


   if(p[c]==NULL)
      return MAX;
```

```
   mu = p[c]->mu;
   Sigma = p[c]->Sigma;

   d = 0;
   for(i=0;i<DIM;i++)
   {
#ifdef PAYL
      tmp = FV[i] - mu[i];
      if(tmp<0)
         tmp = -tmp;
      if(tmp!=0)
         tmp = tmp/Sigma[i];
      d += tmp;
#else
      X[i] = FV[i] - mu[i];
#endif
   }
#ifndef PAYL
   cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 1, DIM, DIM, 1, X, 1, Sigma, DIM, 0, T, 1);
   cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, 1, 1, DIM, 1, T, 1, X, DIM, 0, &d, 1);
   return sqrt(d);
#else
   return d;
#endif
}

int PROFILE_load(PROFILE p, char *filename)
{
   int i;
   int fd;
   struct profile curr;
   struct profile *tmp;

   fd = open(filename, O_RDONLY);
   if(fd < 0)
   {
      fprintf(stderr, "%s:%d: error opening %s\n", __FILE__, __LINE__, filename);
      return 0;
   }
   while(read(fd, &curr, sizeof(struct profile)) == sizeof(struct profile))
   {
      tmp = malloc(sizeof(struct profile));
      if(tmp==NULL)
      {
         fprintf(stderr, "%s:%d: malloc error\n", __FILE__, __LINE__);
         return 0;
      }
```

```
      memcpy(tmp, &curr, sizeof(struct profile));
      for(i=0;i<curr.ref_no;i++)
      {
         p[curr.refs[i]]=tmp;
      }
   }
   return 1;
}

int PROFILE_save(PROFILE p, char *filename)
{
   int i,j;
   int fd;
   struct profile *s[MAX_PROFILES+1];
   struct profile *curr;

   for(i=0;i<=MAX_PROFILES;i++)
   {
      s[i] = p[i];
   }

   fd = open(filename, O_CREAT|O_TRUNC|O_RDWR, 0644);
   if(fd < 0)
   {
      fprintf(stderr, "%s:%d: error opening %s\n", __FILE__, __LINE__, filename);
      return 0;
   }

   for(i=0;i<=MAX_PROFILES;i++)
   {
      curr = s[i];
      if(curr)
      {
         for(j=0;j<curr->ref_no;j++)
         {
            printf("%d ", curr->refs[j]);
            s[curr->refs[j]]=NULL;
         }
         if(write(fd, curr, sizeof(struct profile))!=sizeof(struct profile))
         {
            fprintf(stderr, "%s:%d: write error\n", __FILE__, __LINE__);
         }
      }
   }

   close(fd);
```

```
   return 1;
}

static float manhattan(float *A, float *B, int n)
{
   int i;
   float d;

   d = 0;

   if(A && B)
   {
      for(i=0;i<n;i++)
      {
         if(A[i]>B[i])
         {
            d += A[i]-B[i];
         }
         else
         {
            d += B[i]-A[i];
         }
      }
      return d;
   }
   else
   {
      return -1;
   }
}

static int merge(PROFILE p, int x, int y)
{
   int i;
#ifndef PAYL
   int j;
#endif
   float s1,s2,d;
   int idx,idx2;

   struct profile *a,*b;

   a = p[x];
   b = p[y];

   idx = a->ref_no;
   idx2 = b->ref_no;
```

```
   for(i=0;i<idx2;i++)
   {
      a->refs[idx + i] = b->refs[i];
   }
   a->ref_no += b->ref_no;
   idx = b->ref_no;

   for(i=0;i<idx2;i++)
   {
      p[b->refs[i]] = a;
   }

   s1 = a->n_samples;
   s2 = b->n_samples;
   d = s1 + s2;
   a->n_samples = d;
   s1 = s1 / d;
   s2 = s2 / d;

   for(i=0;i<DIM;i++)
   {
      a->mu[i] = a->mu[i] * s1 + b->mu[i] * s2;
#ifdef PAYL
      a->Sigma[i] = a->Sigma[i] * s1 + b->Sigma[i] * s2;
#else
      for(j=0;j<DIM;j++)
      {
         a->Sigma[i+j*DIM] = a->Sigma[i+j*DIM] * s1 + b->Sigma[i+j*DIM] * s2;
      }
#endif
   }

   free(b);

   return 0;
}

#ifndef PAYL

static int PROFILE_correct(struct profile *p)
{
   int i;

   for(i=0;i<DIM;i++)
   {
      p->Sigma[i+i*DIM] += SMOOTH;
```

```
   }
   return 1;
}
#endif
```

A.6 options.h

```
#ifndef OPTIONS_H
#define OPTIONS_H

#include "params.h"

void usage(void);

void version(void);

int parse_options(int *, char ***, PARAMS *conf);

#endif
```

A.7 options.c

```
#include "common.h"

static char * copy_argv(register char **);

void usage(void)
{
   printf("./main -h -v -d distance -f file -o model_file -r model_file -s\n");
   return;
}

void version(void)
{
   printf("payl clone v0.0.1\n");
   return;
}

int parse_options(int *argc, char ***argv, PARAMS *conf)
{
   int c;

   extern char *optarg;
```

```
extern int optind;

conf->mode = TRAINING;
conf->infile = NULL;
conf->outfile = NULL;
conf->streamfile = NULL;
conf->mahalanobis = 0;
conf->manhattan = 0.5;

while((c=getopt(*argc, *argv, "vhcD:r:o:f:d:")) != -1)
{
   switch(c)
   {
      case 'r':
         conf->mode = DETECTION;
         conf->infile = optarg;
         //printf("reading model from: %s\n", optarg);
         break;
      case 'o':
         printf("saving model to: %s\n", optarg);
         conf->outfile = optarg;
         break;
      case 'D':
         //printf("threshold: %s\n", optarg);
         conf->manhattan = strtod(optarg, NULL);
         break;
      case 'd':
         //printf("threshold: %s\n", optarg);
         conf->mahalanobis = strtod(optarg, NULL);
         break;
      case 'f':
         //printf("reading stream from: %s\n", optarg);
         conf->streamfile = optarg;
         break;
      case 'c':
         conf->classifier = 1;
         break;
      case 'h':
         usage();
         return -1;
         break;
      case 'v':
         version();
         return -1;
         break;
      default:
         usage();
```

```
            return -1;
            break;
      }
   }
   *argc -= optind;
   *argv += optind;
   conf->filter = copy_argv(*argv);
   return *argc;
}

static char * copy_argv(register char **argv)
{
   register char **p;
   register u_int len = 0;
   char *buf;
   char *src, *dst;

   p = argv;
   if (*p == 0)
      return 0;

   while (*p)
      len += strlen(*p++) + 1;

   buf = (char *)malloc(len);
   if (buf == NULL)
      perror("copy_argv: malloc");

   p = argv;
   dst = buf;
   while ((src = *p++) != NULL) {
      while ((*dst++ = *src++) != '\0')
         ;
      dst[-1] = ' ';
   }
   dst[-1] = '\0';

   return buf;
}
```

A.8 params.h

```
#ifndef PARAMS_H
#define PARAMS_H
```

```
#define TRAINING 1
#define DETECTION 2

typedef struct params
{
   int classifier;
   int mode;
   char *infile;
   char *outfile;
   char *streamfile;
   char *filter;
   float mahalanobis;
   float manhattan;
} PARAMS;

#endif
```

A.9 common.h

```
#ifndef COMMON_H
#define COMMON_H

#ifndef _BSD_SOURCE
#define _BSD_SOURCE
#endif

#ifndef BITS
#define BITS 7
#endif

#ifndef MAXPAYLEN
#define MAXPAYLEN 1460
#endif

#define MASK (((unsigned short)-1)>>(16-BITS))
#define DIM ((int)(MASK+1))

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/ethernet.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/igmp.h>
#include <pcap.h>
#include <math.h>

#include "feature_extractor.h"
#include "profile.h"
#include "classifier.h"
#include "options.h"
#include "packet_filter.h"
#include "params.h"

#endif
```

Bibliografia

[1] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. Next-generation intrusion detection expert system (nides), software users manual, beta-update release. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, May 1994.

[2] J. Hoagland. Spade. Silican defense, http://www.silicondefense.com/software/spice, 2000.

[3] J.McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.

[4] D. E. Knuth. *The Art of Computer Programming*, volume Volume 2, Seminumerical Algorithms of *Addison-Wesley Series in Computer Science and Information Processing*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[5] LIBPCAP. Tcpdump/libpcap. 2002. The Tcpdump Group.

[6] M. V. Mahoney. Network traffic anomaly detection based on packet bytes. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 346–350, New York, NY, USA, 2003. ACM Press.

[7] M. V. Mahoney and P. K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *KDD '02: Proceedings of the*

eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 376–385, New York, NY, USA, 2002. ACM Press.

[8] See homepage for details. Atlas homepage. http://math-atlas.sourceforge.net/.

[9] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[10] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. 2004.