Implentation of an Application for Intrusion Detection on Network Virtual Machine

Relatori:
Prof. Mario Baldi
Prof. Fulvio Risso

Candidato:
Giorgio Giuseppe Moscardi

Aprile 2007
A Martino,
al mondo in cui viviamo,
e alla vita in generale
Ringraziamenti

Desidero ringraziare l’intero NetGroup del Politecnico di Torino, in particolare il Prof. Fulvio Risso e l’Ing. Olivier Morandi per il continuo aiuto fornitomi durante lo svolgimento di questo lavoro.

Ringrazio i miei genitori per aver avuto pazienza e fiducia in me.

Vorrei inoltre esternare la mia gratitudine a tutti coloro che mi sono stati vicini durante questi mesi di lavoro, sostenendomi e aiutandomi a portarlo a termine. Un ringraziamento particolare va agli ex-colleghi del laboratorio LABinf del Politecnico di Torino, Nicolò Nepote, Paolo Margara e Flavio Castagno, per il supporto e il tempo che mi hanno permesso di dedicare a questo impegno.

Infine ringrazio l’amico Dott. Ing. Alessandro Giusta per essersi occupato della messa su carta di questo lavoro, e l’amico Dott. Stefano Gazzaniga per avermi fornito l’hardware usato durante i test.
# Table of contents

1 Introduzione ................................. 6
   1.1 I sistemi per il rilevamento delle intrusioni ................. 6
       1.1.1 Snort ........................................ 7
   1.2 NetVM ............................................. 9
       1.2.1 L'architettura NetVM ................................. 9
       1.2.2 Stato dell'arte .................................. 10
   1.3 Il problema ....................................... 10
   1.4 Scelte progettuali ............................... 11
       1.4.1 Architettura ...................................... 11
       1.4.2 Trasporto dei dati ................................ 13
       1.4.3 Codice statico e dinamico ......................... 14
   1.5 Implementazione .................................. 15
       1.5.1 Il parser delle regole ............................... 16
       1.5.2 Il modulo di cattura ............................... 16
       1.5.3 Il modulo Ethernet ................................ 16
       1.5.4 Il modulo IPv4 ..................................... 16
       1.5.5 Il modulo TCP ..................................... 17
       1.5.6 Il modulo UDP ..................................... 17
       1.5.7 Il modulo ICMP ................................... 18
       1.5.8 Il modulo di string-matching ....................... 18
       1.5.9 Il modulo di connection tracking ................... 18
       1.5.10 Il modulo payload ................................. 20
       1.5.11 Il modulo di uscita ............................... 20
   1.6 Limitazioni e alternative ......................... 20
   1.7 Modifiche a NetVM ............................... 20
       1.7.1 Exchange buffer ................................... 20
       1.7.2 Coprocessori ..................................... 21
       1.7.3 NetPE nativi ..................................... 21
1.8 Testing .................................................. 22
1.9 Lavori futuri ........................................... 22

2 Introduction ............................................. 24
  2.1 State of the art ...................................... 24
    2.1.1 Intrusion detection systems ......................... 24
    2.1.2 NetVM ........................................... 25
  2.2 The problem ......................................... 25
  2.3 Work organization .................................. 26

3 Intrusion detection systems ......................... 27
  3.1 Snort ............................................... 28
    3.1.1 Snort internals .................................. 29
    3.1.2 Anatomy of Snort rules............................ 32
    3.1.3 Rule examples ................................... 34
  3.2 Improving the speed of current IDSs ................ 35
    3.2.1 CAM solutions ................................... 35
    3.2.2 FPGA solutions .................................. 36
    3.2.3 Bloom filter solutions............................ 36

4 NetVM ................................................... 37
  4.1 Why a Virtual Machine-based approach ................ 38
  4.2 The NetVM architecture .............................. 38
    4.2.1 The Processing Element ............................ 39
    4.2.2 Memory .......................................... 39
    4.2.3 Stack ............................................ 41
    4.2.4 Connections ...................................... 41

5 Implementation ........................................ 43
  5.1 Design decisions .................................... 43
    5.1.1 Processing elements interconnection ............... 44
    5.1.2 Architecture ..................................... 45
    5.1.3 Data transport ................................... 46
    5.1.4 Static vs. dynamic code ............................ 47
  5.2 The NetVM-Snort architecture ....................... 47
    5.2.1 Overview ........................................ 47
    5.2.2 The rules parser .................................. 48
    5.2.3 The capture and input module ....................... 51
    5.2.4 The Ethernet module .............................. 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2 The Boyer-Moore algorithm</td>
<td>87</td>
</tr>
<tr>
<td>B.2.1 The good-suffix shift</td>
<td>88</td>
</tr>
<tr>
<td>B.2.2 The bad-character shift</td>
<td>88</td>
</tr>
<tr>
<td>B.2.3 Memory and time requirements</td>
<td>89</td>
</tr>
<tr>
<td>B.2.4 Example implementation</td>
<td>90</td>
</tr>
<tr>
<td>B.3 The Turbo Boyer-Moore algorithm</td>
<td>91</td>
</tr>
<tr>
<td>B.3.1 Example implementation</td>
<td>92</td>
</tr>
<tr>
<td>B.4 The Horspool algorithm</td>
<td>94</td>
</tr>
<tr>
<td>B.4.1 Example implementation</td>
<td>94</td>
</tr>
<tr>
<td>Bibliography</td>
<td>96</td>
</tr>
</tbody>
</table>
Capitolo 1

Introduzione

Al giorno d’oggi, i sistemi per il rilevamento delle intrusioni, comunemente detti IDS, dall’inglese Intrusion Detection System, costituiscono un componente essenziale delle reti di computer. Infatti, la diffusione delle connessioni ad Internet ad alta velocità causa un continuo aumento del numero di utenti e servizi, il che risulta in una continua crescita del numero di possibili attaccanti e bersagli. Con queste premesse, la quantità di lavoro che deve essere svolto da un IDS aumenta giorno dopo giorno, e gli IDS devono migliorare le proprie prestazioni altrettanto costantemente. Purtroppo, siamo ormai arrivati ad un punto in cui è difficile migliorare ulteriormente questi programmi, almeno finché rimaniamo nel campo delle implementazioni puramente software.

Per questo motivo, gli ultimi sforzi nel campo sono stati orientati verso l’implementazione in hardware degli IDS, o almeno delle loro parti computazionalmente più intense. Tuttavia, il passaggio non è scontato, dato che il riutilizzo di codice preesistente non è facilmente realizzabile, a causa delle enormi diversità che presenta l’hardware in commercio, sia per quanto riguarda l’architettura, che per quanto riguarda il metodo di programmazione. Spesso, la soluzione più pratica è quella di ricominciare il lavoro da capo, “reinventando la ruota” per ogni dispositivo.

Il framework NetVM, sviluppato al Politecnico di Torino, ha come scopo la risoluzione di questo problema. Attraverso l’astrazione di un network processor e la definizione di un set di istruzioni molto specifico, orientato al processamento di pacchetti, si propone di ottenere la portabilità del codice su un buon numero di dispositivi hardware e software, pur mantenendo una certa efficienza.

1.1 I sistemi per il rilevamento delle intrusioni

I sistemi per il rilevamento delle intrusioni cercano di rilevare attività anomale all’interno di una rete di computer, spesso effettuate guadagnando l’accesso a un sistema remoto attraverso Internet. Tali attività possono essere di diversi tipi: attacchi verso applicazioni, nel tentativo di alterarne o impedirne l’operazione, tentativi di guadagnare il controllo completo della macchina, attraverso
la cosiddetta privilege escalation, scaricamento di file riservati, e ancora altri. Questi attacchi, tipicamente, non possono essere rilevati dai firewall, in quanto spesso si basano sullo sfruttamento di bachi noti di un’applicazione, portato a termine attraverso l’invio di dati formattati in un particolare modo attraverso i suoi normali canali di funzionamento. Ad esempio, è possibile sfruttare un baco di un server web inviando una richiesta HTTP contenente un determinato indirizzo, che un firewall lascerà sicuramente passare.

Un IDS generico è composto da diversi componenti. I sensori si occupano di campionare il traffico, generando di conseguenza eventi, che vengono inviati a un motore centrale, che rileva le anomalie presenti nel traffico basandosi su un insieme di regole impostate in precedenza. L’intero sistema è controllato attraverso una console, alla quale vengono anche inviati i risultati dell’analisi e, quindi, le segnalazioni degli attacchi rilevati. Nella pratica, tali componenti sono spesso combinati all’interno di un’unica entità.

Esistono molte modalità per catalogare gli IDS, ad esempio basandosi sul tipo e sulla posizione dei sensori, oppure sul funzionamento del motore di rilevamento. Per gli scopi di questa tesi, focalizzeremo la nostra attenzione sugli IDS di rete, detti NIDS, Network IDS. Tali sistemi lavorano all’interno di una rete, campionando il traffico attraverso un hub, uno switch configurato per effettuare il port mirroring o attraverso un network tap. In questo modo riescono a tenere sotto controllo tutti gli host della rete, analizzando il traffico destinato a ciascuno di essi e conoscendo i servizi da loro erogati.

Gli IDS possono essere passivi o reattivi: i primi, una volta rilevata un’intrusione, si limitano a segnalare la cosa inviando un avviso alla console, mentre i secondi cercano di prendere provvedimenti per bloccarla, interagendo con il firewall o chiudendo la connessione.

### 1.1.1 Snort

Snort [1] è un’implementazione software di un IDS di rete passivo. È stato originariamente sviluppato da Martin Roesch, che ha poi fondato un’azienda, la Sourcefire, Inc. [6], per ampliarne lo sviluppo. Ciononostante, Snort è software libero, pubblicato sotto la copertura della licenza GNU GPL, anche se Sourcefire ne vende versioni proprietarie ad alcune aziende, corredandole con servizi di supporto.

A quasi un decennio dalla sua prima pubblicazione, Snort è diventato lo standard di fatto per il rilevamento delle intrusioni. Possiede numerose funzionalità, tra cui menzioniamo l’analisi del traffico IP in tempo reale, attraverso la ricerca di contenuti specifici, cosa che permette di rilevare quasi tutte le tipologie di attacco, come le scansioni delle porte, i buffer overflow, i tentativi di fingerprinting del sistema operativo, e molti altri.
Architettura di Snort

Internamente, Snort è composto da parecchi strati. Il traffico viene catturato dalla rete utilizzando libpcap [10], e viene quindi inviato al decodificatore: il compito di questo componente è principalmente quello di rilevare la struttura del pacchetto e la pila di protocolli che esso contiene, inizializzando un insieme di puntatori che permetteranno di accedere rapidamente a ciascuno di essi nelle fasi successive dell’analisi.

Dopo il decodificatore, i pacchetti passano attraverso una serie di preprocessori, che li alterano al fine di rendere possibili certi tipi di analisi, ed effettuano determinati test al fine di rilevare alcuni tipi di intrusione immediatamente riconoscibili. Altri compiti effettuati dai preprocessori sono il riassemblaggio dei flussi TCP e la deframmentazione di IP.

Successivamente, i pacchetti arrivano al motore di rilevamento, che verifica se il pacchetto soddisfa una o più delle regole impostate, e invia i risultati ai plugin di uscita.

I plugin di uscita forniscono i risultati dell’analisi all’operatore. Il metodo utilizzato varia a seconda del plugin, ed è possibile persino inviare avvisi per posta elettronica o mediante messaggi SMS, oppure inserirli in un database.

Le regole

Il formato utilizzato da Snort per le regole è molto potente e flessibile, sebbene mantenga una semplicità di fondo, che fa sì che le regole possano essere scritte su un’unica riga.

Logicamente, le regole sono divise in due parti: l’intestazione e le opzioni. Vediamo un esempio:

```
log tcp any any -> 10.1.1.0/24 80 (content: "GET"; msg: "Webserver access");
```

L’intestazione è costituita dalla parte prima delle parentesi. Essa contiene l’azione da effettuare nel caso che la regola sia soddisfatta da un pacchetto, il protocollo e, infine, gli indirizzi e le porte sorgente e destinazione che devono essere presenti nel pacchetto. Le opzioni, invece, sono racchiuse tra le parentesi, e specificano una serie di test da effettuare sulle varie parti del pacchetto, al fine di verificare completamente la regola, oltre che i parametri per contestualizzare gli eventuali avvisi generati.

La versione corrente di Snort supporta diverse azioni, ma le più comunemente utilizzate sono `alert`, utilizzata per inviare un avviso diretto alla console, `log`, che invece effettua semplicemente il logging del pacchetto, e `pass`, che fa sì che il pacchetto sia ignorato. Quanto ai protocolli, Snort supporta quelli comunemente utilizzati su Internet, ossia `tcp`, `udp` e `icmp`.

Quanto a indirizzi e porte, sono supportate le sintassi comunemente utilizzate, inclusa la notazione `CIDR` per specificare un’intera sottorete IP. Sia l’indirizzo che la porta possono essere specificati come `any`, col risultato che ogni indirizzo/porta soddisferà il parametro.
1.2 NetVM

Il framework *Network Virtual Machine (NetVM)* nasce dall’idea di ottenere la portabilità e l’efficienza delle applicazioni che effettuano processamento intensivo di pacchetti su diverse piattaforme software e hardware, attraverso l’astrazione delle piattaforme stesse. Tale idea è la naturale estensione del concetto di “macchina virtuale” che si è già affermato nel mondo dei processori *general purpose*, mediante implementazioni come la *JVM* di *Sun Microsystems* [21] o la *CLR* di *Microsoft* [22]. Mentre le macchine virtuali citate rappresentano un ambiente ideale per l’esecuzione di applicazioni generiche, esse risultano poco adatte al processamento di pacchetti di rete. Infatti, le operazioni eseguite da quest’ultima classe di applicazioni sono piuttosto ristrette e molto specifiche (ad esempio, gli shift logici e le operazioni bit a bit sono molto ricorrenti) e, di conseguenza, una macchina virtuale che le supporti nativamente offrirebbe sicuramente prestazioni migliori per queste. Di conseguenza, astraendo l’architettura di un network processor, piuttosto che quella di un processore general-purpose, si dovrebbe raggiungere un’ottima efficienza sia sui comuni PC che sui network processor reali.

La precisa definizione dell’architettura e del set di istruzioni permette di ottenere la portabilità a livello di bytecode, e non solo di codice sorgente: il codice compilato una sola volta può essere eseguito su diverse piattaforme, e l’utilizzo di hardware dedicato che implementa alcune funzioni è totalmente trasparente al programmatore, cosa che permette di dedicare l’attenzione alla scelta della piattaforma, piuttosto che a come essa vada programmata. Le funzioni non supportate nativamente possono essere emulati, garantendo comunque la piena compatibilità del bytecode. Allo stesso modo, si velocizza anche lo sviluppo delle applicazioni, dato che il codice può essere scritto e testato a dovere su un semplice PC, e spostato su hardware dedicato solo al termine della fase di sviluppo, senza nessuna modifica e senza bisogno di ricompilare alcunché.

1.2.1 L’architettura NetVM

L’architettura NetVM è costruita attorno al concetto di *processing element (NetPE)*, che virtualizza un *microengine* di un network processor. Una tipica applicazione NetVM è costituita da diversi NetPE, ciascuno dei quali implementa una singola funzionalità. Applicazioni complesse possono essere ottenute interconnettendo diversi NetPE.

Questa visione modulare deriva dall’osservazione che la maggior parte delle applicazioni di rete possono essere decomposte in molte semplici operazioni. Spesso, le operazioni possono essere parallelizzate o eseguite in pipeline per raggiungere prestazioni migliori. La bontà di questo approccio è già stata confermata da altri software, come ad esempio *Netfilter* [8] e *Click* [23], e gli stessi network processor sono spesso organizzati secondo un’architettura simile.

Internamente, i NetPE sono macchine basate su stack e non possiedono, quindi, registri. Questo conferisce portabilità, semplicità di implementazione e compattezza del set di istruzioni. Di
conseguenza, le istruzioni non accedono direttamente alla memoria, bensì operano solo su valori contenuti sullo stack. L’interfacciamento con la memoria avviene unicamente tramite una serie di istruzioni di load e store. Altre istruzioni permettono la copia diretta di aree tra memorie diverse.

Il modello di esecuzione è basato su eventi, che possono essere l’arrivo di un nuovo pacchetto su una porta d’ingresso, la richiesta di un pacchetto da una porta di uscita o lo scadere di un timer. Eventi diversi possono essere fatti corrispondere a diversi segmenti di codice.

La memoria è divisa in quattro segmenti, al fine di separare i diversi tipi di memoria e di garantire efficienza nell’accesso. Ogni NetPE possiede una propria memoria di codice, una propria memoria per i dati e un proprio stack, e può inoltre accedere ad una memoria condivisa da tutti i NetPE, al fine di scambiare dati. Lo scambio di dati avviene, però, più frequentemente tramite gli exchange buffer: si tratta di piccole porzioni di memoria che in ogni istante sono di proprietà esclusiva di un unico NetPE, che ha quindi accesso indiscritto ad esse. Gli exchange buffer possono essere inviati o richiesti agli altri NetPE, creando così un flusso di dati all’interno dell’applicazione.

1.2.2 Stato dell’arte

All’inizio del presente lavoro di tesi, il framework NetVM aveva raggiunto lo stadio alpha. La libreria presentava alcune problematiche, ma era comunque utilizzabile e quasi completa nelle funzionalità offerte. Esistevano alcuni programmi di esempio, ma erano tutti piuttosto basilari e più orientati al debug, piuttosto che a mostrare la potenza della libreria.


Alcune funzionalità, come le eccezioni e i coprocessori, erano previste nell’architettura, ma non ancora implementate. Inoltre, non era ancora stato effettuato il debug di tutte le istruzioni.

1.3 Il problema

In sostanza, la NetVM si trovava ad uno stadio in cui necessitava un testing severo e, soprattutto, la validazione della sua architettura. Questo poteva essere ottenuto mediante l’implementazione di un’applicazione complessa, che dimostrasse, da un lato, l’effettiva usabilità del framework e, da un altro, la sua efficienza. Abbiamo quindi pensato che un clone di Snort fosse l’applicazione ideale per questi scopi: infatti, Snort esegue molte operazioni sui campi delle intestazioni dei protocolli contenuti nei pacchetti, per svolgere il suo compito, e trova quindi nella NetVM il suo ambiente di esecuzione ideale. Inoltre, esegue anche altri test meno specifici, che permetteranno di valutare quanto bene la NetVM si adatti a compiti per i quali non è stata espressamente progettata. Ovviamente, perché un IDS funzioni bene, è necessario che le sue prestazioni siano elevate, quindi potremo verificare anche l’efficienza globale del framework.
Da un altro punto di vista, Snort stesso potrebbe trarre beneficio dall’implementazione, dato che guadagnerà la portabilità istantanea su qualunque dispositivo (per cui esista un’implementazione della NetVM), cosa che potrebbe affermarlo ulteriormente come standard per il rilevamento delle intrusioni.

1.4 Scelte progettuali

Ovviamente, implementare un clone di Snort è un lavoro complesso, che necessita scelte implementative ben ponderate. In realtà, la maggior parte delle scelte sono guidate dalla necessità di raggiungere prestazioni elevate, dato che le analisi di un IDS devono essere compiute praticamente in tempo reale, e la velocità delle reti aumenta di continuo. Nella pratica, prestazioni migliori si ottengono usando più memoria: spesso ci troveremo, quindi, costretti a sacrificare memoria, per guadagnare velocità.

Il primo interrogativo da porsi è se implementare un clone il più possibile esatto di Snort, copiandolo nei minimi particolari, o, se, piuttosto, proporsi di realizzare, in prima approssimazione, un IDS dalle funzionalità ridotte, che raggiunga comunque un buon grado di usabilità, mantenendo una certa capacità di estensione in futuro. Abbiamo optato per quest’ultima soluzione, per svariati motivi. Il principio era la probabile necessità di modifiche al framework NetVM stesso, cosa che avrebbe sdoppiato il lavoro su due fronti, e che ci ha obbligato, dunque, a cercare di ridurlo su uno. Abbiamo quindi definito un insieme di caratteristiche minime, necessarie a raggiungere i nostri scopi. Ad esempio, abbiamo deciso di lasciare perdere il riassembraggio dei flussi TCP e la deframmentazione di IP, il che ci ha permesso di trascurare l’implementazione dei preprocessori di Snort. Allo stesso modo, per i primi test non servirà un sistema di output comprensivo come quello di Snort, ma sarà sufficiente un formato arbitrario, facilmente implementabile, senza pretese di raggiungere tutte le funzionalità garantite dai plugin di output di Snort. Una priorità, invece, è rappresentata dalla decisione di essere compatibili con il formato delle regole usato da Snort, che rappresenta il metodo più utilizzato per la configurazione dei moderni IDS.

Un’altra idea cardine del lavoro è quella di non sentirci limitati dall’implementazione attuale di NetVM, ma di essere pronti a cambiarla e a estenderla, qualora la nostra applicazione possa trarre giovamento dall’operazione.

1.4.1 Architettura

Una volta prese queste decisioni, resta da stabilire come implementare ogni componente di Snort sulla NetVM. Dato che NetVM segue l’approccio della suddivisione del lavoro in compiti semplici, il primo passo consiste nell’identificare tali compiti. Essendo Snort essenzialmente un analizzatore di protocolli, abbiamo identificato i compiti con i vari protocolli potenzialmente presenti nei pacchetti, in prima approssimazione.
La nostra idea è quella di far attraversare ai pacchetti da analizzare una catena di NetPE (come mostrato in figura 1.1), ognuno dedicato a un singolo protocollo, con l’incarico di svolgere due compiti: il primo consiste nell’analizzare il protocollo, estraendo dal pacchetto dati che possono essere utili ai NetPE successivi, mentre il secondo consiste nel verificare quali regole sono soddisfatte dal pacchetto in questione. Per sottolineare la modularità dell’approccio, da questo momento ci riferiremo ai vari NetPE come moduli.

Figura 1.1. Un possibile modo per interconnettere i moduli dell’applicazione

Ovviamente, essendo limitato ad un solo protocollo, un modulo non può decidere da solo sulla veridicità o meno di un’intera regola; si rende quindi necessario un qualche metodo per coordinare i risultati forniti da tutti i vari moduli. Questo si può realizzare facilmente, osservando che, per come sono fatte le regole di Snort, affinché esse siano completamente verificate, ogni singola condizione specificata in esse deve essere verificata, il che equivale a dire che esiste un AND logico implicito tra tutte le opzioni della regola. Basta quindi applicare qualche concetto basilare di algebra booleana per comprendere che una singola opzione non verificata causa la falsità dell’intera regola. Il modo di procedere che ne consegue consiste quindi nel partire considerando tutte le regole come verificate, e impostarle come non verificate appena un modulo rileva la falsità di un’opzione.

In realtà, il modello a catena proposto precedentemente è soggetto a una semplice ottimizzazione. Infatti, se trascuriamo il caso del tunneling dei pacchetti, possiamo constatare che certi moduli sono mutualmente esclusivi. Ad esempio, un pacchetto normalmente contiene un unico protocollo di livello rete (IPv4 o IPv6) e un unico protocollo di livello trasporto (TCP, UDP o ICMP). Possiamo quindi riorganizzare le interconnessioni tra i moduli, ottenendo una struttura a quattro livelli, come mostrato in figura 1.2.

Questa architettura porta diversi vantaggi:

- **Ogni protocollo viene analizzato una volta sola.**

- **I dettagli interni di un protocollo sono racchiusi in un unico componente.** Questo rende semplice il debug e il miglioramento della gestione di un protocollo, e l’aggiunta del supporto per nuovi protocolli.
1 – Introduzione

- Il numero di NetPE attraversati è contenuto. Questo porta a una maggiore efficienza: più corto è il percorso, maggiori sono le prestazioni.

- L’architettura può essere messa in pipeline. Anche se al momento l’applicazione funziona in modo sequenziale, può potenzialmente gestire un pacchetto per strato, aumentando il throughput di un fattore quattro.

1.4.2 Trasporto dei dati

Per fare funzionare il nostro modello, abbiamo bisogno di poter trasportare all’interno della NetVM altri dati, oltre al pacchetto da analizzare. Ad esempio, abbiamo i dati che ogni modulo estrae dal pacchetto, e, soprattutto, dobbiamo tenere traccia dello stato di verifica delle regole. Per raggiungere questo obiettivo, abbiamo proposto ed implementato una modifica agli exchange buffer, che, sostanzialmente, li divide in due parti, come mostrato in figura 1.3. La prima parte contiene il pacchetto, e viene detta, perciò, Packet buffer, mentre la seconda è a disposizione dei NetPE per “appendere” dati arbitrari al pacchetto, e viene definita Info partition.

Come si evince dalla figura, la nostra architettura utilizza i primi $N$ byte della Info partition per tracciare lo stato delle $N$ regole, utilizzando un valore in logica negativa: 0 significa che la regola è verificata, mentre 1 vuol dire che almeno un’opzione non è stata verificata, e quindi l’intera...
regola risulta falsa. La giustificazione per questo utilizzo è che ci permette di risparmiare il tempo necessario all’inizializzazione di tutte le regole come verificate, situazione che rappresenta la nostra condizione di partenza, come abbiamo visto in precedenza. Infatti, il framework garantisce che la Info partition sia completamente inizializzata a 0 quando un pacchetto fa il suo ingresso nella NetVM.

Seguono poi un certo numero di byte di riempimento, per raggiungere l’allineamento su 32 bit, e quindi una serie di campi, tutti di dimensione 32 bit, che sono utilizzati dai vari moduli per salvare i dati che estraggono dai pacchetti. L’assegnazione dei campi è statica, e stabilita nel codice, anche se l’aggiunta di nuovi campi è relativamente semplice.

Dato che la struttura della Info partition deve essere conosciuta da tutti i moduli, e dato che dipende dal numero di regole, essa viene stabilita una volta per tutte subito dopo la lettura e la validazione delle regole, e comunicata ai vari moduli durante la loro inizializzazione.

1.4.3 Codice statico e dinamico

Il codice che ogni NetPE si troverà ad eseguire può essere generato in due modi. Ad esempio, può essere codice scritto a mano una volta per tutte, che legge i parametri delle regole dalla memoria e li verifica sul pacchetto. Questo approccio è più semplice da seguire, ma diminuisce la velocità di esecuzione, dato che richiede un continuo accesso alla memoria.

Alternativamente, il codice può essere generato dinamicamente a partire dalle regole, mediante una sorta di “compilatore”. Questo è ovviamente più complesso, ma risulta in codice più snello,
1 – Introduzione

non richiedendo nessun accesso in memoria, dato che i parametri delle regole risultano incorporati
nel codice stesso.

Per la nostra applicazione abbiamo scelto il secondo approccio, dato che uno dei nostri obiettivi è il raggiungimento di elevate prestazioni.

1.5 Implementazione

Il flusso di esecuzione del programma segue il diagramma mostrato in figura 1.4.

![Diagramma di flusso di NetVMSnort](image)

**Figura 1.4. Diagramma di flusso di NetVMSnort**

Il primo passo consiste nella lettura dei file di configurazione, con la conseguente memorizzazione
delle regole in memoria. Segue quindi l’inizializzazione della NetVM, con la creazione di tutti i NetPE e la generazione del codice per ciascuno di essi, attraverso la compilazione delle regole. A questo punto, con l’inizializzazione di libpcap si completa la fase di preparazione, e si passa al ciclo principale del programma, in cui si catturano i pacchetti dalla rete e si inviano alla NetVM, che li processa, fornendo l’elenco delle regole verificate. L’elenco viene elaborato dal modulo di uscita, che esegue le azioni specificate nelle regole.

Nelle sezioni successive vedremo in dettaglio il funzionamento di ogni componente.

1.5.1 Il parser delle regole

Dato che uno dei nostri scopi è essere compatibili col formato delle regole utilizzato da Snort, il primo componente necessario è un parser in grado di interpretarlo. Abbiamo quindi utilizzato due strumenti tipici per la scrittura di parser, flex e bison, e ne abbiamo messo a punto uno.

Il parser costruito è in grado di interpretare completamente la sintassi definita da Snort, scartando automaticamente le direttive non significative per la nostra implementazione (come quelle per la configurazione di preprocessori e plugin di uscita). Le regole vengono memorizzate in apposite strutture in memoria, e una serie di controlli viene effettuata al fine di accertare il pieno supporto della regola. Le regole che specificano opzioni non supportate vengono automaticamente scartate. In questo modo, la nostra applicazione può selezionare un sottoinsieme delle regole di Snort senza intervento manuale, e con la garanzia che le regole mantengano il corretto significato semantico.

1.5.2 Il modulo di cattura

Il modulo di cattura utilizza libpcap per catturare i pacchetti dalla rete. Si tratta, in realtà, di una soluzione temporanea, necessaria fino a quando il framework NetVM non definirà una propria astrazione delle porte di ingresso, e sarà quindi soggetto a rimozione nelle versioni future dell’applicazione.

1.5.3 Il modulo Ethernet

Il modulo Ethernet si occupa essenzialmente di filtrare i pacchetti che non contengono IPv4 (che, per ora, è l’unico protocollo di livello rete supportato), e di decodificare l’offset al quale IPv4 inizia. Tali compiti sono molto semplici, dato che il formato dell’header Ethernet è fisso, e non esistono opzioni di Snort relative al livello data-link.

1.5.4 Il modulo IPv4

Contrariamente al modulo Ethernet, il modulo IPv4 è uno dei più complessi, nonché uno di quelli sottoposti al maggior carico di lavoro. Esso rappresenta, infatti, un passaggio obbligato per tutti
i pacchetti, e la sua operazione deve quindi essere estremamente ottimizzata. Esso si occupa di verificare se i pacchetti provengono da e vanno verso le sottoreti specificate in ciascuna regola. Per svolgere questo compito velocemente, abbiamo utilizzato una serie di espedienti.

- La prima operazione consiste nel leggere gli indirizzi sorgente e destinazione contenuti del pacchetto, salvandoli in quelle che il framework NetVM definisce *locals*: si tratta di variabili locali, simili ai registri dei processori, e quindi molto veloci. In questo modo possiamo evitare l’accesso al pacchetto per ogni regola.

- Successivamente si esegue il testing delle regole, raggruppandole per indirizzo destinazione. Questo significa che possiamo effettuare un unico controllo per tutte le regole che specificano la stessa destinazione. Abbiamo deciso di utilizzare l’indirizzo destinazione al posto di quello sorgente dopo aver osservato che, nella pratica, la maggior parte delle regole hanno un indirizzo sorgente *any*, mentre specificano un indirizzo di destinazione ben preciso, che corrisponde al server che fornisce un certo servizio all’interno della rete monitorata.

In questo modo, il numero di test da effettuare si riduce notevolmente, garantendo prestazioni accettabili anche con un gran numero di regole.

Il modulo IPv4 si occupa anche di eliminare tutti i pacchetti IP frammentati, senza processarli in nessun modo. Questo è necessario perché, come abbiamo visto, al momento il riassemblaggio dei frammenti IP non è previsto dalla nostra applicazione, e tali pacchetti potrebbero causare dei falsi positivi se la loro elaborazione proseguisse.

Quando il modulo IPv4 ha terminato la sua serie di controlli, analizza il pacchetto per determinare il protocollo di trasporto contenuto nel pacchetto, e smista l’exchange buffer al modulo appropriato.

1.5.5 Il modulo TCP

Il modulo TCP è essenzialmente simile a quello IPv4, e utilizza espedienti simili per ridurre il numero di test da effettuare. Risulta, tuttavia, leggermente più semplice in quanto non ha la complicazione della gestione delle netmask.

Al termine delle operazioni, questo modulo invia il pacchetto al modulo di connection tracking.

1.5.6 Il modulo UDP

Il modulo UDP è virtualmente identico a quello TCP. L’unica differenza di rilievo è l’invio diretto del pacchetti, al termine dell’operazione, al modulo di string-matching, non prevedendo Snort opzioni sui pacchetti UDP correlati. Per questo motivo, il processamento di un pacchetto UDP risulta più veloce, rispetto a quello di un pacchetto TCP.
1.5.7 Il modulo ICMP

Il modulo ICMP utilizza un algoritmo “ingenuo”, per motivi di complessità di implementazione. Infatti, esso rappresenta il primo esempio di modulo il cui comportamento è definito da una serie di opzioni delle regole, che possono essere presenti o meno, ed essere combinate in modo arbitrario. Inoltre, alcune opzioni hanno una sintassi piuttosto flessibile, il che complica ulteriormente la situazione.

Per questi motivi, abbiamo deciso di trattare ogni regola ICMP indipendentemente dalle altre, effettuando in sequenza tutti i test necessari. Questo approccio è accettabile per questo modulo, dato che i pacchetti ICMP rappresentano solitamente una frazione minore del traffico di rete, e, inoltre, anche le regole che trattano ICMP sono solitamente in numero esiguo (ad esempio, nell’insieme di regole predefinito di Snort sono solo 127 su più di 3000).

Come il modulo UDP, terminato il suo compito, questo modulo invia il pacchetto al modulo di string-matching.

1.5.8 Il modulo di string-matching

Il modulo di string-matching rappresenta il cuore di un IDS. Come tale, è quello in cui l’ottimizzazione è più importante, anche perché lo string-matching stesso è un’operazione intrinsecamente complessa. Questa è la ragione per cui un intero modulo è stato dedicato a questa operazione, sebbene logicamente rientri tra i test da effettuare sul payload dei pacchetti.

Nell’implementazione di questo modulo abbiamo deciso di seguire il metodo utilizzato dal ramo 1.x di Snort, utilizzando l’algoritmo Boyer-Moore. Sinteticamente, tale algoritmo presenta un buon rapporto prestazioni/memoria utilizzata, e trova il suo ambiente di utilizzo ideale con il tipo di pattern con cui abbiamo a che fare. D’altro canto, il ramo 2.x di Snort è passato a utilizzare un algoritmo basato su un automa a stati finiti, che risulta più veloce, ma utilizza una quantità di memoria molto maggiore. La nostra scelta è stata guidata più che altro dalla presenza di codice già scritto che implementava l’algoritmo Boyer-Moore, sebbene sia stato necessario apportare ad esso alcune modifiche, per la gestione di pattern arbitrari (mentre gli algoritmi già esistenti prevedevano che i pattern fossero stringhe in stile C) e per la gestione dell’opzione nocase, che rende l’algoritmo insensibile alla differenza tra caratteri maiuscoli e minuscoli.

1.5.9 Il modulo di connection tracking

Il modulo di connection tracking deve tenere traccia di tutte le connessioni TCP, permettendo di identificare la connessione a cui appartiene ogni pacchetto e lo stato in cui essa si trova, in modo da risalire alla direzione che esso sta seguendo (dal client al server o viceversa).

Abbiamo deciso di suddividere il modulo in due sotto-moduli:
Il primo implementa essenzialmente il diagramma degli stati TCP definito nella RFC 793 [27], che è mostrato in figura 1.5, utilizzando una hash table: per ogni pacchetto viene eseguito l’hash degli indirizzi e delle porte sorgente e destinazione, ottenendo così una posizione in una tabella nella quale viene mantenuto lo stato della connessione. Questa parte è stata isolata...
1.5.10 Il modulo payload

Al momento, il modulo payload si occupa unicamente di verificare la lunghezza del payload, come definito dall’opzione \textit{dsize}. Dato che i parametri di tale opzione sono molto variabili, questo modulo tratta ogni regola indipendentemente dalle altre, e non le raggruppa in nessun modo.

Il modulo rappresenta la posizione ideale in cui implementare ulteriori opzioni che riguardano il payload, qualora l’applicazione venga ulteriormente estesa.

1.5.11 Il modulo di uscita

Il modulo di uscita si trova all’esterno della NetVM, e il suo scopo è l’esecuzione delle azioni previste dalle regole. Per svolgere il suo compito, il modulo utilizza i dati forniti dagli altri moduli presenti nella info partition.

1.6 Limitazioni e alternative

L’architettura scelta, sebbene conferisca i vantaggi già citati, presenta anche qualche punto negativo. Il problema maggiore consiste nell’impossibilità di parallelizzare completamente le operazioni compiute dall’applicazione, dato che ogni modulo dipende da quelli che lo precedono. Questo problema si potrebbe aggirare utilizzando un modulo iniziale che calcoli tutti i dati necessari ai vari moduli, ma questo violerebbe l’isolamento dei protocolli in un unico elemento, cosa che complicherebbe l’estensione dell’applicazione. Vale comunque la pena sottolineare che l’applicazione può essere eseguita in quattro stadi di pipeline.

Un secondo problema è che l’architettura presentata non supporta il tunneling dei pacchetti, ma si tratta di un problema minore, aggirabile con modifiche minori al codice e alle interconnessioni.

1.7 Modifiche a NetVM


1.7.1 Exchange buffer

Prima del nostro lavoro, l’exchange buffer era denominato \textit{packet buffer}. Infatti, esso era inizialmente pensato per contenere il solo pacchetto, anche se le specifiche stabilivano che poteva essere
utilizzato per dati arbitrari. Tuttavia, la coesistenza del pacchetto e di dati ad esso correlati non era facilmente ottenibile. L’utilizzo della memoria condivisa era un’alternativa a questo, ma era comunque una soluzione complessa, data la mancanza di un sistema di coordinamento degli accessi, dovuta al fatto che NetVM vede la memoria condivisa più come un modo di condividere dati statici globali.

Abbiamo quindi risolto il problema dividendo, come già detto, l’exchange buffer in due parti, una dedicata al pacchetto e l’altra ai dati. L’exchange buffer rimane comunque una struttura unica, col risultato che i dati “seguono” il pacchetto.

1.7.2 Coprocessori

Le specifiche originali NetVM prevedevano che le funzionalità fornite da hardware specializzato fossero esposte all’interno della NetVM da entità definite coprocessori. Durante lo sviluppo ci siamo resi conto che il concetto di “coprocessore” poteva portare a due diverse implementazioni.

La prima è l’astrazione più immediata dei coprocessori che si trovano comunemente nei dispositivi hardware. L’interazione con essi avviene attraverso un certo numero di registri, e l’invocazione assomiglia al richiamo di un interrupt software.

Abbiamo quindi implementato l’architettura proposta per questi coprocessori, e li abbiamo utilizzati per realizzare un coprocessore di lookup, su cui si basa il funzionamento del modulo di connection tracking.

1.7.3 NetPE nativi

Un problema intrinseco dei coprocessori è che non hanno modo di accedere all’exchange buffer, cosa che impedirebbe, ad esempio, l’implementazione di un coprocessore che controlli il checksum di un pacchetto IP.

Per ovviare a questo, abbiamo definito un altro tipo di coprocessori, che esternamente presentano la stessa interfaccia di un NetPE, e hanno quindi accesso a tutti gli elementi della NetVM. Abbiamo definito tali coprocessori NetPE nativi (NativePE), dato che sembrano di fatto NetPE, ma sono programmati in linguaggio nativo.

I NativePE sono utili anche per il debug e per la prototipazione di applicazioni, permettendo la programmazione in C della NetVM. Tuttavia, bisogna notare che non esiste un mapping diretto di questo tipo di coprocessori su hardware, e quindi è necessaria una definizione più precisa.

Nella nostra applicazione, abbiamo utilizzato i NativePE per implementare il modulo di string-matching.
1.8 Testing

La parte finale del lavoro consiste nel testing delle prestazioni dell’applicazione, confrontandole con quelle di Snort, e suggerendo eventuali miglioramenti futuri.

I risultati dei test sono riassunti nella tabella 1.1.

<table>
<thead>
<tr>
<th></th>
<th>Snort</th>
<th>NetVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacchetti totali processati</td>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>Pacchetti TCP</td>
<td>8439040</td>
<td>8439040</td>
</tr>
<tr>
<td>Pacchetti UDP</td>
<td>1505540</td>
<td>1505540</td>
</tr>
<tr>
<td>Pacchetti ICMP</td>
<td>53429</td>
<td>53429</td>
</tr>
<tr>
<td>Alerts</td>
<td>393</td>
<td>393</td>
</tr>
<tr>
<td>Pacchetti processati al secondo</td>
<td>97002.32</td>
<td>6389.59</td>
</tr>
<tr>
<td>Memoria totale utilizzata</td>
<td>80240 KiB</td>
<td>8592 KiB</td>
</tr>
<tr>
<td>Memoria utilizzata per string-matching</td>
<td>32.84 MiB</td>
<td>1028.94 KiB</td>
</tr>
</tbody>
</table>

A parità di hardware, parametri di compilazione, regole e traffico analizzato, Snort si è rivelato in grado di processare il traffico 15 volte più velocemente. Si tratta di un risultato atteso, per diverse ragioni. Il motivo principale è rappresentato dalla scarsa qualità dell’implementazione dell’interprete del bytecode, che è largamente passibile di miglioramenti. Purtroppo non è stato possibile utilizzare il compilatore Just-In-Time, a causa delle modifiche effettuate al framework, che richiedono l’implementazione delle istruzioni aggiunte. Vi sono, inoltre, punti in cui l’applicazione è migliorabile. Il componente di string-matching, invece, si è rivelato sufficientemente rapido, soprattutto alla luce della ridotta occupazione di memoria in confronto a quella dell’automa utilizzato da Snort, che risulta approssimativamente 32 volte maggiore.

Tutto sommato, le prestazioni sono rassicuranti e lasciano intravedere un futuro roseo per il framework NetVM e le sue applicazioni. Questo sarà comprovato da ulteriori lavori, già in corso, mirati a realizzare la prima implementazione del framework NetVM su un vero network processor, alla riscrittura del compilatore JIT e all’ottimizzazione dell’interprete.

1.9 Lavori futuri

Oltre alle prestazioni, svariati altri aspetti della nostra applicazione possono essere migliorati. La necessità più impellente è rappresentata dall’implementazione di ulteriori opzioni delle regole di Snort, in modo da ampliare la base di regole supportate su cui effettuare i test. In particolare, le opzioni che attualmente causano il maggiore scarso di regole, data la loro mancanza, sono pcre, che riguarda la ricerca di espressioni regolari nei pacchetti, e uricontent, che effettua la ricerca di una stringa nel campo URI delle richieste HTTP.
Infine, un lavoro prioritario è costituito dal riassemblaggio del flussi TCP e dalla deframmentazione di IP, dato che rappresentano le tecniche più comunemente utilizzate per aggirare il rilevamento delle intrusioni.
Chapter 2

Introduction

Intrusion detection systems (IDSs) are an essential element of today’s computer networks. The diffusion of high-speed Internet connections is continuously increasing the number of both users and services, which in turn makes the number of possible attackers and targets grow. Therefore, the amount of work that must be performed by IDSs keeps getting higher and higher. To cope with this, IDSs are continuously improving their processing speed, but we have now reached a point where there is little space for improvements left, as long as they are fully implemented in software. Therefore, recent development has been oriented towards the implementation of IDSs - or parts of them - on hardware devices. The big problem in this scenario is that most existing codebases cannot be easily reused, as the available hardware varies very much in the architecture and in how it is programmed, and the wheel must be reinvented once again. The NetVM framework has been conceived to address these issues once and for all: by abstracting the architecture of a network processor and specifying a precise instruction set oriented towards the processing of network packets, it can achieve code portability and high performance across a number of software and hardware devices.

2.1 State of the art

2.1.1 Intrusion detection systems

Nowadays, the de-facto standard for intrusion detection is Snort [1], which is a software-only implementation of a passive network IDS. It is capable of performing real-time traffic analysis and packet logging on IP networks. Its functions include protocol analysis and content searching/matching, which can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts and many other vulnerabilities.
2.1.2 NetVM

The NetVM architecture developed at the Politecnico di Torino aims to be a portable, yet efficient platform for network applications. NetVM ideally virtualizes a network processor and defines an instruction set oriented towards packet-related operations. Its goal is to solve the performance and portability problems of packet processing-intensive applications.

At the time the work began, NetVM had just reached its alpha stage. The library was still quite rough, but almost complete in functionality. A couple of example programs existed, but they were all very basic in nature, limiting themselves to distinguishing IP packets from ARP packets, or going little further. Such programs were meant to be useful for debugging, more than to showing the power of the library.

The available documentation consisted primarily in a document describing the NetVM architecture [2] and one with the supported instruction set [3]. The C programming interface needed to be inferred from the example programs.

Some elements, such as exceptions and coprocessors, despite being described in the reference architecture were still missing in the implementation. Besides, some instructions were not fully debugged and were behaving differently than expected.

2.2 The problem

The NetVM was at a phase where it needed serious testing and, consequently, a validation of its architecture. Something more complex than the example programs, like a real application, needed to be implemented over it, to demonstrate that the project was actually worthwhile and to see if and how fully it could achieve its goals. Snort seemed an excellent candidate for this task, for a number of reasons: first, it performs many different packet-related tasks, analyzing and extracting header fields, which is a perfect test-ground to see how NetIL suits this Network Processor-typical task. Besides, Snort also performs more general tests, like pattern matching, which can give an idea of how flexible the NetVM architecture is and how well it can be adapted to tasks it was not designed for.

While allowing to test how easily programmable the NetVM is, the implementation of such a time-critical application as an IDS would also allow to see how efficient the NetVM is, and if it can really be used in performance-critical applications.

By having Snort implemented on top of the NetVM framework, it would get instant profit, too: it would be portable to any hardware (to which the NetVM architecture has been ported), making it a lightweight hardware IDS solution which is a somewhat new concept and may represent the next generation of IDSs. With such a solution, we shall be ready to perform intrusion detection on tomorrow's networks without effort.
2.3 Work organization

As fast performance is what this work is all about, a deep understanding of how Snort and the NetVM work is the first necessity. Thus, we will first investigate both of them, dissecting them and describing in detail every component. This will be done in chapters 3 and 4.

The following step will be the mapping of each Snort component to a proper facility in the NetVM, and the implementation of the proposed architecture. Such work will be shown in chapter 5.

During the work, we will not consider ourselves tied to the current NetVM architecture and implementation, but we will also take into consideration the chance of modifying both of them, should this confer our application better performance or flexibility. This will lead to the validation of the NetVM framework. The implemented modifications will be dealt with in chapter 6.

The last step will be performance testing and confrontation with Snort. The results will be presented in chapter 7, together with considerations about further improvements of the NetVM framework and of our Snort implementation.
Chapter 3

Intrusion detection systems

An intrusion detection system (IDS) generally detects unwanted manipulations to computer systems, mainly through the Internet. The manipulations may take the form of attacks carried out by skilled malicious hackers, or script kiddies using automated tools. An intrusion detection system is used to detect all types of malicious network traffic and computer usage that cannot be detected by a conventional firewall. This includes network attacks against vulnerable services, data-driven attacks on applications, host-based attacks such as privilege escalation, unauthorized logins and access to sensitive files, and malware (viruses, trojan horses, and worms).

A generic IDS is usually made up of several components: sensors which generate security events, a console to monitor events and alerts and control the sensors, and a central engine that uses a rules system to generate alerts from the security events received. In many simple implementations all the three components are combined in a single program or device: in fact, IDSs can be implemented both in software or in hardware.

There are several ways to categorize IDSs, depending, for instance, on the type and location of the sensors and the methodology used by the engine to generate alerts:

- A network intrusion detection system (NIDS) is an independent platform which identifies intrusions by examining network traffic and monitors multiple hosts. Network IDSs gain access to network traffic by connecting to a hub, to a network switch configured for port mirroring, or to a physical network tap.

- A protocol-based intrusion detection system consists of a system or agent that would typically sit at the front end of a server, monitoring and analyzing the communication protocol with a connected device (e.g. a remote user or system). For a web server this would typically monitor the HTTPS protocol stream and understand the HTTP protocol relative to the web server/system it is trying to protect. Where HTTPS is in use then this system would need to reside in the “shim” or interface between where HTTPS is unencrypted and immediately
prior to it entering the Web presentation layer.

- **An application protocol-based intrusion detection system** consists of a system or agent that would typically sit within a group of servers, monitoring and analyzing the communication on application-specific protocols. For example, in a web server featuring database interaction, this would monitor the SQL protocol specific to the middleware as it transacts with the database.

- **An host-based intrusion detection system** consists of an agent on a host which identifies intrusions by analyzing system calls, application logs, filesystem modifications (binaries, password files, capabilities/ACLs databases) and activities of other hosts.

- **A hybrid intrusion detection system** combines one or more approaches. Host agent data is combined with network information to form a comprehensive view of the network. An example of a hybrid IDS is **Prelude** [4].

IDSs can also be passive or reactive: in a passive system, the IDS sensor detects a potential security breach, logs the information and signals an alert on the console. In a reactive system (also known as an *intrusion prevention system*) the IDS responds to the suspicious activity by resetting the connection or by reprogramming the firewall to block network traffic from the suspected malicious source. This can happen automatically or at the command of an operator.

IDSs must not be confused with firewalls. Though they both relate to network security, an IDS differs from a firewall in that a firewall looks outwardly for intrusions in order to stop them from happening. An IDS evaluates a suspected intrusion once it has taken place and signals an alarm. This is traditionally achieved by examining network communications, identifying heuristics and patterns (often known as *signatures*) of common computer attacks, and taking action to alert operators. Another practical difference is that firewalls usually look only at packet headers, completely ignoring the payload, while the most important feature of IDSs is their scanning of the packet payload for known patterns [5].

### 3.1 Snort

Snort is a software-only implementation of a passive network IDS. It was originally developed by Martin Roesch, but is now owned and developed by **Sourcefire, Inc.** [6], of which Roesch is the founder and current CTO. It is free software, released under the GNU General Public License, even though proprietary versions with integrated hardware and support services are sold by Sourcefire. The first public release of Snort took place in 1998.

Snort originally advertised itself as “a cross-platform, light-weight network intrusion detection tool that can be deployed to monitor small TCP/IP networks” [7]. Since then it has grown
very much, becoming the most innovative and effective IDS available on the market, and it now claims to be “the de-facto standard for intrusion detection” [1]. Current versions of Snort are capable of performing real-time traffic analysis and packet logging on IP networks. Its functions include protocol analysis and content searching/matching which can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts and many other vulnerabilities. One of the most-recently introduced features even allows the system to be used for intrusion prevention purposes, interacting with the Linux Netfilter/iptables [8] architecture, effectively dropping attacks as soon as they take place.

Snort emphasizes its being lightweight. A lightweight intrusion detection system can easily be deployed on most any node of a network, with minimal disruption to operations. Lightweight IDSs should be cross-platform, have a small system footprint, and be easily configured by system administrators who need to implement a specific security solution in a short amount of time. Lightweight IDSs are small, powerful, and flexible enough to be used as permanent elements of the network security infrastructure. This claims to be in contrast with most commercial NIDSs, which usually have common features such as significant system footprint, complex deployment and high monetary cost, and which usually require dedicated platforms and user training to deploy in a meaningful way.

3.1.1 Snort internals

Internally, Snort is organized in several layers [9]. First, traffic is acquired from the network link via libpcap [10]. Packets are passed through a series of decoding routines that fill out the packet structure for link level protocols, and decode things like TCP and UDP ports. Packets are then sent through a set of preprocessors: each of them does its job with the packet, if required. Packets get then sent through the detection engine, which checks each of them against the various options listed in the Snort rules. Eventually, output plugins process the results provided by the detection engine, sending alerts and logging whatever is required.

The packet decoder

The decoder engine is organized around the layers of the protocol stack present in the supported data-link and TCP/IP protocol definitions. Each subroutine in the decoder imposes order on the packet data by overlaying data structures on the raw network traffic. These decoding routines are called in order through the protocol stack, from the data-link layer up through the transport layer, finally ending at the application layer. Speed is emphasized in this section, and the majority of the functionality of the decoder consists of setting pointers into the packet for quick data access during the later analysis.
Preprocessors

Preprocessors were introduced in version 1.5 of Snort. They allow the functionality of Snort to be extended by allowing users and programmers to add modular plugins to Snort fairly easily. Preprocessor code is run before the detection engine is called, but after the packet has been decoded. The packet can be modified or analyzed in whatever way may be needed using this mechanism.

Snort’s preprocessors perform two main functions: they either manipulate packets so the detection engine can properly analyze them, or they examine traffic for suspicious use that cannot be discovered by signature detection alone. Snort has a variety of preprocessors by default, most of which have been added to detect new methods of IDS evasion. Everything from polymorphic shellcode to fragmented packets can be detected with the aid of preprocessors.

Other functions that can be performed by preprocessors are, for instance, TCP stream rebuilding and IP fragments reassembly.
### The detection engine

The detection engine is the heart of Snort, and it is responsible for the actual matching of signatures against packets. Snort rules are loaded into the detection engine by an ad-hoc parser, and are categorized in a tree-like data structure: the rules are separated into what is referred to as a chain header and a list of chain options. The common attributes such as source/destination IP address and ports identify the chain header. The chain options are instead defined by details such as the TCP flags, ICMP code types, specific type of content, payload size, and so on. Snort 1.x versions then sequentially check every chain header against every packet. If all fields of a chain header are matched, every chain options entry is then sequentially tested, and if all its field are matched as well, the specified action is performed. Packets that do not match any Snort rule are simply discarded. Snort 1.x performs pattern-matching through the Boyer-Moore algorithm (See B.2).

Snort 2.x switched to a different, pattern-centric model [11]. Rule subsets are formed by a rule optimizer [12], based on unique rule and packet parameters using a classification scheme based on set criteria. Since these subsets are based on the unique rule parameters such as source/destination ports, and rule contents, each rule subset consists of the complete set of rules that are applicable to each packet. This guarantees that all applicable rules are tested against each packet, and ensures that rules which cannot possibly match the packet are ignored. Every incoming packet is matched to a corresponding rule set, based on its unique parameters, then an automaton-based multi-rule search engine [13] processes all the content-related options of the selected rule set. Whenever a match is found, the standard Snort processing validates the remaining rule options. If the rule is fully validated, an event is generated and added to the event queue. Once the search engine has completed processing the packet, an event selector processes the event queue.

This pattern-centric model is justified by the fast performance of the automaton-based pattern-matching algorithm [14], which although requires a lot of memory. As such, this approach was not applicable at the time Snort was born, when memory was much more expensive than nowadays. For instance, when Snort was first released in 1998, the cost of 1 gigabyte of RAM was more than 800 US Dollars [15], while today it costs about 80, and today’s memory is even much faster, as new technologies for the making of RAM have been developed during the recent years.

The current version of the detection engine is also modeled after a plug-in approach. Plugins are registered at initialization time, and each of them declares the rule options it handles. This allows anyone with an appropriate plug-in module to add significant detection functionality to Snort and customize the program for specific jobs, by adding to the analysis capability.

### Output plugins

The output plugins are the means Snort has to transmit alerts from the detection engine to the system administrators. As the name suggests, this module is modeled after a plug-in approach, too. Multiple plugins can be used at the same time, to facilitate intrusion data management.
Output plugins can range from simple comma-delimited output to complex relational database output. An output format has even been specifically designed for Snort to outsource the writing to databases, which has traditionally been a bottleneck. Existing plugins also allow most critical alerts to be sent via e-mail or SMS.

### 3.1.2 Anatomy of Snort rules

Snort uses a simple, lightweight rules description language that is flexible and very powerful, allowing rules to be written in a single line. Snort rules are divided into two logical sections: the rule header and the rule options. The rule header contains the rule’s action, protocol, source and destination IP addresses and netmasks, as well as source and destination ports information. The rule option section contains alert messages and information on which parts of the packet should be further inspected to determine if the rule action should be taken.

Example rule:

```
log tcp any any -> 10.1.1.0/24 80 (content: "GET"; msg: "Webserver access");
```

The text up to the first parenthesis is the rule header, while the section enclosed in parentheses contains the rule options. The words before the colons in the rule options section are called option keywords. All the elements that compose a rule must be verified in order to trigger the indicated rule action. The elements of a rule can be considered to form a logical AND statement. At the same time, the various rules in a Snort rules library file can be considered to form a large logical OR statement.

#### Rule actions

The first item in a rule is the rule action. The rule action tells Snort what to do when it finds a packet that matches the rule criteria. The following are the most commonly used actions:

- **alert**: Generate an alert using the selected alert method, and then log the packet.
- **log**: Log the packet.
- **pass**: Ignore the packet.

The 2.x branch of Snort also supports dynamic and user-defined rule actions: the former is used to dynamically enable or disable other rules, while the latter allows arbitrary user-defined actions to be bound to any rule, allowing a certain service to be shut down in case of frequent attacks, for instance.
3 – Intrusion detection systems

Protocols

The next field in a rule is the protocol. Snort currently supports three protocols:

- tcp
- udp
- icmp

Recently, support for ip has been introduced, as well, which allows logging and alerting of packets regardless of the transport protocol data.

IP addresses

The next portion of the rule header deals with the IP address and port information for a given rule. The addresses are formed by a straight numeric IP address and a netmask, in CIDR format. This provides a nice short-hand way to designate large address spaces with just a few characters. The keyword any can be used to define any address. Snort does not currently have a mechanism to provide host name lookup for the IP address fields in the rules file.

There is an operator that can be applied to IP addresses, the negation operator: this operator tells Snort to match any IP address except the one indicated by the listed IP address. The negation operator is indicated with a !.

Lists of IP addresses can also be specified. An IP list is specified by enclosing a comma separated list of IP addresses and CIDR blocks within square brackets.

Port numbers

TCP/UDP port numbers may be specified similarly to IP addresses: a single number represents, obviously, a single port, while any is a wildcard value, meaning literally any port. The negation operator can also be used to invert the meaning of the specification. Finally, port ranges are also supported, and are indicated with the range operator, :.

The direction operator

The direction operator, -> indicates the direction of the traffic that the rule applies to. The IP address and port number on the left side of the direction operator represent the source, while the address and port number on the right side represent the destination.

There is also a bidirectional operator, which is indicated with a <-> symbol: this tells Snort to consider the address/port pairs in both directions. This is handy for analyzing both sides of a conversation, such as telnet or POP3 sessions.

Also, it should be noted that there is no <-> operator, so that rules can be read consistently, knowing the left side is always the source and the right side is always the destination.
The option keywords

The option keywords allow a rule to make very specific tests on a packet. For instance, option keywords have been defined to look for a specific byte value at a specific position in the packet payload, or to look for packets of a given size.

A list of the most commonly used option keywords can be found in Appendix A.

3.1.3 Rule examples

In this section we will take a look at some real Snort rules, taken from the official distribution, explaining the purpose of every keyword. Some options that only have a descriptive meaning, such as reference, sid and rev will be omitted for brevity. The rules have been split on several lines for typographic reasons, but they are usually written on a single line.

1. alert tcp $EXTERNAL_NET any -> $HOME_NET 21 ( \
   msg:"FTP CWD Root directory transversal"; \
   flow:to_server,established; content:"CWD";
   nocase; content:"C\|3A 5C\|"; distance:1;)

This rule detects a flaw in some Microsoft Windows FTP servers, which allows clients to access the full contents of the C drive, by issuing an FTP CWD command to C:\. An alert is triggered upon detection of such a command coming from the external network to a local FTP server. To make the rule more flexible, the rule uses variables, which are usually defined at the beginning of the Snort configuration file. The rule explicitly looks for the string CWD (as specified by the first content directive) regardless of the case (nocase), followed by the string C:\ (second content directive, which makes use of bytecode) at a minimum distance of one character (which is the compulsory space following a command in the FTP protocol). Such a message must also appear in a packet which is heading to the server on an already established TCP connection (flow).

2. alert tcp $EXTERNAL_NET any -> $TELNET_SERVERS 23 ( \
   msg:"TELNET bsd exploit client finishing";
   flow:to_client,established; dsizel:>200; \
   content:"\|FF F6 FF F6 FF FB 08 FF F6\"; depth:50; offset:200; rawbytes;)

This second rule aims at detecting the exploitation of a buffer overflow in the BSD telnet daemon. What is relevant here is the use of the dsizel keyword to look for a packet containing a long sequence of data followed by (offset) an exploit-characteristic string, which must
appear unaltered in the packet (without rawbytes, the string would be searched for in the data altered by Snort’s telnet protocol preprocessor), within the following 50 bytes (depth).

3. alert icmp $EXTERNAL_NET any -> $HOME_NET any ( \
   msg:"ICMP Nemesis v1.1 Echo"; dsize:20; icmp_id:0; icmp_seq:0; itype:8; \
   content:"|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|";)

This last rule shows the use of ICMP-related options to carry out a very specific search for a particular packet, which is usually generated by an ICMP-injection tool.

3.2 Improving the speed of current IDSs

All software-based IDSs require the network to inspect the packet payload at line rates to detect and filter those packets containing worm signatures. These signature sets are large (e.g.: thousands) and complex, and some required operations (e.g.: pattern-matching) are very computation-intensive, hence software-only implementations can hardly meet the performance goals, and recent efforts are oriented towards hardware implementations of the most speed-critical parts of an IDS.

3.2.1 CAM solutions

For instance, pattern-matching could be performed through the use of a Content-Addressable Memory (CAM): it is a special type of computer memory, generally used in applications needing very high speed searching. Unlike standard computer memory (Random-Access Memory, RAM) in which the user supplies a memory address and the RAM returns the data word stored at that address, a CAM is designed such that the user supplies a data word and the CAM searches its entire memory to see if that data word is stored anywhere in it. If the data word is found, the CAM returns a list of the addresses where the word was found.

A variant of CAMs, Ternary CAMs (TCAMs) are already widely used for IP header processing such as longest prefix match [16]. Because of their intrinsic parallel search capability, it should be quite obvious that TCAMs can also be used effectively for the string-matching functions needed in intrusion detection systems. However, TCAMs have a fixed size (which is usually quite small, as they are very expensive), and thus impose limitations on the pattern length that can be directly matched. Another problem is that very often more than one pattern appears in a single rule, and there is no easy way of handling multiple patterns with TCAMs. Algorithms have been developed to overcome those problems [17].
3.2.2 FPGA solutions

FPGAs can be programmed for fast pattern matching due to their exploitation of reconfigurable hardware capability and their ability for parallelism. To search for a regular expression of length $n$ on an FPGA, one solution is to build a serial machine that requires $O(2n)$ memory and takes $O(1)$ time per text character.

Better solutions have been proposed in literature (requiring only $O(n^2)$ space) [18], but all current approaches are optimized for single keyword searching and do not scale well for multiple patterns. The main concern is that patterns are searched sequentially, so the overall latency increases proportionally with the number of patterns.

3.2.3 Bloom filter solutions

The Bloom filter, conceived by Burton H. Bloom in 1970 [19], is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not.

Bloom filters can be used to build an efficient multiple string-matching algorithm, albeit approximate: several Bloom filters are created, each taking care of all patterns of a certain length. When a search has to be performed, all the filters work in parallel, returning results for all the possible pattern lengths. Because of the possibility of having false positives, the matched patterns must be verified again by an exact string-matching algorithm, but as their number is now much smaller, this second search usually takes an acceptable time.

Even though solutions following this scheme have been proposed in literature [20], they are little suitable for an IDS, whose patterns vary from tens to hundreds of bytes and there are many different possible pattern lengths.

It is worth noting that these solutions are often mixed together to achieve flexibility or ease of realization: for instance, bloom filters are often implemented using FPGAs.
Chapter 4

NetVM

At present, the concept of virtual machine is widely adopted by the commercial world and thoroughly investigated by the research community. A virtual machine is a software abstraction that behaves similarly to a physical processor, i.e. it executes programs written in a conventional assembly language. This assembly language is completely independent from the hardware, and the code implementing the virtual machine itself is usually written in a portable manner. All of this results in the fact that programs written for the virtual machine can be run on a wide range of microprocessors. Besides, they can either be interpreted by the target processor, or transformed into native code by a Just-In-Time (JIT) compiler.

The best-known virtual machines, like the Sun JVM [21] or the Microsoft CLR [22], are used as generic, system-independent platforms to run code generated by commonly used programming languages. They are usually virtualizations of general purpose stack machines, with simple and standard instruction sets that provide instructions for memory manipulation, control transfer, arithmetic operations, stack management and so on.

Although powerful, these solutions are by their nature very generic, therefore they often do not map efficiently to specific applications. Let us consider network processing: most applications make use of primitive operations that are different from those used by common applications and have a different program structure (for instance, network-related applications make extensive use of bitwise operations and logical shifts). A virtual machine with native support for such task-specific operations will most likely be more efficient in this particular field.

The Network Virtual Machine (NetVM) aims to be a portable, yet efficient platform for networking applications. NetVM ideally virtualizes a network processor rather than a general purpose processor, therefore it should be profitably mapped not only on traditional general-purpose PC architectures, but also on programmable network devices.
4.1 Why a Virtual Machine-based approach

The world of network processing devices is vast, and many different solutions are available. Low-end solutions are usually software-based and rely on PCs with standard operating systems, while high-end solutions are built on specific hardware, like *ASICs* (Application-Specific Integrated Circuit) or, according to current trends, *ASIPs* (Application-Specific Instruction-set Processors). Despite this heterogeneity, the set of operations is quite uniform, and often the only notable difference among the solutions is mainly performance, which trades-off with price.

The main issue in this scenario is code portability: it is clear that with a portable and scalable software development platform, a lot of common portions of software could be reused. This is not possible nowadays, because it is very difficult, if not impossible, to write software that runs on different network devices: different operating systems provide totally incompatible networking environments, and the plethora of network processors presents very different architectures. Besides, the used languages range from C/C++ for PC-based solutions, to assembly for network processors. Of course, every network processor has its own proprietary instruction set and assembler.

As a consequence, well-known networking functions implemented for a device cannot be reused, and developers need to reinvent the wheel every time they move to a different hardware device, rewriting the entire code from scratch, possibly introducing bugs, which leads to a long development cycle.

The NetVM has been designed to address all of these issues: it implements a standard and well-defined architecture, which makes use of a high-level programming language with specific extensions for network processing and protocol handling (*Network Intermediate Language, NetIL*). Besides, portability is at binary level and not at source level: the bytecode representation allows to execute the same code on different platforms, and the use of specific hardware (rather than software emulation) where available is transparent to the programmer. In some aspects, the NetVM approach even has advantages over pure software solutions, as, for instance, the code can be developed and tested on a standard PC and moved to a network processor once it is finished.

4.2 The NetVM architecture

The NetVM architecture is built around the concept of *processing element* (*NetPE*), which virtualizes the *microengine* of a real network processor. As shown in figure 4.1, a typical NetVM application is assembled from several NetPEs, each implementing a single functionality; complex structures can be built by connecting different NetPEs together. This modular view derives from the observation that many network applications can be decomposed in many separate simple tasks. This architecture can exploit parallelism or sequentiality to achieve better performance. The modular approach is already affirmed: other software solutions, like *Netfilter* or *Click* [23] have demonstrated its efficiency, and many network processors implement a similar architecture,
consisting of many microengines executing simple tasks.

The NetVM specification also describes a standard interface that can be used by NetPEs to exchange packets among themselves, and with user applications.

4.2.1 The Processing Element

The NetPE is the core of the NetVM. It runs the NetIL bytecode, performing actual processing of network data. The NetVM has a stack-based design. A stack-based virtual processor grants portability, a plain and compact instruction set and a simple virtual machine. The direct consequence of this choice is that no general-purpose registers are provided and all the instructions that need to store or process a value have to make use of the stack. Every NetPE has its own private stack.

The execution model is event-based. This means that the execution of a NetPE is activated by external events, which can be bound to different code segments. Typical events are the arrival of a packet from an input endpoint, the request of a packet on an output endpoint or the expiration of a timer.

The main components of a NetPE are shown in figure 4.2.

4.2.2 Memory

The NetVM memory is divided into four segments: one for the program code, one for private data, one for data shared among NetPEs and one for packets. This subdivision is meant to properly isolate the different kinds of memory and to grant access efficiency.
Figure 4.2. Internals of a NetPE

**Code memory**

This segment contains the compiled bytecode implementing the event handlers. Only one handler can be in execution at a given time, and the address of the instruction which is currently being executed is contained in the **Program Counter** register (PC). Of course, this memory segment is private for every PE.

**Data memory**

This segment is private, too, and is used by the programs to store their own long-term data. It is the component abstracting the private memory of a network processor’s microengine.

NetVM implements no high-level memory management systems like garbage collectors that, although very versatile, are too heavy and not necessary for the NetVM purposes, and cannot be easily mapped on embedded systems and network processors. Every PE can specify the size of the data memory it requires, and the requested amount of memory is instead statically allocated during the initialization phase.

The flexibility lost with this approach is balanced by the gain in efficiency: the program can access the memory without intermediation; moreover, knowing the position and the amount of memory before program execution allows very fast accesses when a JIT compiler is used, because memory offsets can be precomputed.

The programs running on a NetPE have direct access to the data segment, with explicit load and store instructions. Specific instructions for copying blocks between different memories (a recurrent operation in network processing) are provided as well.
Shared memory
This memory is meant for data statically shared with other NetPEs. The shared memory is visible by all the NetPEs, and it virtualizes the global memory present in the majority of network processors.

At the moment, there is no means to coordinate concurrent accesses to the shared memory. This aspect will be investigated in the future.

Exchange buffers
Exchange buffers virtualize the memory buffers shared between consecutive microengines in real network processors. They are meant to contain network packets under elaboration, but can also contain arbitrary data. An exchange buffer belongs to a single NetPE in every moment, and can only be accessed by it. Assembler instructions are provided to send/request an exchange buffer to/from another PE.

The size of an exchange buffer is limited (usually to some kilobytes) and the use of shared memory is suggested to pass big structures or tables from one NetPE to another. The size is not determined by the NetPE and can change from packet to packet.

4.2.3 Stack
Every NetPE has its own private stack. Due to the design of the virtual machine, the stack plays a key role in the operation of a NetPE: in fact, the NetIL instructions work on stack values and not on registers. The stack has been conceived to be as simple and efficient as possible: it handles a single data type, the 32-bit integer. Other types (byte, short, etc.) are converted (with or without sign) before being put on the stack. The address of the current top of the stack is kept in the Stack Pointer (SP) register, even though direct manipulation of such register is not possible.

4.2.4 Connections
A NetPE can have a number of input and output connection endpoints. Endpoints are used to connect a NetPE with other NetPEs, with the physical network interfaces and with the user. The NetVM architecture supports an arbitrary number of input and output virtual interfaces for a NetPE.

The NetVM model for connection types is inspired by the one implemented in Click: two kinds of connections are supported, push and pull, as shown in figure 4.3. On a push connection, packets are sent by the source NetPE and are passed downstream to the destination NetPE. This is the way packets usually move in network devices. On a pull connection, in contrast, the destination NetPE initiates packet transfer: it asks the source NetPE to return a packet, if no packet is available it is put in a wait state.
Figure 4.3. The NetPE port connection model
Chapter 5

Implementation

Building a Snort clone on top of the NetVM framework is, obviously, a difficult task. Snort has undergone eight years of development, leading to two major releases, backed up by a million-dollar company, so it is unconceivable to implement a full clone of the current Snort version in a single effort. Instead, we decided to focus on a minimal features set, trying to create an architecture that can be easily extended in the future, eventually reaching the same array of features Snort has.

We chose this approach for various reasons. The main one was the likely need of modifications to the NetVM framework, which would turn the implementation to a two-fronts job, hence the need to reduce the work on one front. Therefore, we opted to only implement a subset of Snort’s rule options, which we deemed to be the essential one able to confer the application usability to a certain degree, while maintaining a reasonable size.

Another aspect needing consideration was whether to make an exact copy of Snort, using its same architecture, or to use a different one. The chosen approach was to take inspiration from the original architecture, adapting it to our needs. We decided to drop - at least for this first implementation - the preprocessors and the output plugins. The absence of preprocessors only involves minor drawbacks, as the work they perform can either be done in a different way, or can even be skipped at all: for instance we are not interested in TCP stream reassembly or IP defragmentation at this stage. The lack of output plugins causes even less troubles, as for the moment a single hardcoded output format will be enough for our needs.

5.1 Design decisions

Computer networks are becoming faster and faster, and, as their utilization spreads, the possible threats almost increase by the same factor. This means that the number of rules that an IDS must cope with keeps growing, while the time it has to check all of such rules against each packet keeps reducing, as it must stay in sync with the wire speed. This is the base principle which controls the
whole design process of an IDS. Therefore, most design decisions are oriented to achieve processing speed, trading off with memory requirements, which mainly drives the choice of algorithms, as will be documented further on.

5.1.1 Processing elements interconnection

A typical packet-processing application is made up of several almost-independent tasks. As we have seen previously, this is the reason behind the processing element approach that the NetVM has undertaken. For our objectives, we have identified such separate tasks with the various protocol layers present in ordinary network traffic. Our idea is to build a chain of processing elements which will be traversed by each incoming packet, as shown in figure 5.1. Each processing element will then deal with a single protocol, and will have to perform two tasks: the first is to analyze the protocol, extracting any information which might be useful to subsequent elements, while the second task is to actually test which rules are matched by the currently inspected packet. To emphasize this modular approach, from now on we will refer to the various PEs as modules.

![Figure 5.1. A possible way to interconnect the application modules](image)

Of course, being limited to a single protocol, a module cannot test all the options specified in a rule. Although, every possible rule option can be assigned to a single module, which will be in charge of testing it. This way, every module will provide a partial match result, and some mechanism to put all the results together must be defined. This can be easily carried out by observing that, for a whole rule to be matched, every single option specified by the rule must be matched, i.e. there exists an implicit logical and among all rule options. By applying some basic boolean algebra concepts, we can say that a rule is false if at least an option specified in it is false. So we can start considering all rules matched, and set them not matched as soon as a PE fails to match an option.
5.1.2 Architecture

Actually, the chain-layout can still be improved. If we do not take into consideration the case of packet tunneling, we can observe that some modules are mutually exclusive. For instance, a packet will always contain only one network-layer protocol (IPv4 or IPv6) and only one transport-layer protocol (TCP, UDP, and we can also consider ICMP as such). We can therefore refactor the connections among the various modules, building a four-tier architecture, as shown in figure 5.2.

![Figure 5.2. The NetVM-Snort architecture](image)

This architecture has many advantages:

- **Each protocol gets analyzed one single time.**

- **The knowledge of a protocol is embedded in a single place.** This makes it easy to debug and improve the handling of a protocol, should any bugs or speedups be found. It also allows easy integration of new protocols: all is needed in this case is a new PE and the relative connections.

- **The number of traversed NetPEs is small.** Of course, the shorter the path, the faster the performance.

- **The architecture is suitable for pipelining.** At the moment, the application handles one packet at a time, but potentially it could handle a packet per tier, improving the throughput.
by a factor of four.

5.1.3 Data transport

Of course, we have more data to move around inside the NetVM than just the bare packet: as we have seen, we also have the protocol data that each module extracts from the packet, and an entry for every rule we must process, to keep track of whether it is matched or not. To achieve this, a modification of the NetVM framework has been proposed and implemented. Such modification allows the exchange buffer to be splitted in two parts, called respectively Packet buffer and Info partition. The former contains the packet, while the latter provides space for arbitrary data which can be attached to the packet.

![Figure 5.3. The exchange buffer layout](image)

The layout chosen for the exchange buffer is shown in figure 5.3: the first $N$ bytes of the info partition are used to keep track of the matching status of each rule. As this status is a boolean value, a single bit per rule would be enough, but the current implementation uses a byte per rule, mainly for practical reasons. Of course, this can be changed with minor modifications. Each byte is mapped to a rule, through an identification number assigned to each rule when it is read. The actual byte value defines the current matching state of the rule, in a sort of negative-logic code: 0 means that the rule is matched, while 1 means that the packet does not satisfy at least one rule option of those which have already been tested, and consequently the rule is not matched by the packet. This - apparently odd - behavior was chosen to save the time needed for the initialization of the info partition. In fact, NetVM guarantees that when a packet is written to it, the info
partition is fully set to zeroes. As we have seen in 5.1.1, we must start considering all the rules initially matched, and setting all the bytes to 1 to use the “normal” (positive) logic values would just be a waste of precious time.

The following bytes are padding bytes, to achieve 32-bit alignment. Then we have a series of 32-bit fields which are used by the various PEs to store data extracted from the packet. The number of fields for each module is hardcoded in the program, but adding new ones is straightforward. As it is essential that all modules agree on the position and use of each field, they should only be referenced via symbolic constants, shared between C code and NetIL code.

It should be noted that the info partition layout actually depends on the number of rules. To make things clear, the layout is calculated once and for all after the rules have been read and validated, and will be made available to all modules at the time of their initialization.

5.1.4 Static vs. dynamic code

The code for every PE can be generated in two different fashions. For instance, it can consist in a loop iterating over rules stored in data structures held in memory. This approach needs the ability to initialize the PE private memory from the control plane, and also that the knowledge of how the memory is organized be shared between the PE bytecode and the control plane. It is, indeed, fairly easy to implement and maintain.

Otherwise, rules data can be embedded in the code. This brings the advantage of needing no data memory access, which leads to faster performance. Although, this approach is definitely more complex, as a sort of “compiler” must be written, which will be able to translate the rules to NetIL code.

For our IDS application, we chose the latter, as performance is one of our main objectives, and in order to avoid the need to define the API for interaction with the PE private memory, which was still lacking in the NetVM framework.

5.2 The NetVM-Snort architecture

5.2.1 Overview

The program operation is made up of a few basic steps, shown in figure 5.4. The first step consists in reading the Snort-format configuration files, building structures in memory with the rules data. Then, NetVM is initialized, all PEs are created and custom NetIL assembly code is created compiling the rules for each of them. The generated code gets then assembled into NetIL bytecode and injected into the PEs. If all the PEs are initialized properly, connections among them are built. Lastly, libpcap is initialized and packet capture begins, marking the end of the initialization phase. In the main program loop, every captured packet is then written
Figure 5.4. Flowchart of the program operation

to the NetVM. Match results are taken from the NetVM output and proper actions are undertaken.
In the following sections we will see in detail how every component behaves.

5.2.2 The rules parser

Overview

One of our objectives is to be 100% compatible with Snort’s configuration files. Therefore a
parser able to interpret them needed to be written. This has been accomplished using two typical
instruments for this matter: *flex* [24] and *bison* [25].
Flex is a lexical analyzer generator. Its role is to scan an input text (i.e. the Snort configuration file) and recognize tokens in it. Such tokens consist of all the Snort keywords, IP addresses, and various other characters defined by the configuration file syntax. Flex is not meant to be used directly when writing complete parsers. It is rather called by Bison when needed.

Bison is a parser generator, the natural complement to flex. The parser is generated starting from a grammar definition, which states how the various tokens can be put together and their semantic value. It automatically calls flex to read tokens from the input file, and then tries to match the tokens sequence to the grammar rules (called productions). Every time a rule is recognized (the proper terminology would be reduced), a programmer-defined action, consisting of C code, can be performed. This way we are able to store the read rules in memory.

Before a rule is added to the list, some sanity checks are performed, as well as a check to make sure that all the options specified in it are actually supported: if the rule fails any check, it is automatically skipped. This allows to use the full Snort ruleset without any modifications, while keeping the certainty that all rules maintain their full semantics and do not trigger any false positives. Other rules may cause a warning in the parser: this usually means that the rule is only partially supported, while keeping its semantics. In this case, it is up to the rule writer to change the rule or to ignore the warning. A detailed log is printed during the parsing, to make problems easily identifiable.

The parser mechanism also supports Snort-style comments and variables:

```
# Path to your rule files
var RULE_PATH ../rules
```

After definition, the value of a variable can be referenced through the \$ operator, as in:

```
include $RULE_PATH/demo.rules
```

As the above examples shows, Snort’s include directive is supported too, which permits to split the configuration file in several files and use only those that are really needed. Other unsupported keywords are automatically skipped. This includes the preprocessor, config, and output directives, which make no sense in our implementation as they deal with preprocessor and output plugins configuration.

A list of supported Snort keywords, with syntax specifications and examples can be found in Appendix A.

**Storing rules in memory**

Some type of structure is needed to store the rules read from the configuration file in memory. Inspiration for the layout of this structure has been taken from an earlier version of Snort, even though the obtained result is quite different as shown in figure 5.5. 
The main fields of the rule are stored in a *struct rulehead* object. Its contents are always present in every rule, and consist of:

- Rule action.
- Protocol.
- Source IP address and port, when relevant for the protocol.
- Destination IP address and port, when relevant for the protocol.
- Message to use when logging.
- Snort rule identifiers, version/revision number, references and classtype.

Each *struct rulehead* instance also has a pointer to a list of *struct ruleopt* objects. These are used to store the remaining options specified in the rule, whose number is undetermined. Hence, the list is managed by a generic list API, which allows basic management of the list (insertion and deletion of nodes), searching in the list with custom comparison functions and iteration over the list elements. Each *struct ruleopt* contains a single option, having simply four fields:

- *name*: A string containing the option name.
- *value*: A string containing the option value.
• **negated**: A boolean flag which is true if the option appeared in the rule preceded by an exclamation sign (!), which is used to state that the option must **not** match.

• **used**: Another boolean flag used to keep track of which options have been used by a module.

The *struct rulehead* objects are also organized in a list, managed with the same API, to allow for an arbitrary number of rules.

The next step, after the list has been built, is the initialization of the NetVM and of the various modules, each of which has its own initialization function. Of course, these functions need the rules, and so the list is passed to all of them. Each of these will iterate over the rules and their options. Whenever they find an option they're in charge of utilizing, they must set the *used* field of that option to true, so that we know it has been actually used.

After everything has been initialized successfully, the rule options are not needed anymore, hence we can safely release the memory they use. While doing this, we must keep an eye on the *used* field: if we find one which is still set to false, something must have gone wrong, as this means that the option is not checked by any module. This way we can be absolutely sure that each rule option is properly checked, and that each rule maintains its intended meaning. One last thing worth noting is that we cannot free the list of *struct rulehead* objects, because they will be needed by the output module.

### 5.2.3 The capture and input module

The capture and input module uses *libpcap* to capture raw Ethernet frames from the wire, then it passes them to the NetVM, which will take care of the processing. Optionally, captured packets can be logged to aid debugging.

This module is subject to be removed when NetVM will incorporate a proper input port abstraction.

### 5.2.4 The Ethernet module

The amount of work that must be done by the Ethernet module is very small. As no Snort keywords deal with the Ethernet level or with network protocols other than IPv4, all this module must do is to make sure that the frame contains an IPv4 packet, locate its start offset and write it in the `ETHERNET_NEXT_OFFSET` field of the info partition. This implies that the code executed by this PE is static, and does not depend on the rules. Besides, the payload type can be recognized very quickly, as the Ethernet header has a fixed format, shown in figure 5.6.

The *EtherType/Frame type* field (offset 12 with regard to the beginning of the frame) contains the value 0x800 for IP packets. The Ethernet header is always 14 bytes long, so the next protocol offset is always 14.

The pseudocode for this module is shown in algorithm 1.
5.2.5 The IPv4 module

Opposite to the Ethernet module, the IPv4 module is very complex, as its job is to check both the source and destination addresses, each of which is a pair of an IPv4 address and a netmask (We treat the special case of a single IP as if it had a mask of 255.255.255.255). Besides, the addresses might be specified as any, which means that any address will match. Such parameters are present in every rule, which means that this is one of the modules requiring the highest amount of computing, and therefore we must behave as efficiently as possible. To achieve high throughput, we have used a few expedients.

- The module operates by loading the source and destination IPs from the packet only once, saving them to two locals. Such locals are facilities available to every NetPE resembling the registers of real processors. They are intended to store frequently-used data for quick access. Thus, this step saves the time needed to access the exchange buffer for every rule.

- Also, we group rules having the same IP and netmask together, so that we only have to check each different combination of IP and netmask once.

- The last trick we use is to test the destination address first, and then, if it matches, the source address. We have used this approach because we have noticed that, in real rule sets most rules have any as the source address, and a determinate IP as destination address (usually specified via variables). This is quite obvious, as attacks may come from anywhere, but we know the IPs of the servers in the LAN we are monitoring.

This way we can narrow down the number of tests we need to perform, achieving good performance even with large rule sets.

The IPv4 module also takes care of discarding all the fragments of fragmented packets, without any further processing, as the handling of such packets is not yet supported by our application, and
they might cause false positives if they reach the subsequent modules without being reassembled first.

The last thing the IPv4 module has to do is to store the datagram length in the `PAYLOAD_LEN` field (subtracting the IP header length) and to detect which transport-layer protocol is encapsulated in the packet. Then it can store its offset in the proper field (`TCP_OFFSET`, `UDP_OFFSET` or `ICMP_OFFSET`) and send the packet to the proper module, which will take care of further processing.

The pseudocode implemented by the module is shown in algorithm 2. It should be noted that what is referred to as “`rule_source_address/netmask`” or similar expressions is the actual value of the parameter specified in the rule.

**Algorithm 2 IPv4 module**

1: Load IPv4 start offset from info partition
2: Load fragmentation flags and fragment offset from packet (offset 6)
3: if `frag_flags.more` == true OR `fragment_offset` != 0 then
4: Return
5: end if
6: Load source address from packet
7: Save source address in `IPV4_SRC` field
8: Save source address in `NetVM_Locals(0)`
9: Load destination address from packet
10: Save destination address in `IPV4_DEST` field
11: Save destination address in `NetVM_Locals(1)`
12: for every group of rules do
13: Load `NetVM_Locals(0)`
14: if `NetVM_Locals(0)` == `group_destination_address/netmask` then
15: for every rule in group do
16: Load `NetVM_Locals(1)`
17: if `NetVM_Locals(1)` != `rule_source_address/netmask` then
18: Mark rule as not matched
19: end if
20: end for
21: else
22: Mark all rules of group as not matched
23: end if
24: end for
25: Detect type of encapsulated protocol
26: Calculate offset of encapsulated protocol
27: Save offset of encapsulated protocol in proper `TCP/UDP/ICMP_OFFSET` field
28: Send exchange buffer to TCP/UDP/ICMP module

It should be noted that, even though no IPv6 module exists, yet, its operation would be pretty similar to that of this module.
Code example

For clarity, let us make an example, considering the following rules:

log tcp 10.0.0.1 any -> 10.9.9.9 any (msg: "Rule 8");
log tcp 10.0.0.2 any -> 10.9.9.9 any (msg: "Rule 9");

As both rules have the same destination address, they will be grouped together, and get translated to the following NetIL code:

ipentd00001:
ipdst00001: ; (1)
push -1
push 168364297
locload 1
mcmp
j.eq ipents00001
ipaltd00008: ; (2)
push 1
push 8
istore.8
ipaltd00009:
push 1
push 9
istore.8
jump.w ipentd00002
ipents00001: ; (3)
ipsrc00008:
push -1
push 167772161
locload 0
mcmp
j.eq ipsrc00009
ipalts00008: ; (4)
push 1
push 8
istore.8
ipsrc00009: ; (5)
push -1
push 167772162
locload 0
mcmp
j.eq ipentd00002

1. As we have said, we first check the destination IP: as NetIL does not support IP address notation, 10.9.9.9 is converted to its integer representation, 168364297. The j.eq instruction will cause execution to jump to 3 if the result of mcmp is zero, which happens if the address in the packet is 10.9.9.9.

2. This block is executed if the j.eq did not jump, which means that the destination address in the packet does not match 10.9.9.9. All it does is set to 1 bytes 8 and 9 of the info partition, which means that rules 8 and 9 are not matched. After that is done, we can skip all the tests on the source address at once and jump straight to the next test on the destination address, via the jump.w to 7.

3. If execution gets here, the destination address in the packet has already been matched, and we must now check the source address for each rule of the group. Here we check if the packet comes from 167772161, which is 10.0.0.1, as specified in rule 8. If it does, the j.eq brings us to test the next source, at 5.

4. If the source address at 3 did not match, we set rule 8 as not matched, then we go on with the next source test.

5. Here we test the source address against 10.0.0.2 (for rule 9), exactly as in 3.

6. If the test at 5 failed, we set rule 9 as not matched.

7. This group of rules has been fully tested, so a new group begins here.

5.2.6 The TCP module

As can be deduced from the pseudocode shown in algorithm 3, the operation of the TCP module is quite similar to that of the IPv4 module, with a couple of notable differences. The first difference is that it only has to deal with ports, avoiding the slight complication of handling netmasks. The second one is more important: as not all rules might include TCP details, this module has to treat them differently.
From the working model, it should be clear that if a packet reaches the TCP module, it surely contains TCP data. Hence, all the rules specifying UDP or ICMP as protocol are surely false. So, the first thing the TCP module does is to set all of these to false. Then the processing goes on similarly to the IPv4 module, with rules grouped by destination port. One last minor difference is that the next module is fixed (the connection tracking module), so no decisions have to be made.

The TCP module also updates the PAYLOAD_LEN, subtracting the length of the TCP header to its current value.

Algorithm 3 TCP module

1: Load TCP start offset from info partition
2: Load source port from packet
3: Save source port in TCP_SRC field
4: Save source port in NetVM_locals(0)
5: Load destination port from packet
6: Save destination port in TCP_DEST field
7: Save destination port in NetVM_locals(1)
8: for every group of rules do
9:   Load NetVM_locals(0)
10:  if NetVM_locals(0) == group_destination_port then
11:    for every rule in group do
12:      Load NetVM_locals(1)
13:      if NetVM_locals(1) != rule_source_port then
14:         Mark rule as not matched
15:      end if
16:    end for
17:  else
18:    Mark all rules of group as not matched
19:  end if
20: end for
21: Send exchange buffer to connection tracking module

5.2.7 The UDP module

The UDP module is, in line of principle, identical to the TCP module. The only notable difference takes place at the end of execution, as this module sends the packet straight to the string-matching module, as no Snort keywords specify tests needing the correlation of UDP packets.

The pseudocode implemented by the module is shown in algorithm [4]

5.2.8 The ICMP module

The ICMP module is the first example of a module whose behavior is set by rule options. Things get a little complicated, as any combination of them - including none - might appear in a rule. Furthermore, the syntax of such options is not fixed, as the “greater than” and “smaller than”
Algorithm 4 UDP module

1: Load UDP start offset from info partition
2: Load source port from packet
3: Save source port in $UDP_{SRC}$ field
4: Save source port in $NetVM_{locals}(0)$
5: Load destination port from packet
6: Save destination port in $UDP_{DEST}$ field
7: Save destination port in $NetVM_{locals}(1)$
8: for every group of rules do
9:   Load $NetVM_{locals}(0)$
10:  if $NetVM_{locals}(0) == group_{destination}_{port}$ then
11:     for every rule in group do
12:        Load $NetVM_{locals}(1)$
13:        if $NetVM_{locals}(1) != rule_{source}_{port}$ then
14:           Mark rule as not matched
15:        end if
16:     end for
17:  else
18:     Mark all rules of group as not matched
19:  end if
20: end for
21: Send exchange buffer to string-matching module

operators can be used. Our generated code must therefore be very flexible, and to achieve this we chose to use a naive approach again, where every single rule is treated separately from the rest. This can even be justified noting that ICMP rules are usually just a fraction of the total (137 over 3209 in the example Snort ruleset, $\sim 4.25\%$), so our performance constraints are a bit looser for this module.

The naive algorithm basically checks every ICMP header field required by the rule, making sure its value matches the one specified in the rule. As soon as a match attempt fails, the rule is marked false. This module also subtracts the ICMP header length from the $PAYLOAD_{LEN}$ field.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Type</th>
<th>Code</th>
<th>Checksum</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7. Format of an ICMP packet
**Code example**

Let us consider the following rule:

```plaintext
alert icmp any any -> 10.11.12.13 any ( \
    msg:"ICMP Address Mask Reply undefined code"; icode: >0; itype: 18;)
```

It will be translated to the following NetIL code:

```plaintext
icmpent00477:
icmpcd00477:
    push ICMP_OFFSET ; (1)
    uiload.8
    inc1
    upload.8
    push 0 ; (2)
    jcmp.le icmpalt00477
icmptp00477:
    push ICMP_OFFSET ; (3)
    uiload.8
    upload.8
    push 18 ; (4)
    jcmp.neq icmpalt00477
    jump.w icmpent00478
icmpalt00477:
    push 1 ; (5)
    push 477
    istore.8
icmpent00478: ; (6)
```

1. We use the ICMP_OFFSET field of the info partition (precalculated by the IPv4 module) to load the second byte of the ICMP header, where the ICMP code resides (see figure 5.7).

2. Here we check if the header value is less than the value specified in the rule. In such case, the rule is false (as it specifies greater than 0), and so we jump to 5, otherwise execution just goes on to 3.

3. We now test the ICMP type value, loading the first byte of the ICMP header.

4. If the header value is different from the value specified in the rule, we jump to 5, to mark the rule as not verified, otherwise tests for this rule are ended and we can jump to the next rule.
5. Here we mark the rule as not verified, then go on with the following rule.

6. Tests for the next rule begin here.

5.2.9 The string-matching module

The string-matching module is the core of an IDS application. This module is very processing-intensive, as string-matching is a complex task by its nature [28]. This is the reason why it has been splitted in a separate module, even though its operations would logically belong to the payload module. Besides, being put in this position, it reduces the work of the payload module (and of all the possible subsequent modules), as after passing this module, a small number of rules will still be true.

See Appendix 13 for a detailed description of the string-matching problem and of the most commonly used string matching algorithms.

Our choices

The Snort 1.1 branch used the Boyer-Moore algorithm for string-matching. The 2.x branch switched to an automaton-based multi-pattern algorithm, able to check all patterns at the same time, which is faster, but more complex, and which uses more memory, as we have seen. We chose to follow the first approach, as already-debugged code was available, and as it was considered good enough for our needs: in fact, we have seen that Boyer-Moore gives its best results when the length of the pattern is small with regard to the alphabet size, which is exactly our case: our alphabet size is 256, as we are considering patterns made up of raw bytes, and most patterns are only a few dozens of characters long. Although, we chose to implement the Boyer-Moore algorithm and two of its variants all together, to be able to evaluate different memory occupation - speed trade-offs. At the moment the algorithm to be used can be chosen with a command-line options, but the framework already supports utilizing different algorithms for different patterns. This could be used to tailor the choice of the algorithm to the patterns, whose properties are what most affects their efficiency.

The already available algorithms needed to be changed in some aspects, though. For instance, they were only able to deal with C-style strings, which only contain ASCII characters and are terminated by a NULL character. This cannot apply to Snort patterns, as they may contain arbitrary binary data (embedded in the rules with the bytecode syntax described in A.4), including the NULL character. Thus we needed to change them so that they keep track of the pattern lengths and ignore NULL characters. Another modification was needed because of the nocase keyword, as the algorithms were all case-sensitive. We changed the preprocessing and comparison phases, so that all characters are turned to upper-case. This leads to a little overhead when comparing strings, but that cannot be avoided.
5.2.10 The connection tracking module

The connection tracking module must keep track of the state of all the TCP connections it sees, and of who started them, so that it will be able to identify if a packet belongs to an established connection (which means that packets have already traveled in both directions) and if it was sent by the server or by the client.

The common way to implement connection tracking is through an hash table: the protocol, the source and destination IPs and the source and destination ports are concatenated, and then hashed. The obtained value is used to access a table where information about the connection is stored.

We planned to use this method, and we observed that keeping track of TCP connections is a generic task, which can potentially be used in several applications. Hence, we chose to split the connection tracking module in two submodules: the first has static code and updates tracking structures in memory, while the second uses the data elaborated by the first to test if the packet matches the rule options. This makes code reusing and maintaining much easier.

The connection tracking core

The core basically implements the standard TCP state diagram, as specified by RFC 793 \[27\] (figure 5.8).

The most interesting part is how the hash table was implemented. We used the coprocessor architecture provided by the NetVM, writing a lookup coprocessor. Every time a packet passes through this submodule, the relevant fields get sent to the coprocessor, and the data it returns indicate the status of the connection the packet belongs to, and what direction the packet is traveling in. If necessary, the entry is updated, while if the coprocessor returns no data, a new entry is created. Only the three-way handshake phase is supported at the moment, leaving the implementation of connection closing, aborting and timeout to future work, as this was not vital for our goals.

A little problem arises when dealing with connections that have been set up before the application was started: for such connections, we have no means to distinguish packets coming from the client from those coming from the server, as we do not know who sent the initial SYN packet. All we can do, if we see packets in both directions, is assume that the connection is established. Therefore, packets belonging to such connections can be treated in two ways: they can either always match every to_client/to_server rule, or they can match none. We used the first approach for simplicity, but it seems that Snort uses the second one, hence this aspect should be investigated more deeply and changed, if necessary.
The rule matching submodule

After the connection tracking core has completed its task, it stores the current state of the connection to which the packet belongs in the CT_STATE field of the info partition. This is used by
the rule matching submodule to check which rule options are matched by the packet. The value is actually a bitmask:

- Bit 0 (TCP_ESTABLISHED) is set to 1 if the packet belongs to an established connection.
- Bit 8 (CT_TO_SERVER) is set to 1 if the packet comes from a client and is directed to a server.
- Bit 9 (CT_TO_CLIENT) is set to 1 if the packet comes from a server and is directed to a client.
- Bit 10 (CT_UNKNOWN) is set to 1 if the direction of the packet cannot be established. This usually means that the connection has been set up before the application was started, and therefore there is no way of figuring out the direction of the packet.

This is actually the same format that is used for the value stored in the hash table, and it makes the work of this module straightforward.

Algorithm 5 Connection tracking rule matching submodule

1: Load connection tracking state from info partition
2: if CT_TO_SERVER bit is set then
3: Mark all rules with to_client as not matched
4: else
5: if CT_TO_CLIENT bit is set then
6: Mark all rules with to_server as not matched
7: else
8: Mark all rules with to_client or to_server as not matched
9: end if
10: end if
11: if TCP_ESTABLISHED bit is not set then
12: Mark all rules with established as not matched
13: end if
14: Send exchange buffer to string matching module

5.2.11 The payload module

The payload module has a simple job, in the current implementation. All it has to do is to see if the payload size matches the one specified in the various rules. If more Snort options are implemented, this would be the place where to implement all the options that somehow deal with the data payload.

The payload size is loaded from the PAYLOAD_LEN field of the info partition. This field is updated by the previous modules, when necessary, so that when the exchange buffer arrives to this module, it contains the exact size of the payload. Detection of the size cannot be performed by
this PE only, as it requires knowledge of the various protocols, which is something that we want to avoid with the current architecture.

No rule grouping is performed by this module, as the payload size specified by the various rules is usually very different from rule to rule. This might be reconsidered if further rule options will be processed by this module. Although, the usual method of saving frequently used values in locals is used.

Algorithm 6 Payload module
1: Load payload size from info partition
2: Save payload size in local 0
3: for every rule do
4: Load NetVM_locals(0)
5: if NetVM_locals(0) != rule_dsize then
6: Mark rule as not matched
7: end if
8: end for
9: Send exchange buffer to output module

It is worth noting that the payload size can be specified in the rule with a relational operator, hence the actual code is slightly more complicated than the pseudocode, to handle all possible cases.

5.2.12 The output module

The output module constitutes the final step of our application. It reads the exchange buffer from the NetVM and outputs alerts for the matched rules. To do so, it iterates over the rule bits in the info partition, and whenever it finds one that is still set to true, it uses the struct rulehead list that was in memory to output the proper message, along with information about the packet and an optional packet dump.

This step actually takes place out of the NetVM, and uses a callback mechanism.

Algorithm 7 Output module
1: for every rule do
2: Load rule bit from info partition
3: if rule_bit == 0 then
4: Do output
5: end if
6: end for
5.3 Limitations and alternatives

Despite the advantages we have already seen, the chosen architecture also imposes some constraints. The biggest problem is that it is unsuitable to pure asynchronous parallelism, as every module depends on data which must be computed by the preceding modules. A way to work around this problem could be the use of a single module right at the NetVM entrance, which analyzes every incoming packet and calculates the offset of all the encapsulated protocols, much like the packet decoder does in Snort. The drawback with this approach is that the knowledge of protocols would not be concentrated in a single place anymore, making the addition of support for more protocols slightly harder. Anyway, the current architecture can be pipelined, which should still boost performance.

Another problem with the architecture is that it does not support tunnelled packets, as they contain more than one network-level protocol. Although, this can be solved by adding some connections between the network-level modules and with some minor code changes.

5.4 Integration with the POSITIF project

POSITIF is a project funded by the European Commission whose aim is to develop a modular architecture to handle security-related tasks in a policy-driven fashion. Without going into detail, POSITIF can make use of various types of probes to detect intrusions, including Snort.

As the POSITIF project is being partly developed at the Politecnico di Torino and it can use Snort as a sensor, we thought that it could be an ideal environment for stress-tests of our NetVMSnort application, hence some investigation has been made to make the integration possible.

All communications within the framework use a well-defined XML format (See figure 5.9, keeping in mind that Prelude uses Snort as a sensor). Every component which has its own configuration format and/or output format is then wrapped by another module, whose task is to translate between the formats so that all communications remain consistent.

Figure 5.9. Proactive Security Monitor (PSM) of the POSITIF framework

Therefore, should our application be integrated in the POSITIF framework, we could make
direct use of the existing Snort wrapper, as our aim is to be 100% compatible with Snort. Although, as our output system is not yet defined, we could even write a custom wrapper module which only translates input data, while already outputting data in the required XML format, called IDMEF. Figure 5.10 shows how the wrapping is done with the currently supported IDSs.

Figure 5.10. Wrapping of an IDS in the POSITIF framework
Chapter 6

Modifications to NetVM

At the beginning of this thesis, the NetVM was in alpha stage. Many bugs were still present in its code, and some things were not working as expected. Besides finding and correcting all of such problems, during the course of the work we identified some aspects of the NetVM to be unsatisfying or subject to some kind of improvement. Some things had to be done in a too complicated way, while others could just not be done. Thus, after some investigation, we have proposed and implemented some modifications to the architecture and implementation of NetVM.

6.1 Exchange buffers

Before our work, Exchange buffers were actually called packet buffers. This emphasizes the fact that they were originally intended to contain the bare packet data. The documentation stated that they could contain any kind of data, but this was only partially true: problems turned up when we had the need to append some NetPE-elaborated data to the packet, so that NetPEs further in the chain could access and use them. It may be objected that this could be done through shared memory, but this actually introduces the need to coordinate data accesses, to avoid race conditions, which is an aspect that has been overlooked by the NetVM, which sees shared memory more as a way to share static global data to all the NetPEs. Anyway, even if we wanted to store packet-related data in shared memory, we would still need to pass the pointer to the data on to the following NetPEs. Therefore a mechanism had to be designed to address this issue.

We wanted to keep the packet buffer design as simple and efficient as possible, so we simply splitted it in two parts, called respectively packet partition and info partition. The former is meant to hold the packet, while the second can contain arbitrary data. The division is purely logical: the whole buffer is exchanged among the NetPEs as a single entity, which results in the data being “attached” to the packet. This is the reason behind the name change to exchange buffer.

The size of both partitions can be specified in a NetIL program. It is a once-for-all definition,
which must be the same for all the NetPEs on every path. If a NetPE does not agree on the size, the NetVM will throw an exception during its initialization.

From a programmer’s point of view, access to the packet data works the same as before. New NetIL instructions have been introduced to manipulate data in the info partition. They are modelled after those used to access the other types of memory:

**Info partition load instructions**

- **uiload.8**: Load an unsigned 8 bit int from the info field onto the stack after conversion to a 32 bit int.
- **uiload.16**: Load an unsigned 16 bit int from the info field onto the stack after conversion to a 32 bit int.
- **siload.8**: Load a signed 8 bit int from the info field onto the stack after conversion to a 32 bit int.
- **siload.16**: Load a signed 16 bit int from the info field onto the stack after conversion to a 32 bit int.
- **siload.32**: Load a signed 32 bit int from the info field onto the stack.

**Info partition store instructions**

- **istore.8**: Store an 8bit int from the stack to info field, possibly truncating it from 32bit int (only least significant bits are stored).
- **istore.16**: Store a 16bit int from the stack to info field, possibly truncating it from 32bit int (only least significant bits are stored).
- **istore.32**: Store a 32bit int from the stack to info field.

The framework guarantees that the info partition is initialized so that it contains all zeroes when an exchange buffer first enters the NetVM.

### 6.2 Coprocessors

The original NetVM specification planned that functions provided by specialized hardware were to be exposed within the NetVM framework by the means of *coprocessors*, even though details had still to be defined and no interface had been designed. During development, we found out that the concept of “coprocessor” could lead to two different implementations.
The first one is the most immediate abstraction of real hardware coprocessors. This kind of coprocessor is meant to extend the NetIL instruction set, providing custom features. I/O is register-based and invocation resembles a software interrupt. All registers have the same size, 32-bit, and they can either be read-only, write-only or read-write. The intended architecture is shown in figure 6.1.

![Architecture of a coprocessor](image)

Figure 6.1. Architecture of a coprocessor

The normal NetIL program execution is suspended while a coprocessor is working, as this greatly simplifies the architecture, and saves us the need to implement support for asynchronous communications. All coprocessors are ideally connected to a common bus, shared with all NetPEs, as shown in figure 6.2. This means that every NetPE can make use of every coprocessor, even though this aspect arises problems with parallel execution of NetPEs, which should be solved in the future. The intended working model is the following:

1. The application writes data to the coprocessor, through its input registers.
2. The application invokes an operation on the coprocessor.
3. The coprocessor executes the requested operation, processing input data and creating output data.
4. The application reads the result of the operation through the output registers of the coprocessor.

Coprocessors are identified in NetIL code by a well-known unique name, while their registers and the operations they support are identified by a number. The application must be aware of how to control the coprocessor. Thus, every coprocessor should be accompanied by the documentation of its interface.
A standard set of coprocessors will be defined and implemented in the NetVM framework. Although, implementations do not have to include the full set. They may include only a subset, or even none, and they can also provide customized coprocessors. Besides, some coprocessors may correspond to real hardware, while others can be emulated. A new NetIL section has been introduced, to allow NetIL code to specify which coprocessors it needs. Obviously, if a module requests an unimplemented coprocessor, an exception will be thrown.

Programming Interface

To make use of a coprocessor in NetIL code, an application must declare it in its .coprocessors segment. The following is an example to make use of the lookup coprocessor:

```
segment .coprocessors
  use_coprocessor lookup
ends
```

NetIL instructions have been added to interface with coprocessors:

- **copro.out**: This instruction is used to write data to an input register of a coprocessor. The data are taken from the top of the stack.
  - Syntax: `copro.out <coprocessor>, <register>`
  - Example: Write 10 to register 0 of the lookup coprocessor:
push 10

copro.out lookup, 0

- **copro.in**: This instruction reads data from an output register of a coprocessor and pushes them on the stack.
  - Syntax: `copro.in <coprocessor>, <register>`
  - Example: Read from register 1 of the `lookup` coprocessor:
    `copro.in lookup, 1`

- **copro.invoke**: This instruction passes control to a coprocessor and waits for its operation to finish.
  - Syntax: `copro.invoke <coprocessor>, <operation>`
  - Example: Invoke operation 4 of the `lookup` coprocessor:
    `copro.invoke lookup, 4`

### The lookup coprocessor

As we have seen in 5.2.10, our connection tracking module was implemented using a `lookup` coprocessor, written for the purpose. Interaction with the coprocessor is performed through two I/O registers. The first register is called *data* register, and is used to read data from the coprocessor and to write data to it. The second register is called *valid* register, and is used when performing a lookup to recognize a valid result.

The coprocessor supports five operations:

1. **Initialize**: Quite obviously, this must be performed to initialize the coprocessor to a valid state. It sets up the hash table to be totally empty.

2. **Add data**: Data must be written to the coprocessor in 32-bit chunks, because the size of the I/O registers is fixed. Therefore this operation must be used several times to provide the coprocessor the data that needs to be hashed. In our case we will use it with the protocol, the source and destination IPs and the source and destination ports every time we need to add a new connection to the table or to read the state of an existing connection.

3. **Add value**: This operation is used to store the data which must be attached to a hash. We use it when we add a new connection to the table. The value can only be 32-bit long.

4. **Read value**: This operation reads the value attached to the hash of the current data. After using it, the *valid* register must be checked. If it is set to zero, the data does not appear in the table and thus the *data* register has no valid data. If it is set to anything different
from zero, the data is present in the table and the attached value can be read from the data register.

5. **Reset**: This must be called to reset the coprocessor state between two hashes of different data.

   It is important to note that if a read value operation returned invalid data, then an add value operation can immediately follow. The coprocessor itself is quite generic, and can be used in other contexts as well. This approach would make our application benefit very much from an hardware implementation of the NetVM. In this case, the coprocessor could be mapped to a hardware device, like a content-addressable memory (better known as CAM), delivering instant results.

### 6.3 NativePEs

The common way of using coprocessors has a problem with our architecture: it does not allow access either to the packet data, nor to the shared memory. This reduces the possible uses of coprocessors, as, for instance, a coprocessor checking the IP checksum of a packet could not be implemented. For this reason, we also designed an alternative coprocessor architecture, which basically consists in a coprocessor having the same interface as NetPE. This means that it exposes a number of input and output ports and works in an event-oriented fashion. We decided to call this kind of coprocessor *NativePE*, because it looks like a PE, but it is programmed in native C code.

The NativePE is also useful as a debug tool, as it has the practical effect of allowing complicated NetPE code to be written in the C language rather than in NetIL, when using a general-purpose implementation of the NetVM. This also leads to faster prototyping of NetVM applications: initial code can be written in C and later converted to NetIL, after making sure that all NetPEs are properly connected and the whole NetVM is working as expected.

Although, NativePEs do not have a direct mapping to any components in possible hardware implementations of the NetVM framework. They are not either portable, therefore they should not be used until details are further defined.

**Programming Interface**

An application wanting to make use of a NativePE, must define the `ENABLE_NATIVE_PE` symbol before including `netvm.h`. This enables the following function, used to create a NativePE object:

```c
nvmNativePEState *nvmNativeNetPeCreate (nvmNetVMState *VmState);
```

Once the object has been created, the following function allows to define I/O ports and the handlers which will be called upon incoming events:
int32_t nvmNativeNetPeRegister (nvmNativePEState *pe,
    u_int32_t ports_no, u_int32_t ports[],
    nvmInitFunction *_init, nvmPushFunction *_push,
    nvmPullFunction *_pull, void *_userdata);

The meaning of the parameters is as follows:

- *ports_no* is the number of ports the PE should have.
- *ports* is an array of *ports_no* elements, defining the type of each port. The following constants can be used:
  - nvmPORT_EXPORTER defines an input port.
  - nvmPORT_COLLECTOR defines an output port.
  - nvmCONNECTION_PUSH defines a push port.
  - nvmCONNECTION_PULL defines a pull port.

Each port must be an input or output port and must define a push or pull type. Hence, the above constants can be OR'ed as necessary.

- *_init* is a function which will be called when the NativePE is initialized. It should conform to the following prototype:

  int32_t nvmInitFunction (nvmPEState *, nvmInternalPort *);

- *_push* and *_pull* are the handler functions which will be called when a push or pull port is triggered. Their prototype is the same as the *_init* function. Any pointer can be set to *NULL* if no function should be triggered by the corresponding event.

- Eventually, *userdata* is a pointer to arbitrary data, which will be available to the even-handling functions. It is provided to avoid the use of global variables. The use of a structure is recommended.

When a handler takes control, it should use the following initialization code:

```c
int32_t copro_push (nvmPEState *pe, nvmInternalPort *port) {
    u_int8_t *pkt, *info, *shmem, *data;
    nvmNativePEState *npe = nvmNATIVE_PE (pe);

    nvmNativePeGetData (npe, port, &pkt, &info, &shmem, &data);

    /* ... */
}
```

72
This way, pointers to the packet partition, the info partition, the shared memory and the data provided when calling \texttt{nvmNativeNetPeRegister} are properly initialized. Finally, when the exchange buffer has to be sent through an output port, the \texttt{nvmPacket\_Send()} function should be used.

The string-matching NativePE

In our work, we used the NativePE to implement the string-matching module. Of course, such a module needs to access the packet data, so the use of a coprocessor is not straightforward. Besides, the Boyer-Moore algorithm and its derivatives are complex to implement in NetIL code, and any implementation would need a long and thorough debugging phase, so the choice of a NativePE seemed a forced step.
Chapter 7

Results and conclusions

7.1 Testing methods and environment

The easiest way to measure the performance of our application is, of course, comparison with Snort. To make a fair comparison, the two applications should be put in the most similar conditions possible.

7.1.1 Hardware

All tests will be performed on the same hardware, that is a general-purpose machine featuring an Athlon64 3200+ processor with 512 KB of cache memory, running at 2000 MHz. Dynamic clock scaling will be disabled so that the processor will always run at full-speed. The machine has 1 gigabyte of RAM and is running the Linux kernel, version 2.6.19.2 in 32-bit mode. Tests have also been carried out under a pure 64-bit environment, but the results did not differ significantly, therefore they will not be presented.

7.1.2 Compilation

The GNU C compiler (GCC) will be used, version 3.4.6. The compilation flags and optimizations will be the same for both applications, oriented to achieve maximum execution speed on the hardware used for the testing. Therefore, we have chosen the following settings:

CFLAGS="-O2 -march=athlon-xp -m3dnow -fomit-frame-pointer"

This instructs GCC to generate code optimized for the Athlon XP family processors making use of the 3DNow! SIMD (Single Instruction - Multiple Data) extensions, and to perform two levels of optimizations. This way it will perform nearly all supported optimizations that do not involve a space-speed trade-off. Loop unrolling is therefore not performed.
As shown in table 7.1, sample runs show that our application gets a large benefit from this choice of flags. Other settings might provide even better performances, but this will be investigated in the future.

The obtained binaries will also be stripped of debugging symbols, to make them smaller and able to get more profit from caching.

### 7.1.3 Runtime

We define as “runtime” only the time spent processing packets, ignoring the time needed to read the rules from disk and to setup the NetVM configuration. Although, runtime includes the time needed to read the captured packets from disk or to capture them from the network. These choices have been made to reproduce the behaviour of the Snort built-in timers.

The time will be measured through calls to the system `gettimeofday()` function, which allows for microseconds accuracy, even though hundreds of seconds will be enough for our needs.

Every type of logging will be disabled, in order to avoid I/O operations and measure only the packet processing time.

Besides, both applications should have exactly the same inputs. This means that the same ruleset and the same input traffic should be used. As our application does not support the full default Snort ruleset yet, we added an option `-u` to dump the accepted rules to a new configuration file that we can use to run Snort.

To process the same traffic, we added a few more command-line options: `-w` is used to dump the captured traffic to a file, in the same format used by `tcpdump`, which can be read by all applications using `libpcap` (which includes Snort). On the contrary, the `-r` option can be used to read traffic from a similar file, instead of capturing packets from the network.

Our application will use the Boyer-Moore algorithm for string-matching, as it is the most thoroughly tested at the moment.

The Snort features not yet implemented in our application (like IP checksum verification or TCP flow reassembly, for instance) will be completely disabled.
7.1.4 Ruleset

As dictated by our initial design decisions, at the moment, our application does not support all of
the Snort rule options. Currently supported are tests on IP addresses, TCP ports, ICMP header
fields, payload length, TCP connection state, TCP flow direction, and presence of static patterns
in the payload, either at a random or preset position. The patterns can also be case-sensitive or
case-insensitive. Details about the supported options are presented in appendix A.

The currently-supported keywords lead to an accepted ruleset of 1002 rules, over the 3782
available in the official Snort ruleset, amounting to about 26%. As will be discussed later, the
major cause of this dropping is the lack of implementation of the `pcre` and `uricontent` keywords.

7.1.5 Input network traffic

The network traffic that will be used in the tests consists of 10.000.000 packets captured on a real
network, which means the tests should be quite representative of the real environment an IDS
is supposed to work in. All TCP sessions beginning before the start of the capture have been
removed.

7.1.6 Repetitions

To get more precise results, the tests will be run twelve times, recording the results of each run.
Then, the best and worst performances will then be eliminated, and the average of the ten remaining
results will be calculated.

The main data of interest will be the packets per second that the applications can process.

7.2 Test results

7.2.1 Performance data

Tables 7.2 and 7.3 show the performance data gathered from twelve runs of Snort and NetVMSnort,
respectively. Table 7.4 shows the memory used by both applications, emphasizing the fraction used
for the pattern matching components, while table 7.5 shows the size of the source and binary code
obtained compiling the rules for each module. Finally, table 7.6 shows the number of processed
packets, divided by protocol, and the number of generated alerts.

7.2.2 Performance considerations

Considering the packet processing time, the original Snort is roughly fifteen times faster than our
implementation. This was an expected result, because of many factors. First of all, our code is not
native, as it is run by the NetVM interpreter: this affects performance according to the quality
Table 7.2. Snort performance data

<table>
<thead>
<tr>
<th>Run time</th>
<th>Processed packets per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>102.358308</td>
<td>97696.03</td>
</tr>
<tr>
<td>103.101504</td>
<td>96991.80</td>
</tr>
<tr>
<td>102.414851</td>
<td>97642.09</td>
</tr>
<tr>
<td>102.607572</td>
<td>97458.69</td>
</tr>
<tr>
<td>104.958474</td>
<td>95275.78 W</td>
</tr>
<tr>
<td>102.124317</td>
<td>97919.87 B</td>
</tr>
<tr>
<td>104.304167</td>
<td>95873.45</td>
</tr>
<tr>
<td>102.245087</td>
<td>97804.21</td>
</tr>
<tr>
<td>104.456277</td>
<td>95733.84</td>
</tr>
<tr>
<td>102.378591</td>
<td>97676.67</td>
</tr>
<tr>
<td>104.225046</td>
<td>95946.23</td>
</tr>
<tr>
<td>102.880434</td>
<td>97200.21</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>97002.32</strong></td>
</tr>
</tbody>
</table>

Table 7.3. NetVMSnort performance data

<table>
<thead>
<tr>
<th>Run time</th>
<th>Processed packets per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1566.8</td>
<td>6382.44</td>
</tr>
<tr>
<td>1567.04</td>
<td>6381.46</td>
</tr>
<tr>
<td>1567.23</td>
<td>6380.68 W</td>
</tr>
<tr>
<td>1564.85</td>
<td>6390.39</td>
</tr>
<tr>
<td>1566.43</td>
<td>6383.94</td>
</tr>
<tr>
<td>1564.11</td>
<td>6393.41</td>
</tr>
<tr>
<td>1565.17</td>
<td>6389.08</td>
</tr>
<tr>
<td>1563.85</td>
<td>6394.48</td>
</tr>
<tr>
<td>1563.17</td>
<td>6397.26 B</td>
</tr>
<tr>
<td>1565.05</td>
<td>6389.57</td>
</tr>
<tr>
<td>1563.83</td>
<td>6394.56</td>
</tr>
<tr>
<td>1563.33</td>
<td>6396.60</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6389.59</strong></td>
</tr>
</tbody>
</table>

of the interpreter layer itself. At the moment, the only available implementation consists in a reference interpreter, which can still be largely improved. The importance of this step is confirmed

Table 7.4. Memory usage data

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Pattern matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort</td>
<td>80240 KiB</td>
<td>32.84 MiB</td>
</tr>
<tr>
<td>NetVMSnort</td>
<td>8592 KiB</td>
<td>1028.94 KiB</td>
</tr>
</tbody>
</table>
Table 7.5. Generated NetIL size

<table>
<thead>
<tr>
<th>Module</th>
<th>Source size (bytes)</th>
<th>Bytecode size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet</td>
<td>1313</td>
<td>267</td>
</tr>
<tr>
<td>IPv4</td>
<td>62775</td>
<td>15853</td>
</tr>
<tr>
<td>TCP</td>
<td>72722</td>
<td>15025</td>
</tr>
<tr>
<td>UDP</td>
<td>69295</td>
<td>13804</td>
</tr>
<tr>
<td>ICMP</td>
<td>81879</td>
<td>15921</td>
</tr>
<tr>
<td>Connection tracking</td>
<td>62425</td>
<td>15628</td>
</tr>
<tr>
<td>Payload</td>
<td>16856</td>
<td>822</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>367265</strong> (~ 359 KB)</td>
<td><strong>77320</strong> (~ 75 KB)</td>
</tr>
</tbody>
</table>

Table 7.6. Processed packets data

<table>
<thead>
<tr>
<th>Total packets</th>
<th>TCP</th>
<th>UDP</th>
<th>ICMP</th>
<th>Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>8439040</td>
<td>1505540</td>
<td>53429</td>
<td>393</td>
</tr>
</tbody>
</table>

by profiling information extracted from the program, shown in figure 7.1, which shows that the `nvmExecute_Instructions` function, which implements the interpreter, runs for more than 70% of the total execution time.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>70.57</td>
<td>1365.06</td>
<td>1365.06</td>
<td>2985137</td>
<td>nvmExecute_Instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.98</td>
<td>1538.72</td>
<td>173.65</td>
<td>17310</td>
<td>gettimeofday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.95</td>
<td>1711.81</td>
<td>7487360</td>
<td>499135</td>
<td>boyer_moorescasefindn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.66</td>
<td>1782.53</td>
<td>70.72</td>
<td>499135</td>
<td>copro_stringmatchpush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.89</td>
<td>1838.38</td>
<td>55.85</td>
<td>4425600</td>
<td>boyer_mooresfindn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1. NetVMSnort profiling information

It should be noted that the profiling information are gathered throughout the whole execution of the program, not only at runtime. The ~ 5% time not shown is mostly taken by initialization functions, and constitutes an error affecting the data in an asymptotic way. The large number of packets to be processed was chosen to minimize such error.

The second most intensive task which could potentially need improvements is, of course, string matching, which appears in the profile with the `boyer_mooresfindn`, `boyer_moorescasefindn` and `copro_stringmatchpush` functions. Although, the profiling data show that, at the moment, it does not represent a bottleneck for our application: it only covers about the 14% of the execution time, globally, while string-matching tasks in Snort take about 31% of the time [29]. Of course, this last comparison is improper, since the two applications use totally different algorithms for string-matching, which cannot be directly compared. Anyway, as our algorithm is surely slower, it cannot take less time, in percentage, than the one used in Snort, so this constitutes evidence that
the bottleneck must be somewhere else.

Besides, code generation for some modules can still be improved. For instance, the IP and TCP modules can be further optimized: when more than one rule shares both the same source and destination addresses/ports, the destination ones are checked only once, while the sources are checked for every rule, introducing some redundancy that could be avoided.

Our choice of the Boyer-Moore algorithm shows its goodness in the small memory amount necessary at runtime, which is around one megabyte, compared by the more than 32 needed by the deterministic finite automaton built by Snort, which, although, might be one of the reasons for its greater speed.

The memory used for code by our application seems very reasonable, amounting to about 75 kilobytes in bytecode format. Even considering the memory used by the rules and by the other structures the application needs, we can infer that a large portion of the memory is used by the NetVM, and that such amount might further decrease as optimizations to the framework are made. Furthermore, the NetVM bytecode is stack-based and, as such, tends to be more verbose than that of a real processor.

As a whole, we strongly believe that the main reason for the slowness of our application is the quality of the implementation of the interpreter. Actually, a JIT NetIL compiler was available when the work was started, but it could not be used at the time of testing because of the modifications applied to the NetVM framework, which introduced new instructions that will need to be implemented in the JIT engine. Furthermore, there is already work in progress to get an implementation of the NetVM framework on a real network processor. It is expected that performance will be boosted by these works.

7.3 Future improvements

Despite performance, there are many other aspects which can be improved in out NetVMSnort application. The most immediate is the support for more Snort keywords, so that a bigger number of rules can be used. The outstanding keywords that cause the biggest dropping of rules are the pcre and uricontent keywords. The former deals with matching Perl-Compatible Regular Expressions in packet payloads, and has not been implemented in the course of this work for complexity reasons. Although, now that the interface for coprocessors is mostly defined, it should be quite straightforward to implement a regular expression matching engine. The latter keyword, instead, makes Snort protocol-aware, allowing a specific string to be matched in the URI field of an HTTP request. This has also been overlooked for this initial implementation, but could easily be done by implementing an HTTP module. This is a further evidence of the ease of extension conferred to our application by the architecture choice.

Another aspect needing improvement is accuracy: there are some cases where discrepancies
between Snort and our application appear. Even though some bugs seem to be present in the way Snort counts packets, a revision of the used algorithms seems necessary, in order to make sure that they behave exactly the same way as Snort. Nevertheless, the results of this first batch of tests are quite reassuring.

Two things that we have completely overlooked are TCP flow reassembly and IP defragmentation. These should be investigated and implemented, as packet fragmentation is one of the most common IDS evasion techniques.

Lastly, the callback mechanism used for the output module is a potential bottleneck. This is evident when enabling the output of generated alerts. To avoid it, the callback function should return as soon as possible, delegating the results processing to a separate entity. This can be achieved through the use of threads, exactly like the Fast Logging Project for Snort \cite{fast-logging} does.
Appendix A

Supported keywords

Following is a list of the Snort rule options currently supported by our application, grouped by the modules that are in charge of checking them. More details can be found in the official Snort users manual [31].

A.1 Output module options

These options mainly have description and classification purposes. They do not specify anything to be matched in packets and, as such, are only used by the output module.

- **msg**: This option specifies the message to print along with a packet dump or with an alert.
  
  - Syntax: `msg: "<message text>";`

- **sid**: The *sid* keyword is used to uniquely identify Snort rules. This information allows output plugins to identify rules easily. This option should be used with the *rev* keyword. It is required by Snort and must be different for every rule.
  
  - Syntax: `sid: <snort rules id>;
  
  - Allowed values:
    
    * 100: Reserved for future use.
    * 100 - 1,000,000: Rules included with the Snort distribution.
    * 1,000,000: Used for local rules.

- **rev**: This keyword is used to uniquely identify revisions of Snort rules. Revisions, along with Snort rule IDs, allow signatures and descriptions to be refined and replaced with updated information. This option should be used with the *sid* keyword.
A – Supported keywords

- Syntax: `rev: <revision integer>`;

- **classtype**: This keyword specifies the type of attack detected by the rule. In Snort rules classes can be differently prioritized.
  - Syntax: `classtype: <class name>`;
  - Examples:
    - `classtype: attempted-dos;`
    - `classtype: web-application-attack;`
    - `classtype: suspicious-filename-detect;`

- **reference**: The `reference` keyword allows rules to include references to external attack identification systems. This option is to be used by the output module to provide links to additional information about the alert produced. This option can be used multiple times, to provide multiple references.
  - Syntax: `reference: <id system>,<id>`;
  - Examples:
    - `reference: bugtraq,1387;`
    - `reference: cve,2003-0727;`

A.2 ICMP module

These options can be used to search for determinate values in the ICMP header fields.

- **itype**: This keyword is used to check for a specific value in the ICMP header “type” field.
  - Syntax: `itype:[<>]<number>[<>]<number>;`
  - Example: Look for an ICMP type greater than 30:
    - `itype: >30;`

- **icode**: This keyword is used to check for a specific value in the ICMP header “code” field.
  - Syntax: `icode: [<>]<number>[<>]<number>;`
  - Example: Look for an ICMP code smaller than 20:
    - `icode: <20;`
• **icmp_id**: This keyword is used to check for a specific value in the ICMP header “id” field. This is useful because some covert channel programs (for instance, the stackeldruht distributed denial of service agent) use static ICMP fields when they communicate.
  
  - Syntax: `icmp_id: <number>;`
  - Example: Look for an ICMP ID of 0:
    ```
    icmp_id: 0;
    ```

• **icmp_seq**: This keyword is used to check for a specific value in the ICMP header sequence number field. This was introduced for the same reasons of the `icmp_id` keyword.
  
  - Syntax: `icmp_seq: <number>;`
  - Example: Look for an ICMP Sequence of 0:
    ```
    icmp_seq: 0;
    ```

### A.3 Payload module

These options can be used to specify some properties of the data payload of the packet. Note that the payload is intended as that contained in the inner protocol (i.e., usually transport layer protocol).

• **dsize**: This keyword is used to test the packet payload size. This may be used to check for abnormally sized packets. It is useful for detecting buffer overflows.
  
  - Syntax: `dsize: [<>]<number>[<>]<number>;`
  - Example: Look for payload size between 300 and 400 bytes:
    ```
    dsize: 300<>400;
    ```

### A.4 Content module

These keywords allow to configure the most important feature of Snort: looking for specific contents in the data payload.

• **content**: This keyword allows the user to set rules that search for specific content in the packet payload and trigger response based on that data. By default this test is case sensitive. The option data syntax is somewhat complex, as it can contain mixed text and binary data. The binary data is generally enclosed within pipe characters and represented as `bytecode`: bytecode represents binary data as hexadecimal numbers and is a good shorthand method
for describing complex binary data. Multiple content rules can be specified in one rule. This allows rules to be tailored for less false positives. If the option is preceded by a !, the alert will be triggered on packets that do not contain the specified content. The behavior of this keyword can be modified by the subsequent options, all of which relate to the immediately preceding content keyword.

- Syntax: \texttt{content: [!] "<content string>";}
- Examples: Look for the typical \texttt{GET} keyword of the HTTP protocol and for some binary data (example of bytecode):

  \begin{itemize}
  \item \texttt{content: ! "GET";}
  \item \texttt{content: "|5c 00|P|00|I|00|P|00|E|00 5c|"};
  \end{itemize}

- \textbf{nocase}: This keyword allows to specify that Snort should look for the specific pattern ignoring case. nocase modifies the previous content keyword in the rule.
  - Syntax: \texttt{nocase;}

- \textbf{depth}: This keyword allows to specify how far into the payload Snort should search for the specified pattern. depth modifies the previous content keyword in the rule.
  - Syntax: \texttt{depth: <number>;}
  - Example: Look for the specified pattern within the first 20 bytes of the payload:

  \begin{itemize}
  \item \texttt{content: "cgi-bin/phf"; depth: 20;}
  \end{itemize}

- \textbf{offset}: This keyword allows to specify where to start searching for a pattern within the payload, skipping a certain number of bytes at its beginning. offset modifies the previous content keyword in the rule.
  - Syntax: \texttt{offset: <number>;}
  - Example: Look for the specified pattern skipping the first 4 bytes of the payload:

  \begin{itemize}
  \item \texttt{content: "cgi-bin/phf"; offset: 4;}
  \end{itemize}

- \textbf{distance}: This keyword allows to ignore a certain number of bytes before starting to search for the specified pattern relative to the end of the previous pattern match. This works exactly the same as depth, except it is relative to the end of the last pattern match, instead of the beginning of the payload.
  - Syntax: \texttt{distance: <byte count>;}
  - Example: Look for “ABC” followed by any character and then by “DEF“:

  \begin{itemize}
  \item \texttt{content:"ABC"; content: "DEF"; distance: 1;}
  \end{itemize}
• **within**: This keyword makes sure that at most a certain number of bytes is between two pattern matches. It’s designed to be used in conjunction with the *distance* option.

  - Syntax: `within: <byte count>;`
  - Example: Look for “ABC” followed by “EFG” in up to 10 characters.

    `content:"ABC"; content: "EFG"; within:10;`

A.5 Connection tracking module

These options can be used to specify some properties of the connection the packet belongs to. Obviously they can only be used with a stateful protocol (i.e. TCP).

• **flow**: This keyword can be used to specify the state of the connection and the direction of a packet, using the client-server paradigm. For instance, this allows packets related to clients viewing web pages to be distinguished from servers serving them in the same network.

  - Syntax:

    ```
    flow: [(established|stateless)]
    [, (to_client|to_server|from_client|from_server)]
    ```

  - Example: Look for packets coming from the client side.

    `flow: from_client;`
Appendix B

String-matching algorithms

The string-matching problem consists in finding one, or more generally, all the occurrences of a string (more generally called a pattern) in a text. The pattern is usually denoted by $x = x[0...m-1]$, and its length is equal to $m$. The text is denoted by $y = y[0...n-1]$, and its length is equal to $n$. Both strings are built over a finite set of character called alphabet, denoted by $\Sigma$ whose size is equal to $\sigma$.

There are many string-matching algorithms, which can be grouped into classes. The first distinction can be made depending on which string, the pattern or the text, is given first. As our case is the former, we will only be focusing on algorithms which have been developed to solve that problem. Such algorithms can be further divided in two classes: those based on the use of automata and those using a sliding window mechanism.

Automaton-based algorithms build a deterministic finite state machine which is able to recognize the pattern, and they are mostly used when there is the need to search for multiple patterns at the same time. The problem with this kind of algorithms is that they tend to consume a lot of memory, as the number of states is often huge. Hence, they are only used when the sliding-window approach cannot be used, as, for instance, when dealing with an infinite number of patterns, represented by a regular expression.

The basic operation of sliding window algorithms is as follows: they scan the text with the help of a window whose size is generally equal to $m$. They first align the left ends of the window and the text, then compare the characters of the window with the characters of the pattern - this specific work is called an attempt - and after a whole match of the pattern or after a mismatch they shift the window to the right of a certain number of characters. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. We associate each attempt with the position $j$ in the text when the window is positioned on $y[j...j + m - 1]$. The various algorithms mainly differ in the preprocessing phase and in how they shift the window. The basic idea is to gain information on the patterns preprocessing them, so that the window can be shifted.
by several characters, leading to the end of the search more quickly, while keeping the certainty that no match will be missed.

In the following sections the most commonly-used algorithms (according to literature [32]) for string matching will be examined. Most results are very difficult to prove (some, indeed, were not proven until years after the development of the algorithms), therefore we shall have to take them for given.

B.1 The naive algorithm

The easiest algorithm that could be implemented has no preprocessing at all. It just tries all the possible positions in which the pattern could appear. This means that the window is shifted by a single character at each iteration. Quite obviously, this algorithm sets the lower performance limit, with its searching phase $O(mn)$.

B.1.1 Example implementation

```c
void BF(char *x, int m, char *y, int n) {
    int i, j;

    /* Searching */
    for (j = 0; j <= n - m; ++j) {
        for (i = 0; i < m && x[i] == y[i + j]; ++i);
        if (i >= m) OUTPUT(j);
    }
}
```

B.2 The Boyer-Moore algorithm

The Boyer-Moore algorithm is considered the most efficient algorithm for normal applications. Actually, most better-performing algorithms are simply improvements of Boyer-Moore, which therefore offers very good performance with a low-complexity implementation.

The first peculiarity of the Boyer-Moore algorithm is that it performs character comparisons from right to left. If it starts a search at the beginning of a text for the word “ANPANMAN”, for instance, it checks the eighth position of the text to see if it contains an $N$. If it finds the $N$, it moves to the seventh position to see if that contains the last $A$ of the word, and so on, until it checks the first position of the text for a $A$.

Why Boyer-Moore takes this backward approach is clearer when we consider what happens if the verification fails: for instance, let us suppose that instead of an $N$ in the eighth position, we find an $X$. The $X$ does not appear anywhere in $ANPANMAN$, and this means there is no match
for the search string at the very start of the text, or at the next seven positions following it, since those would all fall across the X as well, as shown in figure [B.1]. After checking just one character, we are able to skip ahead and start looking for a match starting at the ninth position of the text, just after the X.

\[
- - - - - - X - - - - - - \\
A N P A N M A N - - - - - - \\
- A N P A N M A N - - - - - - \\
- - A N P A N M A N - - - - - - \\
- - - A N P A N M A N - - - - - - \\
- - - - A N P A N M A N - - - - - - \\
- - - - - A N P A N M A N - - - - - - \\
- - - - - - A N P A N M A N - - - - - - \\
- - - - - - - A N P A N M A N
\]

Figure B.1. The X in position 8 excludes all 8 of the possible starting positions shown

This explains why the best-case performance of the algorithm, is $n/m$: in the best case, only one in $m$ characters needs to be checked. This also explains the somewhat counter-intuitive result that the longer the pattern we are looking for, the faster the algorithm will be usually able to find it.

The algorithm uses two functions, both computed in the preprocessing phase to decide by how much the window should be shifted. These two shift functions are called **good-suffix shift** (also called **matching shift**), and **bad-character shift** (also called **occurrence shift**): the former calculates how many positions ahead to start the next search based on how many characters were matched successfully before the match attempt failed, while the latter makes a similar calculation based on the identity of the character that caused the match attempt to fail.

### B.2.1 The good-suffix shift

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i + j] = b$ of the text during an attempt at position $j$. Then, $x[i + 1...m - 1] = y[i + j + 1...j + m - 1] = u$ and $x[i] \neq y[i + j]$. The good-suffix shift consists in aligning the segment $y[i + j + 1...j + m - 1] = x[i + 1...m - 1]$ with its rightmost occurrence in $x$ that is preceded by a character different from $x[i]$.

If there exists no such segment, the shift consists in aligning the longest suffix $v$ of $y[i + j + 1...j + m - 1]$ with a matching prefix of $x$.

### B.2.2 The bad-character shift

The bad-character shift consists in aligning the text character $y[i + j]$ with its rightmost occurrence in $x[0...m - 2]$. 

88
Figure B.2. The good-suffix shift, \( u \) re-occurs preceded by a character \( c \) different from \( a \).

Figure B.3. The good-suffix shift, only a suffix of \( u \) re-occurs in \( x \).

If \( y[i+j] \) does not occur in the pattern \( x \), no occurrence of \( x \) in \( y \) can include \( y[i+j] \), and the left end of the window is aligned with the character immediately after \( y[i+j] \), namely \( y[i+j+1] \).

### B.2.3 Memory and time requirements

The good-suffix shift function is stored in a table of size \( m + 1 \), while the bad-character shift function is stored in a table of size \( \sigma \). Both tables can be precomputed in time \( O(m + \sigma) \) before the searching phase and require an extra-space of \( O(m + \sigma) \). The searching phase time complexity is quadratic, but at most \( 3n \) text character comparisons are performed when searching for a non-periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for \( a^{m-1}b \) in \( b^n \) the algorithm makes only \( O(n/m) \) comparisons, which is the absolute theoretical minimum for any string-matching algorithm in the model where

Figure B.4. The bad-character shift, \( a \) occurs in \( x \).
only the pattern is preprocessed.

Note that each time the window needs to be shifted, the Boyer-Moore algorithm uses the maximum value between the good-suffix shift and the bad-character shift. This is particularly important, as the bad-character shift can be negative and can therefore cause the algorithm to loop indefinitely. Although, the good-suffix shift is always positive, so the value it provides will be used in such cases.

### B.2.4 Example implementation

```c
void preBmBc(char *x, int m, int bmBc[]) {
    int i;
    for (i = 0; i < ASIZE; ++i) 
        bmBc[i] = m;
    for (i = 0; i < m - 1; ++i)
        bmBc[x[i]] = m - i - 1;
}

void suffixes(char *x, int m, int *suff) {
    int f, g, i;
    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && x[g] == x[g + m - 1 - f])
                --g;
            suff[i] = f - g;
        }
    }
}
```
B – String-matching algorithms

B.3 The Turbo Boyer-Moore algorithm

The Turbo Boyer-Moore algorithm is an amelioration of the Boyer-Moore algorithm. It needs no extra preprocessing, but only a constant small extra space with respect to the original algorithm. It consists in remembering the factor of the text that matched a suffix of the pattern during the last attempt, only if a good-suffix shift was performed. This technique presents two advantages:
• It is possible to jump over this factor.

• It allows to perform a turbo-shift, in some cases.

A \textit{turbo-shift} can occur if during the current attempt the suffix of the pattern that matches the text is shorter than the one remembered from the preceding attempt. In this case, let us call $u$ the remembered factor and $v$ the suffix matched during the current attempt such that $uzv$ is a suffix of $x$. Let $a$ and $b$ be the characters that caused the mismatch during the current attempt in the pattern and the text respectively. Then $av$ is a suffix of $x$, and thus of $u$ since $|v| < |u|$. The two characters $a$ and $b$ occur at a distance $p$ in the text, and the suffix of $x$ of length $|uzv|$ has a period of length $p = |zv|$ since $u$ is a border of $uzv$, thus it cannot overlap both occurrences of two different characters $a$ and $b$, at distance $p$, in the text. The smallest shift possible has length $|u| - |v|$, which constitutes the turbo-shift.

Still in the case where $|v| < |u|$, if the length of the bad-character shift is larger than the length of the good-suffix shift and the length of the turbo-shift, then the length of the actual shift must be greater or equal to $|u| + 1$. Indeed, in this case the two characters $c$ and $d$ are different since we assumed that the previous shift was a good-suffix shift. Then a shift greater than the turbo-shift but smaller than $|u| + 1$ would align $c$ and $d$ with a same character in $v$. Thus in this case the length of the actual shift must be at least equal to $|u| + 1$.

The preprocessing phase can be performed in $O(m + \sigma)$ time and space complexity. The searching phase is in $O(n)$ time complexity. The number of text character comparisons performed by the Turbo Boyer-Moore algorithm is bounded by $2n$.

\subsection{Example implementation}

The precomputing functions $preBmBc$ and $preBmBc$ used by the following implementation are the same of the Boyer-Moore algorithm shown above.
**B – String-matching algorithms**

```c
void TBM(char *x, int m, char *y, int n) {
    int bcShift, i, j, shift, u, v, turboShift,
        bmGs[XSIZE], bmBc[ASIZE];

    /* Preprocessing */
    preBmGs(x, m, bmGs);
    preBmBc(x, m, bmBc);

    /* Searching */
    j = u = 0;
    shift = m;
    while (j <= n - m) {
        i = m - 1;
        while (i >= 0 && x[i] == y[i + j]) {
            --i;
            if (u != 0 && i == m - 1 - shift)
                i -= u;
        }
        if (i < 0) {
            OUTPUT(j);
            shift = bmGs[0];
            u = m - shift;
        } else {
            v = m - 1 - i;
            turboShift = u - v;
            bcShift = bmBc[y[i + j]] - m + 1 + i;
            shift = MAX(turboShift, bcShift);
            shift = MAX(shift, bmGs[i]);
            if (shift == bmGs[i])
                u = MIN(m - shift, v);
            else {
                if (turboShift < bcShift)
                    shift = MAX(shift, u + 1);
                u = 0;
            }
        }
    }
}
```

Figure B.7. \( c \neq d \), so they cannot be aligned with the same character in \( v \).
B.4 The Horspool algorithm

The bad-character shift used in the Boyer-Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it alone produces a very efficient algorithm in practice. Horspool proposed to use only the bad-character shift of the rightmost character of the window to compute the shifts in the Boyer-Moore algorithm.

The preprocessing phase is in $O(m + \sigma)$ time and $O(\sigma)$ space complexity. The searching phase has a quadratic worst case but it can be proven that the average number of comparisons for a single character is between $1/\sigma$ and $2/\sigma + 1$.

B.4.1 Example implementation

The precomputing function $preBmBc$ used by the following implementation is the same of the Boyer-Moore algorithm shown above.

```c
void HORSPOOL(char *x, int m, char *y, int n) {
    int j, bmBc[ASIZE];
    char c;

    /* Preprocessing */
    preBmBc(x, m, bmBc);

    /* Searching */
    j = 0;
    while (j <= n - m) {
        c = y[j + m - 1];
        if (x[m - 1] == c && memcmp(x, y + j, m - 1) == 0)
            OUTPUT(j);
        j += bmBc[c];
    }
}
```
Bibliography

[1] “Snort - The de-facto standard for intrusion detection”,


[8] Paul Russell et al. - “Netfilter”,


[10] “libpcap”,


http://flex.sourceforge.net

http://www.gnu.org/software/bison/


http://www.positif.org


http://www.geschke-online.de/FLoP/


[32] Christian Charras, Thierry Lecroq - “Exact string matching algorithms”,
http://www-igm.univ-mlv.fr/~lecroq/string/index.html
List of Figures

1.1 Un possibile modo per interconnettere i moduli dell’applicazione . . . . . . . . . . 12
1.2 L’architettura di NetVM-Snort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.3 La struttura dell’exchange buffer . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1.4 Diagramma di flusso di NetVMSnort . . . . . . . . . . . . . . . . . . . . . . . . . . 15
1.5 Il diagramma degli stati TCP definito dalla RFC 793 . . . . . . . . . . . . . . . . 19
3.1 Internal architecture of the Snort IDS . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1 The NetVM architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.2 Internals of a NetPE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3 The NetPE port connection model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.1 A possible way to interconnect the application modules . . . . . . . . . . . . . . . 44
5.2 The NetVM-Snort architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.3 The exchange buffer layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.4 Flowchart of the program operation . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.5 Layout of the structures used to store rules in memory . . . . . . . . . . . . . . . 50
5.6 Format of the IEEE 802.3/Ethernet II frame . . . . . . . . . . . . . . . . . . . . . 52
5.7 Format of an ICMP packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.8 The TCP state transition diagram, as specified in RFC 793 . . . . . . . . . . . . . 61
5.9 Proactive Security Monitor (PSM) of the POSITIF framework . . . . . . . . . . . 64
5.10 Wrapping of an IDS in the POSITIF framework . . . . . . . . . . . . . . . . . . . . 65
6.1 Architecture of a coprocessor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.2 Ideal connection of a coprocessor . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.1 NetVMSnort profiling information . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
B.1 The X in position 8 excludes all 8 of the possible starting positions shown . . . . . 88
B.2 The good-suffix shift, $u$ re-occurs preceded by a character $c$ different from $a$ . . 89
List of Figures

B.3 The good-suffix shift, only a suffix of $u$ re-occurs in $x$. 

B.4 The bad-character shift, $a$ occurs in $x$. 

B.5 The bad-character shift, $b$ does not occur in $x$. 

B.6 A turbo-shift can apply when $|v| < |u|$. 

B.7 $c \neq d$, so they cannot be aligned with the same character in $v$. 

89

89

90

92

93
<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ethernet module</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>IPv4 module</td>
<td>53</td>
</tr>
<tr>
<td>3</td>
<td>TCP module</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>UDP module</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Connection tracking rule matching submodule</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Payload module</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>Output module</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Risultati dei test ................................................................. 22
7.1 NetVMSnort performance with different compiler settings .............. 75
7.2 Snort performance data ................................................................. 77
7.3 NetVMSnort performance data ......................................................... 77
7.4 Memory usage data ............................................................... 77
7.5 Generated NetIL size ................................................................. 78
7.6 Processed packets data ................................................................. 78