



**Politecnico  
di Torino**

Master of Science in Computer Engineering

Tesi di Laurea Magistrale

# Implementazione e sviluppo di software Demo per Verefoo

## **Supervisors**

prof. Fulvio Valenza

prof. Riccardo Sisto

dott. Daniele Bringhenti

## **Candidate**

Benito MARRA

ANNO ACCADEMICO 2023-2024



# Summary

Text of the summary

# Acknowledgements

Acknowledgement (optional)

# Contents

<b>List of Figures</b>	7
<b>List of Tables</b>	8
<b>Listings</b>	9
<b>1 Introduzione</b>	11
1.1 Obiettivo della Tesi . . . . .	11
1.2 Descrizione della tesi . . . . .	11
<b>2 Network Security Automation e Verefoo</b>	13
2.1 Service Function Chain . . . . .	13
2.2 Verefoo . . . . .	15
2.2.1 Introduzione . . . . .	15
2.2.2 Descrizione del modello . . . . .	15
2.3 Definizione ed esempio di Grafo di servizio e di allocazione . . . . .	18
2.4 Definizione ed esempi delle proprietà di sicurezza . . . . .	22
2.4.1 Reachability Requirements . . . . .	23
2.4.2 Isolation Requirements . . . . .	25
2.4.3 Protection Requirements . . . . .	27
<b>3 Docker</b>	31
3.1 Differenze fra Container e Macchine Virtuali . . . . .	31
3.2 Docker e la sua architettura . . . . .	34
3.3 Il tool Compose . . . . .	35
3.3.1 Introduzione . . . . .	35
3.3.2 YAML . . . . .	36
3.3.3 Services e Networking . . . . .	37

<b>4</b>	<b>Obiettivi della tesi</b>	<b>40</b>
<b>5</b>	<b>Correzione, sviluppo e ottimizzazione Demo A</b>	<b>42</b>
5.1	Introduzione alla Demo . . . . .	42
5.2	Sviluppo Installer . . . . .	43
5.3	Implementazione . . . . .	47
5.4	Output . . . . .	50
5.5	Verifiche e Test . . . . .	53
<b>6</b>	<b>Merge</b>	<b>57</b>
<b>7</b>	<b>DemoB</b>	<b>58</b>
<b>8</b>	<b>Conclusions</b>	<b>61</b>
	<b>Bibliography</b>	<b>62</b>

# List of Figures

2.1	Esempio di Service Function Chain . . . . .	14
2.2	Architettura di VEREFOO [1] . . . . .	16
2.3	Esempi di Service (in alto) e Allocation(in basso) Graphs [1] . . . . .	17
2.4	Esempio di un Service Graph . . . . .	19
2.5	Grafo di allocazione per requisito di raggiungibilità . . . . .	24
2.6	Output grafo di esempio per requisito di raggiungibilità . . . . .	25
2.7	Output grafo di esempio per requisito di isolamento . . . . .	26
2.8	Grafo di allocazione d'esempio per requisiti di protezione . . . . .	29
2.9	Output grafo di esempio per requisito di protezione . . . . .	29
3.1	Architettura Virtual Machine e Containers . . . . .	32
3.2	Architettura Docker . . . . .	34
5.1	Service Graph Demo A . . . . .	47
5.2	Verefoo Output Demo A . . . . .	50
5.3	VpnGateway4 . . . . .	53
5.4	VpnGateway5 . . . . .	53
5.5	Verifica Tunnel VPN . . . . .	54
5.6	Tunnel stabilito . . . . .	55
5.7	Prova di correttezza . . . . .	55

# List of Tables

5.1	Node definitions and functionalities . . . . .	48
5.2	Security Requirements Definition DemoA . . . . .	48
5.3	VPN Gateway 1 . . . . .	51
5.4	VPN Gateway 2 . . . . .	51
5.5	VPN Gateway 3 . . . . .	51
5.6	VPN Gateway 4 . . . . .	51
5.7	VPN Gateway 5 . . . . .	51
5.8	VPN Gateway 6 . . . . .	52
7.1	Esempio di tabella con tre colonne senza linee verticali. . . . .	58
7.2	Esempio di tabella con tre colonne senza linee verticali. . . . .	59
7.3	Esempio di tabella con tre colonne senza linee verticali. . . . .	59
7.4	Esempio di tabella con tre colonne senza linee verticali. . . . .	59
7.5	Esempio di tabella con tre colonne senza linee verticali. . . . .	60



# Listings

2.1	Definizione nodi del ServiceGraph 2.4 . . . . .	19
2.2	Definizione Allocation Place per Allocation Graph . . . . .	21
2.3	Definizione di una proprietà di sicurezza generica . . . . .	22
2.4	Esempio di requisito di raggiungibilità . . . . .	24
2.5	Esempio di requisito di isolamento . . . . .	26
2.6	Definizione di una proprietà di protezione . . . . .	27
2.7	Esempio di requisito di protezione . . . . .	29
3.1	Esempio definizione di un file YAML . . . . .	36
3.2	Esempio file di configurazione docker-compose . . . . .	39
5.1	Installazione packages curl e pv . . . . .	44
5.2	Installazione java openjdk . . . . .	45
5.3	Installazione Docker e Docker Compose . . . . .	45
5.4	Installazione Z3 . . . . .	46
5.5	Definizione di Services dell'ambiente virtuale DemoA . . . . .	49



# Chapter 1

## Introduzione

### 1.1 Obiettivo della Tesi

Nell'ultimo decennio diverse tecnologie di rete si sono sviluppate, creando reti sempre più robuste ed efficienti. In questo scenario, una tecnologia in particolare, la "*Network Function Virtualization*"(NFV) ha reso possibile creare delle reti che svolgono le funzioni di sicurezza e trasporto dei dati tramite la virtualizzazione di quest'ultime. Ciò ha permesso di definire le "*Software Defined Network*"(SDN) che introducono la possibilità di controllare le operazioni di rete tramite software.

Sfruttando queste tecnologie è stato sviluppato Verefoo(VERified REfinement and Optimized Orchestration) cioè un framework in grado di riuscire ad implementare nei nodi della rete i vari "*Network Security Requirements*"(NSR) in una topologia predefinita e fornita come input al framework.

### 1.2 Descrizione della tesi

Dopo aver spiegato nel Capitolo [1] gli obiettivi ed il lavoro prodotto per raggiungerli, il resto della tesi è definito nel seguente modo:

- Nella prima parte del Capitolo [2] si introduce il problema della Network Security Automation e si descrive il framework di Verefoo, ponendo particolare attenzione sul suo funzionamento ad alto e basso livello. Nella seconda parte sono descritte le definizioni delle Proprietà di sicurezza da passare come input al framework, con una spiegazione dettagliata di come queste intervengono nella definizione della topologia finale che verrà fornita come output dal framewrok.  
Infine verranno introdotti i grafi che verefoo richiede ed utilizza nella computazione dei vari *NSF*.
- Il Capitolo [3] definisce l'architettura di docker, specificando la differenza tra usare docker per la virtualizzazione e delle semplici macchine virtuali. Successivamente viene fatto un approfondimento sul docker-compose, un tool in

grado di poter istanziare più container velocemente tramite script. Nella parte finale viene spiegato come effettuare il networking sui container istanziati, come definirlo tramite docker-compose e come testare le comunicazioni in modo efficiente.

- Il Capitolo [4] descrive gli obiettivi posti all’inizio di questo lavoro di tesi. Più specificatamente, per ogni obiettivo presente verranno specificate le modalità e le scelte effettuate per portarlo a termine con una descrizione accurata dei vari passi che sono stati svolti prima della soluzione definitiva. Inoltre viene descritto in maniera più profonda rispetto a questo indice la descrizione dei futuri capitoli.
- Il Capitolo [5] descrive i lavori svolti nella prima delle due demo di cui questa tesi tratterà. Inizialmente viene descritto tramite pezzi di codice lo sviluppo dell’installer prodotto affinché un qualsiasi utente possa utilizzare la demo in maniera pratica ed agile. Nei paragrafi successivi vengono evidenziati i punti critici incontrati, elencando le modifiche apportate affinché essa possa funzionare correttamente. Nell’ultimo paragrafo infine verranno specificati ulteriori upgrade che si possono inserire nella demo per mettere in mostra in maniera ancora più evidente il lavoro svolto da Verefoo.
- Il Capitolo [6] descrive i lavori svolti ed implementati su Verefoo. In questo capitolo viene descritto il processo di merge fra le versioni precedentemente esistenti di Verefoo. Successivamente verrà quindi spiegato, anche tramite frammenti di codice, gli step che il framework eseguirà per produrre in output una rete che soddisfi contemporaneamente tutti i requisiti di sicurezza passati come input. Infine si evidenziano anche le difficoltà che sono emerse lavorando al framework, e verranno proposte alcune soluzioni per poter evitare simili problematiche in futuro.
- Il Capitolo [7] descrive lo sviluppo della seconda Demo. In un primo momento viene mostrata la topologia di rete scelta da virtualizzare, con la finalità di indicare le nuove funzionalità di verefoo sviluppate al completamento del secondo obiettivo della tesi. Successivamente vengono descritti tutti i passi svolti per implementare la demo, con un commento per il codice che è stato utilizzato. Infine si evidenziano anche i limiti della demo prodotta con alcuni futuri aggiornamenti possibili.
- Il Capitolo [8] elenca i lavori futuri da svolgere all’interno del framework, la necessità di poter implementare soluzioni alternative a quella proposta in questo documento, e i limiti che devono essere superati affinché il framework possa essere utile in un ambiente reale e non solo di testing virtualizzato. Infine vengono descritte le conclusioni del lavoro, con un riassunto generale di tutto ciò che è stato prodotto.

## Chapter 2

# Network Security Automation e Verefoo

Nel mondo odierno le reti internet hanno rivestito un'importanza sempre maggiore, evolvendosi da piccole e semplici scenari per reti private domestiche a grandi e complicate topologie per le aziende e la comunicazione in tutto il mondo. Trattandosi di un mondo sempre in evoluzione, anche la configurazione e l'installazione di queste reti è diventata sempre più complessa, tanto da far notare sempre di più l'errore umano nelle impostazioni delle reti. Per queste situazioni nasce l'idea di *Network Security Automation*, che pone come obiettivo principale quello di riuscire a rendere la sicurezza delle reti il più possibile autonoma, riducendo la possibilità di errore umano e delegando all'automatizzazione tutte le criticità della configurazione delle varie funzioni di rete.

In questo capitolo si introduce la definizione di ***Security Function Chain (SFC)*** specificando la loro capacità nel migliorare la sicurezza delle reti, successivamente verrà introdotto il framework Verefoo che utilizza le SFC per poter produrre delle topologie di rete robuste e sicure e automatizzare il processo di creazione e configurazione delle reti.

L'ultima sezione del capitolo infine spazia sugli input che il framework accetta, le *Network Security Functions (NSFs)*, cioè tutte le funzioni che la rete deve rispettare come ad esempio filtrare dei pacchetti o criptare del traffico dati.

## 2.1 Service Function Chain

All'interno delle reti è possibile far passare il traffico in maniera *End-to-End*, *Site-to-Site* o *End-to-Site*. Durante la comunicazione nelle reti moderne è solito far transitare i pacchetti attraverso nodi che si occupano di funzioni specifiche all'interno della rete (Ad esempio un Packet Filter o un Network Address Translator NAT), che sono necessari per poter far rispettare alla rete determinate caratteristiche l'utente richiede. I nodi che sono adibiti a svolgere le funzioni prendono il nome di *Service Function (SF)* ed il collegamento di più nodi adibiti a SF viene definito *Service Function Chain (SFC)*. Una definizione formale di SF e SFC è stata presentata nel RFC 7665 [2] che definisce le seguenti:

- **Service Function:** Una funzione che è responsabile del trattamento specifico dei pacchetti ricevuti. Una Service Function può agire su varie livelli di uno stack di protocollo (ad esempio, al livello di rete o ad altri livelli OSI). Come componente logica, una SF può essere realizzata come un elemento virtuale o essere incorporata in un elemento di rete fisico. Una o più SF possono essere incorporate nello stesso elemento di rete. Possono esistere più occorrenze della funzione di servizio nello stesso dominio amministrativo.
- **Service Function Chain** Una Service Function Chain definisce un insieme ordinato di funzioni di servizio astratte e vincoli di ordinamento che devono essere applicati a pacchetti e/o frame e/o flussi selezionati come risultato di una classificazione. Un esempio di una Service Function astratta è un "firewall". L'ordine implicito potrebbe non essere una progressione lineare poiché l'architettura consente SFC che si ramificano su più di un percorso e consente anche casi in cui c'è flessibilità nell'ordine in cui le Service Function devono essere applicate.

La possibilità di definire SF separate e di combinarle fra loro nell'ordine che si preferisce garantisce alle reti la possibilità di essere flessibili e scalabili facilmente. Come infatti è descritto dalla definizione di SFC la concatenazione di più SFC permette ramificazioni su più percorsi, garantendo molteplici comunicazioni fra due host con caratteristiche di sicurezza differenti. Per comprendere meglio il concetto di SFC, un esempio fornito è il seguente:



Figure 2.1. Esempio di Service Function Chain

Come si può notare, vi sono diversi elementi all'interno di questo esempio. Per quanto riguarda i vari SF possiamo trovare i seguenti:

- **Firewall:** Si occupa di fare da packet filter fra i due webclient, per filtrare solo i pacchetti che effettivamente sono necessari alla comunicazione
- **Monitor:** Si occupa di monitorare il traffico in transito fra i due nodi, può essere un nodo da interrogare in caso di problematiche all'interno della rete per controllare che il firewall svolga il suo ruolo correttamente.
- **VPN Gateway:** Si occupa di criptare e decriptare il traffico. In questa topologia è fondamentale la loro presenza in quanto vi è un nodo considerato non affidabile, di conseguenza tutte le comunicazioni che passano attraverso quel nodo sono criptografate.

L'unione dei tre SF in questo specifico ordine definisce una Service Function Chain. E' importante notare che se l'ordine fosse stato diverso (ad esempio mettendo prima i 2 VPN Gateway e dopo il firewall) la SFC risultante sarebbe stata diversa da quella di partenza, garantendo una ramificazione.

## 2.2 Verefoo

### 2.2.1 Introduzione

Il potenziamento progressivo delle tecnologie appena descritte ha portato rapidamente allo sviluppo di reti che automatizzavano i lavori di configurazione che solitamente venivano svolti manualmente. Un esempio di queste nuove tecnologie viene svolto da VEREFOO[1](VERified REfinement and Optimized Orchestration), un framework che si pone diversi obiettivi fra i quali la definizione ad alto livello dei requisiti di sicurezza di rete, l'allocazione automatica ed ottimale delle varie Service Function per ottenere la maggior efficienza di rete allocando le minori risorse possibili, e la configurazione automatica delle varie *Network Security Functions*, eliminando l'errore umano che in reti di grandi dimensioni è solito capitare. Come è possibile intuire, la sfida di produrre una rete configurata automaticamente e correttamente è molto difficile da ottenere, perciò per assicurare la correttezza formale dei risultati ottenuti da Verefoo l'intero framework utilizza un metodo formale che si basa sulla risoluzione di un *Maximum Satisfiability Modulo Theories* (MaxSMT) tramite l'engine Z3 di Microsoft.

Questo, essendo basato su 3 pilastri quali *Ottimizzazione, Ottimalità e Correttezza Formale*, Verefoo si pone 2 obiettivi da soddisfare:

1. L'allocazione ottimale dei vari NSF's
2. La configurazione ottimale dei vari NSF's.

### 2.2.2 Descrizione del modello

Il modello di Verefoo, descritto in figura 2.2 richiede in input due elementi fondamentali:

- **Service Graph:** Una topologia logica delle funzioni della rete che insieme formano un collegamento end-to-end. A differenza delle SFC il Service Graph può avere diversi percorsi per collegare due punti nella rete presentando la ramificazione tipica della concatenazione di SFC. Durante la creazione del Service Graph non è necessario specificare nessun requisito di sicurezza, come ad esempio Firewall, Monitors o Filtering Databases.
- **Network Security Requirements:** Sono i requisiti di sicurezza che la rete in output dovrà avere dopo la computazione di Verefoo. Questo elemento è fondamentale per costruire il modello di MaxSMT da risolvere tramite Z3. Allo stato attuale, Verefoo consente di avere 3 requisiti di sicurezza principali:
  1. **Reachability Property:** Indica che un nodo Y di destinazione deve essere raggiungibile da un nodo di partenza X in almeno un percorso della topologia.
  2. **Isolation Property:** Indica che un nodo Y di destinazione **NON** deve essere raggiungibile da un nodo di partenza X in tutti i possibili percorsi all'interno della topologia.

3. **Protection Property:** Indica che la comunicazione tra un nodo di partenza X ed un nodo di destinazione Y deve essere sicura. In questo requisito è anche possibile specificare un nodo definito "*Untrusted Node*" ovvero un nodo che potrebbe essere un possibile punto di debolezza nella rete e che quindi deve poter vedere solo il traffico criptografato.

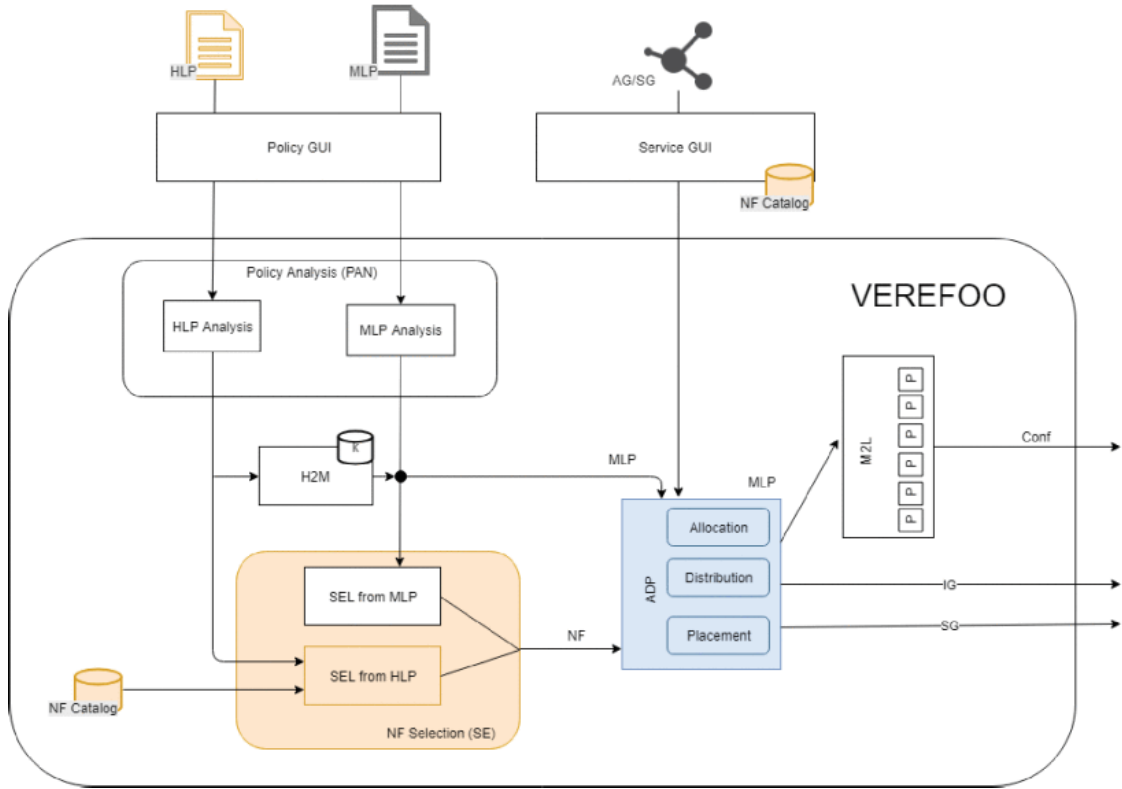


Figure 2.2. Architettura di VEREFOO [1]

Successivamente all'input, il framework esegue una serie di passi che possono essere così riassunti:

1. **Fase di controllo dell'input:** In questa fase Verefoo accetta l'input passato sotto forma di file XML e controlla la coerenza dell'input fornito. Il framework oltre ad accettare l'input composto da Service Graph e NSRs accetta anche la possibilità di fornire un *Allocation Graph* al posto del Service Graph (figura 2.3). La differenza fondamentale fra i due è che nel secondo oltre ai vari nodi della rete descritti nel Service Graph si specificano anche dei nodi aggiuntivi, chiamati *Allocation Places* che rappresentano i punti nella topologia dove è possibile istanziare una funzione di sicurezza di rete.
2. **Fase di analisi del modulo PAN:** qui Verefoo esegue un'analisi delle policy che sono state passate in input. Più specificatamente viene controllato che i vari NSRs siano coerenti fra loro, evidenziando eventuali errori (ad esempio non si può avere una Reachability Property ed una Isolation Property con gli



stessi nodi di partenza e destinazione). Alla fine dell'analisi delle policy viene prodotto il numero minimo di vincoli che devono essere rispettati affinché la topologia soddisfi i NSRs richiesti. In caso di errore un report viene solitamente fornito in output per comprendere il perché un determinato input non è soddisfabile.

3. **Trasformazione a Medium Level Language:** Una volta definiti ad alto livello i vincoli da far rispettare alla topologia, questi vengono tradotti da un linguaggio di alto livello ad uno di medio livello tramite il componente H2M.
4. **Selezione delle Network Function:** Ricevuto l'output dal modulo H2M il modulo Network Functions Selection (SE) si occupa di selezionare da un catalogo le NSF necessarie a rispettare i requisiti di alto e medio livello. Questo catalogo è anche accessibile al designer della rete nella fase di design disponibile nella Service GUI di Verefoo.
5. **Allocazione, Distribuzione e Piazzamento:** Durante questo passo del framework viene eseguito, come suggerito dal titolo, l'allocazione delle varie funzioni di rete calcolate tramite il modulo NF Selection e i vincoli di medio livello tradotti nell'H2M. Questo compito è affidato al modulo ADP che è il cuore di Verefoo, perché decide in quali punti della rete e con quali configurazioni le varie NFs devono essere allocate. L'ADP produce quindi un nuovo Service Graph nel quale sono allocate anche le funzioni di sicurezza di rete, e produce dei file di configurazione per ciascuna funzione allocata nel nuovo Service Graph. Queste configurazioni sono create in un linguaggio di basso livello grazie al modulo M2L presente all'interno dell'ADP.

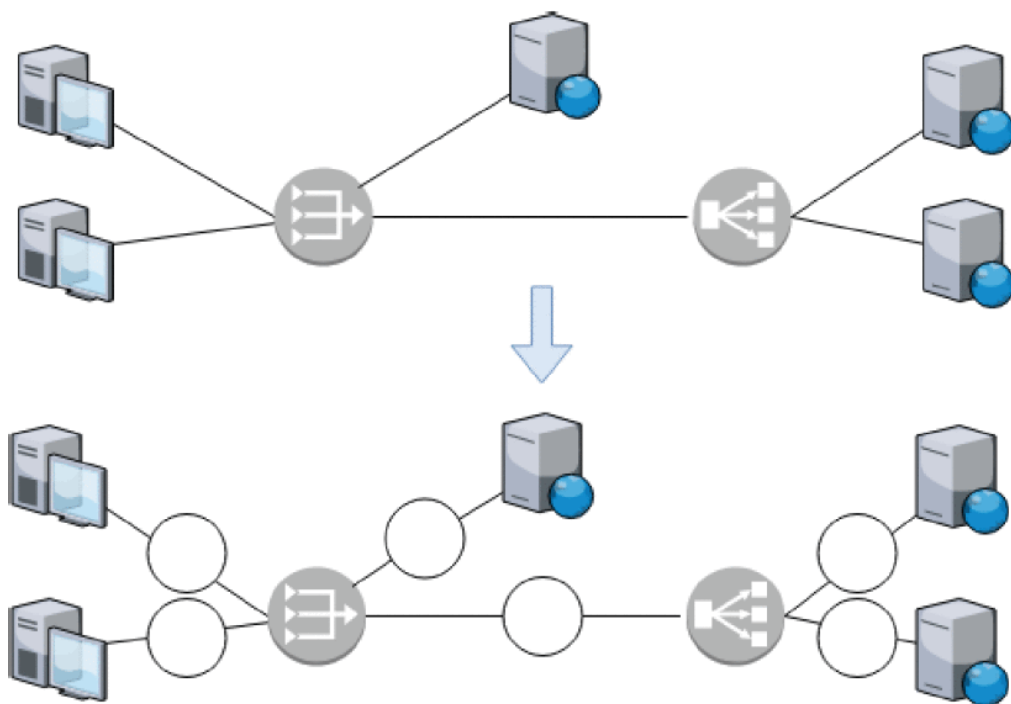


Figure 2.3. Esempi di Service (in alto) e Allocation(in basso) Graphs [1]

## 2.3 Definizione ed esempio di Grafo di servizio e di allocazione

Il primo fondamentale input che deve essere fornito a Verefoo è il grafo di servizio o il grafo di allocazione. Viene definito fondamentale perchè è l'unico modo che ha il framework per comprendere la topologia di rete con la quale dovrà interfacciarsi per soddisfare le richieste dell'utente. Allo stato attuale all'interno di Verefoo è possibile definire come SF le seguenti:

- **Load Balancer:** è uno strumento di controllo di flusso della rete. Il load balancer è infatti in grado di distribuire il carico di lavoro in maniera equa nella rete evitando delle situazioni nelle quali alcuni collegamenti fra i nodi risultano sovraccaricati mentre altri inattivi. È quindi in grado di migliorare l'affidabilità e l'efficienza di un sistema distribuito.
- **Network Address Translator (NAT):** è un servizio che consente la traduzione degli indirizzi IP tra due reti. Il suo obiettivo principale è consentire a dispositivi in una rete privata di condividere un singolo indirizzo IP pubblico per accedere a risorse esterne su Internet.
- **Web Client:** è un'applicazione software o un dispositivo che consente agli utenti di accedere a risorse e servizi su Internet utilizzando il protocollo HTTP (Hypertext Transfer Protocol) o il suo equivalente sicuro HTTPS (Hypertext Transfer Protocol Secure). Questo tipo di client è progettato per interagire con i server web, recuperare informazioni e visualizzare contenuti web.
- **Web Server:** è un software o un'applicazione che fornisce risorse e servizi su Internet, rispondendo alle richieste provenienti dai web client. Il suo ruolo principale è accettare richieste HTTP (Hypertext Transfer Protocol) o HTTPS (Hypertext Transfer Protocol Secure) da parte dei client e inviare loro le risorse richieste.
- **Packet Filter(Firewall):** è un componente di sicurezza della rete progettato per monitorare, analizzare e controllare il traffico di rete in base a regole predefinite. La sua funzione principale è quella di decidere quali pacchetti di dati possono attraversare il firewall e raggiungere la destinazione e quali devono essere bloccati.
- **Web Cache:** è un meccanismo di memorizzazione temporanea che conserva copie di risorse web come pagine HTML, immagini, fogli di stile e altri contenuti multimediali. L'obiettivo principale della web cache è accelerare il caricamento delle pagine web e ridurre il carico sulla rete e sui server web, fornendo ai client risorse già memorizzate localmente anziché scaricarle nuovamente da Internet.

Questi elementi possono essere inseriti nella creazione di una rete da fornire a Verefoo. Il framework è in grado di accettare file XML che descrivono il grafo della topologia di rete definendo i nodi nel seguente modo:

- **Nome:** é l'indirizzo ip che caratterizza il nodo.
- **Funzionalità:** descrive quale delle SF descritte precedentemente il nodo implementa.
- **Nodi adiacenti:** viene fornita una lista di nodi adiacenti al nodo in questione definiti dal loro indirizzo IP.
- **Configurazione:** viene specificata la configurazione, ove necessaria, per poter istanziare la funzionalità definita al campo precedente.

Di seguito viene proposto un esempio della definizione di un Service Graph:

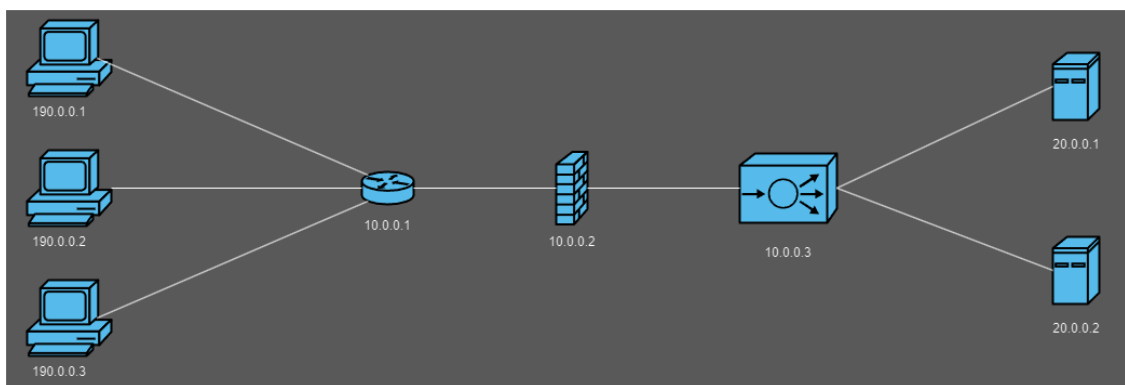


Figure 2.4. Esempio di un Service Graph

In questo scenario, sono presenti tre WebClient con gli indirizzi IP corrispondenti a 190.0.0.1, 190.0.0.2 e 190.0.0.3, tutti appartenenti a una specifica sottorete. Questa sottorete è gestita da un dispositivo NAT con indirizzo 10.0.0.1, il quale instrada il traffico dati al firewall e maschera gli indirizzi IP locali dei client. Successivamente, il firewall entra in gioco e prende la decisione di filtrare soltanto le comunicazioni provenienti dall'IP 190.0.0.1 e dirette verso 20.0.0.1. Il traffico filtrato viene quindi inoltrato al load balancer con indirizzo 10.0.0.3. Quest'ultimo si occupa di distribuire il traffico sulla linea di connessione in modo da garantire una gestione efficiente delle risorse, privilegiando la linea meno congestionata.

Al fine di esempio, la seguente è la definizione in termini di codice XML da fornire a Verefoo per specificare una rete con le condizioni sopra descritte:

Listing 2.1. Definizione nodi del ServiceGraph 2.4

```
<NFV>
<graphs>
  <graph id="0">
    <node functional_type="WEBCLIENT" name="190.0.0.1">
      <neighbour name="10.0.0.1"/>
      <configuration description="Client1" name="conf1">
        <webclient nameWebServer="30.0.5.2"/>
      </configuration>
    </node>
```

```

<node functional_type="WEBCLIENT" name="190.0.0.2">
  <neighbour name="10.0.0.1"/>
  <configuration description="Client2" name="conf1">
    <webclient nameWebServer="30.0.5.2"/>
  </configuration>
</node>
<node functional_type="WEBCLIENT" name="190.0.0.3">
  <neighbour name="10.0.0.1"/>
  <configuration description="Client3" name="conf1">
    <webclient nameWebServer="30.0.5.2"/>
  </configuration>
</node>
<node functional_type="NAT" name="10.0.0.1">
  <neighbour name="190.0.0.1"/>
  <neighbour name="190.0.0.2"/>
  <neighbour name="190.0.0.3"/>
  <neighbour name="10.0.0.2"/>
  <configuration description="NAT" name="conf1">
    <nat>
      <source>190.0.0.1</source>
      <source>190.0.0.2</source>
      <source>190.0.0.3</source>
    </nat>
  </configuration>
</node>
<node functional_type="LOADBALANCER" name="10.0.0.3">
  <neighbour name="10.0.0.2"/>
  <neighbour name="20.0.0.1"/>
  <neighbour name="20.0.0.2"/>
  <configuration description="LB" name="conf1">
    <loadbalancer>
      <pool>20.0.0.1</pool>
      <pool>20.0.0.2</pool>
    </loadbalancer>
  </configuration>
</node>
<node functional_type="WEBSERVER" name="20.0.0.1">
  <neighbour name="10.0.0.3"/>
  <configuration description="Server1" name="conf1">
    <webserver>
      <name>20.0.0.1</name>
    </webserver>
  </configuration>
</node>
<node functional_type="WEBSERVER" name="20.0.0.2">
  <neighbour name="10.0.0.3"/>
  <configuration description="Server2" name="conf1">
    <webserver>
      <name>20.0.0.2</name>
    </webserver>
  </configuration>
</node>
<node functional_type="FIREWALL" name="10.0.0.2">
  <neighbour name="10.0.0.1"/>
  <neighbour name="10.0.0.3"/>
  <configuration description="" name="conf1">
    <firewall defaultAction="DENY">

```

```
        <elements>
          <action>ALLOW</action>
          <source>190.0.0.1</source>
          <destination>20.0.0.1</destination>
          <protocol>ANY</protocol>
          <src_port>22</src_port>
          <dst_port>22</dst_port>
        </elements>
      </firewall>
    </configuration>
  </node>
</graph>
</graphs>
</NFV>
```

In continuità con il grafo precedentemente descritto, possiamo estendere la rappresentazione introducendo un esempio di "Allocation Graph". Come specificato precedentemente, questo grafo prevede l'inclusione di uno o più "Allocation Places", che sono nodi specifici funzionanti come spazi temporanei per il futuro collocamento di una service function (SF). Tale approccio agevola il processo di comunicazione con Verefoo, consentendo una migliore comprensione della topologia di rete e accelerando l'individuazione di soluzioni ottimali. Di seguito è presente un esempio di codice XML per definire un Allocation Place che si interpone fra il client 190.0.0.1 e il NAT definiti nella figura 2.4:

Listing 2.2. Definizione Allocation Place per Allocation Graph

```
<node name="1.0.0.1">
  <neighbour name="190.0.0.1"/>
  <neighbour name="10.0.0.1"/>
</node>
```

## 2.4 Definizione ed esempi delle proprietà di sicurezza

Le proprietà di sicurezza sono il secondo input che può essere fornito a Verefoo per stabilire i vincoli necessari affinché si possa creare una rete sicura. A differenza del primo parametro di input, la definizione della proprietà di sicurezza è responsabilità dell'amministratore della rete, in quanto deve decidere tramite le possibilità offerte da Verefoo e dalla GUI ad esso associata quante e quali proprietà inserire nella propria rete per garantire la sicurezza richiesta.

Per garantire flessibilità al framework è possibile inserire le definizioni sia con un linguaggio a basso livello definendo rispetti IP, porte e protocolli da accettare o rifiutare che con un linguaggio ad alto livello nel quale i vari elementi della rete verranno definiti da dei numeri che poi saranno mappati a degli IP.

I requisiti prodotti all'interno di verefoo saranno formulati con la seguente definizione:

Listing 2.3. Definizione di una proprietà di sicurezza generica

<code>[ruleType, IPSrc, IPDst, portSrc, portDst, transportProto]</code>
---

- **ruleType**: specifica quale proprietà stiamo definendo. Allo stato attuale del framework ci sono 3 possibili opzioni: Reachability Property, Isolation Property e Protection Property.
- **IPSrc**: indica l'indirizzo IP sorgente, cioè il primo nodo della rete dal quale il requisito deve essere applicato.
- **IPDst**: indica l'indirizzo IP destinazione, ovvero l'ultimo nodo della rete dal quale il requisito deve essere applicato.
- **portSrc**: specifica la porta di rete del nodo sorgente che il protocollo di trasporto utilizzerà per inoltrare i pacchetti di rete. In questa opzione è possibile inserire il valore "\*" che rappresenterà il range di tutte le porte possibili.
- **portDst**: specifica la porta di rete del nodo destinazione che il protocollo di trasporto utilizzerà per ricevere i pacchetti di rete. Come per il valore portSrc anche in questo caso è possibile inserire il valore "\*" per rappresentare tutte le possibili porte di destinazione.
- **transportProto**: è il campo che specifica quale protocollo di trasporto verrà utilizzato per soddisfare la regola. Allo stato attuale Verefoo consente di utilizzare 2 possibili protocolli, UDP e TCP.

I campi IPSrc e IPDst non sono limitati a descrivere un singolo host sorgente e un singolo host destinazione per ogni requisito. È infatti possibile utilizzare i due campi per definire anche delle sottoreti. Il framework di Verefoo, infatti, definisce gli IP con la classica notazione decimale:

$$ip = ip1.ip2.ip3.ip4$$

Ogni elemento da ip1 a ip4 deve appartenere al range [0-255] oppure è possibile utilizzare il valore "\*" per definire l'intera sottorete. Grazie a questa implementazione è possibile definire le sottoreti come ad esempio 40.5.0.0\16 utilizzando la notazione 40.5.\*.\* o anche sottoreti più piccole come 20.1.1.0\24 scrivendo 20.1.1.\*.

Di seguito viene fornita una descrizione più dettagliata dei vari requirements specificabili su Verefoo e di come il framework li traduca in requisiti più generali che la topologia di rete deve avere affinché si possa soddisfare la richiesta dell'utente.

### 2.4.1 Reachability Requirements

I requisiti di raggiungibilità sono definiti nel framework di Verefoo per garantire che un determinato Host sorgente che verrà definito Host-S sia in grado di comunicare in maniera diretta e definita con un host destinazione che verrà nominato Host-D. Per garantire ciò è necessario assicurarsi che tutti i packet filter presenti nella connessione fra Host-S ed Host-D non scartino mai i pacchetti di questa comunicazione. Per ottenere questo obiettivo è quindi possibile configurare i packet filter della rete in due modalità: blacklist e whitelist. Nella prima i packet filter faranno transitare tutto il traffico tranne quello specificato nelle regole definite, mentre nella seconda bloccherà tutto il traffico in entrata escluso quello specificato nelle regole che gli sono state imposte.

Per poter dare per certo che il requisito sia applicabile alla topologia Verefoo svolge le seguenti operazioni:

1. Viene svolto un controllo formale sulla definizione della regola, cioè viene controllata la correttezza di tutti i campi inseriti nella proprietà. In questa prima fase Verefoo controlla se gli input inseriti sono tutti presenti e definiti nella maniera corretta (Come ad esempio Stringhe e indirizzi ip come numeri decimali).
2. Viene svolto un controllo logico sulla definizione della regola, cioè viene controllato che l'indirizzo IP sorgente e quello destinazione appartengano effettivamente a degli host definiti nella rete. Inoltre viene controllato anche il protocollo di trasporto, assicurandosi che sia UDP o TCP.
3. Si ispeziona l'Host-S, e ci si assicura che sia possibile inoltrare il traffico destinato all'Host-D in almeno uno dei nodi adiacenti poichè è sufficiente garantire che esista almeno un percorso definito per la comunicazione fra Host-S e Host-D

4. Infine viene attenzionato l'Host-D, sincerandosi che sia possibile ricevere il traffico proveniente dall'Host-S da almeno uno dei nodi adiacenti, perchè garantisce che almeno un percorso fra l'Host-S e l'Host-D sia stato trovato.

Di seguito è possibile trovare un esempio svolto da Verefoo per soddisfare un requisito di raggiungibilità:

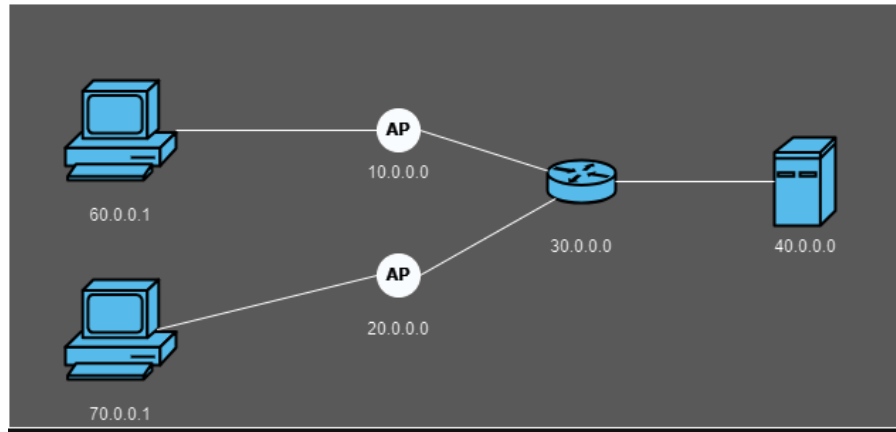


Figure 2.5. Grafo di allocazione per requisito di raggiungibilità

Successivamente è necessario definire la definizione di requisito di raggiungibilità tra l'Host-S 60.0.0.1 e l'Host-D 40.0.0.0. Nel caso di questo esempio proveremo a far comunicare i due Host tramite solo il protocollo TCP:

Listing 2.4. Esempio di requisito di raggiungibilità

```
<PropertyDefinition>
<Property graph="0" name="ReachabilityProperty" src="60.0.0.1"
dst="40.0.0.0" lv4proto="TCP" src_port="*" dst_port="*" />
</PropertyDefinition>
```



Il risultato atteso è il seguente:

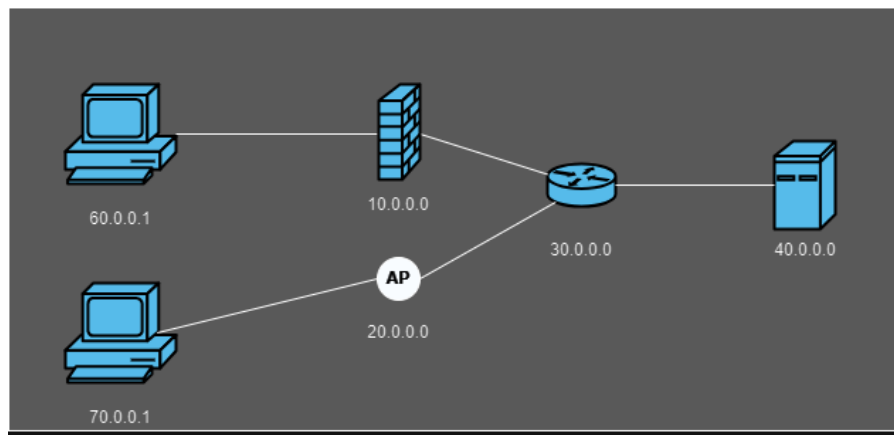


Figure 2.6. Output grafo di esempio per requisito di raggiungibilità

Come si può notare, è stato posto un firewall che funge da packet filter che esclude tutte le comunicazioni uscenti da 60.0.0.1 e dirette a 40.0.0.0 che non utilizzano il protocollo TCP. Il firewall è quindi stato impostato in blacklist mode negando tutto il traffico che non soddisfa la regola definita sopra.

## 2.4.2 Isolation Requirements

I requisiti di isolamento sono definiti nel framework di Verefoo per garantire che un determinato Host sorgente che verrà definito Host-S non sia in grado di comunicare in maniera diretta e definita con un host destinazione che verrà nominato Host-D. Al fine di poter garantire ciò è necessario che tutti i packet filter presenti nel collegamento tra Host-S e Host-D scartino a priori ogni pacchetto. In maniera equivalente ai requisiti di raggiungibilità è possibile implementare nelle reti questo requisito tramite l'utilizzo di packet filter. Analogamente, sarà possibile impostare i packet filter in blacklist, cioè bloccando solo le comunicazioni che vengono specificate dalle regole di filtraggio, oppure in whitelist, permettendo il transito solo dei pacchetti che fanno match con le regole di filtraggio.

Affinchè ciò sia implementato correttamente, Verefoo analizza ed opera nel seguente modo ogni requisito di isolamento ricevuto:

1. Come per i requisiti di raggiungibilità, viene svolto un controllo formale sulla definizione della regola, cioè viene controllata la correttezza di tutti i campi inseriti nella proprietà. In questa prima fase Verefoo controlla se gli input inseriti sono tutti presenti e definiti nella maniera corretta (Come ad esempio Stringhe e indirizzi ip come numeri decimali).
2. Viene svolto un controllo logico sulla definizione della regola, cioè viene controllato che l'indirizzo IP sorgente e quello destinazione appartengano effettivamente a degli host definiti nella rete. Inoltre viene controllato anche il protocollo di trasporto, assicurandosi che sia UDP o TCP.

3. Si ispeziona l'Host-S, e ci si assicura che sia possibile inoltrare il traffico destinato all'Host-D in almeno uno dei nodi adiacenti poichè è sufficiente garantire che esista almeno un percorso definito per la comunicazione fra Host-S e Host-D
4. Infine viene attenzionato l'Host-D, sincerandosi che non sia possibile ricevere il traffico proveniente dall'Host-S da nessuno dei nodi adiacenti, così da garantire che non esiste nessun percorso in grado di far comunicare l'Host-S e l'Host-D.

Al fine di poter portare all'attenzione un esempio di questo requisito di sicurezza, si utilizzerà la stessa topologia di rete consultabile nell'immagine 2.5. A differenza del requisito di raggiungibilità, in questo caso la definizione sarà la seguente:

Listing 2.5. Esempio di requisito di isolamento

```
<PropertyDefinition>
<Property graph="0" name="IsolationProperty" src="70.0.0.1"
dst="40.0.0.0" lv4proto="UDP" /> </PropertyDefinition>
```

In questo caso viene espresso che tutto il traffico proveniente dall'Host-S 70.0.0.1 e diretto all'Host-D 40.0.0.0 che transita tramite il protocollo di livello 4 UDP, deve essere bloccato e quindi non arrivare a destinazione. La computazione svolta da Verefoo porta alla seguente:

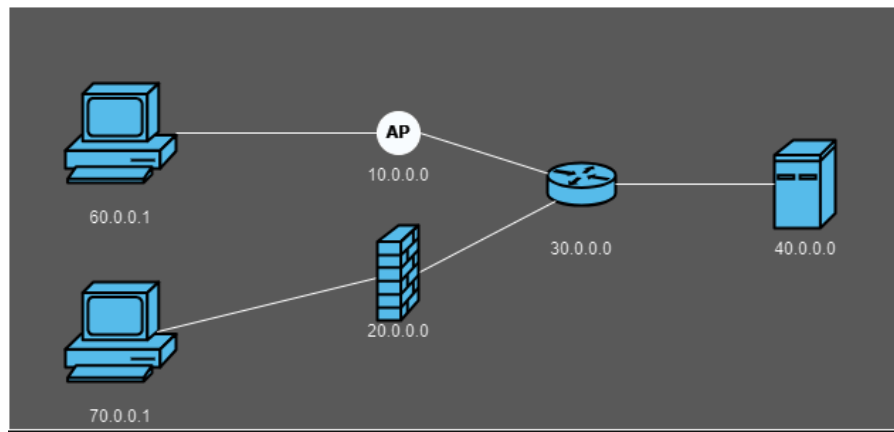


Figure 2.7. Output grafo di esempio per requisito di isolamento

A differenza della 2.6, il packet filter allocato da Verefoo è stato impostato in blacklist mode. Facendo ciò le comunicazioni fra i due host 70.0.0.1 e 60.0.0.1 sono ancora garantite e l'unica regola che blocca il traffico dati è specifica per le comunicazioni in UDP da 70.0.0.1 a 40.0.0.0.

### 2.4.3 Protection Requirements

I requisiti di protezione sono definiti nel framework di Verefoo per garantire che un determinato Host sorgente che verrà definito Host-S possa comunicare con un host destinazione definito Host-D in maniera sicura garantendo le proprietà di integrità, confidenzialità e segretezza. Per assicurare tali prestazioni Verefoo istanzia dei tunnel VPN che vengono allocati nella rete attraverso dei VPN Gateway in grado di prendere il traffico in ingresso del tunnel, criptarlo, farlo transitare all'interno del tunnel e decriptarlo quando il messaggio è arrivato a destinazione. Per fare ciò su Verefoo è presente un translator in grado di creare delle configurazioni StrongSwan [3], un software open-source che permette di istanziare tunnel VPN utilizzando IPsec [4].

A differenza dei requisiti di raggiungibilità e di isolamento, la definizione della regola varia leggermente aggiungendo qualche campo:

Listing 2.6. Definizione di una proprietà di protezione

```
[ruleType, IPSrc, IPDst, portSrc, portDst,
  secTechnology, authAlg, encAlg, untrustedNodes,
  inspectorNodes, untrustedLinks ]
```

Come si può notare paragonando questa nuova definizione di regola alla 2.3 i campi `ruleType`, `IPSrc`, `IPDst`, `portSrc` e `portDst` sono gli stessi. Ad essi si aggiungono:

- **secTechnology**: specifica quale tecnologia VPN stiamo definendo.
- **authAlg**: definisce l'algoritmo di autenticazione da utilizzare all'interno del tunnel VPN.
- **encAlg**: definisce l'algoritmo da utilizzare per eseguire la cifratura dei pacchetti in transito nel tunnel.
- **untrustedNodes**: definisce il set di nodi non sicuri della rete. Questa definizione è fondamentale perchè Verefoo deve conoscere l'insieme dei nodi nel quale è obbligatorio imporre le regole di sicurezza definite dall'utente. Solitamente il set di nodi non sicuri è definito dal percorso di collegamento fra l'Host-S e l'host-D, tuttavia è possibile anche aggiungere altri nodi non appartenenti al percorso se lo si ritiene necessario.
- **inspectorNodes**: definisce il set di nodi nei quali il traffico deve transitare senza alcuna protezione. Questo set di nodi ha il compito di analizzare il traffico per controllarne la correttezza e la sicurezza, e non potrebbe operare se il traffico fosse cifrato. Verefoo ha quindi l'obbligo di trovare una soluzione ottima che permetta ai nodi d'ispezione di analizzare il traffico.
- **untrustedLinks**: è il set di collegamenti nei quali l'applicazione dei requisiti di sicurezza è obbligatoria. Similmente al set dei nodi non sicuri, questo

parametro è una vera e propria estensione, in quanto tiene in considerazione tutti i possibili path che si potrebbero delineare dall'Host-S all'Host-D come link non sicuri. In aggiunta, come accade per gli `untrustedNodes`, è possibile definire dei link aggiuntivi se lo si ritiene necessario.

Al fine di implementare correttamente ogni requisito descritto dalla regola, Verefoo analizza ed opera nel seguente modo ogni requisito di protezione ricevuto:

1. Similmente ai requisiti precedentemente spiegati, viene svolto un controllo formale sulla definizione della regola, cioè viene controllata la correttezza di tutti i campi inseriti nella proprietà. In questa prima fase Verefoo controlla se gli input inseriti sono corretti. A differenza dei precedenti requisiti non tutti i parametri nella definizione della regola sono obbligatori, è infatti possibile non definire alcun `untrustedNode`, `inspectorNode` o `untrustedLink`, utilizzando i requisiti di default che il framework calcola.
2. Viene svolto un controllo logico sulla definizione della regola, cioè viene controllato che l'indirizzo IP sorgente e quello destinazione appartengano effettivamente a degli host definiti nella rete.
3. Si pone attenzione a tutti i possibili flussi di traffico del requisito specificato. Per qualsiasi flusso, ogni qual volta viene incontrato un nodo considerato non sicuro si calcola il numero di nodi precedenti e ci si assicura che i nodi che definiscono e implementano un qualsiasi tipo di protezione siano sempre in numero maggiore di quelli che invece la rimuovono. Questa condizione è fondamentale per garantire che il traffico che passa attraverso i nodi non sicuri sia sempre cifrato e non ispezionabile dal nodo.
4. Sempre considerando tutti i possibili flussi di traffico, ogni qual volta viene incontrato un nodo considerato d'ispezione si calcola il numero di nodi precedenti e ci si assicura che i nodi che definiscono e implementano una qualsiasi protezione siano in numero uguale di quelli che invece la rimuovono. Questa condizione è fondamentale per garantire che il traffico che passa attraverso i nodi d'ispezione sia sempre decifrato ed in chiaro, così da essere analizzato.
5. Infine, come per i due precedenti casi, considerando tutti i flussi di traffico, se si ha un collegamento non sicuro si calcola il numero di nodi precedenti e ci si assicura che i nodi che definiscono e implementano una qualsiasi protezione siano in numero maggiore di quelli che invece la rimuovono. Sincerandosi di ciò, è possibile garantire che per ogni collegamento non sicuro nel quale il traffico transita, i pacchetti saranno sempre cifrati e non ispezionabili in alcun punto del collegamento.

Per poter porre all'attenzione un esempio semplice di requisito di protezione è necessario considerare una topologia differente da quella analizzata nei due precedenti esempi, che è la seguente:

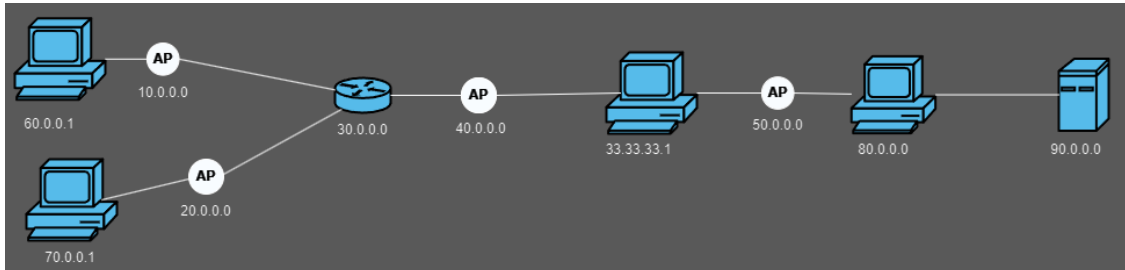


Figure 2.8. Grafo di allocazione d'esempio per requisiti di protezione

In questo esempio, si vuole creare un canale sicuro per far comunicare l'host-S 60.0.0.1 con l'host-D 90.0.0.0. A scopo illustrativo poniamo l'host 33.33.33.1 come nodo non sicuro (untrustedNode) e l'host 80.0.0.0 come nodo d'ispezione per controllare il traffico in ingresso verso il nodo 90.0.0.0 come se fosse un IDS (inspectorNodes). Per attuare ciò la definizione da passare in input a Verefoo è la seguente:

Listing 2.7. Esempio di requisito di protezione

```
<PropertyDefinition>
  <Property graph="0" name="ProtectionProperty" src="60.0.0.1"
    dst="90.0.0.0" src_port="80" dst_port="80">
    <protectionInfo encryptionAlgorithm="AES_128_CBC"
      authenticationAlgorithm="SHA2_256">
      <untrustedNode node="33.33.33.1"/>
      <inspectorNode node="80.0.0.0"/>
      <securityTechnology>TLS</securityTechnology>
      <securityTechnology>IPSEC</securityTechnology>
    </protectionInfo>
  </Property>
</PropertyDefinition>
```

Una soluzione possibile scelta da Verefoo è la seguente:

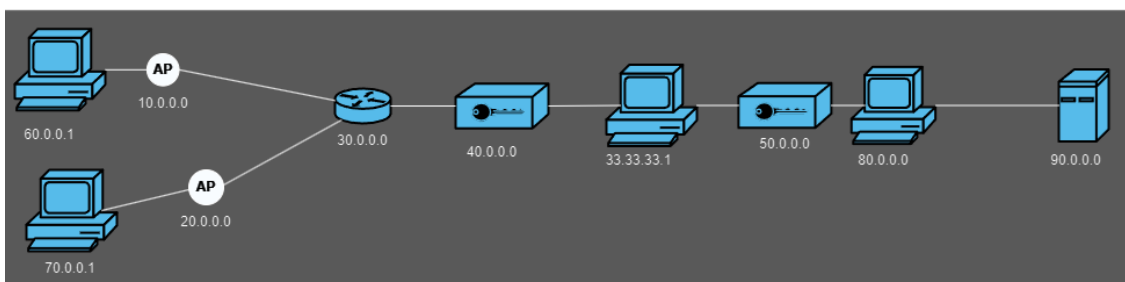


Figure 2.9. Output grafo di esempio per requisito di protezione

Come si può notare 2 VPN Gateway sono state allocati al fine di garantire i requisiti specificati: il primo al nodo 40.0.0.0 è stato configurato in "*ACCESS MODE*" permettendo a tutti i pacchetti provenienti da 60.0.0.1 e diretti a 90.0.0.0 di entrare nel tunnel VPN, il secondo invece è stato configurato in *EXIT MODE* così da poter rimuovere la protezione per tutti i pacchetti inserita precedentemente. È inoltre fondamentale sottolineare come i requisiti che abbiamo richiesto sono stati tutti rispettati. È infatti possibile notare come il nodo considerato non sicuro (33.33.33.1) si trova all'interno del tunnel e ogni pacchetto che riceve sarà stato precedentemente protetto dal nodo 40.0.0.0. Similmente, il nodo 80.0.0.0 che dovrebbe agire da IDS per la topologia di esempio può svolgere la sua funzione in quanto tutti i pacchetti in transito vengono precedentemente decifrati dal nodo 50.0.0.0. Infine non avendo specificato alcun collegamento non sicuro verefoo ha garantito soltanto che il collegamento di default sia sicuro.

# Chapter 3

## Docker

Negli ultimi anni numerose aziende che forniscono servizi agli utenti si sono trovate in serie difficoltà a causa dell' aumento sempre costante del numero dei propri utenti e delle richieste di questi ultimi. Tali aziende si sono trovate quindi nella posizione di dover incrementare proporzionalmente le loro risorse disponibili, sia hardware che software, e di dover assumere sempre più frequentemente del personale altamente specializzato per gestirle. Questa situazione di crisi ha portato ad un cambio di prospettiva in ambito aziendale, orientando l'attenzione verso un sistema di virtualizzazione basato sull'utilizzo dei container invece che con le classiche macchine virtuali. Così facendo si è trovata una valida soluzione a questo problema, in quanto grazie alle loro caratteristiche i container consentono di garantire la qualità del servizio offerto dalle aziende, ottimizzando l'utilizzo delle risorse hardware già in uso, senza la necessità di effettuare ulteriori investimenti significativi in hardware o personale aggiuntivo.

L'obiettivo di questo capitolo è di fornire una descrizione approfondita di una delle tecnologie più diffuse ed utilizzate in questo ambito, Docker[5]. Inoltre viene anche descritto l'utilizzo di uno strumento, docker-compose, molto utile per creare e gestire applicazioni multi-container. Nella parte finale del capitolo è infine possibile trovare una descrizione e degli esempi su come eseguire il networking all'interno dei container.

### 3.1 Differenze fra Container e Macchine Virtuali

Prima di scendere nello specifico descrivendo il funzionamento di docker all'interno di un sistema operativo è necessario definire nel dettaglio cosa sia una macchina virtuale e cosa un container e perchè è più efficiente la seconda soluzione rispetto alla prima.

Per quanto riguarda le macchine virtuali una definizione ci viene fornita dal sito ufficiale di VMware [6]:

”Una Macchina Virtuale (VM) è una risorsa di elaborazione che utilizza software al posto di un computer fisico per eseguire programmi e distribuire applicazioni. Una o più macchine virtuali (guest) vengono eseguite su una macchina fisica (host). Ogni macchina virtuale esegue il proprio sistema operativo e funziona in modo separato dalle altre VM, anche quando sono tutte in esecuzione sulla stessa macchina

host. Ciò significa che, ad esempio, una macchina virtuale con sistema operativo MacOS può essere eseguita su un PC fisico.”

Sotto la prospettiva dei container, è possibile ottenere una definizione precisa consultando la documentazione ufficiale di Docker[7].

”Un container è un’unità standard di software che raggruppa il codice e tutte le sue dipendenze in modo che l’applicazione possa essere eseguita rapidamente e in modo affidabile da un ambiente di calcolo all’altro.”

Seppur le due definizioni possano sembrare molto simili, la struttura software che sta dietro ad entrambe le tecniche di virtualizzazione è diversa, come viene mostrato nella seguente figura:

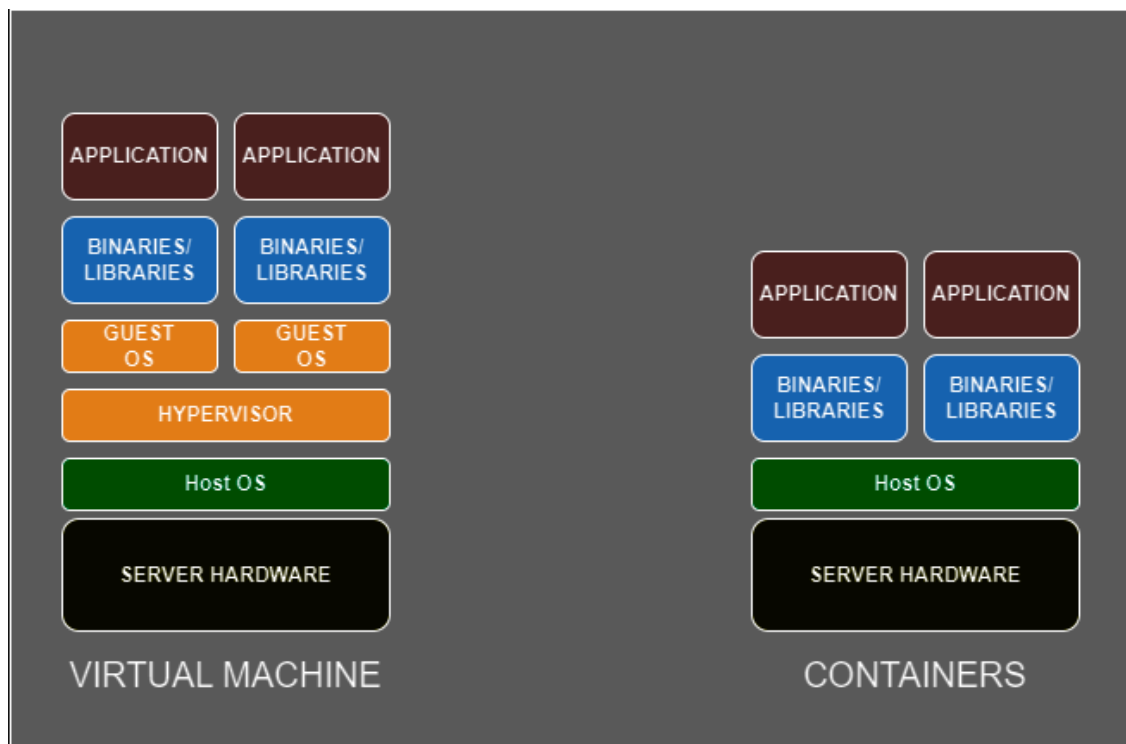


Figure 3.1. Architettura Virtual Machine e Containers

Le macchine virtuali hanno infatti una struttura più complessa rispetto ai container. È infatti presente un hypervisor[8], che è un software che permette di creare e gestire le macchine virtuali in maniera veloce, efficiente, flessibile e portabile. Sopra all’hypervisor ogni macchina virtuale ha il suo proprio sistema operativo, costringendo l’host OS ad allocare numerose risorse per istanziare anche solo 1 macchina virtuale. Si può notare come questa soluzione sia difficilmente scalabile perchè ciò porta il server hardware sul quale poggia tutto il sistema ad essere ulteriormente stressato con l’aggiunta di ogni macchina virtuale. Viceversa, i container richiedono solo le risorse minime necessaria a fare funzionare l’applicazione di turno, installando solo i file binari e le librerie necessarie.



Più in generale le caratteristiche vantaggiose dei container rispetto alle macchine virtuali sono riassunte dalle seguenti parole chiave:

- **Modularità:** avere la possibilità di creare un container per ogni possibile task permette di suddividere i container in vari moduli, ognuno che svolge una sepcifica funzione del progetto di riferimento. Operando in tale direzione, è possibile sviluppare progetti con un approccio Bottom-Up , portando ad un ambiente di testing e validation più veloce ed immediato sui singoli moduli.
- **Isolamento:** ogni container che esegue una immagine viene visto come un ambiente isolato, indipendente dagli altri container in esecuzione. Questo approccio semplifica notevolmente l'individuazione di possibili bug ed errori nel progetto.
- **Peso in Memoria:** come analizzato nel lavoro di Martin Lindström[9], diferentemente dalle macchine virtuali, i container sono delle virtualizzazioni molto più leggere e che richiedono meno risorse alla macchina ospitante. Questo è derivato dal fatto che i container contengono solo lo stretto necessario all'applicazione per funzionare correttamente, mentre le VM hanno bisogno anche di istanziare un proprio sistema operativo, che richiede una discreta quantità di spazio.
- **Scalabilità:** Avendo un peso molto ridotto rispetto alle macchine virtuali, la necessità di aumentare le performance e le dimensioni di un progetto trova nei container un ottimo fattore di scalabilità. È infatti possibile scalare i sistemi sia verticalmente, perchè all'aumentare delle risorse del sistema operativo ospitante corrisponde un aumento della velocità di reazione dei container che orizzontalmente, perchè aggiungere una feature corrisponde nell'aggiungere un container al sistema già funzionante.
- **Condivisone Risorse:** Attraverso i file di configurazione dei container è possibile condividere file con il container, che nell'ambiente isolato verranno considerate come risorse dedicate, anche se in realtà sono condivise. Ciò estende questa funzionalità se si mettono in comune le stesse risorse per più container. In questo caso sulle risorse verrà messo un lock che bloccherà le risorse fino a che uno dei container non abbia finito di utilizzarle, rilasciandole.
- **Fast Boot:**Non dovendo dipendere da nessun sistema operativo, i container possono avviarsi ed essere operativi molto più velocemente rispetto alle macchine virtuali.
- **Operazioni su disco:** Avendo un collegamento diretto con il sistema operativo le operazioni su disco (scrittura, lettura e cancellazione) sono più veloci, portando ad un aumento delle performance per processi parallelizzabili.

## 3.2 Docker e la sua architettura

Tra le piattaforme software disponibili per istanziare e gestire containers per applicazioni Docker riveste il ruolo di software leader nel settore. Nato nel 2013 e progettato da Solomon Hykes nell'azienda dotCloud, Docker è un progetto open source. La differenza fondamentale dagli altri software è che basa la maggior parte delle operazioni eseguibili su un demone chiamato appunto Docker, che svolge le operazioni di istanziazione dei container e la loro gestione. Al fine di garantire ciò, il demone richiede come file di input delle *"Immagini"*. Come è descritto nella documentazione di Docker[7]: "Un'immagine di container Docker è un pacchetto leggero, autonomo ed eseguibile di software che include tutto il necessario per eseguire un'applicazione: codice, runtime, strumenti di sistema, librerie di sistema e impostazioni."

L'architettura di Docker sfrutta il meccanismo client-server, della quale viene mostrata una rappresentazione:

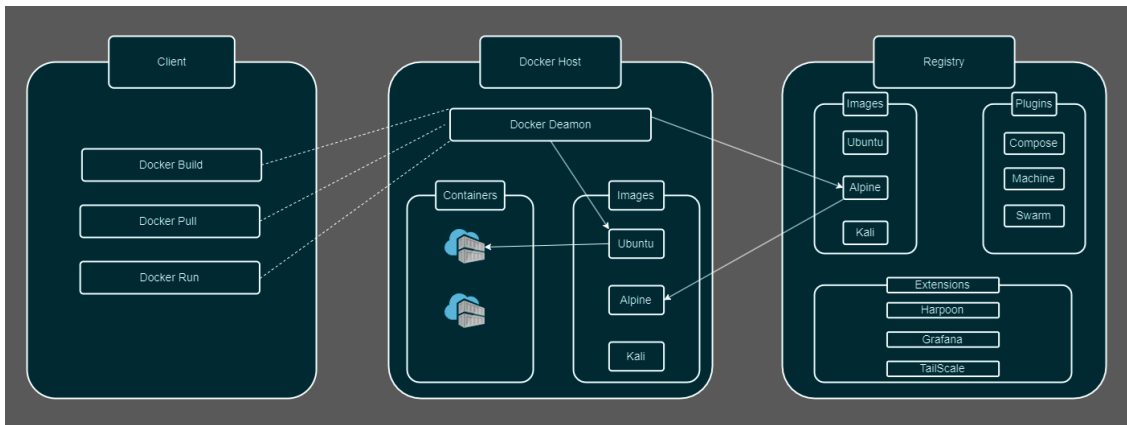


Figure 3.2. Architettura Docker

Di seguito una spiegazione di tutti gli elementi che intervengono nella creazione di un container:

- **Docker Client:** Rappresenta la macchina fisica nel quale è installato Docker. Può comunicare con il controller principale (Docker Host) tramite delle Rest API. Le chiamate che il client può effettuare sono le seguenti:
  - *Docker Pull:* viene utilizzato per scaricare un'immagine da un Registro. Il demone controllerà se questa immagine è presente nel registro locale indicato, altrimenti andrà a cercare online la versione più recente dal registro predefinito di Docker Hub.
  - *Docker Build:* questo comando permette la creazione di un'immagine dato un file di configurazione definito dall'utente (deve avere il nome di Dockerfile) e una cartella di riferimento.

– *Docker Run*: questa chiamata fa creare al demone un container con l'immagine specificata da linea di comando ed avvia il container.

- **Docker Deamon**: è il cuore dell'architettura di Docker. Il ruolo del docker deamon (anche chiamato dockerd) è di ascoltare le richieste tramite call API del client e gestire il registro Docker dove sono contenute le immagini, i plugin e le estensioni. Inoltre può comunicare con altri demoni per gestire i servizi Docker.
- **Docker Registry**: è una zona di memoria che memorizza le immagini Docker. In questa zona sono anche presenti eventuali plugin installati su Docker e le estensioni sviluppate per sfruttare i servizi di Docker. Talvolta potrebbe capitare che le immagini ricercate nel registro non sono presenti, ed in questo caso viene effettuato un collegamento diretto con un registro pubblico online definito Docker Hub per poter usufruire di alcune immagini già pronte.

## 3.3 Il tool Compose

### 3.3.1 Introduzione

Come si è potuto notare nella sezione precedente, Docker garantisce un grado di flessibilità molto elevato che consente di creare sia container che svolgono ruoli molto semplici che container più articolati, i quali richiedono anche l'installazione di diverse librerie tramite Dockerfile. Tuttavia capita molto spesso che isolare un container da tutti gli altri sia una limitazione in quanto per svolgere determinati task due o più macchine virtuali devono potere comunicare efficacemente, inoltre la gestione di reti complesse tramite singoli Dockerfile e linea di comando può risultare tediosa e molto scomoda da utilizzare. Basti pensare che nel caso di un container non funzionante bisognerebbe modificare non solo il Dockerfile del singolo elemento, ma anche rieseguire tutte le chiamate di sistema per fare rebuild dell'ambiente virtuale.

Per venire incontro a questi problemi molto comuni nello sviluppo di applicazioni aziendali, Docker propone delle soluzioni innovative e che cercano non solo di proporre una soluzione ai problemi sopracitati, ma anche di introdurre dei meccanismi di semplificazione per la gestione di reti complesse. Più precisamente le caratteristiche di docker compose possono essere riassunte, come specificato nella documentazione[\[10\]](#), dalle seguenti:

1. **Ambienti isolati multipli**: Il plugin fornisce la possibilità di definire il nome del progetto per isolare i vari ambienti di sviluppo. Inoltre, fornisce la possibilità di richiamare il nome di un progetto per istanziare molteplici copie dello stesso ambiente, per evitare che le build interferiscano fra loro o per evitare che diversi progetti che hanno gli stessi nomi per i servizi definiti vadano in contrasto.
2. **Conservare i volumi di dati**: compose memorizza e ricorda i container esistenti precedentemente. In questo modo è possibile, ogni volta che si avvia

l'ambiente virtuale, trovare i container utilizzati nelle precedenti iterazioni e copiare i dati dal vecchio container a quello nuovo. È quindi garantito che si evitino perdite di dati importanti fra le varie iterazioni.

3. **Reboot Efficiente:** questa proprietà consente di poter fare un reboot completo dell'ambiente virtuale evitando di istanziare nuovamente i container che non sono stati modificati. Compose è infatti in grado di riconoscere i cambiamenti effettuati in ogni container e nella fase di reboot del sistema verranno distrutti e ricreati solo gli elementi modificati, evitando di sovraccaricare la macchina con del calcolo computazionale inutile.
4. **Variabili e Flessibilità:** all'interno dei file di configurazione dell'ambiente virtuale è possibile definire variabili locali. Grazie ad esse è possibile creare delle configurazioni custom a seconda dell'utente o dell'ambiente fisico del sistema.

Per poter definire un ambiente virtuale tramite un unico file di configurazione, docker-compose accetta come input un file YAML che definisce tutti gli elementi presenti da virtualizzare.

### 3.3.2 YAML

YAML[11], acronimo di "Yet Another Markup Language", è un formato di serializzazione dei dati universalmente utilizzato grazie alla sua semplicità, facilità di scrittura e di lettura e comprensione. Queste caratteristiche sono ottenibili grazie a diversi elementi che questo formato ha derivato da linguaggi come Pearl, C o Python, fornendo sia allo sviluppatore che al lettore dei file una sintassi semplice e di immediata comprensione, come l'utilizzo dei caratteri di indentazione per definire i blocchi (allo stesso modo di Python) o la definizione dei dati con mappe chiave valore come i file JSON.

Grazie a queste sue caratteristiche YAML è uno dei formati più utilizzati per la scrittura di file di configurazione, per scambiare dati fra programmi che utilizzano diversi linguaggi di programmazione o rappresentare dati molto complessi in modo chiaro e leggibile da un utente. Per fornire maggiore contesto, si propone un esempio di definizione di un ipotetico file YAML:

Listing 3.1. Esempio definizione di un file YAML

```
topology_name: "Example"
vertices:
  node1:
    name: "Alfa"
    ip: "120.10.0.1"

  node2:
    name: "Beta"
    ip: "120.10.0.2"
  node3:
    name: "Gamma"
    ip: "120.10.0.3"
```

```

edges:
  edge1:
    name:"Alfa-Beta"
    ipstart: "120.10.0.1"
    ipend: "120.10.0.2"
  edge2:
    name:"Alfa-Gamma"
    ipstart: "120.10.0.1"
    ipend: "120.10.0.3"
  edge3:
    name:"Gamma-Beta"
    ipstart: "120.10.0.3"
    ipend: "120.10.0.2"

```

Come si può notare, le somiglianze con un file JSON sono evidenti. In questo esempio vi è la definizione di un grafo, infatti è presente la definizione degli archi e dei vertici. Il gruppo dei vertici è formato a sua volta da 3 sottogruppi, ciascuno rappresentante un nodo che è definito all'interno della topologia con nome ed ip. Analogamente, anche il gruppo degli archi contiene 3 sottoinsiemi, che definiscono i vari archi della topologia di rete dandogli un nome e definendo l'ip di partenza e di destinazione di ogni arco.

### 3.3.3 Services e Networking

All'interno del plugin docker-compose è quindi possibile utilizzare un unico file YAML che definisce e stabilisce le relazioni fra tutti i container. Più precisamente all'interno di un file è possibile definire, ad alto livello, i seguenti elementi:

- **Services:** Come è possibile ricavare dalla documentazione<sup>[12]</sup> "Un servizio è una definizione astratta di una risorsa di elaborazione all'interno di un'applicazione che può essere scalata o sostituita indipendentemente da altri componenti. I servizi sono supportati da un insieme di contenitori, gestiti dalla piattaforma in base ai requisiti di replicazione e ai vincoli di posizionamento. Poiché i servizi sono supportati da contenitori, sono definiti da un'immagine Docker e un insieme di argomenti di runtime." Tramite la definizione di questi servizi è quindi possibile definire quali container il nostro sistema monterà e utilizzerà. Al fine di questo lavoro di tesi ogni container rappresenta un nodo all'interno della topologia di rete che verrà presa in esempio. La funzionalità del singolo nodo di rete verrà poi specificata all'interno dello stesso servizio. È infatti possibile definire, all'interno di ogni servizio, numerosi elementi che permettono di personalizzare il singolo servizio a preferenza dell'utente, i più importanti sono:

- *Volumes:* rappresentano memorie persistenti istanziate al momento di avvio del container dall'host fisico. Questo permette quindi di poter istanziare dei container che contengono fin dall'inizio dei file di configurazione necessari. A differenza delle macchine virtuali, montare un

volume non rappresenta una porzione di memoria condivisa fra il container e la macchina ospitante, in quanto questo negherebbe il requisito di isolamento del container, quanto piuttosto una modo per gestire i dati che devono essere conservati anche dopo che un container è stato arrestato o eliminato.

- *Configs*: è un attributo possibile in fase di definizione del servizio, che consente di gestire le configurazioni distribuite, ovvero di fornire al servizio i file di configurazione necessari durante la fase di esecuzione del container.
- *Secrets*: questo attributo è molto simile ai configs definiti precedentemente, con la differenza principale di proteggere dati considerati sensibili o importanti, come delle password o delle chiavi private di crittazione. Un servizio può quindi accedere ai file solo se viene definito esplicitamente l'attributo "secret" su quel file.
- **Networks**: Le reti sono il secondo macro elemento definibile all'interno del file per docker-compose. All'interno è possibile definire i metodi di comunicazione fra i vari container, definendo delle reti all'interno delle quali i vari container possono comunicare. È importante sottolineare che non è sufficiente definire all'interno di questo elemento una rete affinché i container si connettano, ma è obbligatorio inserire all'interno di ogni servizio appartenente alla rete l'elemento "network" specificando il nome della rete che viene definita in questa sezione. Attraverso queste reti è possibile anche configurare le interfacce di rete per i bridge, gli indirizzi ip per i client ed i server esplicitamente, favorendo così la comunicazione all'interno dell'ambiente virtuale e testando tramite i comandi da terminale noti come il ping.  
Analogamente ai servizi, anche le reti hanno a disposizione degli attributi che è possibile definire per personalizzare la rete:
  - *driver*: viene utilizzato per specificare il driver di rete da utilizzare. Al fine di questa tesi il driver principale utilizzato è il "bridge" che consente ai container di comunicare tra loro sullo stesso host tramite il bridge Docker predefinito.
  - *ipam*: permette la gestione degli indirizzi ip per la rete in questione. Consente non solo di definire un indirizzo ip, ma anche di definire multiple interfacce di rete e intere sottoreti utilizzando la classica notazione con "\".
  - *internal*: specifica se la rete può essere accessibile solo da altri container all'interno della stessa applicazione, oppure anche da container esterni.
  - *external*: specifica se il network è esterno al file docker-compose. In questo caso, il network deve essere creato al di fuori del file yaml utilizzato all'interno di docker-compose.
  - *name*: permette di definire il nome di una rete, che sarà l'identificativo per tutti i servizi che vorranno accedervi.
  - *attachable*: specifica se i container possono essere connessi a questa rete. Se non è specificato esplicitamente dall'utente il valore di default è "true".

Le caratteristiche del file di configurazione appena definite garantiscono quindi una completa personalizzazione dei container, fornendo anche un ottimo metodo di automatizzazione dei container. Di seguito è presentato un esempio, mantenendo la topologia descritta in [3.1], di alcuni degli attributi che verranno utilizzati all'interno di questo lavoro di tesi per definire gli ambienti virtuali delle topologie di rete che verranno studiate:

Listing 3.2. Esempio file di configurazione docker-compose

```
services:
  host1:
    container_name: host1
    hostname: host1
    image: nginx
    cap_add: NET_ADMIN
    command: sh -c "route del default"
    networks:
      clients:
        ipv4_address: 120.10.0.1

  host2:
    container_name: host2
    hostname: host2
    image: nginx
    cap_add: NET_ADMIN
    command: sh -c "route del default"
    networks:
      clients:
        ipv4_address: 120.10.0.2

  host3:
    container_name: host3
    hostname: host3
    image: nginx
    cap_add: NET_ADMIN
    command: sh -c "route del default"
    networks:
      clients:
        ipv4_address: 120.10.0.3

networks:
  clients:
    name: clients
    driver: bridge
    ipam:
      driver: default
      config:
        subnet: 120.10.0.0/24
        gateway: 120.10.0.200
```

# Chapter 4

## Obiettivi della tesi

Questo capitolo introduce gli obiettivi di questa tesi, descrivendo studi e metodologie utilizzate al fine di raggiungerli.

Nei capitoli precedenti è stato infatti descritto lo stato dell'arte di Docker e Verefoo, che sono i due strumenti principali utilizzati per svolgere questo lavoro di tesi. Il primo è infatti uno strumento fondamentale per poter garantire un ambiente di testing efficiente e isolato, il secondo invece è il framework principale nel quale la quasi totalità del lavoro si è svolta. Comprendere l'utilizzo corretto dei due elementi è quindi di fondamentale importanza al fine di poter capire, continuare e migliorare lo stato attuale di Verefoo. Inoltre, molti dei risultati prodotti precedentemente su Verefoo rispetto a questo lavoro pur essendo corretti non offrivano alcun modo di mostrare in maniera diretta le innovazioni prodotte ai nuovi utenti che si avvicinavano al framework. Parte del lavoro svolto è quindi basato sull'ideare e produrre dei metodi efficaci e semplici per mostrare all'utente le capacità e caratteristiche di Verefoo.

Entrando più nello specifico, gli obiettivi della tesi possono essere definiti dal seguente elenco:

1. Come primo obiettivo ci si è focalizzati su una demo già presente all'interno dell'ecosistema. All'interno di questa, tuttavia, diversi elementi all'interno erano considerabili obsoleti o scorretti, di conseguenza ci si è posti come scopo principale di questa prima parte correggere e perfezionare la demo per mostrare correttamente le potenzialità del framework. Allo stato iniziale, il framework era in grado di accettare solo un determinato requisito di sicurezza di rete ovvero la *Protection Property* cioè la possibilità di far passare il traffico crittografato da un nodo ad un altro della topologia in maniera sicura. Al fine di garantire ciò vengono allocati nella topologia dei VPN Gateway in grado di poter cifrare il traffico in ingresso e decifrare quello in uscita. La topologia proposta utilizzerà uno scenario verosimile a quello che ci si potrebbe aspettare in un'azienda di piccole-medie dimensioni, nella quale al fine di poter garantire la correttezza dei requisiti proposti, verranno istanziati 6 VPN Gateway. I lavori svolti per questo obiettivo sono consultabili nel capitolo 5 di questa tesi.



2. Una volta terminato il restauro della demo sulle VPN, è emersa la necessità di integrare alle funzionalità già presenti la possibilità di configurare anche i packet filter. Come secondo obiettivo ci si è quindi concentrati per trovare una soluzione al fine di poter integrare le varie versioni di Verefoo. Inizialmente il framework era diviso in differenti branch, due dei quali permettevano rispettivamente l'allocazione solamente dei VPN Gateway o dei Firewall configurati come packet filter per garantire la *Isolation Property* e la *Reachability Property*. Il traguardo previsto è quello di creare un ulteriore branch che permettesse la fusione dei due precedentemente descritti. Per ottenere ciò diverse soluzioni sono state esplorate. Inizialmente si è pensato di avere una soluzione mista tramite due versioni del framework attive contemporaneamente che comunicavano fra loro in sequenza, per poi passare a soluzioni che permettevano con un solo file jar di svolgere entrambe le funzioni in una sola esecuzione. Anche in questo caso sono stati analizzate entrambe le possibili soluzioni per implementare questo obiettivo, sia istanziando prima i Firewall che i gateway VPN che il viceversa. La soluzione finale scelta è stata quella di allocare prima i VPN Gateway e successivamente i Firewall, con delle motivazioni a supporto che verranno estese nel capitolo 6.
3. Concluso il lavoro sul framework è risultato essenziale trovare un modo per mostrare i risultati ottenuti. L'ultimo obiettivo del lavoro svolto è stato quindi la progettazione, lo sviluppo e l'implementazione di un'altra demo, diversa dalla precedente che mostrasse le nuove potenzialità del framework. A differenza della prima, che da questo momento verrà definita come Demo-A, la seconda, che chiameremo Demo-B, propone un esempio di topologia di rete molto più complessa e con diverse proprietà di sicurezza aggiuntive. Lo sviluppo di questa ha richiesto, come nella precedente, la realizzazione di un'ambiente virtuale dedicato creato con Docker-Compose nel quale mostrare come le varie proprietà venissero rispettate. Infine, per agevolare i futuri lavori nel framework è stato prodotto in linguaggio Bash un installer per rendere semplice ed immediato l'installazione del framework. Ulteriori approfondimenti sul codice e le scelte effettuate sono descritte nel capitolo 7.

Come ultima appendice al lavoro svolto ai fini di questa tesi, è infine presente una breve conclusione del lavoro che oltre a fare un riassunto generale sugli obiettivi raggiunti definisce i futuri lavori possibili e suggerisce anche alcuni aggiornamenti e perfezionamenti che possono essere svolti nelle demo e nel framework prodotti.

## Chapter 5

# Correzione, sviluppo e ottimizzazione Demo A

All'interno di questo capitolo sarà fornita una descrizione completa dei lavori svolti nel primo progetto di demo che è stato preso in analisi inizialmente, corretto e migliorato successivamente.

Inizialmente viene descritta con una breve introduzione gli obiettivi che ci si aspettava di raggiungere con lo sviluppo della demo, i problemi presenti all'inizio del lavoro svolto e le soluzioni possibili attuabili.

Nella seconda parte viene trattato, entrando più nello specifico, l'implementazioni delle soluzioni proposte e presentato il lavoro finale.

Nell'ultima parte del capitolo viene infine mostrato una prova di correttezza della demo con la verifica dei suoi output e delle sue funzionalità.

### 5.1 Introduzione alla Demo

Come descritto al Capitolo[2] Verefoo è un framework in grado di definire dei requisiti di sicurezza ad alto livello, allocare in maniera ottimale varie Network Security Functions (NSF) all'interno della topologia di rete fornita in input e configurare automaticamente tali funzioni automaticamente. Allo stato attuale il framework è ancora in fase di sviluppo e, nonostante si ponga gli obiettivi appena descritti, non tutte le funzionalità sono attualmente possibili all'interno del framework. Più precisamente, varie versioni del framework sono presenti in stato di sviluppo, e ognuna si occupa di allocare una possibile NSF separatamente alle altre. All'interno di questo capitolo verrà presa in considerazione per il lavoro una di queste possibili versioni, ovvero quella che si occupa della verifica dei Network Security Requirements di protezione, dell'allocazione del numero minimo ottimale di VPN Gateway per rispettare i requisiti in input, e della configurazione di questi ultimi.

L'obiettivo principale di questo lavoro di demo è quindi mostrare all'utente come questa versione sia in grado, data una topologia di rete simile a quelle di una azienda di medie dimensioni, di verificare i requisiti, allocare le VPN e configurarle correttamente.

Un altro elemento emerso durante i lavori sullo sviluppo di questa demo è stata la difficile accessibilità di quest'ultima. Per far funzionare sia il framework che la demo sono infatti necessari numerosi tool da installare all'interno della macchina in alcune versioni specifiche e potrebbe essere non immediato installare correttamente il dispositivo per far funzionare framework e demo. Di conseguenza come obiettivo secondario, ma di uguale importanza è stato prodotto un installer che permette all'utente di ottenere automaticamente tutti i programmi nelle versioni corrette per poter utilizzare il framework a proprio piacimento.

Per portare a termine questi due obiettivi ci si è quindi interfacciati con un progetto di demo che risultava incompleto e mal funzionante. Analizzando più approfonditamente possiamo dividere le modifiche effettuate all'interno di questa demo nei seguenti punti:

1. **Versione Framework:** Il file del framework iniziale era malfunzionante, in quanto ogni qual volta che si provava ad avviarlo il terminale segnalava il file come corrotto ed inutilizzabile.
2. **Chiamate API:** La demo utilizzava delle chiamate API considerabili obsolete, di conseguenza qualsiasi relazione con il framework non produceva alcun risultato.
3. **Installer:** Come accennato precedentemente la mancanza di un installer della Demo rendeva il framework poco accessibile e di difficile installazione manuale.
4. **Certificati VPN:** Alcuni dei certificati di chiavi pubbliche e private erano ormai scaduti, sono quindi stati sostituiti ed i file di configurazione modificati opportunamente.
5. **Docker Compose:** Il file di configurazione dell'ambiente virtuale di Docker Compose conteneva errori e molti dei container non venivano istanziati correttamente.
6. **Forwarding Rules:** Alcune forwarding rules all'interno dell'ambiente virtuale erano scorrette, sono state quindi corrette ed aggiornate per garantire la comunicazione fra tutti i nodi della rete.

## 5.2 Sviluppo Installer

Come menzionato nell'elenco precedente, uno dei punti fondamentali dei lavori effettuati su questo progetto è stata la programmazione e implementazione di un installer all'interno della demo.

Verefoo è un framework che per funzionare necessita le seguenti specifiche:

- **Linux Ubuntu 20.04 Long Term Support(LTS)** come sistema operativo della macchina ospitante il framework.
- **Pv** come programma per monitorare i dati mandati attraverso la pipe.

- **Docker Engine** come tecnologia per istanziare e gestire container virtuali.
- **Docker-Compose v1** come plugin di Docker per poter istanziare e gestire molteplici container utilizzando un unico file di configurazione.
- **Java openjdk-1.8** per poter eseguire il framework correttamente.
- **Curl** come programma per poter effettuare chiamate API con il framework.
- **Z3 4.8.15** come programma per risolvere il problema MaxSMT che Verefoo crea durante la computazione dell'output.

Tuttavia al fine di poter implementare tutti questi elementi tramite terminale linux non è sufficiente installare i vari packages tramite il comando *"apt-get install [packageName]"* ma è anche necessario modificare opportunamente alcune variabili d'ambiente all'interno del sistema operativo. Inoltre molti di questi package devono essere installati con delle versioni specifiche per evitare problemi di compatibilità, di conseguenza per l'utente può risultare ostico riuscire a reperire ed installare tutte le versioni correttamente in quanto molti software non sono aggiornati alle versioni più recenti ma bisogna ricercare la versione specifica sui github dei vari programmi, rendendo quindi il comando apt-get install inutilizzabile.

Di conseguenza è stato prodotto uno script bash che da questo momento in poi verrà denominato installer che si occupa di controllare se i package sono già effettivamente presenti all'interno del sistema operativo. In caso affermativo si controlla se la versione del package è quella corretta per utilizzare Verefoo, altrimenti verrà effettuato un downgrade della versione corrente. In caso negativo il package verrà invece istanziato direttamente.

Di seguito vengono mostrati e commentati alcuni snippet di codice appartenenti all'installer:

Listing 5.1. Installazione packages curl e pv

```
1  #!/bin/bash
2  if which pv &> /dev/null
3  then
4  printf "${GREEN}pv installed... OK${COLOR_RESET}\n"
5  else
6  printf "${BLUE}pv package not installed. I'm going to install
   it\n${COLOR_RESET}\n"
7  sudo apt-get update
8  sudo apt-get --yes install pv
9  printf "${GREEN}pv installed... OK${COLOR_RESET}\n"
10 fi
11 if which curl &> /dev/null
12 then
13 printf "${GREEN}curl installed... OK${COLOR_RESET}\n"
14 else
15 printf "${BLUE}curl package not installed. I'm going to install
   it\n${COLOR_RESET}\n"
16 sudo apt-get update
17 sudo apt-get --yes install curl
```

```

18  printf "${GREEN}curl installed... OK${COLOR_RESET}\n"
19  fi

```

Per quanto riguarda i package `pv` e `curl` l'installazione è piuttosto semplice, infatti si può notare come sia alla riga 2 che alla riga 11 venga controllato che i comandi `pv` e `curl` siano presenti reindirizzando l'output sia in caso di successo che di errore al folder `/dev/null` che è un folder nel quale i dati vengono cancellati automaticamente ma che restituisce il successo o il fallimento dell'operazione. Nel caso l'operazione restituisca successo vuol dire che il package è già presente all'interno del sistema operativo, in caso contrario viene installato eseguendo precedentemente una `apt-get update` per scaricare localmente la versione più aggiornata del package. Per quanto riguarda l'installazione di `java` invece l'installazione è leggermente più complessa:

Listing 5.2. Installazione java openjdk

```

1  VERSION=$(java -version 2>&1 >/dev/null | grep "java
    version\|openjdk version")
2  if [ "" = "$VERSION" ]; then
3  printf "${BLUE}Java-openjdk not installed. I'm going to install
    it\n${COLOR_RESET}\n"
4  sudo apt-get update
5  sudo apt-get --yes install openjdk-8-jdk
6  echo "JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64/jre" | sudo
    tee -a /etc/environment
7  printf "${GREEN}Java installed... OK${COLOR_RESET}\n"
8  else
9  printf "${GREEN}Java openjdk installed... OK${COLOR_RESET}\n"
10 fi

```

In questo caso per controllare se la versione corretta di `java` sia già installata viene invocato il comando `"java -version"` effettuando un redirect dell'output e intercettando delle righe contenenti le stringhe `"java version"` o `"openjdk version"` assegnando questo risultato alla variabile `VERSION`.

Successivamente viene controllato se il risultato dato da questo comando corrisponde alla stringa vuota o se effettivamente è stata trovata una versione di `java`. Nel caso in cui `java` non sia presente all'interno del sistema operativo viene installato tramite il comando `"apt-get install openjdk-8-jdk"` e infine viene definito il path dove il sistema operativo dovrà andare a cercare le varie librerie di sistema ogni qualvolta un comando `java` verrà eseguito, che viene salvato all'interno del file contenente tutte le variabili d'ambiente, che si trova al path: `"etc/environment"`.

Listing 5.3. Installazione Docker e Docker Compose

```

1  if [ -x "$(command -v docker)" ]; then
2  printf "${GREEN}docker installed... OK${COLOR_RESET}\n"
3  else
4  printf "${BLUE}Docker engine not installed. I'm going to install
    it\n${COLOR_RESET}\n"
5  sudo apt-get update
6  sudo apt-get install ca-certificates curl gnupg
7  sudo install -m 0755 -d /etc/apt/keyrings

```

```

8      curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
      --dearmor -o /etc/apt/keyrings/docker.gpg
9      sudo chmod a+r /etc/apt/keyrings/docker.gpg
10     echo \
11     "deb [arch=$(dpkg --print-architecture)
      signed-by=/etc/apt/keyrings/docker.gpg]
      https://download.docker.com/linux/ubuntu \
12     "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
13     sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
14     sudo apt-get --yes install docker-ce docker-ce-cli containerd.io
      docker-buildx-plugin docker-compose-plugin
15     printf "${GREEN}docker engine installed installed...
      OK${COLOR_RESET}\n"
16     fi
17     if [ -x "$(command -v docker-compose)" ]; then
18     printf "${GREEN}docker-compose installed... OK${COLOR_RESET}\n"
19     else
20     printf "${BLUE}Docker-compose package not installed. I'm going to
      install it\n${COLOR_RESET}\n"
21     sudo apt-get update
22     sudo apt-get --yes install docker-compose
23     printf "${GREEN}docker-compose installed... OK${COLOR_RESET}\n"
24     fi

```

In questa parte di codice viene installato Docker ed il suo plugin docker-compose. Similmente all'installazione di curl e pv viene controllata l'esistenza di entrambi alle righe 1 e 17 e successivamente vengono installati utilizzando la documentazione ufficiale di Docker[13] e di Docker-Compose[14]

Listing 5.4. Installazione Z3

```

1      FILE=/home/z3
2      if [ -d "$FILE" ]; then
3      printf "${GREEN}z3 exists in /home directory... OK${COLOR_RESET}\n"
4      else
5      printf "${BLUE}z3 doesnt exist in home directory, im going to
      install it\n${COLOR_RESET}\n"
6      cd /home
7      sudo curl -LO https://github.com/Z3Prover/z3/releases/download/
      z3-4.8.15/z3-4.8.15-x64-glibc-2.31.zip
8      sudo unzip z3-4.8.15-x64-glibc-2.31.zip
9      sudo rm z3-4.8.15-x64-glibc-2.31.zip
10     sudo mv z3-4.8.15-x64-glibc-2.31 z3 #rename z3-4.8.15 into z3
11     echo "LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/z3/bin/" | sudo tee
      -a /etc/environment # Care!! >> not > or it will over write
      environment variables
12     sudo echo "Z3=/home/z3/bin/" | sudo tee -a /etc/environment
13     printf "${GREEN}Z3 installed and environment variables setted...
      OK${COLOR_RESET}\n"
14     fi

```

Come ultimo passo l'installer controlla l'esistenza del tool di Z3 all'interno del

sistema operativo. Diversamente dai packages standard installati finora, per potersi far rilevare correttamente dal framework è necessario che Z3 sia installato nella Home directory con un path definito come `"/home/z3"`. Conseguentemente per controllare l'esistenza del tool è necessario quindi verificare solo la presenza del folder z3 nella home directory. In caso di installazione necessaria tramite una chiamata API con curl viene scaricato dal github ufficiale di Z3 [15] la versione desiderata di Z3, estratta nella home directory e rinominata in `"z3"`. Infine vengono istanziate 2 variabili d'ambiente ovvero `"LD-LIBRARY-PATH"` e `"Z3"` all'interno del file con le variabili d'ambiente che si trova al path `"/etc/environment"`.

### 5.3 Implementazione

Il secondo e più corposo lavoro all'interno di questa prima parte della tesi è stata l'effettiva realizzazione della Demo A tramite ambiente virtuale. L'obiettivo principale che ci si è posti da raggiungere è stato di mostrare le potenzialità di Verefoo proponendo all'utente una topologia di rete che si avvicinasse il più possibile a quella di una rete di un'azienda di piccole dimensioni. Per raggiungere tali scopi la soluzione che è stata implementata è la seguente:

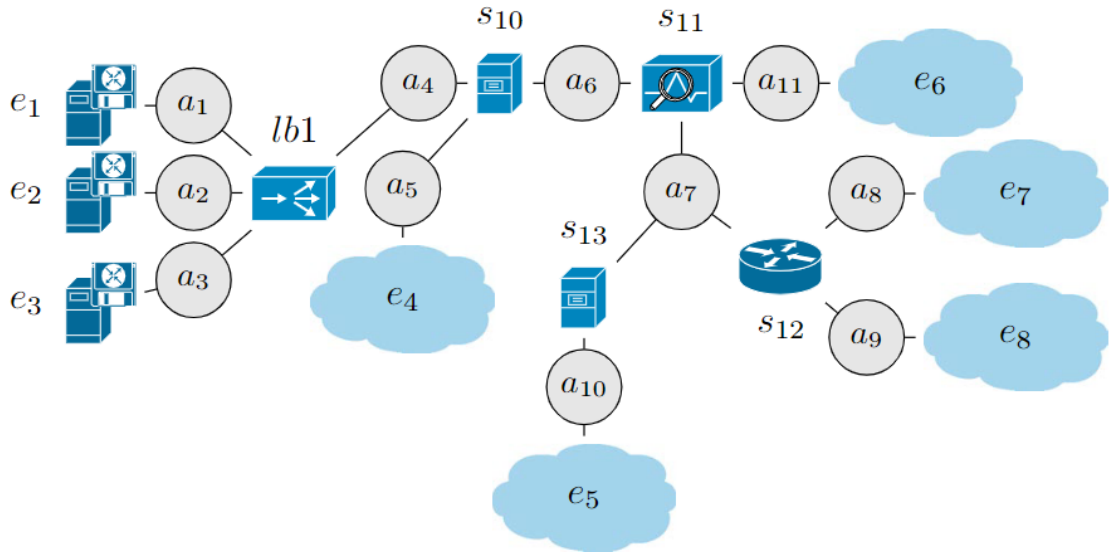


Figure 5.1. Service Graph Demo A

Come è possibile notare, la topologia proposta presenta sulla sinistra 3 WebServer (e1,e2,e3) il cui traffico viene gestito da un load balancer (lb1) il quale si occupa di evitare congestioni di traffico durante le comunicazioni fra clients e servers. Congiuntamente ai WebServer sono presenti anche 5 endpoints (e4,e5,e6,e7,e8) che all'interno dell'ambiente virtuale verranno istanziati come fossero dei WebClient. Al fine di poter testare il corretto funzionamento del framework all'interno della rete sono anche presenti dei nodi che fungeranno da monitor, come il nodo s11, ed altri che invece svolgeranno la semplice funzione di forwarder con le rispettive static routes, per simulare dei router generici. Infine sono presenti diversi nodi

al momento vuoti che rappresentano gli allocation places che Verefoo richiede fra i suoi possibili input per velocizzare il processo di risoluzione del problema MaxSMT. All'interno di questi nodi, il framework potrà collocare delle network security functions per assicurarsi il corretto funzionamento dei requisiti di sicurezza all'interno della rete.

Di seguito viene fornita una tabella con la definizione di ogni nodo, del suo indirizzo IP che verrà utilizzato nell'ambiente virtuale e della sua funzionalità all'interno della topologia.

Name	IP	Functionality
e1	130.10.0.1	Web servers behind load balancer b1
e2	130.10.0.2	*
e3	130.10.0.3	*
e4	40.40.41.1	Web Client
e5	40.40.42.1	Web Client
e6	88.80.84.1	Web Client
e7	192.168.1.1	Web Client
e8	192.168.2.1	Web Client
lb1	130.10.0.4	Load Balancer
s10	33.33.33.2	Web Cache
s11	33.33.33.3	Forwarder
s12	220.124.30.1	Forwarder
s13	33.33.33.4	Forwarder
a7	1.0.0.7	Forwarder

Table 5.1. Node definitions and functionalities

Oltre alla definizione del Service Graph per la demo è necessario fornire al framework anche l'insieme di security requirement che la rete deve avere, ovvero una traduzione di tutte le proprietà di sicurezza che vorremmo fossero presenti all'interno della nostra azienda.

Per poter simulare il più possibile un'azienda reale è stato stabilito di creare numerosi requisiti di protezione con l'obiettivo di avere una topologia di output che contenesse un numero elevato di VPN Gateway (6), in quanto è molto comune anche in ambienti di smart working avere dei tunnel dedicati per ogni Host che lavora all'esterno della rete aziendale.. Al fine di poter ottenere una topologia simile sono state definite le seguenti regole:

Policy	IPSrc	IPDst	pSrc	pDst	tProto	Confidentiality	Integrity	Untrusted nodes
Protection	40.40.41.1	130.10.0.1	*	22	ANY	AES-256-CBC	SHA2-256	33.33.33.2
Protection	88.80.84.1	130.10.0.*	*	80	ANY	AES-256-CBC	SHA2-256	33.33.33.2/33.33.33.3
Protection	192.168.1.1	130.10.0.1	*	*	ANY	AES-256-CBC	SHA2-256	33.33.33.2/33.33.33.3
Protection	40.40.42.1	192.168.2.1	*	*	ANY	AES-256-CBC	SHA2-256	33.33.33.4/220.124.30.1

Table 5.2. Security Requirements Definition DemoA



- **Prima Regola:** Il Web Client e4 deve poter comunicare in maniera sicura con il Web Server e1. Il traffico originato da e4 può utilizzare qualsiasi porta di uscita per il protocollo di trasporto ma il Server e1 deve ricevere i dati in ingresso unicamente dalla porta 22. È possibile utilizzare sia il protocollo UDP che TCP per il trasporto. Viene specificato infine il nodo s10 come nodo non sicuro e attraverso il quale il traffico deve passare cifrato.
- **Seconda regola:** Il Web Client e6 deve poter comunicare in maniera sicura con tutti i Web Server (e1,e2,e3). Il traffico originato da e6 può utilizzare qualsiasi porta di uscita per il protocollo di trasporto ma i Server devono ricevere i dati in ingresso solo dalla porta 80. È possibile utilizzare sia il protocollo UDP che TCP per il trasporto. In questo caso i nodi considerati non sicuri sono s10 e s11.
- **Terza Regola:** Il Web Client e7 deve poter comunicare in maniera sicura con il Web Server e1. Non ci sono limitazioni sulle porte per il traffico in entrata ed in uscita e può essere utilizzato qualsiasi protocollo di quarto livello per il trasporto. I nodi non sicuri, come per la seconda regola, sono s10 e s11.
- **Quarta Regola:** Il Web Client e5 deve poter comunicare in maniera sicura con il Web Client e8. Come per la terza regola, non ci sono limitazioni sulle porte e protocollo di trasporto da utilizzare. I nodi considerati non sicuri per questa regola sono s12 ed s13.

Definire questi elementi all'interno del framework di Verefoo non è comunque sufficiente per creare una demo, in quanto come già visto al Capitolo[2] è sufficiente descrivere in file XML la definizione dei nodi ed i corrispettivi indirizzi ip e nodi adiacenti.

Per avere una demo che invece dimostri la correttezza delle operazioni del framework è necessario istanziare gli elementi descritti in output e successivamente tradurli in un ambiente virtuale di container Docker. Di seguito è quindi disponibile la definizione di alcuni services per la topologia descritta in precedenza:

Listing 5.5. Definizione di Services dell'ambiente virtuale DemoA

```
services:
  server1:
    container_name: server1
    hostname: server1
    image: endpoint
    cap_add:
      - NET_ADMIN
    command: sh -c "route del default && route add -net 0.0.0.0 netmask 0.0.0.0
      gw 130.10.0.4 && tail -F anything"
    networks:
      servers:
        ipv4_address: 130.10.0.1

  end4:
    container_name: end4
    hostname: end4
    image: endpoint
    cap_add:
```

```

– NET_ADMIN
command: sh -c "route del default && route add -net 0.0.0.0 netmask 0.0.0.0
          gw 40.40.41.100 && tail -F anything"
networks:
endpoints4:
  ipv4_address: 40.40.41.1

```

Scendendo nel dettaglio sono mostrati principalmente un server ed un endpoint, che appartengono a sottoreti differenti ma che hanno la stessa immagine di partenza nella costruzione del container. Entrambi infatti verranno configurati come endpoint e le loro forwarding route verranno cancellate completamente, indicando come unico gateway un ip stabilito arbitrariamente nella sottorete in cui appartengono. Inoltre tramite il cap-add viene dato a ciascun endpoint la capacità di NET-ADMIN che consente al container di configurare l'interfaccia di rete e le route.

## 5.4 Output

Fornendo gli input definiti precedentemente il framework cercherà una soluzione che non solo soddisfi tutte le regole, ma che impieghi anche il minor numero di risorse necessarie per garantire tali proprietà. Verefoo risolverà quindi un problema di tipo MaxSMT (Maximum Satisfiability Modulo Theories). Nel caso specifico di questa topologia con i requisiti di sicurezza previamente definiti nel paragrafo precedente il risultato prodotto in output sarà il seguente:

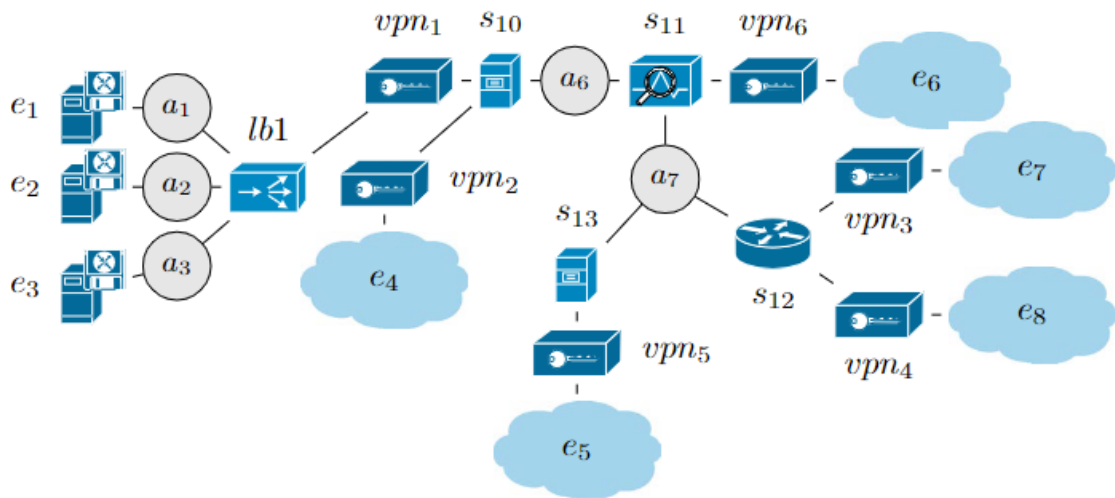


Figure 5.2. Verefoo Output Demo A

Come era prevedibile, diversi VPN Gateway sono stati allocati al posto dei vari Allocation Places definiti nell'input fornito a Verefoo. Data questa soluzione che dovrebbe essere la soluzione al problema definito prima, proveremo a testare dentro l'ambiente virtuale la funzionalità dei vari tunnel VPN, per assicurarci che il framework lavori correttamente. Prima di procedere però è importante anche analizzare i vari elementi che Verefoo ha prodotto. Il framework infatti non ha solo

allocato negli allocation places corretti i Gateway, ma ha anche fornito anche una configurazione automatica da utilizzare. Nel caso specifico la configurazione per i 6 Gateway è la seguente:

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	EXIT	192.168.1.1	130.10.0.1	*	*	ANY
2	EXIT	88.80.84.1	130.10.0.3	*	80	ANY
3	EXIT	40.40.41.1	130.10.0.1	*	22	ANY
4	EXIT	88.80.84.1	130.10.0.1	*	80	ANY
5	EXIT	88.80.84.1	130.10.0.2	*	80	ANY
6	ACCESS	130.10.0.1	192.168.1.1	*	*	ANY
7	ACCESS	130.10.0.3	88.80.84.1	80	*	ANY
8	ACCESS	130.10.0.1	40.40.41.1	22	*	ANY
9	ACCESS	130.10.0.1	88.80.84.1	80	*	ANY
10	ACCESS	130.10.0.2	88.80.84.1	80	*	ANY

Table 5.3. VPN Gateway 1

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	ACCESS	40.40.41.1	130.10.0.1	*	22	ANY
2	EXIT	130.10.0.1	40.40.41.1	22	*	ANY

Table 5.4. VPN Gateway 2

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	ACCESS	192.168.1.1	130.10.0.1	*	*	ANY
2	EXIT	130.10.0.1	192.168.1.1	*	*	ANY

Table 5.5. VPN Gateway 3

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	ACCESS	192.168.2.1	40.40.42.1	*	*	ANY
2	EXIT	40.40.42.1	192.168.2.1	*	*	ANY

Table 5.6. VPN Gateway 4

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	ACCESS	40.40.42.1	192.168.2.1	*	*	ANY
2	EXIT	192.168.2.1	40.40.42.1	*	*	ANY

Table 5.7. VPN Gateway 5

#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	ACCESS	88.80.84.1	130.10.0.1	*	80	ANY
2	ACCESS	88.80.84.1	130.10.0.2	*	80	ANY
3	ACCESS	88.80.84.1	130.10.0.3	*	80	ANY
4	EXIT	130.10.0.1	88.80.84.1	80	*	ANY
5	EXIT	130.10.0.2	88.80.84.1	80	*	ANY
6	EXIT	130.10.0.3	88.80.84.1	80	*	ANY

Table 5.8. VPN Gateway 6

Prendendo in esempio il VPN Gateway 1 è possibile notare come Veref00 non si limita ad eseguire delle configurazioni semplici ma è in grado di produrre anche configurazioni miste, ovvero non unicamente di ingresso o di uscita dal tunnel. È infatti possibile notare come tutto il traffico in transito può sia cifrato in Accesso (ACCESS) al tunnel che decifrato in uscita (EXIT) al tunnel.

Oltre alle configurazioni dei gateway VPN, Veref00 offre, tramite un traduttore automatico, dei file di configurazione per istanziare i tunnel VPN utilizzando Strongswan. A fine di esempio viene fornito il file di configurazione di Strongswan di uno dei vari gateway della topologia, che verrà utilizzato per istanziare il tunnel VPN nell'ambiente virtuale:

```
connections {
  site-site {
    local_addrs = 20.0.1.1
    remote_addrs = 20.0.7.2
    local {
      auth = pubkey
      certs = VpnConfig1Cert.pem
      id = VpnConfig1.strongswan.org
    }
    remote {
      auth = pubkey
      id = VpnConfig2.strongswan.org
    }
  }
  children {
    net-net {
      local_ts = 130.10.0.1/24
      remote_ts = 192.168.1.1/24
      start_action = trap|start
      rekey_time = 5400
      rekey_bytes = 500000000
      rekey_packets = 1000000
      esp_proposals = aes256-sha2_256-modp2048
    }
  }
}
version = 2
mobike = no
```

```

    reauth_time = 10800
  }
}

```

## 5.5 Verifiche e Test

Definito l'ambiente virtuale e impostato le configurazioni di Verefio prodotte in output è necessario testare e dimostrare che il framework ha prodotto una soluzione corretta.

Al fine di eseguire tutte le operazioni di verifica e test in maniera più trasparente possibile è stato inserito, in aggiunta alla topologia già definite nella figura (5.2) un nodo aggiuntivo, di collegamento fra il vpn gateway 4 e l'endpoint 8. In questo modo, utilizzando tcpdump e analizzando le interfacce di rete sarà possibile notare in quali nodi della topologia il traffico passerà in chiaro e in quali invece verrà cifrato. È importante sottolineare come all'interno dell'ambiente virtuale tutti gli elementi sono stati configurati precedentemente per velocizzare le operazioni di testing, di conseguenza i vari certificati pubblici e privati sono stati già generati ed inseriti all'interno dei nodi VPN, e le route di trasmissione statiche sono state già inserite per tutta la topologia tramite il file di docker-compose.

All'interno di questa tesi, per semplificare le operazioni di verifica effettuate non verranno verificati tutti i requisiti di sicurezza definiti precedente ma ci si limiterà a testare la connessione fra l'endpoint e5 e l'endpoint e8.

Per iniziare il test di correttezza sul tunnel vpn che collega i due endpoint è necessario attivare strongswan con i certificati dei due gateway vpn, utilizzando quindi il comando *"swanctl -q"* all'interno del container di vpn4 e vpn5:

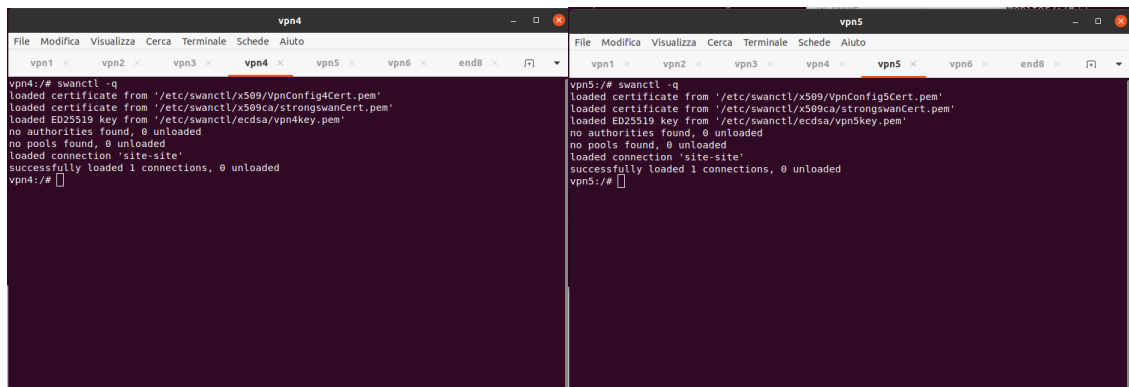
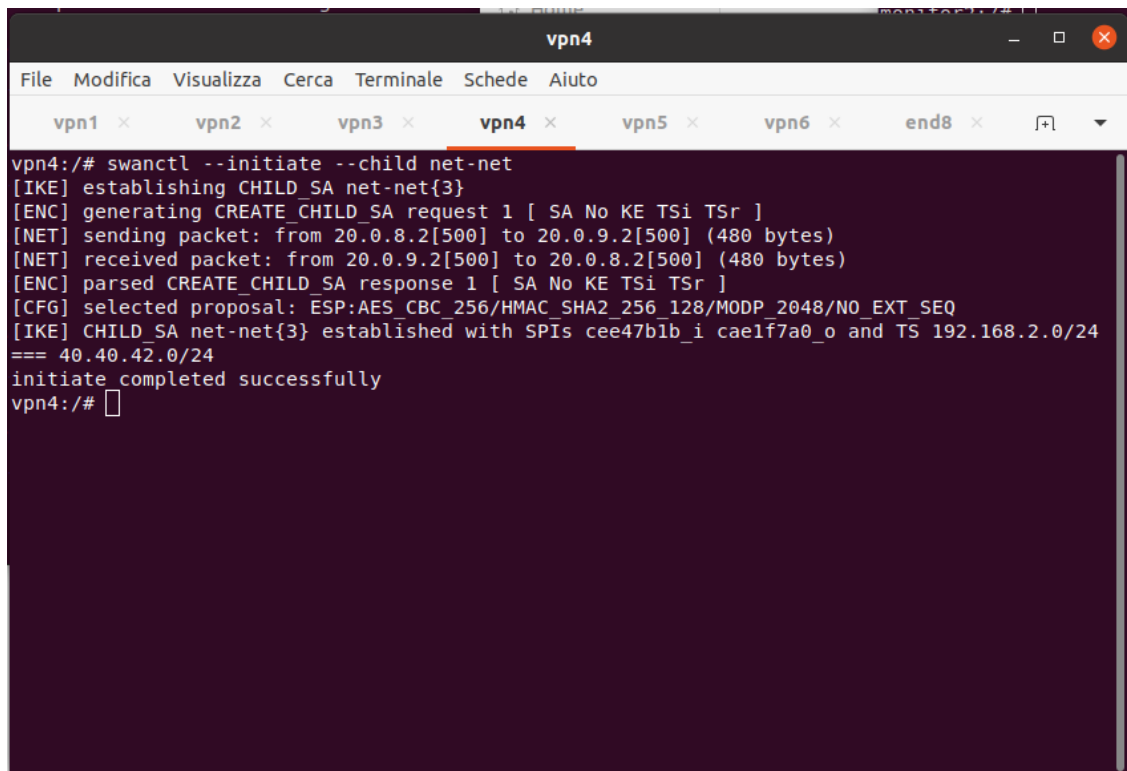


Figure 5.3. VpnGateway4

Figure 5.4. VpnGateway5

Una volta caricati correttamente i vari certificati all'interno dei container, è necessario utilizzare strongswan per creare il tunnel esplicitamente. Per fare ciò è necessario invocare il seguente comando: `swanctl --initiate --child net-net`. Attraverso l'output mostrato su terminale è possibile verificare che i tunnel siano stati correttamente istanziati o se vi è stato qualche problema nello scambio dei parametri di sicurezza.



```

vpn4:/# swanctl --initiate --child net-net
[IKE] establishing CHILD_SA net-net{3}
[ENC] generating CREATE_CHILD_SA request 1 [ SA No KE TSr ]
[NET] sending packet: from 20.0.8.2[500] to 20.0.9.2[500] (480 bytes)
[NET] received packet: from 20.0.9.2[500] to 20.0.8.2[500] (480 bytes)
[ENC] parsed CREATE_CHILD_SA response 1 [ SA No KE TSr ]
[CFG] selected proposal: ESP:AES_CBC_256/HMAC_SHA2_256_128/MODP_2048/NO_EXT_SEQ
[IKE] CHILD_SA net-net{3} established with SPIs cee47b1b_i cae1f7a0_o and TS 192.168.2.0/24
=== 40.40.42.0/24
initiate completed successfully
vpn4:/#

```

Figure 5.5. Verifica Tunnel VPN

Come si può notare dall'output stampato sul terminale il protocollo IPsec utilizza IKE per scambiarsi le chiavi e successivamente viene stabilita la security association definita "net-net" nel file di configurazione swanctl. Una volta che viene stabilita la security association viene definito il traffic selector, ovvero si specifica quali pacchetti dovranno entrare nel tunnel VPN (con rispettivo IPsrc ed IPdst). Infine vengono scelti gli algoritmi per l'autenticazione e la cifratura dei pacchetti. Se entrambi i nodi riescono ad accettare le stesse condizioni allora il tunnel viene creato. Nel caso in esempio quindi come si può leggere nell'output il tunnel è stato configurato da Verefoo correttamente.

Da questo momento in poi quindi qualsiasi pacchetto inviato dall'endpoint e8 all'endpoint e5 e viceversa verrà cifrato e non sarà possibile ispezionarlo all'interno del tunnel VPN. Per verificare ciò verranno utilizzati due monitor, il primo definito monitorESP sarà fornito dal container del nodo s12, mentre il secondo definito monitorICMP sarà il nodo precedentemente aggiunto fra l'endpoint e8 e il gateway vpn4. In questo modo è possibile utilizzare tcpdump e osservare le interfacce di rete nelle quali passano i pacchetti fra e8 ed e5. Per quanto riguarda il monitorESP la

corretta interfaccia è eth1 mentre per il monitorICMP è eth0. Eseguiamo quindi il seguente comando: `tcpdump -i [interfacename]` su entrambi i monitor per osservare il traffico dati.

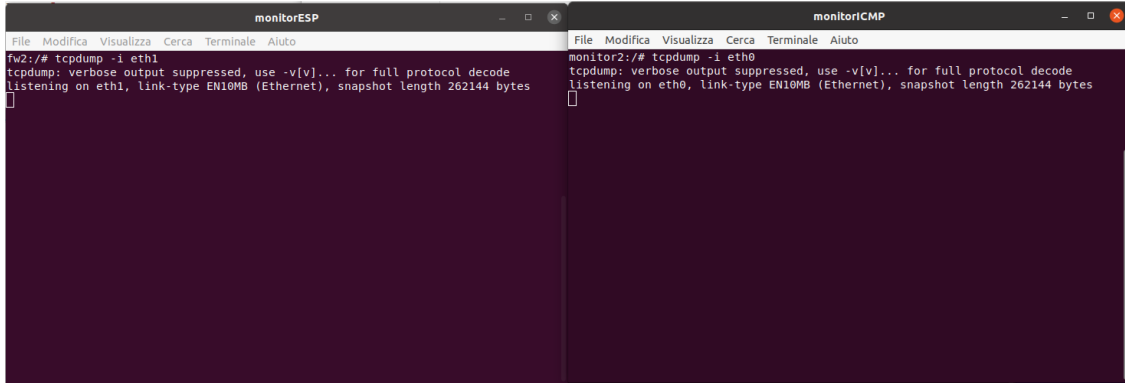


Figure 5.6. Tunnel stabilito

Una volta configurato tcpdump in entrambi i monitor tutti i pacchetti in transito attraverso questi due nodi verranno registrati e controllati, indicando ip e protocollo di livello 3 che viene utilizzato. Per testare che i pacchetti vengano effettivamente cifrati correttamente quindi mandiamo dei pacchetti di ping da e8 ad e5 e controlliamo l'output dei due monitor:

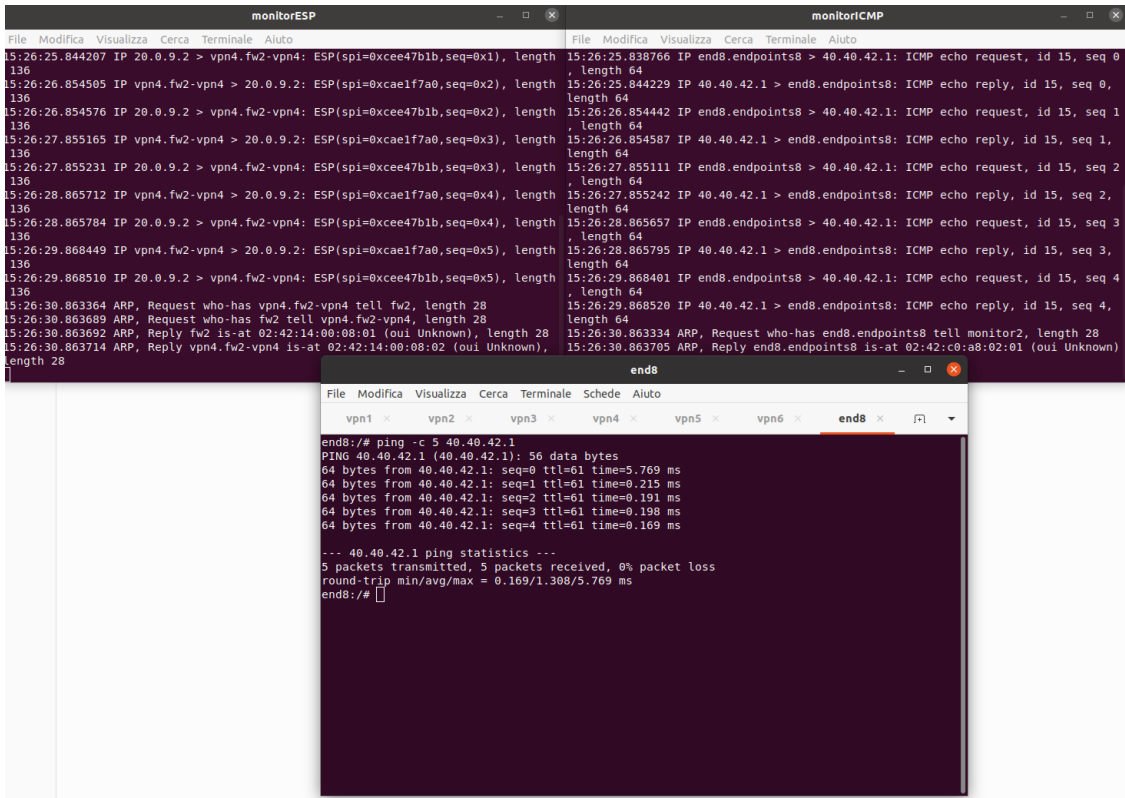


Figure 5.7. Prova di correttezza

Osservando l'ultimo output si può dedurre che i tunnel funzionano correttamente. Per il monitor ICMP che si trova fra l'endpoint e il vpn gateway 4 il traffico viene trasmesso in chiaro ed i pacchetti vengono visualizzati come pacchetti ICMP da tcpdump, invece i pacchetti che transitano su s12 cioè dopo il passaggio dal vpn gateway vengono visualizzati come pacchetti ESP che è il protocollo utilizzato da IPsec per incapsulare i dati nei tunnel. Di conseguenza l'output prodotto da verfoo soddisfa le regole fornite ed utilizza il minor numero di risorse allocate possibili, verificando la correttezza del framework.

Con questo output è possibile quindi affermare il corretto funzionamento del framework e della Demo A e, di conseguenza, il raggiungimento del primo obiettivo di questo lavoro di tesi.



# Chapter 6

## Merge

Merge works of the 2 verefoo versions

# Chapter 7

## DemoB

Name	IP	Functionality
C1-1	20.1.1.1	Web Client behind load balancer
C1-2	20.1.2.1	Web Client behind load balancer
C1-3	20.1.3.1	Web Client behind load balancer
Lb	33.33.33.1	Load balancer
C2-1	20.2.1.1	Web Client behind NAT
C2-2	20.2.2.1	Web Client behind NAT
C2-3	20.2.3.1	Web Client behind NAT
C2-4	20.2.4.1	Web Client behind NAT
FW1	33.33.33.3	Forwarder
C3-1	20.3.1.1	Web Client
C4-1	20.4.1.1	Web Client
Mnt	33.33.33.2	Traffic Monitor
S1	10.0.0.1	Web Server
S2	10.0.0.2	Web Server

Table 7.1. Esempio di tabella con tre colonne senza linee verticali.

Policy	IPSrc	IPDst	pSrc	pDst	tProto	Confidentiality	Integrity	Untrusted no
Isolation	20.1.1.1	10.0.0.1	*	*	ANY	//	//	//
Reachability	20.1.1.1	10.0.0.2	*	*	ANY	//	//	//
Isolation	20.1.2.1	10.0.0.1	*	*	ANY	//	//	//
Reachability	20.1.2.1	10.0.0.2	*	*	ANY	//	//	//
Isolation	20.1.3.1	10.0.0.1	*	*	ANY	//	//	//
Reachability	20.1.3.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.2.1.1	10.0.0.1	*	*	ANY	//	//	//
Isolation	20.2.1.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.2.2.1	10.0.0.1	*	*	ANY	//	//	//
Isolation	20.2.2.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.2.3.1	10.0.0.1	*	*	ANY	//	//	//
Isolation	20.2.3.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.2.4.1	10.0.0.1	*	*	ANY	//	//	//
Isolation	20.2.4.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.3.1.1	10.0.0.1	*	*	ANY	//	//	//
Reachability	20.3.1.1	10.0.0.2	*	*	ANY	//	//	//
Reachability	20.4.1.1	10.0.0.1	*	*	ANY	//	//	//
Reachability	20.4.1.1	10.0.0.2	*	*	ANY	//	//	//
Protection	20.1.1.1	10.0.0.2	*	22	ANY	AES-256-CBC	SHA2-256	33.33.33.2

Table 7.2. Esempio di tabella con tre colonne senza linee verticali.

Name	IP	Functionality
C1-1	20.1.1.1	Web Client behind load balancer
C1-2	20.1.2.1	Web Client behind load balancer
C1-3	20.1.3.1	Web Client behind load balancer
Lb	33.33.33.1	Load balancer
C2-1	20.2.1.1	Web Client behind NAT
C2-2	20.2.2.1	Web Client behind NAT
C2-3	20.2.3.1	Web Client behind NAT
C2-4	20.2.4.1	Web Client behind NAT
FW1	33.33.33.3	Forwarder
C3-1	20.3.1.1	Web Client
C4-1	20.4.1.1	Web Client
Mnt	33.33.33.2	Traffic Monitor
S1	10.0.0.1	Web Server
S2	10.0.0.2	Web Server

Table 7.3. Esempio di tabella con tre colonne senza linee verticali.

Name	Action	IPSrc	IPDst	pSrc	pDst	tProto
VPN1	ACCESS	20.1.1.1	10.0.0.2	*	22	ANY
VPN2	EXIT	20.1.1.1	10.0.0.2	*	22	ANY

Table 7.4. Esempio di tabella con tre colonne senza linee verticali.

Default Action	Action	IPSrc	IPDst	pSrc	pDst	tProto
ALLOW	DENY	20.1.*.*	10.0.0.1	*	*	ANY
ALLOW	DENY	20.2.*.*	10.0.0.2	*	*	ANY

Table 7.5. Esempio di tabella con tre colonne senza linee verticali.

# Chapter 8

## Conclusions

Conclusion and future works

# Bibliography

- [1] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, “Towards a fully automated and optimized network security functions orchestration,” in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, 2019, pp. 1–7.
- [2] J. M. Halpern and C. Pignataro, “Service function chaining (SFC) architecture,” *RFC*, vol. 7665, pp. 1–32, 2015. [Online]. Available: <https://doi.org/10.17487/RFC7665>
- [3] StrongSwan Project, *StrongSwan Documentation*, 2024. [Online]. Available: <https://www.strongswan.org/documentation.html>
- [4] “IPsec - Internet Protocol Security,” <https://tools.ietf.org/html/rfc4301>, 2005, rFC 4301.
- [5] (2024) Docker documentation. Docker, Inc. [Online]. Available: <https://docs.docker.com/>
- [6] (2024) What is a virtual machine? VMware, Inc. Accessed on February 13, 2024. [Online]. Available: <https://www.vmware.com/topics/glossary/content/virtual-machine.html>
- [7] Docker, Inc. (2024) What is a container? Accessed on February 13, 2024. [Online]. Available: <https://www.docker.com/resources/what-container/>
- [8] (2024) What is a hypervisor? VMware, Inc. Accessed on February 13, 2024. [Online]. Available: <https://www.vmware.com/topics/glossary/content/hypervisor.html>
- [9] M. Lindström, “Containers & virtual machines: A performance, resource & power consumption comparison,” Master’s thesis, Blekinge Institute of Technology, SE-371 79 Karlskrona, June 2024. [Online]. Available: <http://www.diva-portal.org/smash/get/diva2:1665606/FULLTEXT01.pdf>
- [10] (2024) Docker documentation. Docker, Inc. [Online]. Available: <https://docs.docker.com/compose/features-uses/>
- [11] “Yaml,” <https://yaml.org/>, definition of the markup language.
- [12] (2024) Docker compose - services. Docker Documentation. Accessed on February 13, 2024. [Online]. Available: <https://docs.docker.com/compose/compose-file/05-services/>
- [13] Docker Documentation. (2024) Install docker engine on ubuntu. Docker. [Online]. Available: <https://docs.docker.com/engine/install/ubuntu/>
- [14] ——. (2024) Install docker compose on linux. Docker. [Online]. Available: <https://docs.docker.com/compose/install/linux/>
- [15] Microsoft. (2024) Z3 theorem prover. [Online]. Available: <https://github.com/Z3Prover/z3>