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

List of Figures

List of Tables

Listings

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 instanziati, 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 affinchè 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 affinchè 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 affinchè 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 rete. 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 carateristiche di sicurezza differenti. Per comprendere meglio il concetto di SFC, un esempio fornito è il seguente:

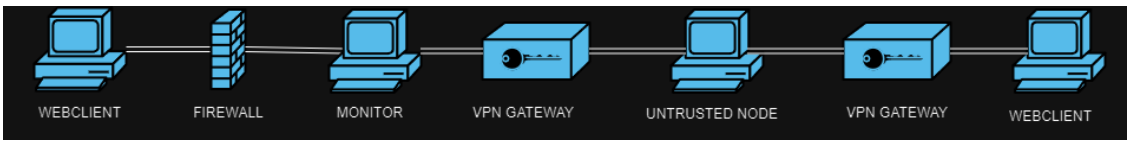

Figure 2.1. Esempio di Service Function Chain

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 definice 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 efficenza 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 NSFs
2. La configurazione ottimale dei vari NSFs.

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 percoso 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 comunicazone 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.

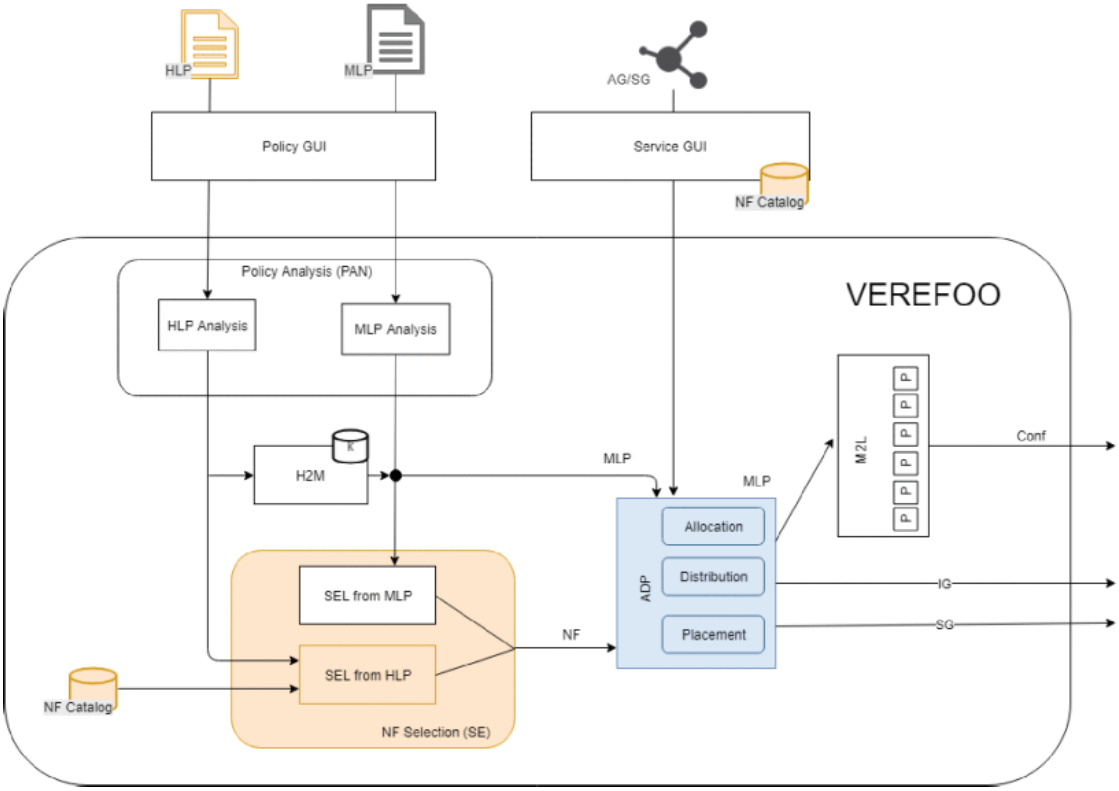

Figure 2.2. Architettura di VEREFOO [1]

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 affinchè la topologia soddisfi i NSRs richiesti. In caso di errore un report viene solitamente fornito in output per comprendere il perchè 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, perchè 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.

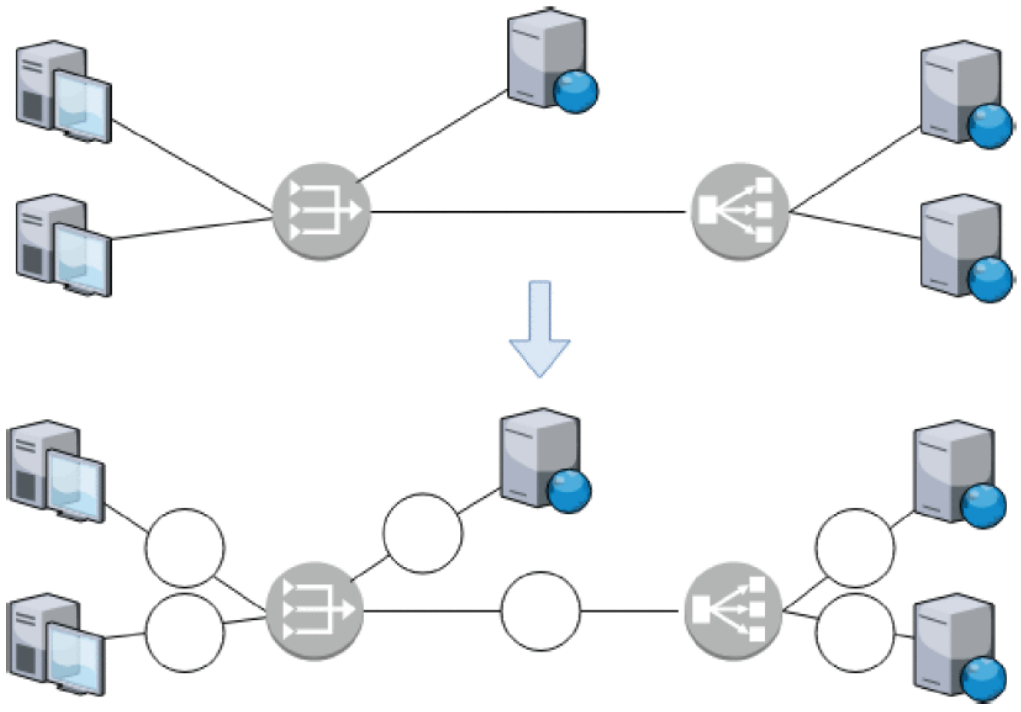

Figure 2.3. Esempi di Service (in alto) e Allocation(in basso) Graphs [1]

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:

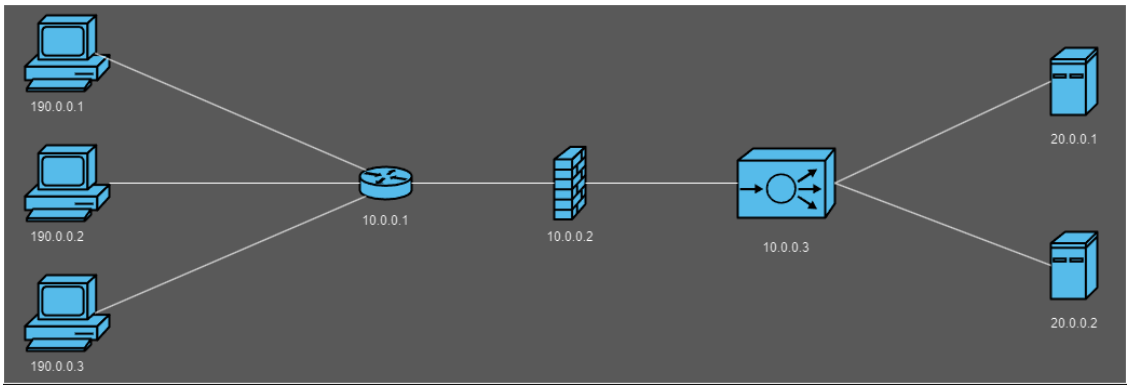

Figure 2.4. Esempio di un Service Graph

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 affinchè 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

```
[ruleType, IPSrc, IPDst, portSrc, portDst, transportProto]
```

- **ruleType**: specifica quale proprietà stiamo definendo. Allo stato attuale del framework ci sono 3 possibili opzione: 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 affinchè 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:

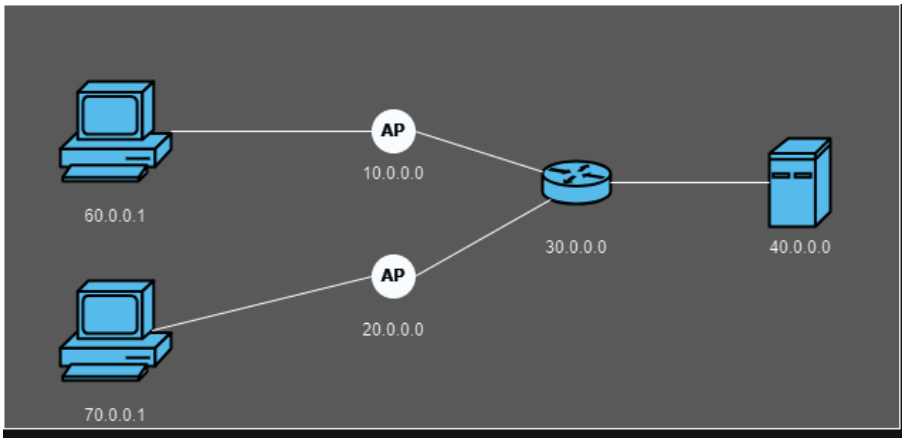

Figure 2.5. Grafo di allocazione per requisito di raggiungibilità

Successivamente è necessario definire la definizione di requisito di raggiungibilità tra l'Host-S 60.0.01 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 aspettato è il seguente:

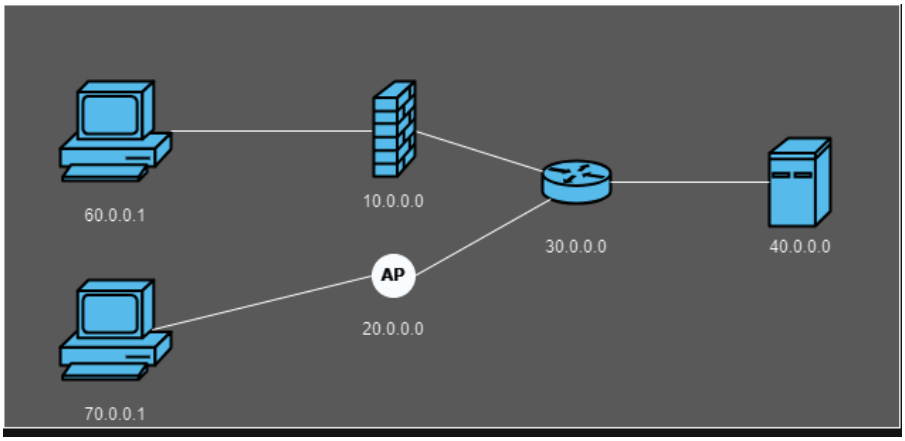

Figure 2.6. Output grafo di esempio per requisito di raggiungibilità

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:

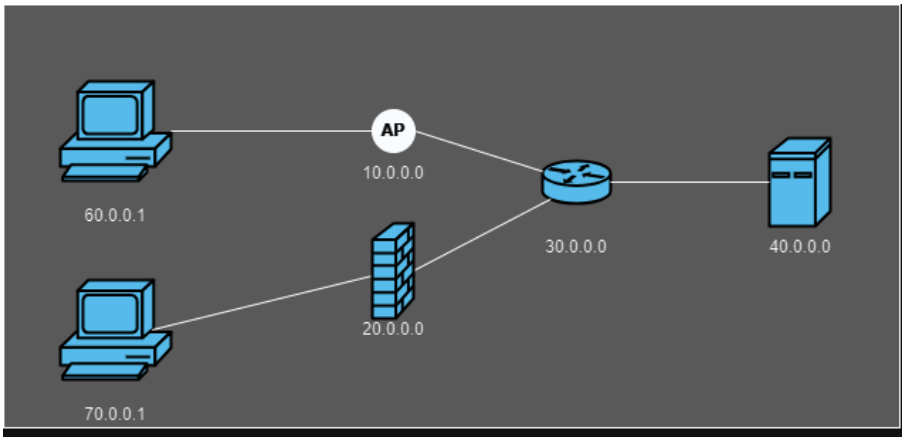

Figure 2.7. Output grafo di esempio per requisito di isolamento

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,
   secTecnology,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:

- **secTecnology**: 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 appartententi 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 trafico.
- **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 flusso 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 definiscano e imnplementano 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:

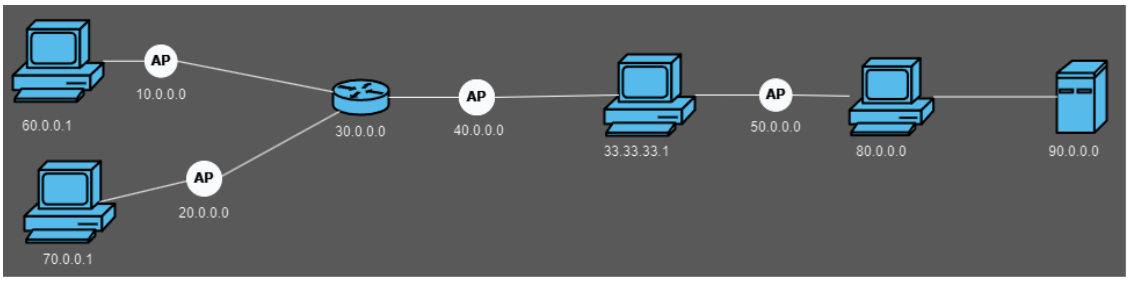

Figure 2.8. Grafo di allocazione d'esempio per requisiti di protezione

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:

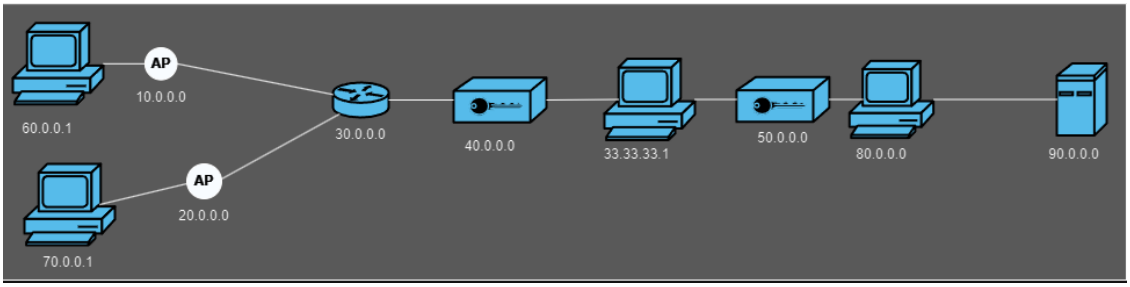

Figure 2.9. Output grafo di esempio per requisito di protezione

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:

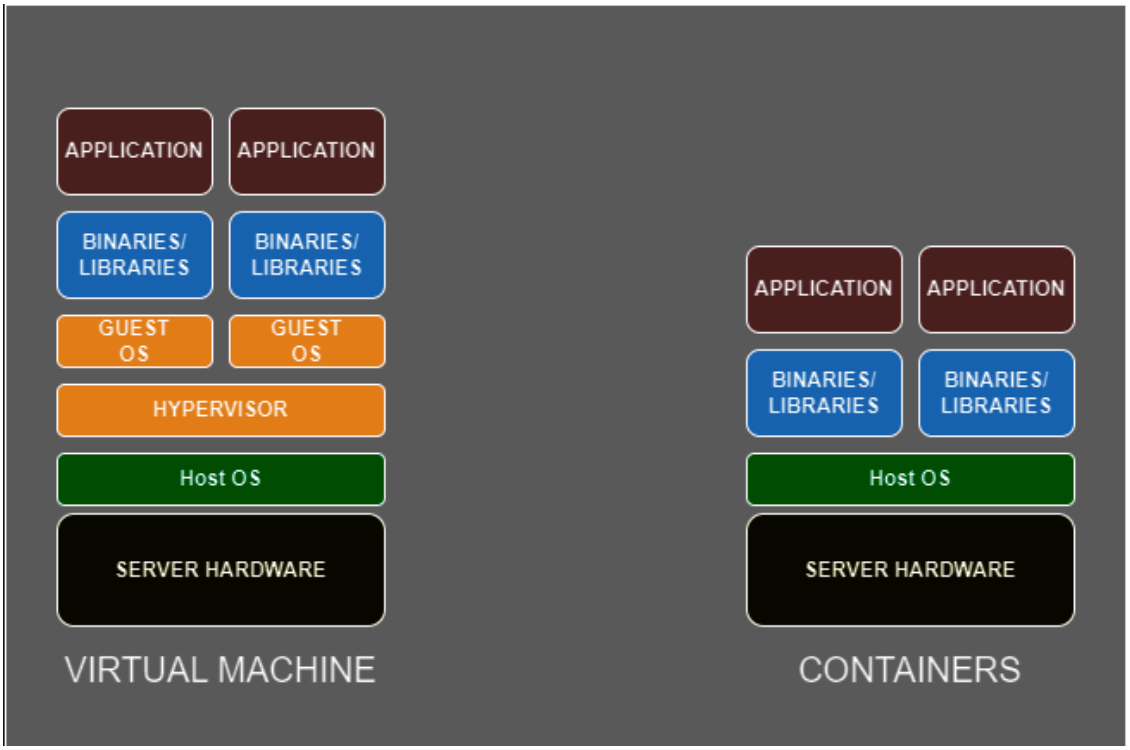

Figure 3.1. Architettura Virtual Machine e Containers

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 difficlmente 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], differentemente 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:

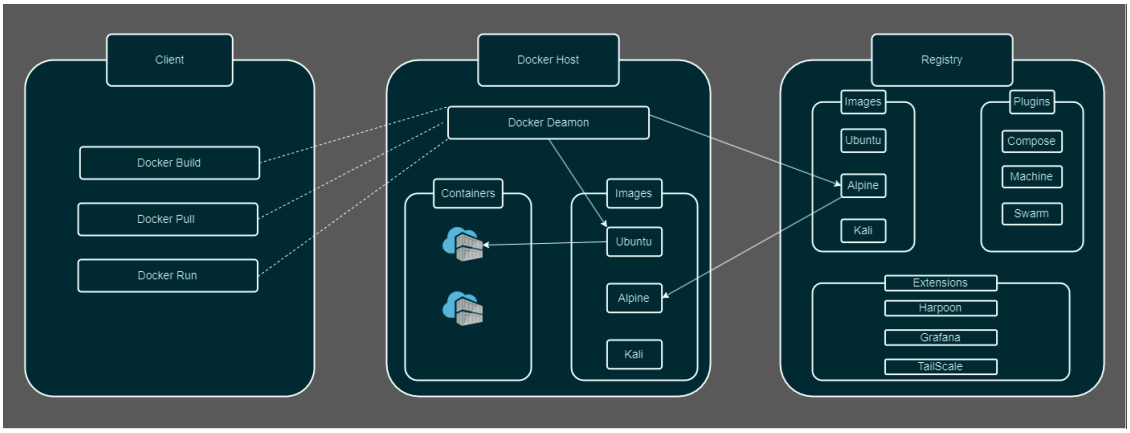

Figure 3.2. Architettura Docker

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 esistiti 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 universamentalmente utilizzato grazie alla sua semplicità, facilità di scrittura e di lettuera 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[12]"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-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 affinchè 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 esplicatemente 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 correto 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 approcciavano 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 proprità 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

DemoA

5.1 Introduction

Welcome to the demo of Verefoo for VPNs. What you will see written here will be a simple stand-alone demo in which the features and capabilities of Verefoo are highlighted. Before we begin we need to define a network topology that Verefoo will need to relate to, more precisely you will also need to define the respective IP addresses and links between the different nodes. During the course of the demo, a virtual environment developed with Docker Containers will be instantiated that will reproduce the topology presented in this document.

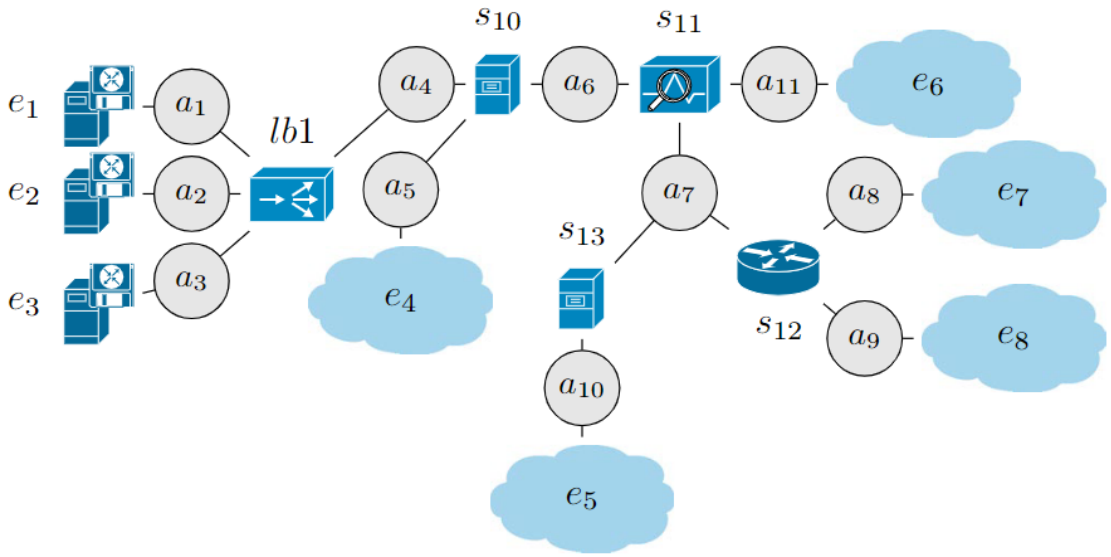

Figure 5.1. Service Graph

As can be seen, the proposed topology has 3 webservers on the left whose traffic is handled by a load balancer and 5 endpoints that are represented by web clients. Within the network there are nodes that will act as monitors, such as node s11, and others that will instead perform the simple forwarder function with their respective static routes. Finally, there are several currently empty nodes that we will call

allocation places. Within these nodes, the framework will be able to place network security functions to make sure that the security requirements are working properly within the network. A table is provided below with the definition of each node, its IP address that will be used in the virtual environment, and its functionality within the topology.

Name	IP	Functionality
e1	130.10.0.1	Web servers behind load balancer b1
e2	130.10.0.2	*
e3	130.10.0.*	*
e4	40.40.41.*	Web Client
e5	40.40.41.*	Web Client
e6	88.80.84.*	Web Client
e7	192.168.1.*	Web Client
e8	192.168.2.*	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

The second and final input that must be provided to the framework is the set of security requirements that the network must have. With this demo, the goal was to have an output topology that contained a large number of VPN Gateways (6), in order to be able to show an example that would come as close as possible to a hypothetical real case of a small-to-medium sized company. In order to be able to achieve a similar topology, the following rules were defined:

Policy	IPSrc	IPDst	pSrc	pDst	tProto	Confidentiality	Intregrity	Untr
Protection	40.40.41.1	130.10.0.1	*	22	ANY	AES-256-CBC	SHA2-256	3
Protection	88.80.84.1	130.10.0.*	*	80	ANY	AES-256-CBC	SHA2-256	33.33.3
Protection	192.168.1.1	130.10.0.1	*	*	ANY	AES-256-CBC	SHA2-256	33.33.3
Protection	40.40.42.1	192.168.2.1	*	*	ANY	AES-256-CBC	SHA2-256	33.33.33

Table 5.2. Security Requirements Definition

- **First Rule**: Web Client e4 must be able to communicate securely with Web Server e1. Traffic originating from e4 can use any port for the transport protocol but the e5 Server must receive data only from port 22. Both UDP and TCP protocols can be used for communications. Finally, node s10 is specified as a non-secure node and through which traffic must pass encrypted.

- **Second Rule**: Web Client e8 must be able to communicate securely with all Web Servers (e1,e2,e3). Traffic originating from e8 can use any port for the transport protocol but the servers must receive data only from port 80. It is possible to use either UDP or TCP protocol for communication. In this case the nodes that are considered non-secure are s10 and s11.
- **Third Rule**: Web Client e7 must be able to communicate securely with Web Server e1. There are no limitations on ports for incoming and outgoing traffic, and any fourth-level protocol can be used. The unsecured nodes, as with the second rule, are s10 and s11.
- **Fourth Rule**:The Web Client e5 must be able to communicate securely with the Web Client e8. As with the third rule, there are no limitations on the ports and transport protocol to be used. The nodes considered insecure for this rule are s12 and s13.

5.2 Output

By providing the previously defined inputs, the framework will look for a solution that not only satisfies all the rules, but also employs the least amount of resources necessary to guarantee these properties. Verefoo will then solve a problem defined as MaxSMT (Maximum Satisfiability Modulo Theories). In the specific case of this demo, the result produced in output will be as follows:

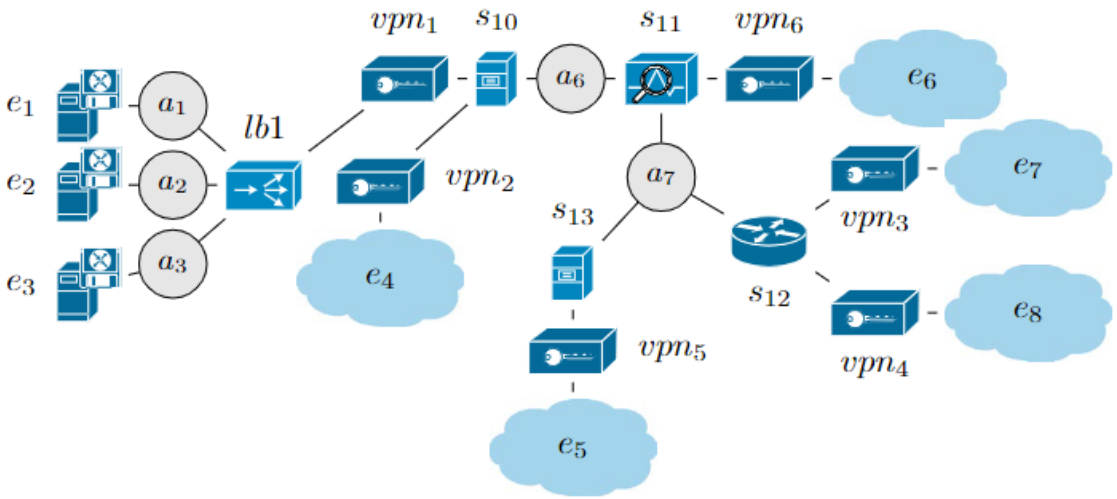

Figure 5.2. Verefoo Output

As could be imagined, several VPN Gateways were allocated in place of the various Allocation Places defined in the input provided to Verefoo. Given this should be the solution to the problem defined earlier, we will try to test inside the virtual environment the functionality of the various VPN tunnels to make sure that the framework works properly. Before proceeding, however, it is also important to analyze the various elements that Verefoo has produced. In fact, the framework not

only allocates the gateways in the correct places, but also provides an automatic configuration to use. In this specific case, the configuration is as follows:

#	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

Chapter 6

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 6.1. Esempio di tabella con tre colonne senza linee verticali.

Policy	IPSrc	IPDst	pSrc	pDst	tProto	Confidentiality	Intregrity	Untrusted nc
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 6.2. 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 6.3. 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 6.4. 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 6.5. Esempio di tabella con tre colonne senza linee verticali.

Chapter 7

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 2, 2024. [Online]. Available: https://www.vmware.com/topics/glossary/content/virtual-machine.html

[7] Docker, Inc. (2024) What is a container? Accessed on February 2, 2024. [Online]. Available: https://www.docker.com/resources/what-container/

[8] (2024) What is a hypervisor? VMware, Inc. Accessed on February 2, 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 2, 2024. [Online]. Available: https://docs.docker.com/compose/compose-file/05-services/