



Master of Science in Computer Engineering

Tesi di Laurea Magistrale

Deployment automatico di funzioni di sicurezza di rete con Docker Compose

Supervisors

prof. Fulvio Valenza

prof. Riccardo Sisto

dott. Daniele Bringhenti

Candidato

Benito MARRA

ANNO ACCADEMICO 2023-2024

Sommario

Nel mondo moderno uno dei problemi emergenti più importanti è l'automatizzazione della sicurezza nelle reti. Diverse tecnologie sono state sviluppate con l'intento di semplificare tutti i processi di sicurezza all'interno delle aziende. In questo ambito due delle scoperte più significative sono state le tecnologie denominate Network Function Virtualization (NFV) e Software Defined Network (SDN). Attraverso queste è possibile definire delle reti dentro le quali alcuni nodi non sono definiti da un hardware specifico ma vengono virtualizzati, permettendo di svolgere delle funzioni tramite l'ausilio di software. All'interno di questo ambiente è stato progettato VEREFOO(VERified REFinement and Optimized Orchestration) un framework capace di allocare e configurare automaticamente Network Security Functions (NSF) per soddisfare dei requisiti di sicurezza definiti in input dall'utente. Le prime versioni del framework erano in grado di poter allocare in una iterazione univocamente un solo tipo di Network Security Functions, generalmente dei Firewall o dei VPN Gateway. Il lavoro presente in questo documento descrive e promuove una versione aggiornata del framework, che riesce in un'unica iterazione ad allocare contemporaneamente entrambe le NSF descritte. Inoltre, descrive il processo di design, sviluppo e implementazione di due Demo prodotte per mostrare le potenzialità del framework. La prima riguarda principalmente la versione primitiva di Verefoo capace di verificare e soddisfare i requisiti di sicurezza di protezione delle comunicazione tramite l'istanziazione di VPN Gateway, la seconda dimostra il funzionamento della nuova versione del framework prodotta tramite una topologia di rete che necessita contemporaneamente dell'utilizzo di Firewall per scartare determinati pacchetti in transito all'interno della rete e di VPN Gateway per garantire la comunicazione sicura tramite l'autenticazione e la cifratura dei pacchetti.

Ringraziamenti

Acknowledgement (optional)

Indice

Elenco delle figure	7
Elenco delle tabelle	9
Listings	10
1 Introduzione	12
1.1 Obiettivo della Tesi	12
1.2 Descrizione della tesi	12
2 Network Security Automation e Verefoo	14
2.1 Service Function Chain	14
2.2 Verefoo	16
2.2.1 Introduzione	16
2.2.2 Descrizione del modello	16
2.3 Definizione ed esempio di Grafo di servizio e di allocazione	19
2.4 Definizione ed esempi delle proprietà di sicurezza	23
2.4.1 Reachability Requirements	24
2.4.2 Isolation Requirements	26
2.4.3 Protection Requirements	28
3 Docker	32
3.1 Differenze fra Container e Macchine Virtuali	32
3.2 Docker e la sua architettura	35
3.3 Il tool Compose	36
3.3.1 Introduzione	36
3.3.2 YAML	37
3.3.3 Services e Networking	38

4 Obiettivi della tesi	41
5 Correzione, sviluppo e ottimizzazione Demo A	43
5.1 Introduzione alla Demo	43
5.2 Sviluppo Installer	44
5.3 Implementazione	48
5.4 Output	51
5.5 Verifiche e Test	54
6 Merge Funzioni di Verefoo	58
6.1 Versione Ibrida con doppio Jar File	59
6.2 Versioni Firewall-to-VPN e VPN-to-Firewall	61
6.3 Trasposizione Teorica in Codice: Analisi dell'Implementazione	64
6.3.1 Definizione del Verefoo Serializer	64
6.3.2 Definizione del Verefoo Proxy	68
7 Design, progetto e sviluppo Demo B	71
7.1 Introduzione	71
7.2 Implementazione	72
7.3 Output	79
7.3.1 Topologia di Rete	79
7.3.2 Configurazioni Network Security Functions	80
7.3.3 Configurazioni Strongswan	81
7.4 Verifiche e Test	82
8 Conclusioni e Lavori Futuri	86
Bibliografia	88

Elenco delle figure

2.1	Esempio di Service Function Chain	15
2.2	Architettura di VEREFOO [1]	17
2.3	Esempi di Service (in alto) e Allocation(in basso) Graphs [1]	19
2.4	Esempio di un Service Graph	20
2.5	Grafo di allocazione per requisito di raggiungibilità	25
2.6	Output grafo di esempio per requisito di raggiungibilità	26
2.7	Output grafo di esempio per requisito di isolamento	27
2.8	Grafo di allocazione d'esempio per requisiti di protezione	30
2.9	Output grafo di esempio per requisito di protezione	30
3.1	Architettura Virtual Machine e Containers	33
3.2	Architettura Docker	35
5.1	Service Graph Demo A	48
5.2	Verefoo Output Demo A	51
5.3	VpnGateway4	54
5.4	VpnGateway5	54
5.5	Verifica Tunnel VPN	55
5.6	Tunnel stabilito	56
5.7	Prova di correttezza	56
6.1	Definizione esempio caso limite per duplice allocazione	62
6.2	Allocazione parziale del caso limite	62
6.3	Soluzione all'allocazione parziale del caso limite	63
6.4	Soluzione finale caso limite	63
7.1	Grafo di Allocazione della Demo B	72
7.2	Verefoo Output	79
7.3	Verifica pacchetti scartati dal Firewall per rete 20.1.*.*	82

7.4	Verifica pacchetti scartati dal Firewall per rete 20.2.*.*	83
7.5	Setup tcpdump per verifiche di sicurezza	84
7.6	Verifica cifratura dei pacchetti in transito	84

Elenco delle tabelle

5.1	Node definitions and functionalities	49
5.2	Security Requirements Definition DemoA	49
5.3	VPN Gateway 1	52
5.4	VPN Gateway 2	52
5.5	VPN Gateway 3	52
5.6	VPN Gateway 4	52
5.7	VPN Gateway 5	52
5.8	VPN Gateway 6	53
7.1	Definizione nodi della topologia per Demo B	73
7.2	Definizione requisiti di sicurezza della topologia per Demo B	74
7.3	Configurazione dei due VPN gateway Demo B	80
7.4	Configurazione Firewall Demo B	80

Listings

2.1	Definizione nodi del ServiceGraph	2.4	21
2.2	Definizione Allocation Place per Allocation Graph		22
2.3	Definizione di una proprietà di sicurezza generica		23
2.4	Esempio di requisito di raggiungibilità		25
2.5	Esempio di requisito di isolamento		27
2.6	Definizione di una proprietà di protezione		28
2.7	Esempio di requisito di protezione		30
3.1	Esempio definizione di un file YAML		37
3.2	Esempio file di configurazione docker-compose		40
5.1	Installazione packages curl e pv		45
5.2	Installazione java openjdk		46
5.3	Installazione Docker e Docker Compose		46
5.4	Installazione Z3		47
5.5	Definizione di Services dell'ambiente virtuale DemoA		50
6.1	Inizializzazione e normalizzazione input		65
6.2	Calcolo dei Paths e dei profili nell'input		65
6.3	Iterazione di allocazione VPN		66
6.4	Iterazione di allocazione Firewall		67
6.5	Esempio di codice Java		68
7.1	Definizione di Services dell'ambiente virtuale DemoB		76
7.2	Definizione Dockerfile Endpoint		77
7.3	Definizione Dockerfile Firewall		78
7.4	Definizione Dockerfile VPN Gateway		78

Capitolo 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 framework.

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.

Capitolo 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 richieste dall'utente. 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 vari 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:



Figura 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. È 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*, 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 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.

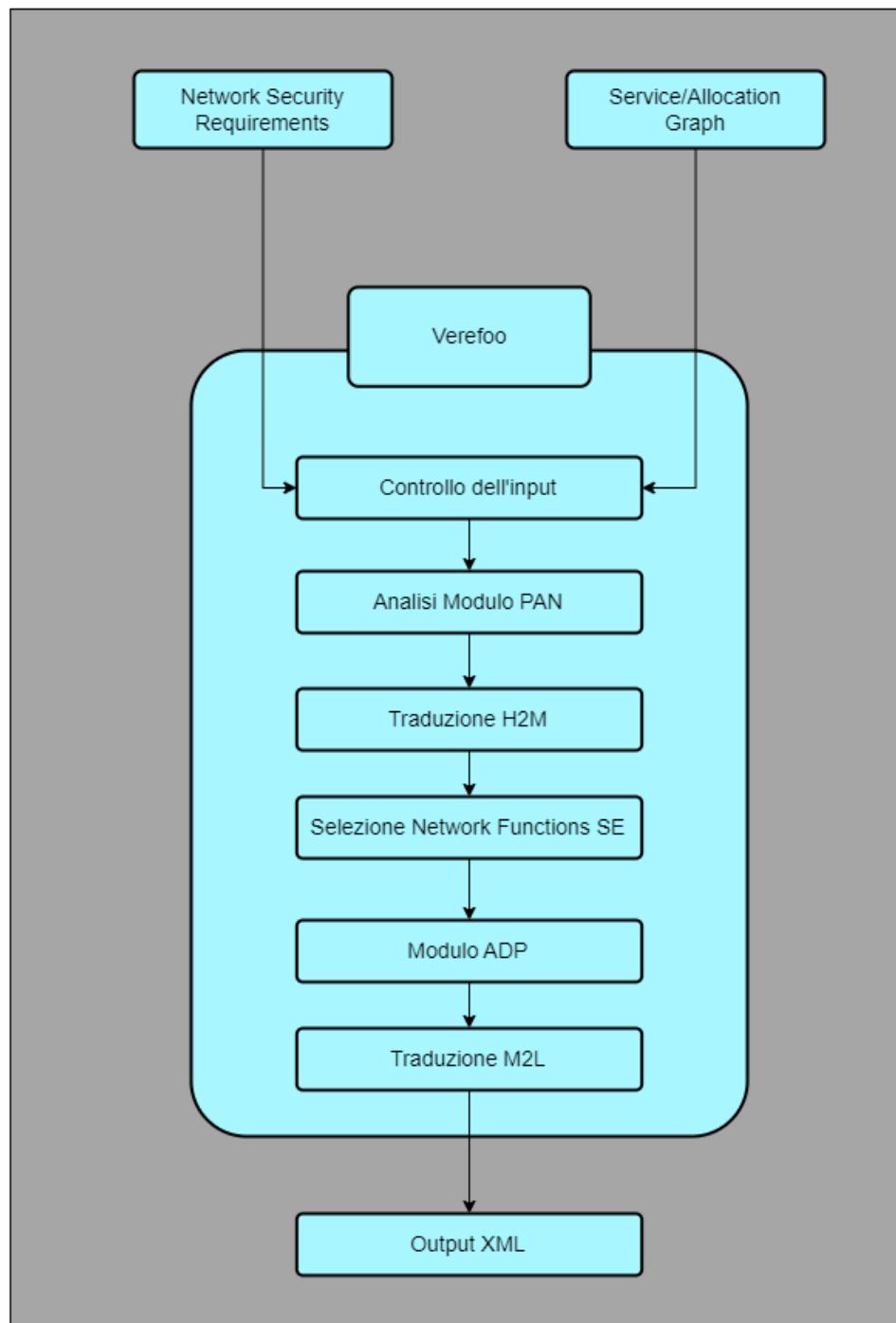


Figura 2.2. Architettura di VEREFOO [1]

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:** All'interno di questa sezione 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 soddisfacibile.
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.

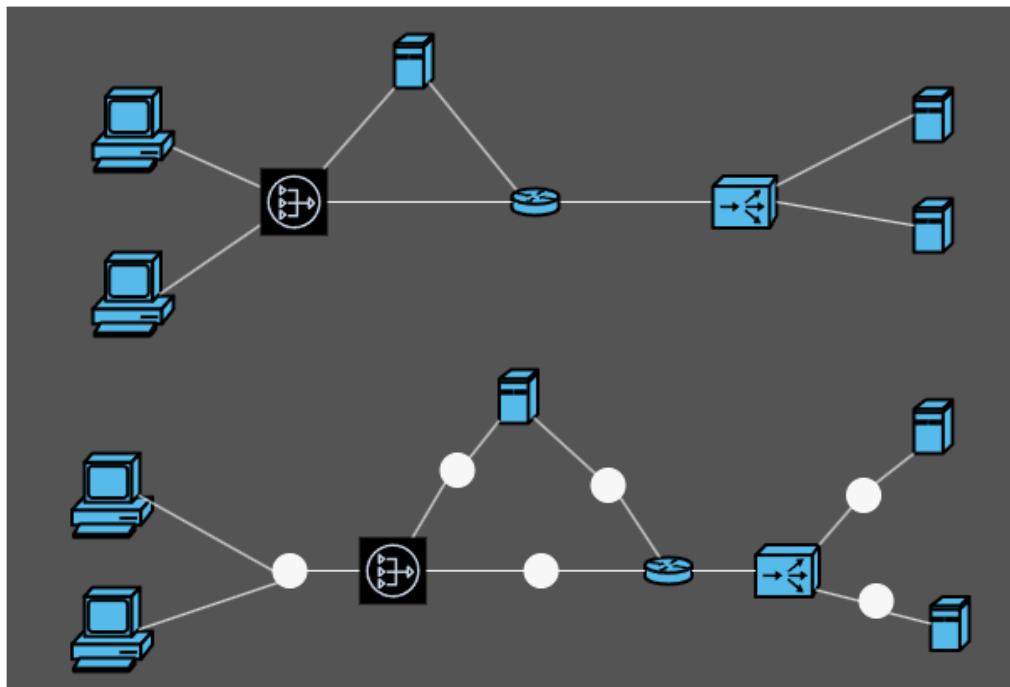


Figura 2.3. Esempi di Service (in alto) e Allocation(in basso) Graphs [1]

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 perché 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 i seguenti elementi:

- **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:** è 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:

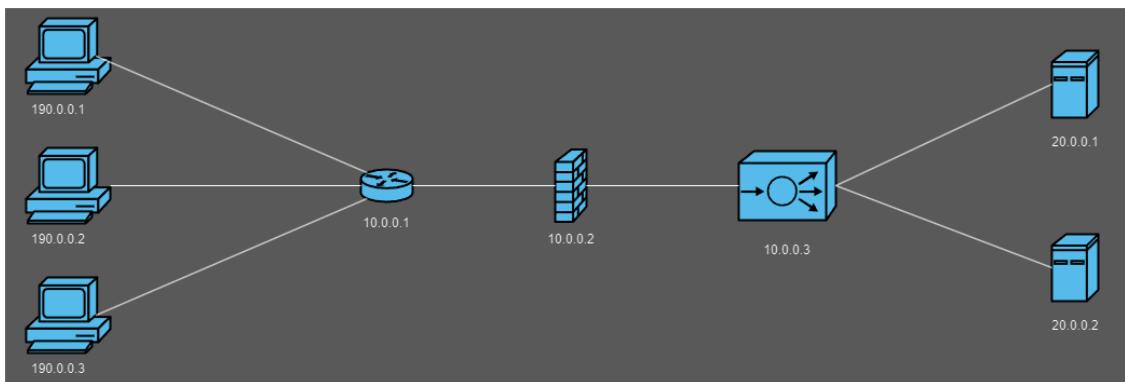


Figura 2.4. Esempio di un Service Graph

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>
                </loadbalancer>
            </configuration>
        </node>
    </graph>
</graphs>

```

```

        <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 rispettivamente 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:

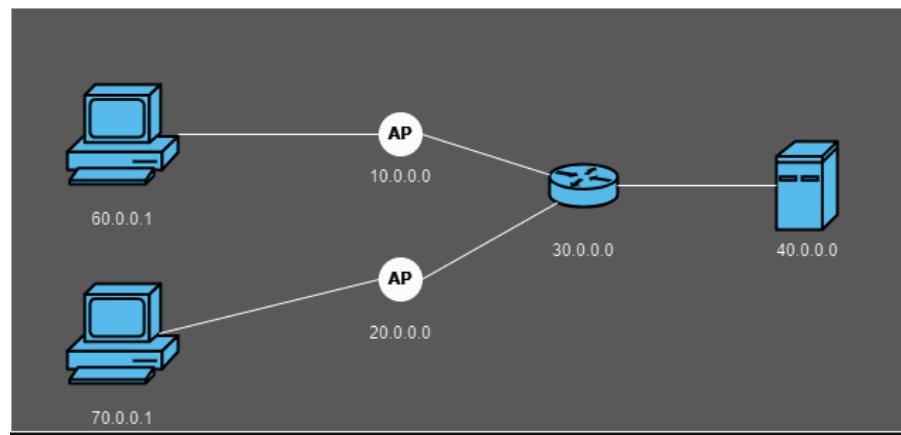


Figura 2.5. Grafo di allocazione per requisito di raggiungibilità

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 aspettato è il seguente:

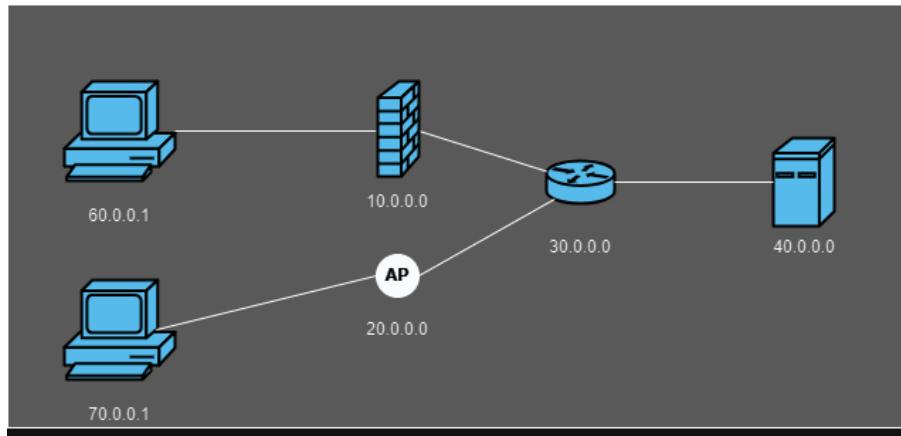


Figura 2.6. Output grafo di esempio per requisito di raggiungibilità

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 poiché 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:

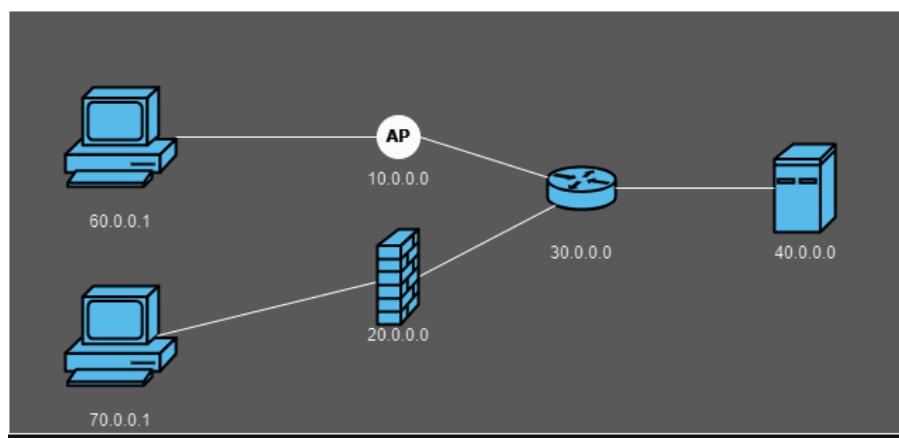


Figura 2.7. Output grafo di esempio per requisito di isolamento

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 secretezza. 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 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 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 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:

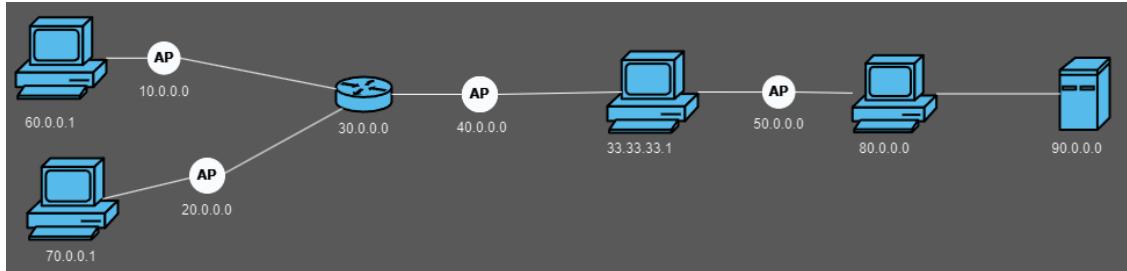


Figura 2.8. Grafo di allocazione d'esempio per requisiti di protezione

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:

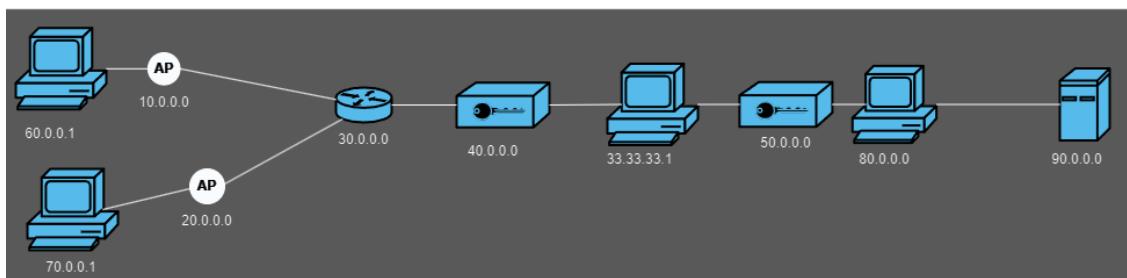


Figura 2.9. Output grafo di esempio per requisito di protezione

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.

Capitolo 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 perché 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:

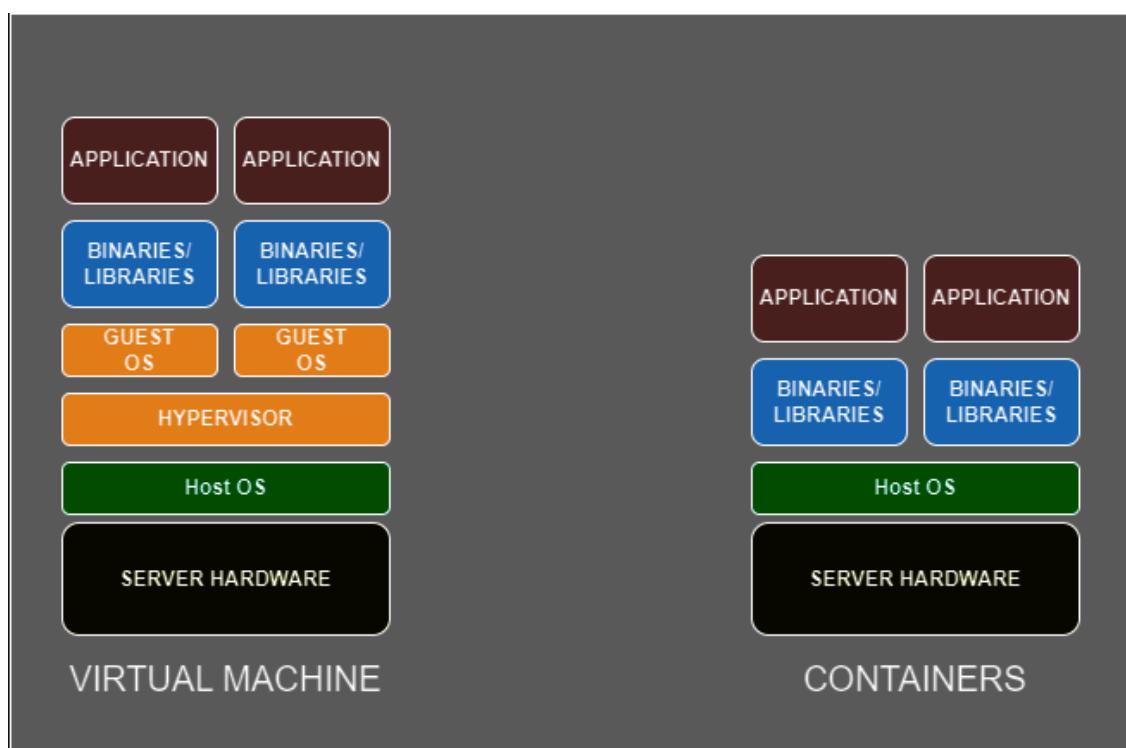


Figura 3.1. Architettura Virtual Machine e Containers

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 perché 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 necessarie 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 specifica 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, perché all'aumentare delle risorse del sistema operativo ospitante corrisponde un aumento della velocità di reazione dei container che orizzontalmente, perché aggiungere una feature corrisponde nell'aggiungere un container al sistema già funzionante.
- **Condivisione 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:

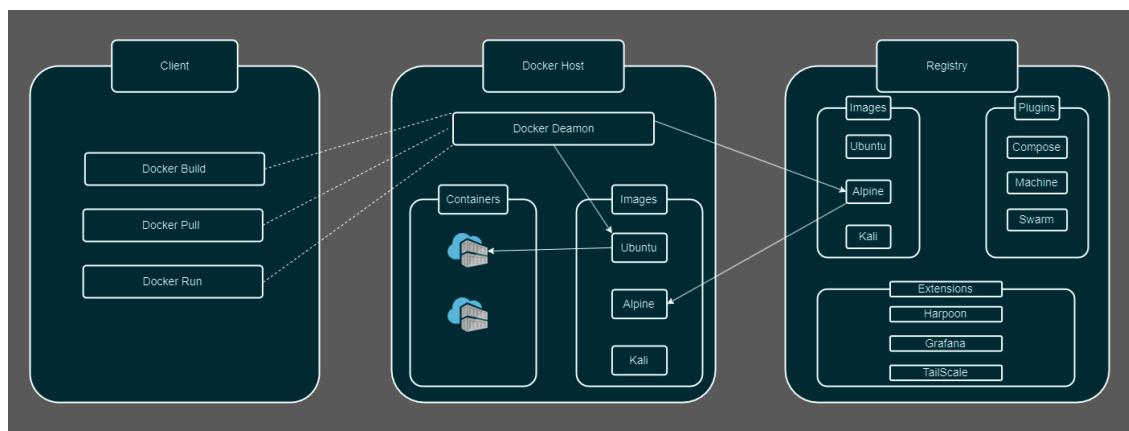


Figura 3.2. Architettura Docker

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 sopraccitati , 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 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[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 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

```

Capitolo 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 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 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.

Capitolo 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 di conseguenza non operativo. 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 opportunatamente.
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 reinderizzando 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 contentente 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${COLOR_RESET}\n"
6 cd /home
7 sudo curl -L0 https://github.com/Z3Prover/z3/releases/download/
8           z3-4.8.15/z3-4.8.15-x64-glibc-2.31.zip
9 sudo unzip z3-4.8.15-x64-glibc-2.31.zip
10 sudo rm z3-4.8.15-x64-glibc-2.31.zip
11 sudo mv z3-4.8.15-x64-glibc-2.31 z3 #rename z3-4.8.15 into z3
12 echo "LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/z3/bin/" | sudo tee
      -a /etc/environment # Care!! >> not > or it will over write
      environment variables
13 sudo echo "Z3=/home/z3/bin/" | sudo tee -a /etc/environment
14 printf "${GREEN}Z3 installed and environment variables setted...
      OK${COLOR_RESET}\n"
15 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:

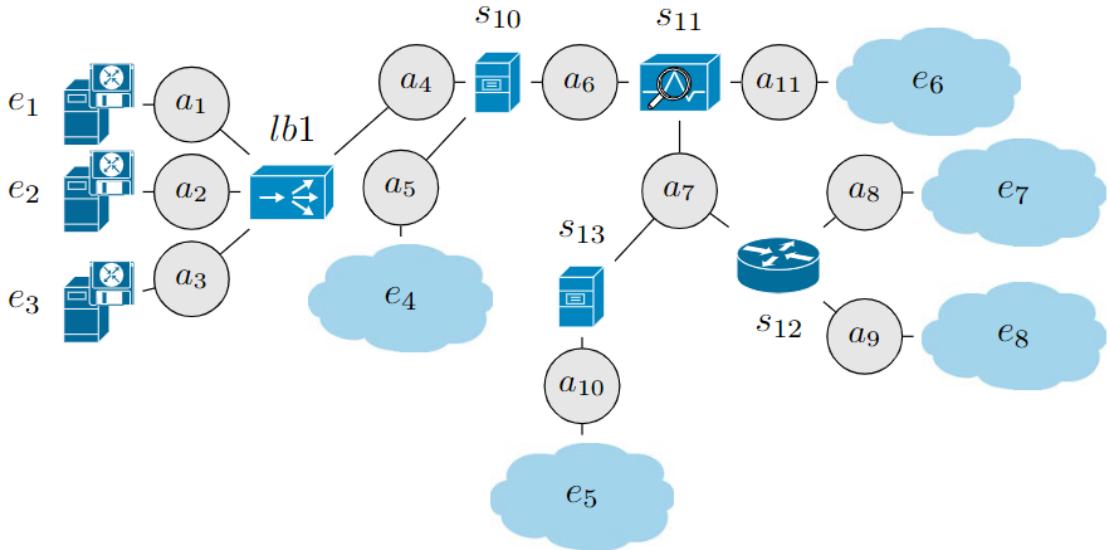


Figura 5.1. Service Graph Demo A

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 WebClients. 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

Tabella 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	Intregrity	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

Tabella 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:

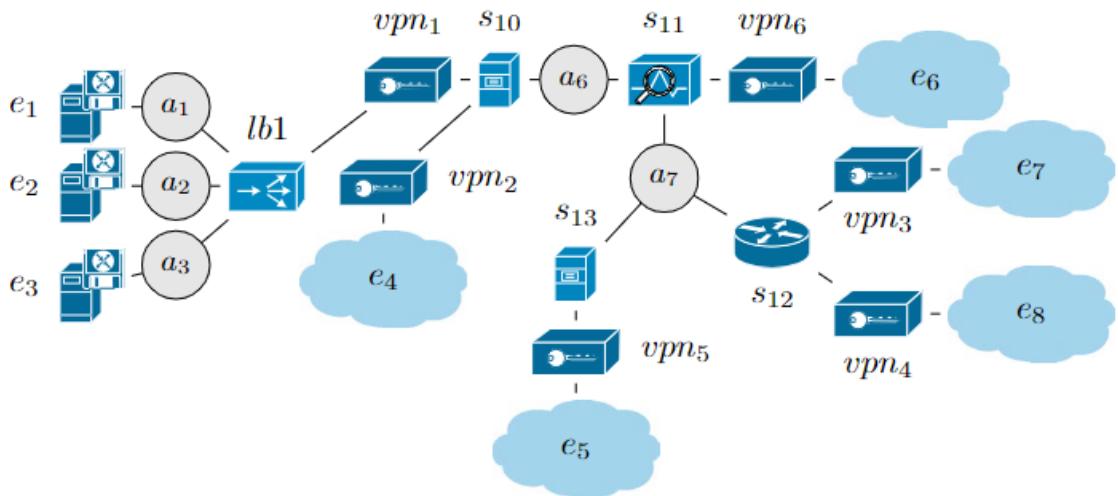


Figura 5.2. Verefoo Output Demo A

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

Tabella 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

Tabella 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

Tabella 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

Tabella 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

Tabella 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

Tabella 5.8. VPN Gateway 6

Prendendo in esempio il VPN Gateway 1 è possibile notare come Verefoo 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, Verefoo 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 Verefoo 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 definita 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:

```

vpn4:~# swanctl -q
loaded certificate from '/etc/swanctl/x509/VpnConfig4Cert.pem'
loaded certificate from '/etc/swanctl/x509ca/strongswanCert.pem'
loaded E025519 key from '/etc/swanctl/ecdsa/vpn4key.pem'
no authorities found, 0 unloaded
no pools found, 0 unloaded
loaded connection 'site-site'
successfully loaded 1 connections, 0 unloaded
vpn4:~# 

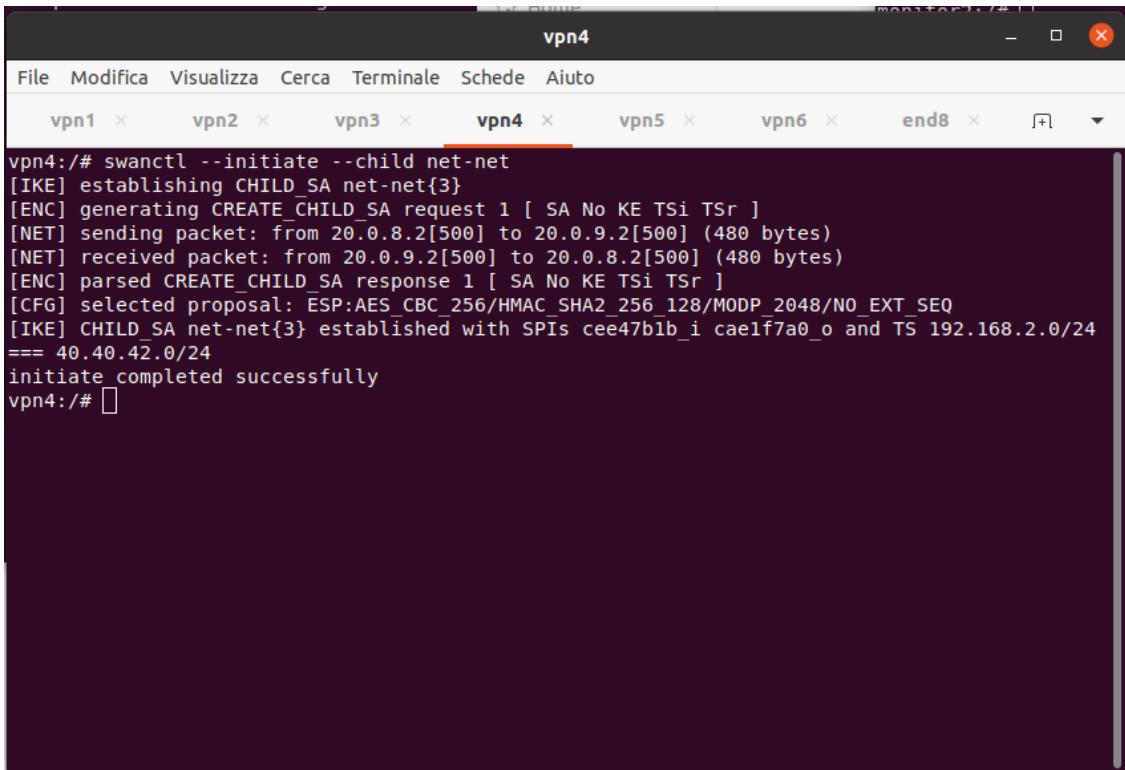
vpn5:~# swanctl -q
loaded certificate from '/etc/swanctl/x509/VpnConfig5Cert.pem'
loaded certificate from '/etc/swanctl/x509ca/strongswanCert.pem'
loaded E025519 key from '/etc/swanctl/ecdsa/vpn5key.pem'
no authorities found, 0 unloaded
no pools found, 0 unloaded
loaded connection 'site-site'
successfully loaded 1 connections, 0 unloaded
vpn5:~# 

```

Figura 5.3. VpnGateway4

Figura 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
File Modifica Visualizza Cerca Terminale Schede Aiuto
vpn1 × vpn2 × vpn3 × vpn4 × vpn5 × vpn6 × end8 ×
vpn4:/# swanctl --initiate --child net-net
[IKE] establishing CHILD_SA net-net{3}
[ENC] generating CREATE_CHILD_SA request 1 [ SA No KE TSi 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 TSi 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 caef7a0_o and TS 192.168.2.0/24
== 40.40.42.0/24
initiate completed successfully
vpn4:/#
```

Figura 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 monitor-ESP 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.

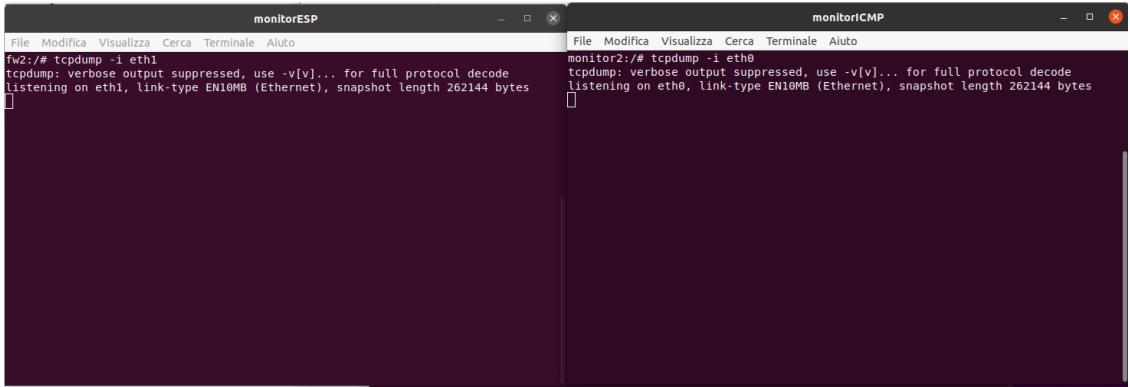


Figura 5.6. Tunnel stabilito

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:

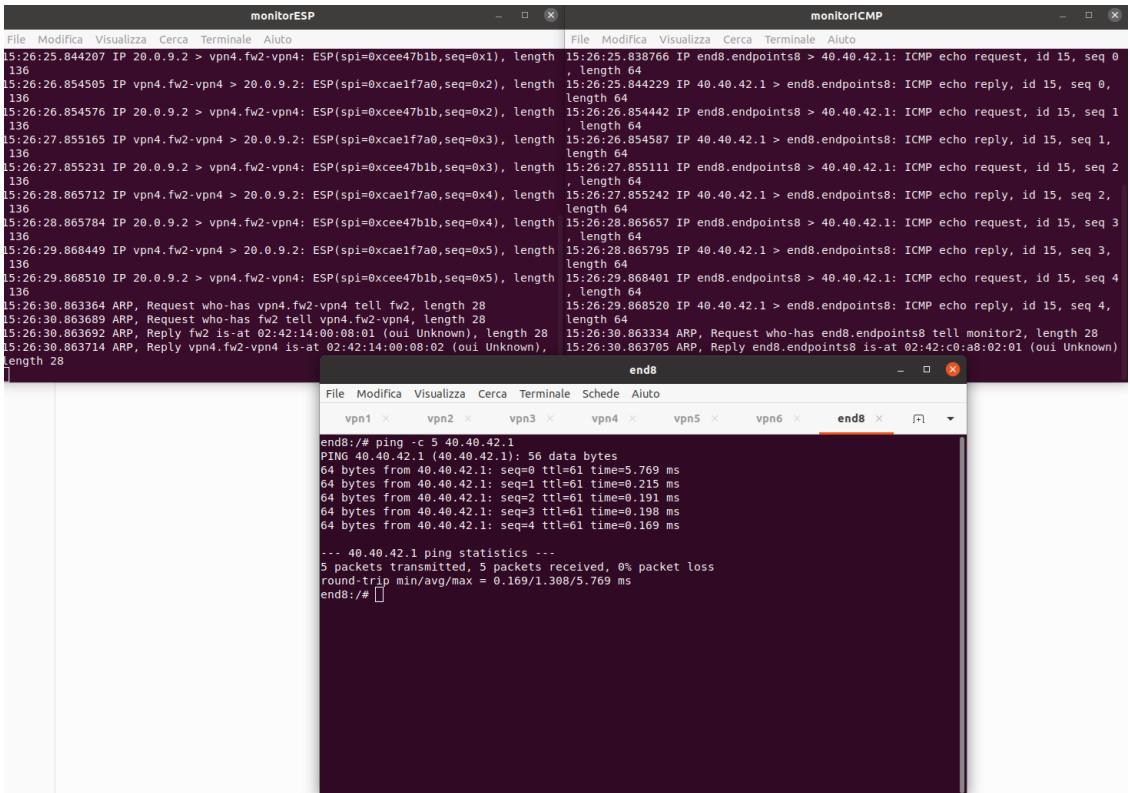


Figura 5.7. Prova di correttezza

Osservando l'ultimo output si può dedurre che i tunnel funzionano correttamente. Per il monitorICMP 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 verefoo 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.

Capitolo 6

Merge Funzioni di Verefoo

All'interno di questo capitolo verrà descritto il processo di ricerca e di sviluppo che è stato svolto per poter raggiungere il secondo macro obiettivo di questa tesi, ovvero la creazione di una nuova versione di Verefoo che sia in grado di istanziare, in un'unica iterazione, sia dei Firewall configurati per agire da Packet Filter che dei VPN Gateway che permettono la comunicazione cifrata fra due endpoints.

Il lavoro prodotto al fine di raggiungere questo obiettivo pur essendo sufficiente è tuttavia non pienamente completo per garantire una versione di Verefoo priva di errori di correttezza delle soluzioni prodotte, ma garantisce una prima versione stabile e parzialmente corretta dell'implementazione. Durante gli studi e lo sviluppo del Merge delle due versioni sono infatti state eseguite numerose prove per garantire un merge che mantenesse quanto più possibile le peculiarità delle versioni utilizzate precedentemente cercando di ridurre al minimo i problemi verificatisi durante l'implementazione.

Nei paragrafi seguenti verrà descritto il processo di sviluppo che è stato effettuato per ottenere la versione più stabile possibile e che minimizzasse gli errori di output possibili. Inizialmente verrà descritta la soluzione ibrida che si è tentato di raggiungere evidenziando le motivazioni del perchè non fosse efficiente, successivamente invece saranno illustrate due soluzioni possibili all'obiettivo da raggiungere elencando le motivazioni che hanno portato alla scelta di una delle due soluzioni. Infine verrà mostrato brevemente con degli snippet di codice Java il funzionamento generale della versione di Verefoo prodotta.

6.1 Versione Ibrida con doppio Jar File

Il primo approccio tentato per raggiungere l'obiettivo è stato quello di utilizzare le due versioni distinte di Verefoo contemporaneamente, creando una soluzione che richiedesse due iterazioni del framework con diversi input.

Scendendo nel dettaglio, questa soluzione richiede lo svolgimento dei seguenti passaggi:

1. **Suddivisione dei security requirements:** Il file di input che un'utente definisce per ottenere una configurazione di rete desiderata tendenzialmente contiene contemporaneamente sia requisiti di isolamento e raggiungibilità che requisiti di protezione. Di conseguenza è necessario suddividere i requisiti in input in due gruppi, il primo contenente i requisiti necessari per la definizione dei Firewall nella topologia(ovvero isolamento e raggiungibilità) ed il secondo contenente i requisiti per la definizione dei VPN Gateway (ovvero protezione). Per costruire quindi il primo input verrà mantenuta la topologia di rete eliminando uno dei due gruppi di requirements dall'input iniziale.
2. **Iterazione della prima versione di Verefoo:** Ottenuto il primo file di input è possibile eseguire il primo jar file contenente la versione di verefoo che alloca univocamente i Firewall o i VPN Gateway. In questa fase è necessario utilizzare il pacchetto curl([5.2](#)) per eseguire una chiamata API di tipo POST passando come argomento della chiamata l'input prodotto al punto precedente. La chiamata restituirà un output nel quale i Firewall o i VPN Gateway saranno allocati(a seconda della versione utilizzata per prima), ove necessario, al posto degli allocation places disponibili mentre tutti gli allocation places aggiuntivi verranno sostituiti da dei semplici forwarder.
3. **Traduzione Output prima versione e creazione del nuovo input:** L'output prodotto nel passaggio precedente non è utilizzabile per la seconda iterazione di Verefoo. Risulta quindi necessario tradurlo in una versione comprensibile da Verefoo. I nodi dove sono state allocate Network Security Functions verranno mantenute nell'nuovo input, assieme all'insieme di nodi definiti precedentemente. Tutti i nodi che erano invece stati trasformati da allocation places a forwarder devono essere ritradotti come allocation places, ed infine il secondo gruppo di requirements deve essere aggiunto alla descrizione del grafo di input.
4. **Iterazione della seconda versione di Verefoo:** Una volta terminata la produzione del secondo input è possibile eseguire il secondo file jar che si occupa dell'allocazione della Network Security Function rimanente in base alla scelta effettuata al punto 2. Come fatto precedentemente viene quindi eseguita una chiamata API di tipo POST tramite il pacchetto curl passando come argomento della chiamata il secondo input prodotto. Il risultato definito in output rappresenta la topologia di rete che l'utente ha richiesto soddisfacendo contemporaneamente sia i requisiti di raggiungibilità ed isolamento che i requisiti di protezione.

Seppur teoricamente questa soluzione sembra un buon tentativo primitivo di risolvere il problema delle versioni multiple con Verefoo, essa presenta numerose limitazioni e difficoltà che ne hanno reso impossibile la realizzazione.

Uno degli ostacoli principali è stata la traduzione del primo output nel secondo input descritta al punto 3. Infatti Verefoo non fornisce un metodo per comprendere in maniera univoca quali sono i Forwarder presenti nella rete precedentemente all'allocazione delle Network Security Functions rispetto a quelli che vengono tradotti successivamente al non utilizzo di un allocation places. Questo rende praticamente impossibile creare un software che traduca automaticamente l'output fornito dalla prima iterazione in un input idoneo per la seconda iterazione, non rendendo questa soluzione automatizzabile. Inoltre, delegare all'utente la configurazione del secondo file manualmente reintrodurrebbe la variabile di errore umano per cui Verefoo è stato sviluppato vanificando l'intero lavoro.

In seconda battuta è stato analizzato quale delle due versioni del framework andasse utilizzata per la prima chiamata API(step 2) e quale per la seconda(step 4). Sebbene la scelta possa sembrare indifferente, a livello implementativo potrebbe portare a delle computazioni più complesse a causa delle operazioni che le NSFs svolgono all'interno della topologia. I firewall per agire da packet filter devono infatti ispezionare i pacchetti che transitano attraverso il nodo e questo va in diretta contrapposizione con i tunnel VPN che vengono istanziati. Se infatti viene allocato un tunnel VPN nel quale path è presente un firewall questo non sarà in grado di poter ispezionare il pacchetto essendo completamente incapsulato dal protocollo ESP di IPsec.

Come ultima criticità di questa soluzione, che ha portato la ricerca di una soluzione alternativa, è stata la vera e propria incompatibilità delle due versioni del framework. Scendendo più nel dettaglio, è stato notato come pur traducendo momentaneamente manualmente il primo output in un input elegibile per la seconda versione di Verefoo quest'ultimo continuava a non trovare una soluzione ottimale o in alcuni casi addirittura a dare errore di computazione restituendo un "*503: Internal Server Error*" come risposta alla POST effettuata. Indagando è emerso che la versione che si occupava di Firewall non poteva in alcun modo accettare un input contenente dei Gateway VPN in quanto questi non erano stati definiti all'interno del codice del file jar e di conseguenza non venivano riconosciuti come Network Security Functions. Viceversa anche allocando prima i Firewall e successivamente i Gateway VPN la soluzione non veniva comunque computata in quanto, seppur presente la definizione dei Firewall all'interno del file Jar, i vincoli utilizzati per definire il problema MaxSMT riguardanti i Firewall non erano aggiornati allo stato più recente della ricerca, andando in conflitto con i nuovi security requirements in input.

Di conseguenza lo sviluppo di questa soluzione si è interrotto privilegiando una soluzione che prevedesse un'unica iterazione per l'allocazione di entrambe le Network Security Function.

6.2 Versioni Firewall-to-VPN e VPN-to-Firewall

Dovendo sviluppare una soluzione univoca una delle prime scelte effettuate è stata la decisione di cosa dover allocare in maniera prioritaria fra Firewall e VPN Gateway. Inizialmente si è preferito optare per una soluzione nella quale i Firewall venissero allocati prima dei Gateway VPN, in quanto la versione di Verefoo contentente l'allocazione e la configurazione dei Firewall era considerabile come la versione più aggiornata e testata rispetto alle altre e quindi un'ottima base di partenza per il lavoro di sintesi delle versioni.

Tuttavia questa scelta è stata celermente scartata in quanto avere una soluzione intermedia nella quale sono già presenti i Firewall rispetto alle VPN porta a due problemi principali:

1. **Firewall all'interno dei Tunnel VPN:** Potrebbe capitare che il firewall non riesca a svolgere la sua funzione allocandolo precedentemente ai gateway VPN. Infatti è possibile che il framework inserisca nel path di alcuni pacchetti che devono essere scartati dal firewall dei tunnel VPN che ovviamente, per garantire l'integrità dei dati, cifrano il pacchetto e modificheranno il proprio header inserendo come IPsrc e IPdst i due estremi del VPN Gateway. Se tali pacchetti dovessero passare da un firewall configurato precedentemente per bloccare determinati pacchetti provenienti con un IP diverso da quello di encapsulamento potrebbe capitare che il pacchetto attraversi il tunnel senza venir scartato.
2. **Maggiore calcolo computazionale:** Allocare precedentemente un Firewall richiede un calcolo medio computazionale maggiore, perché nella successiva allocazione dei Gateway VPN il framework dovrà trovare, al minimo, 2 nodi di tipo allocation places disponibili per poter istanziare un tunnel. Viceversa invece istanziano precedentemente un tunnel VPN servirà solo 1 nodo per l'allocazione del firewall, rendendo la ricerca nello spazio delle soluzioni possibili più rapida e computazionalmente leggera.

Considerando queste due motivazioni è stato quindi deciso di utilizzare univocamente soluzioni del tipo VPN-to-Firewall rispetto a quelle Firewall-to-VPN.

Tuttavia, per futuri lavori, le soluzioni Firewall-to-VPN non sono da escludere a priori in quanto è possibile realizzarle senza che entrino in conflitto con i Firewall, ma per garantire ciò è fondamentale che le configurazioni del firewall vengano sovra-scritte tenendo conto di eventuali tunnel VPN che il framework potrebbe allocare, modificando le regole del firewall con i nuovi indirizzi IP del tunnel. Per fare ciò tuttavia è necessario modificare diverse features di Verefoo che non sono state argomento di approfondimento nello svolgimento di questo lavoro di tesi e perciò questa possibilità viene delegata al prossimo studio di implementazioni di soluzioni miste. Un altro elemento che è stato materia di studio all'interno dello sviluppo di questa soluzione è stata l'analisi della disponibilità dei singoli allocation places per la ricerca di soluzioni ottimali. Più precisamente, essendo la collocazione delle network security functions suddivisa per ogni funzione di rete è possibile trovarsi nella situazione in cui un allocation places che sarebbe stato un candidato ideale per la

collocamento di una funzione di sicurezza viene occupato precedentemente da un'altra funzione di sicurezza. Questa situazione potrebbe portare quindi non solo a delle soluzioni che si possono definire sub-ottimali ma anche a delle situazioni limite in cui gli allocation places non sono sufficienti rispetto al numero di Network Security Functions che devono essere istanziate nella topologia di rete. Di seguito viene rappresentato un esempio molto semplice di caso limite che mostra questa circostanza:

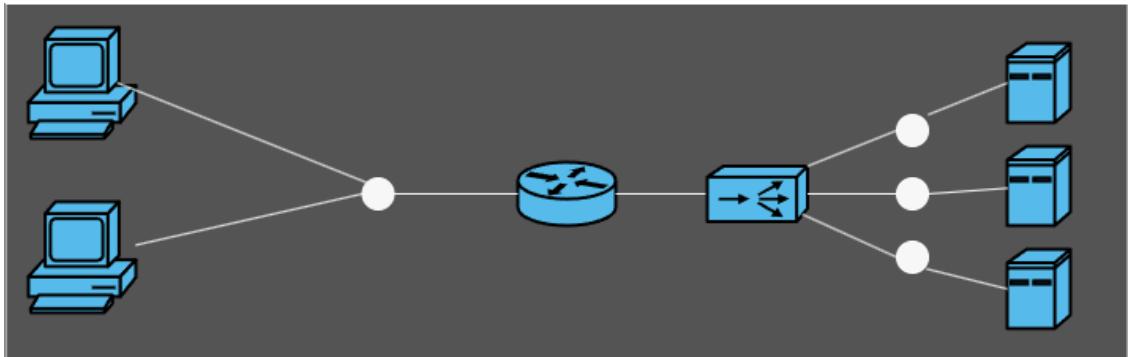


Figura 6.1. Definizione esempio caso limite per duplice allocazione

All'interno di questo esempio l'utente vuole creare una rete nella quale solo uno dei due Web Client a sinistra possa comunicare con i Web Server a destra mentre l'altro Web Client, che possiamo definire "client zombie" deve essere completamente isolato dal resto della topologia di rete. Inoltre viene richiesto che il collegamento fra uno dei tre server ed il Web Client sia protetto, e quindi il traffico che transita per quello specifico percorso deve venire cifrato da un endpoint all'altro.

In questo caso allocando prima i VPN gateway per garantire i requisiti di protezione il risultato che ci si aspetta è il seguente:

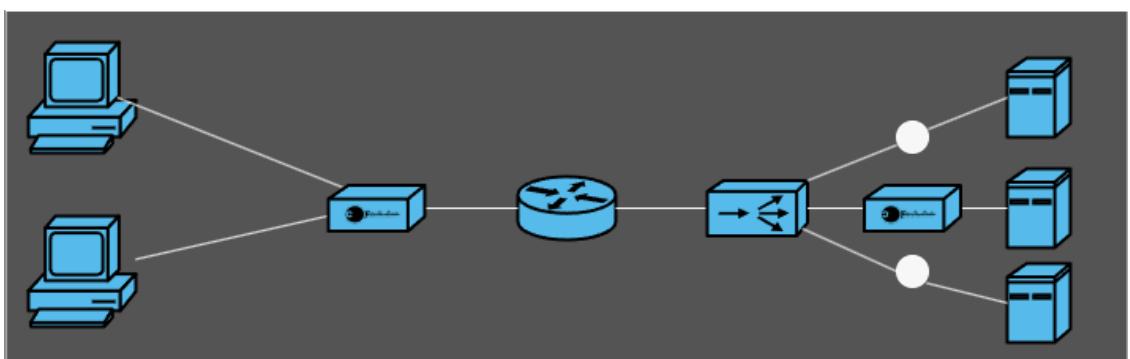


Figura 6.2. Allocazione parziale del caso limite

I gateway, seppur istanziati correttamente, fanno emergere chiaramente un grande problema, ovvero che al fine di poter rispettare anche i requisiti di isolamento il numero di allocation point definiti inizialmente non è sufficiente all'interno dell'input fornito dall'utente. Inoltre se fossero stati allocati i Firewall precedentemente

alle VPN si sarebbe arrivati ad una situazione speculare nel momento in cui sarebbe stato necessario allocare i gateway.

Al fine di proporre una soluzione valida a questo problema, è stato deciso di inserire un ulteriore passo durante l'allocazione delle prime Network Security Functions, ovvero di istanziare, contestualmente alla funzione corrente, altri N allocation places, uno per ogni nodo adiacente alla singola Network Function istanziata nell'allocation place scelto. Riprendendo l'esempio precedente l'output parziale prodotto dal framework sarà il seguente:

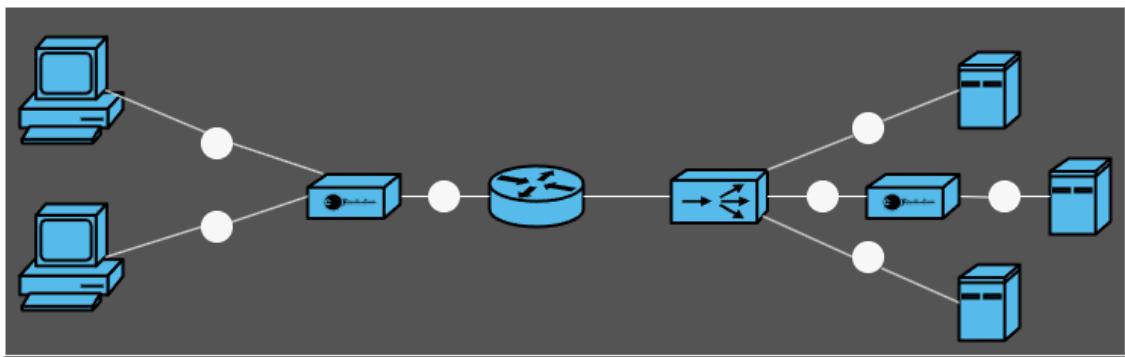


Figura 6.3. Soluzione all'allocazione parziale del caso limite

Attraverso l'introduzione di ulteriori allocation places la topologia risulterà più complessa, ma allo stesso tempo verrà concessa a Verefoo la possibilità di avere uno spazio di soluzioni possibili per risolvere l'input fornito maggiore. Di conseguenza la soluzione ottimale che verrà prodotta sarà non solo possibile nel caso limite, ma anche tendenzialmente migliore delle precedenti nei casi più comuni, in quanto sarà possibile applicare le restrizioni di sicurezza in più punti della rete intervenendo prima sui pacchetti che necessitano di essere scartati ma anche posizionando i vari gateway VPN in posizioni più strategicamente valide all'interno della topologia. Infine viene fornito, nella figura sottostante, la soluzione finale che Verefoo produrrà in output per il caso descrito in questo paragrafo:

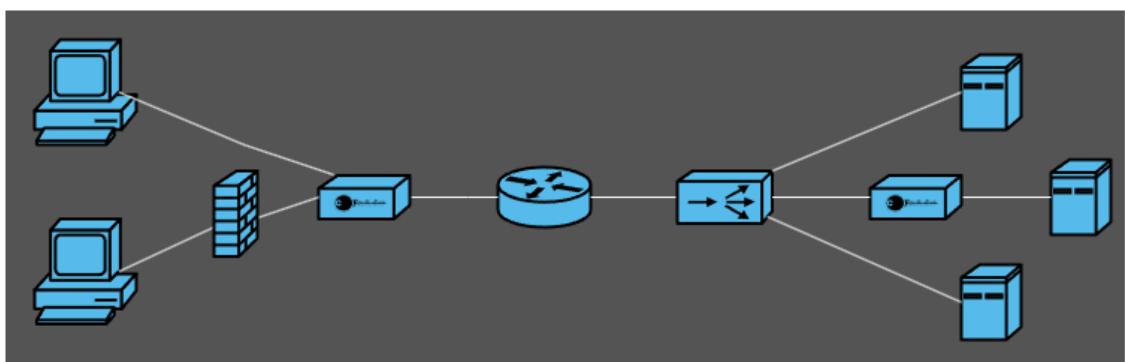


Figura 6.4. Soluzione finale caso limite

6.3 Trasposizione Teorica in Codice: Analisi dell'Implementazione

Definite le implementazioni desiderate all'interno dell'analisi delle possibili soluzioni, il framework è stato modificato adeguatamente per poter effettuare, come descritto prima, in un'unica iterazione l'allocazione dei VPN Gateway necessari per garantire i requisiti di protezione e dei Firewall per garantire i requisiti di isolamento e raggiungibilità in questo ordine. All'interno di questo paragrafo, verranno mostrati i punti salienti del codice al più alto livello effettuati, ovvero le classi di Verefoo che agiscono come Proxy e come Serializer. L'implementazione di più basso livello invece, allo stato precedente alle modifiche effettuate da questo lavoro, era già stato definito tramite vincoli descritti nella comunicazione con il Theorem Prover Z3 e tramite la definizione delle classi delle varie Network Security Functions. Il linguaggio di sviluppo del framework è Java, pertanto il codice che verrà descritto di seguito utilizzerà tale linguaggio di programmazione.

Prima di dare uno sguardo alle implementazioni delle classi è bene descrivere a grandi linee il flow chart del framework. Ad ogni esecuzione di una chiamata API Verefoo esegue le seguenti istruzioni:

1. **Lettura Input e Traduzione:** Nelle primissime fasi tramite un marshaller vengono effettuate le traduzioni dell'input da un file di testo ad un file XML comprensibile e memorizzabile all'interno della memoria di Verefoo. In questa fase viene quindi effettuata una validazione dell'input nelle sue varie componenti.
2. **Invocazione del Serializer:** Definito l'input valido viene creata una istanza della classe del Serializer che è una classe che si occupa di "incapsulare" tutte le attività di Verefoo. Viene definito serializer perché nel caso di molti Service Graph mandati in input esegue in maniera serializzata tutte le procedure necessarie per far lavorare Z3 correttamente per ogni grafo.
3. **Invocazione del Normalizer:** All'interno del Serializer la prima operazione che viene effettuata è l'invocazione del Normalizer che si occupa di produrre, dato l'input fornito dall'utente e reso comprensibile con il marshaller, di produrre una versione normalizzata del grafo.
4. **Invocazione del Proxy:** Dopo aver creato un input normalizzato viene chiamato il Proxy che si occupa di svolgere il calcolo del problema MaxSMT per produrre una soluzione.
5. **Traduzione output:** Infine come ultimo passo, il framework traduce l'output prodotto all'interno del proxy sotto forma di file XML per renderlo più comprensibile all'utente.

6.3.1 Definizione del Verefoo Serializer

Le modifiche più significative effettuate all'interno del work flow di verefoo sono state eseguite all'interno dei file VerefooSerializer e VerefooNormalizer. Di seguito vengono forniti snippet di codice del Serializer:

Listing 6.1. Inizializzazione e normalizzazione input

```
public VerefooSerializer(NFV root) {

    int flag=0;
    this.nfv = root;
    AllocationGraphGenerator agg = new AllocationGraphGenerator(root);
    root = agg.getAllocationGraph();
    VerefooNormalizer norm = new VerefooNormalizer(root);
    root = norm.getRoot();
    ...
}
```

All'interno di questa prima fase il parametro in ingresso che viene fornito è un NFV (2.1). Come descritto precedentemente è impossibile utilizzare l'NFV all'interno di Verefoo, di conseguenza viene istanziato un AllocationGraphGenerator che si occupa, dato un input NFV in partenza, di generare un allocation graph. Successivamente questo grafo viene normalizzato all'interno della classe VerefooNormalizer ed è finalmente pronto ad essere utilizzato per calcolare l'output.

Listing 6.2. Calcolo dei Paths e dei profili nell'input

```
...
List<Path> paths = null;
if (root.getNetworkForwardingPaths() != null)
    paths = root.getNetworkForwardingPaths().getPath();
for (Graph g : root.getGraphs().getGraph()) {
    if (root.getProfile()==null) {
        ProfileType pt = new ProfileType();
        pt.setName(ProfileNameType.HYBRID);
        root.setProfile(pt);
    }
}
...
```

Prima di eseguire l'algoritmo di calcolo del problema MaxSMT posto dall'utente è necessario eseguire alcuni step di configurazione dopo l'inizializzazione dell'input. In questo caso specifico, se i NetworkForwardingPaths esistono, essi vengono estratti dalla root ed inseriti in una lista che servirà successivamente come parametro al VerefooProxy.

Inoltre, potendo inviare all'interno dell'NFV più di un grafo, viene controllato il profilo descritto in ognuno, e per ogni profilo non esistente viene impostato come profilo *HYBRID*, che rappresenta il tipo di soluzione che viene trovata all'interno di questa versione del framework. In questo modo ogni utente potrà comprendere subito quali file di output sono stati generati da questa versione rispetto agli altri.

Listing 6.3. Iterazione di allocazione VPN

```

...
List<Property> propProtection =
    root.getPropertyDefinition().getProperty().stream()
    .filter(p -> p.getGraph() == g.getId() &&
        p.getName().name().equals("PROTECTION_PROPERTY"))
    .collect(Collectors.toList());
if (propProtection.size() != 0) {
    flag=1;
    VerefooProxy test = new VerefooProxy(g, root.getHosts(),
        root.getConnections(), root.getConstraints(),
        propProtection, paths, root.getProfile().getName().name()
        , "VPN");

    long beginAll = System.currentTimeMillis();
    VerificationResult res = test.checkNFFGProperty();
    long endAll = System.currentTimeMillis();
    resultChecker = endAll - beginAll;
    time = (int) res.getTime();

    if (res.result != Status.UNSATISFIABLE && res.result !=
        Status.UNKNOWN) {
        Translator t = new Translator(res.model.toString(), root, g,
            test.getAllocationNodes(), test.getTrafficFlowsMap(), "VPN");
        z3Model = res.model.toString();
        t.setNormalizer(norm);
        result = t.convert();
        root = result;
        sat = true;
    } else {
        sat = false;
        result = root;
    }
    root.getPropertyDefinition().getProperty().stream().filter(p ->
        p.getGraph() == g.getId())
        .forEach(p -> p.setIsSat(res.result != Status.UNSATISFIABLE));
}
...

```

Terminate le configurazioni necessarie per istanziare il Proxy, dallo stream delle proprietà di sicurezza (2.4) descritte nell'NFV viene estratto tramite un filtro una lista formata univocamente dalle proprietà di protezione, cioè quelle responsabili dell'allocazione dei VPN gateway. Se la lista contiene almeno una proprietà di sicurezza dovranno essere allocati almeno 2 gateway per poterla soddisfare, di conseguenza viene istanziato il VerefooProxy che è il cuore dell'algoritmo di verefoo, passando fra i vari parametri il grafo, gli host e le connessioni, le proprietà ricavate precedentemente e i paths possibili per soddisfarle. Viene inoltre definito il nuovo nome del profilo(VPN). Completata la computazione della soluzione all'interno del VerefooProxy l'output viene verificato per controllare se tutte le proprietà sono

effettivamente ancora soddisfatte. Se l'output è corretto viene quindi definito un Translator che si occupa di tradurre l'output parziale in un formato valido per la computazione delle altre proprietà rimaste da verificare. In questo caso la variabile di soddisfacibilità viene posta a *true* così che si possa procedere anche all'allocazione successiva. Viceversa in caso di un risultato negativo il risultato viene posto uguale alla root iniziale e la variabile di soddisfacibilità viene posta a *false*.

Listing 6.4. Iterazione di allocazione Firewall

```

...
List<Property> prop =
    root.getPropertyDefinition().getProperty().stream()
    .filter(p -> p.getGraph() == g.getId() &&
    (p.getName().name().equals("ISOLATION_PROPERTY")
    || p.getName().name().equals("REACHABILITY_PROPERTY")))
    .collect(Collectors.toList());

if(propProtection.size()!=0){
    for(Property p:propProtection){
        prop.add(createProperty(p));
    }
}

if(prop.size()!=0 && sat)
{
    ServiceGraph serviceGraph= new ServiceGraph(root);
    root=serviceGraph.addAP();

VerefooProxy test = new VerefooProxy(g, root.getHosts(),
    root.getConnections(), root.getConstraints(),
    prop, paths, root.getProfile().getName().name() , "FW");
long beginAll = System.currentTimeMillis();
VerificationResult res = test.checkNFFGProperty();
long endAll = System.currentTimeMillis();
resultChecker = endAll - beginAll;
time = (int) res.getTime();
if (res.result != Status.UNSATISFIABLE && res.result != Status.UNKNOWN) {
    Translator t = new Translator(res.model.toString(), root, g,
        test.getAllocationNodes(), test.getTrafficFlowsMap(), "FW");
    z3Model = res.model.toString();
    t.setNormalizer(norm);
    result = t.convert();
    root = result;
    sat = true;
    serviceGraph.RemoveAP();
} else {
    sat = false;
    result = root;
}
root.getPropertyDefinition().getProperty().stream().filter(p ->
    p.getGraph() == g.getId())

```

```

        .forEach(p -> p.setIsSat(res.result != Status.UNSATISFIABLE));

        flag=1;
    }
    ...

```

Il passo successivo all’allocazione dei VPN Gateway è l’allocazione dei Firewall configurati come Packet Filter. Per fare ciò il meccanismo che si utilizza è simile al passo precedente. Viene quindi definita una lista di proprietà da soddisfare che in questo caso saranno soltanto le proprietà di isolamento e di raggiungibilità, ovvero le proprietà che determinano la necessità di inserire dei Packet Filter all’interno della topologia di rete. Diversamente dal caso precedente, dopo aver inserito nella lista tutte le proprietà relative ai Firewall, viene controllata la lista delle proprietà di protezione e ogni proprietà viene aggiunta alla lista precedente. Questo step intermedio è necessario perché dopo aver chiamato per la seconda volta il VerefooProxy per allocare i Firewall, al fine di verificare la correttezza della soluzione prodotta non sarà sufficiente avere la lista con i requisiti di isolamento e raggiungibilità ma è necessario considerare tutte le proprietà nella topologia finale prodotta. Dopo aver prodotto la seconda lista, se è presente almeno una proprietà e la variabile di soddisfabilità è impostata a true (ovvero se lo step precedente è andato a buon fine) viene ricalcolato il nuovo Service Graph aggiungendo gli allocation places (AP) come descritto nella figura(6.3). Successivamente viene ripetuto lo step precedente istanziando un nuovo VerefooProxy, controllando il risultato e traducendolo in un output comprensibile all’utente. L’unica differenza con lo step precedente è che una volta terminata la conversione vengono rimossi gli allocation places aggiunti precedentemente dal Service Graph come illustrato in figura(6.4).

6.3.2 Definizione del Verefoo Proxy

Nonostante il Serializer svolga una funzione molto importante all’interno dell’ecosistema di Verefoo, il cuore pulsante delle operazioni di calcolo del problema MaxSMT, dell’allocazione delle Network Security Functions e della loro configurazione viene effettuato all’interno del Proxy, che svolge la funzione di vera e propria interfaccia con le funzionalità del framework.

Di seguito viene fornita la definizione del Verefoo Proxy con un breve commento delle modifiche effettuate alle funzioni che svolge:

Listing 6.5. Esempio di codice Java

```

public VerefooProxy(Graph graph, Hosts hosts, Connections
        conns, Constraints constraints, List<Property> prop,
        List<Path> paths, String profile, String type) throws
        BadGraphError {
    this.profile = profile;
    this.type = type;
    // Initialitation of the variables related to the nodes
    allocationNodes = new HashMap<>();
}

```

```

nodes = graph.getNode();
nodes.forEach(n -> allocationNodes.put(n.getName(), new
    AllocationNode(n)));
wildcardManager = new WildcardManager(allocationNodes);
properties = prop;
securityRequirements = new HashMap<>();
int idRequirement = 0;
for(Property p : properties) {
    securityRequirements.put(idRequirement, new
        SecurityRequirement(p, idRequirement));
    idRequirement++;
}

this.paths = paths;
this.nodeMetrics =
    constraints.getNodeConstraints().getNodeMetrics();

//Creation of the z3 context
HashMap<String, String> cfg = new HashMap<String, String>();
cfg.put("model", "true");
ctx = new Context(cfg);

//Creation of the NetContext (z3 variables)
nctx = nctxGenerate(ctx, nodes, prop, allocationNodes);
nctx.setWildcardManager(wildcardManager);

trafficFlowsMap = generateFlowPaths();
allocationManager = new AllocationManager(ctx, nctx,
    allocationNodes, nodeMetrics, prop, wildcardManager,type);
allocationManager.instantiateFunctions();
allocateFunctions();
distributeTrafficFlows();
allocationManager.configureFunctions(type);
check = new Checker(ctx, nctx, allocationNodes);

formalizeRequirements();

}

```

All'interno del Proxy è stata inserita una variabile definita type che permette di comprendere, ad ogni istanza del Proxy, la Network Security Function che è necessario allocare. Al momento è possibile utilizzare due parametri possibili: "VPN" per definire l'allocazione dei VPN Gateway e "FW" per definire l'allocazione dei Firewall.

Dopo aver definito il profilo e il tipo di funzione da allocare vi è una breve inizializzazione delle variabili necessarie al calcolo del risultato, tra le quali viene creata una HashMap per definire gli allocationNodes, un wildcardManager che serve per gestire la comunicazione con il tool Z3 e una HashMap per contenere i vari security requirements. Dopo aver creato il contesto di z3 per il calcolo del problema

MaxSMT, il proxy esegue le seguenti funzioni per trovare una soluzione:

1. **Calcolo Flow Paths:** forniti i requisiti di sicurezza vengono calcolati tutti i percorsi possibili all'interno della topologia di rete per soddisfarli.
2. **Definizione e istanziazione di Funzioni:** ogni elemento all'interno della topologia viene istanziato dentro alla variabile di contesto di z3 definita precedentemente. In questo caso le definizioni sono possibili grazie a delle modifiche ai file di configurazione di z3 avvenute precedentemente a questo lavoro, che consentono di poter comprendere contemporaneamente sia i vincoli necessari per definire i firewall che quelli per i VPN gateway.
3. **Allocazione delle funzioni di sicurezza:** una volta istanziate le funzioni di rete vengono allocate tenendo conto della nuova variabile type. Grazie a questo marker che consente di capire in quale passo dell'allocazione siamo indirizzando il codice ad allocare solo i Gateway o solo i Firewall e ad associarli ad un determinato allocation place.
4. **Distribuzione del traffico:** viene calcolato il traffico in input per ciascun nodo e viene distribuito cercando la soluzione migliore.
5. **Configurazione delle funzioni di sicurezza:** avendo soddisfatto tutto il traffico necessario descritto nei requisiti viene proposta una configurazione delle funzioni a seconda della variabile type che è stata allocata.
6. **Formalizzazione dei requisiti:** terminate le configurazioni l'ultimo step si occupa di controllare formalmente se i requisiti vengono rispettati grazie ad un checker.

Grazie all'ausilio dei lavori precedenti, con le modifiche effettuate è ora possibile allocare in un'unica istanza d'esecuzione del framework sia dei Firewall configurati come packet filter che dei VPN Gateway per cifrare il traffico in transito all'interno di una topologia di rete. Di conseguenza si può considerare raggiunto il secondo macro obiettivo di questo lavoro di tesi.

Capitolo 7

Design, progetto e sviluppo Demo B

All'interno di questo capitolo viene descritto il design, lo sviluppo e la realizzazione del secondo lavoro di Demo prodotto. Questo lavoro si differenzia dal precedente descritto al Capitolo [5] in quanto ogni elemento appartenente alla Demo è stato creato da zero, senza avere nessun riferimento precedente.

Inizialmente verrà fornita una breve introduzione alla demo, descrivendo obiettivi e motivazioni che han portato allo sviluppo, successivamente invece verrà descritta la topologia di rete progettata e i requisiti di sicurezza richiesti, analizzando le motivazioni che hanno portato alla scelta di questi ultimi.

Terminata l'introduzione verrà poi descritto lo sviluppo dell'ambiente virtuale, facendo riferimenti a snippet di codici effettivamente implementati all'interno dell'ambiente e descrivendo accuratamente i file utilizzati per creare immagini e container all'interno di Docker Compose.

Infine l'ultima parte del capitolo, in maniera simile al capitolo sulla Demo A, proporrà dei test e delle verifiche della correttezza degli output forniti da Verefoo.

7.1 Introduzione

Come è stato ampiamente discusso nel Capitolo [6] la nuova versione di Verefoo prodotta è ora in grado di allocare, in un'unica iterazione, contemporaneamente due tipi di Network Security Functions: i Firewall configurati come dei Packet Filter e dei VPN Gateway che consentono l'autenticazione e la cifratura dei pacchetti durante le comunicazioni fra due endpoint. Tramite queste novità è quindi possibile definire contemporaneamente sia le proprietà di isolamento e raggiungibilità che quelle di protezione, garantendo flessibilità all'utente. Essendo questo un risultato definibile come una milestone nel percorso di sviluppo di Verefoo si è pensato di realizzare una Demo che possa mostrare i progressi raggiunti tramite l'istanziazione di un nuovo ambiente virtuale. In questo caso rispetto al precedente non ci si è concentrati sul definire dei requisiti il più simile possibile a quelli di una ipotetica azienda di medie dimensioni quanto più a mostrare in maniera evidente come tutti e 3 i requisiti di sicurezza vengono rispettati. Nonostante questo obiettivo si è però deciso di utilizzare una topologia più complessa rispetto a quella della Demo A,

così da mostrare anche come con diversi elementi che aumentano il carico computazionale del framework, la soluzione che viene fornita in output è computata in maniera rapida, ottimale e soprattutto corretta.

Avendo già sviluppato un installer per la Demo A (paragrafo 5.2) all'interno del repository non è stato necessario crearne uno nuovo, in quanto i package utilizzati in questa Demo sono gli stessi di quella precedente, è quindi stato fornito all'utente lo stesso file.

7.2 Implementazione

Per raggiungere gli obiettivi descritti nell'introduzione si è pensato di creare molti più host rispetto alle versioni precedenti. In questo modo è possibile notare anche le potenzialità di virtualizzazione che l'utilizzo dei container tramite Docker Compose ci permette di avere. Inoltre ogni host che viene rappresentato in figura non rappresenterà un unico elemento all'interno della topologia ma uno dei possibili host della sottorete definita nel quadrato in cui l'host è contenuto. Di seguito viene quindi fornita una rappresentazione grafica della topologia:

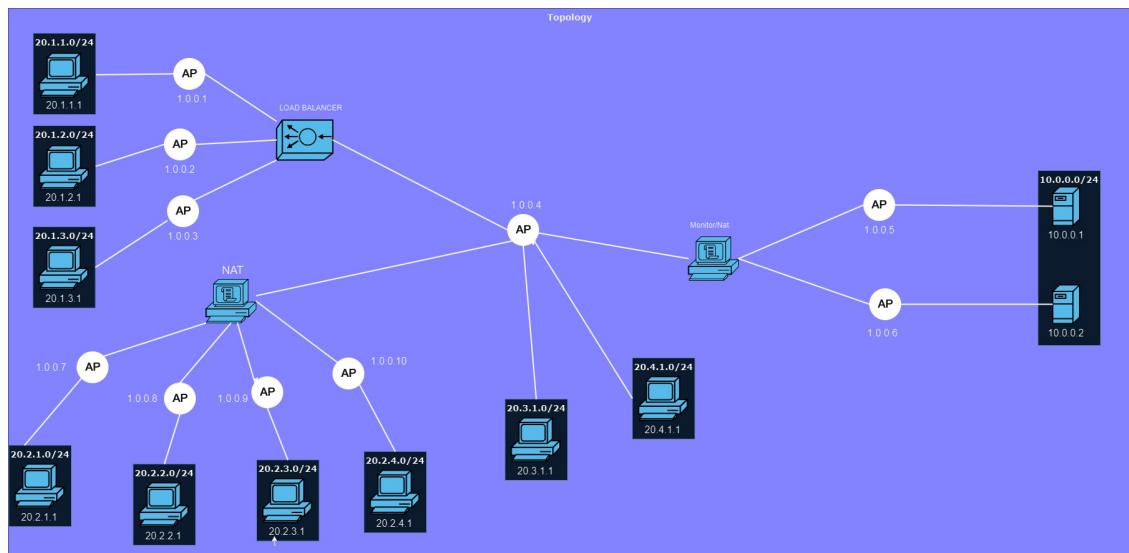


Figura 7.1. Grafo di Allocazione della Demo B

Entrando più nel dettaglio è possibile notare diversi gruppi di host, che rappresentano delle ipotetiche sedi aziendali separate. La prima, in alto a sinistra, viene descritta con la sottorete 20.1.0.0/16. All'interno di questo cluster sono presenti tre sottoreti rappresentate dagli ip 20.1.1.0/24, 20.1.2.0/24, 20.1.3.0/24. Le comunicazioni di queste sottoreti vengono regolate da un Load Balancer, che in caso di congestione del traffico decide arbitrariamente in quale nodo della rete inoltrare il traffico per decongestionare alcuni collegamenti. La seconda sede aziendale è rappresentata in basso a sinistra dalla rete 20.2.0.0/16. Come per la prima sede sono presenti 4 sottoreti di host definite dagli ip 20.2.1.0/24, 20.2.2.0/24, 20.2.3.0/24, 20.2.4.0/24. Diversamente dalla prima, all'esterno del cluster 20.2.0.0/16 è presente

un NAT che si occupa di mascherare gli indirizzi ip all'interno della rete con il resto della topologia. In basso sono inoltre presenti altre due sedi più piccole delle precedenti, definite dalle sottoreti 20.3.1.0/24 e 20.4.1.0/24, queste dovrebbero simulare eventuali dipendenti in smart working che quindi non si connettono da una grande rete aziendale ma dalla propria rete di casa privata.

Sulla destra è invece presente una Server Farm definita dalla rete 10.0.0.0/24; all'interno sono stati definiti due Web Server con IP 10.0.0.1 e 10.0.0.2. Infine è presente un nodo che funge da monitor (Mnt) per ispezionare il traffico in entrata ed uscita dai Web Server. La topologia appena descritta può essere sintetizzata dalla tabella seguente:

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

Tabella 7.1. Definizione nodi della topologia per Demo B

Entrambi i server sono i punti d'interesse più importanti della topologia perchè le comunicazioni che verranno effettuate durante la simulazione utilizzeranno come uno dei due host almeno uno dei due server. Entrando più nello specifico l'obiettivo dei requisiti di sicurezza di isolamento e raggiungibilità sarà quello di rendere le comunicazioni della prima sede possibili solo con il server 10.0.0.2, della seconda sede solo con il server di 10.0.0.1 e delle restanti due sedi con entrambi i server. Per quanto riguarda invece i requisiti di protezione l'obiettivo che si ci si è prefissati è quello di inserire almeno 2 VPN Gateway affinchè un host specifico all'interno della rete possa comunicare in maniera sicura con i server per lo scambio di informazioni riservate. Di conseguenza è stato scelto uno degli host della prima sede, immaginandosi quindi la rete 20.1.0.0/16 come la sede centrale dell'azienda. La definizione dei requisiti di sicurezza è quindi definibile in input al framework tramite la seguente tabella:

Policy	IPSrc	IPDst	pSrc	pDst	tProto	Confidentiality	Intregrity	Untrusted nodes
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

Tabella 7.2. Definizione requisiti di sicurezza della topologia per Demo B

- **Prima coppia di Regole:** Il Web Client C1-1 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S2 e non deve poter raggiungere con alcun percorso il server S1 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Seconda coppia di Regole:** Il Web Client C1-2 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S2 e non deve poter raggiungere con alcun percorso il server S1 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Terza coppia di Regole:** Il Web Client C1-1 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S2 e non deve poter raggiungere con alcun percorso il server S1 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Quarta coppia di Regole:** Il Web Client C2-1 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e non deve poter

raggiungere con alcun percorso il server S2 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.

- **Quinta coppia di Regole:** Il Web Client C2-2 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e non deve poter raggiungere con alcun percorso il server S2 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Sesta coppia di Regole:** Il Web Client C2-3 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e non deve poter raggiungere con alcun percorso il server S2 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Settima coppia di Regole:** Il Web Client C2-4 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e non deve poter raggiungere con alcun percorso il server S2 all'interno della topologia. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Ottava coppia di Regole:** Il Web Client C3-1 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e con almeno un percorso anche il server S2. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Nona coppia di Regole:** Il Web Client C4-1 deve poter essere sempre in grado di raggiungere con almeno un percorso il server S1 e con almeno un percorso anche il server S2. Per entrambe le regole non ci sono limitazioni di utilizzo sulla porta e sul protocollo di quarto livello, è quindi possibile utilizzare una qualsiasi porta nel range [0-65536] sia per il traffico in entrata che in uscita ed un protocollo a scelta fra UDP e TCP.
- **Decima Regola:** Il Web Client C1-1 deve poter comunicare in maniera sicura con il Web Server S2. Il traffico originato da C1-1 può utilizzare qualsiasi porta di uscita per il protocollo di trasporto ma il Server S2 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 Mnt come nodo non sicuro e attraverso il quale il traffico deve passare cifrato.

Come ampiamente è stato già discusso per la Demo A, anche in questo caso non è sufficiente definire le regole su Verefoo per assicurarsi di avere una soluzione ottimale, ma è necessario creare un ambiente isolato in grado di testare la correttezza delle configurazioni prodotte da Verefoo. Anche in questo caso è stato scelto di utilizzare Docker Compose per la realizzazione di una rete virtuale, nella quale ogni host, server, forwarder, firewall e VPN gateway è rappresentato univocamente da un container in esecuzione. Di seguito viene presentata una parte dei servizi definiti all'interno del docker-compose file per istanziare una topologia del genere:

Listing 7.1. Definizione di Services dell'ambiente virtuale DemoB

```

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
    10.0.0.100 && tail -F anything"
  networks:
    servers:
      ipv4_address: 10.0.0.1
server2:
  container_name: server2
  hostname: server2
  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
    10.0.0.199 && tail -F anything"
  networks:
    servers:
      ipv4_address: 10.0.0.2

forwarder1:
  container_name: forwarder1
  hostname: forwarder1
  image: endpoint
  cap_add:
    - NET_ADMIN
  volumes:
    - './RouterVPNConfig:/mnt:ro'
  command: sh -c "/mnt/staticroutes/forone && tail -F anything"
  networks:
    servers:
      ipv4_address: 10.0.0.100
    forwarder1_vpn3:
      ipv4_address: 130.0.3.2
    forwarder1_fw:
      ipv4_address: 80.0.1.2
client1_1:
  container_name: client1_1
  hostname: client1_1
  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
    20.1.1.100 && tail -F anything"
  networks:
    clients1_1:
      ipv4_address: 20.1.1.1

```

In questo design, diversamente dal precedente, è possibile notare come i due Web Server condividono la stessa rete, ma viene impostato un default gateway differente. Questa scelta è stata necessaria in quanto come è possibile notare nella figura (7.1) sono presenti due diversi allocation place per collegare i due elementi. Grazie a questa configurazione sarà quindi possibile instanziare Network Security Functions differenti a seconda del server con cui si comunica. Nelle definizioni d'esempio è presente pure quella di un forwarder, in questo caso seppur l'immagine utilizzata per la costruzione del container è identica a quella di un endpoint generico durante l'instanziazione del container viene eseguito un comando bash aggiuntivo che carica il file "forone" dalle static route. Tramite ciò è possibile definire tutti i forwarding path per ogni elemento che non influenza le proprietà di sicurezza della rete.

Per quanto riguarda i Web Client all'interno della topologia la singola definizione è identica a quella proposta all'interno della Demo A con l'unica maggiore differenza è che è stato scelto l'ip terminante con ".100" come il default gateway per ogni host.

Entrando più nello specifico, è stato necessario definire delle immagini dalle quali istanziare i container grazie al Docker deamon. Per tutte le immagini si è deciso di utilizzare delle immagini di alpine, una distribuzione Linux estremamente leggera in termini di dimensioni su disco e generalmente sicura, nella quale sono poi installabili eventuali package aggiuntivi che permettono di creare delle versioni custom dell'immagine a seconda di ciò che vogliamo far svolgere al container. Nel caso degli endpoint la definizione del Dockerfile è la seguente:

Listing 7.2. Definizione Dockerfile Endpoint

```

1  FROM alpine:latest
2
3  RUN apk update
4  RUN apk add hping3 --update-cache --repository
   http://dl-cdn.alpinelinux.org/alpine/edge/testing
5  RUN apk add tcpdump
6  RUN apk add net-tools
7  ENV PS1='\h:\w\$ '
8
9  CMD ["/bin/sh"];

```

Partendo dall'ultima immagine disponibile nel repository generale di Docker viene installato un container alpine, successivamente viene fatto un update dei vari package disponibili e installato il package hping3 che consente di inviare dei pacchetti ICMP/UDP/TCP e tracciare il loro percorso all'interno della rete utilizzando il comando "-traceroute". Questo package è puramente necessario per eseguire le

operazioni di debug e controllo all'interno della rete. Completano il container i package tcpdump e net-tools che permettono rispettivamente di osservare le interfacce di rete per osservare i pacchetti in transito e di eseguire comandi per la configurazione di rete come ifconfig, arp, ipmaddr eccetera.

Nonostante gli output non siano ancora stati prodotti, dai requisiti di sicurezza ci si aspetta di avere degli allocation places nel quale verranno istanziati Firewall e VPN. Di conseguenza dei Dockerfile sono stati scritti per la futura allocazione di queste Security Network Functions. Di seguito viene fornito il Dockerfile dei container destinati a diventare Firewall:

Listing 7.3. Definizione Dockerfile Firewall

```

1  FROM alpine:latest
2
3  RUN apk update
4  RUN apk add iptables sudo
5  RUN apk add tcpdump
6  RUN apk add bash
7  RUN apk add --no-cache quagga \
8      && touch /etc/quagga/zebra.conf \
9      && touch /etc/quagga/vtysh.conf \
10     && touch /etc/quagga/ripd.conf
11 ENV PS1='\[ \w \$ '
12 CMD ["/bin/sh"];

```

All'interno di questa configurazione di alpine viene installato il package iptables, che è necessario in quanto permette di definire a livello software un firewall, di monitorare il traffico in ingresso ed uscita dal nodo e di scartare eventuali pacchetti che non soddisfano la configurazione. A questo package si aggiunge quagga che è un package in grado di definire delle funzioni di routing avanzate per comunicazioni basate su TCP o IP. All'interno della cartella di installazione di quagga vengono poi definiti dei file di configurazione necessari ad un corretto funzionamento del software. I packages di tcpdump e bash sono presenti solamente per effettuare operazioni di debug e test.

Infine è disponibile una definizione dell'immagine di un ipotetico VPN Gateway:

Listing 7.4. Definizione Dockerfile VPN Gateway

```

1  FROM alpine:latest
2  RUN apk update
3  RUN apk add net-tools
4  RUN apk add iptables sudo
5  RUN apk add tcpdump
6  RUN apk add --update strongswan && \
7      rm -rf /var/cache/apk/* && \
8      mkdir -p /etc/strongswan
9  ENV PS1='\[ \w \$ '
10 CMD ["/bin/sh"]

```

Quest'ultima configurazione oltre ad utilizzare alcuni packages come iptables, tcpdump e net-tools discussi precedentemente installa il software fondamentale per effettuare operazioni di tunneling VPN ovvero strongswan. Questo package è in grado di accettare le configurazioni che vengono fornite in output dal Translator di Verefoo, velocizzando quindi le operazioni di testing delle configurazioni prodotte.

7.3 Output

7.3.1 Topologia di Rete

Una volta definito tutto il necessario affinchè sia possibile istanziare in sicurezza e in maniera corretta un ambiente virtuale basato sui container Docker, è possibile iniziare a verificare la correttezza della nuova versione del framework analizzando i suoi output. È importante sottolineare che il numero di allocation places definiti in input è volutamente inferiore a quello necessario per rispettare i vincoli in maniera ottimale. Tramite questa scelta di design è quindi possibile controllare che i nuovi allocation places istanziati dopo l'allocazione delle VPN vengano considerati nel calcolo delle soluzioni ottimali.

Fornendo quindi l'output definito nel paragrafo precedente la soluzione prodotta in output da Verefoo è la seguente:

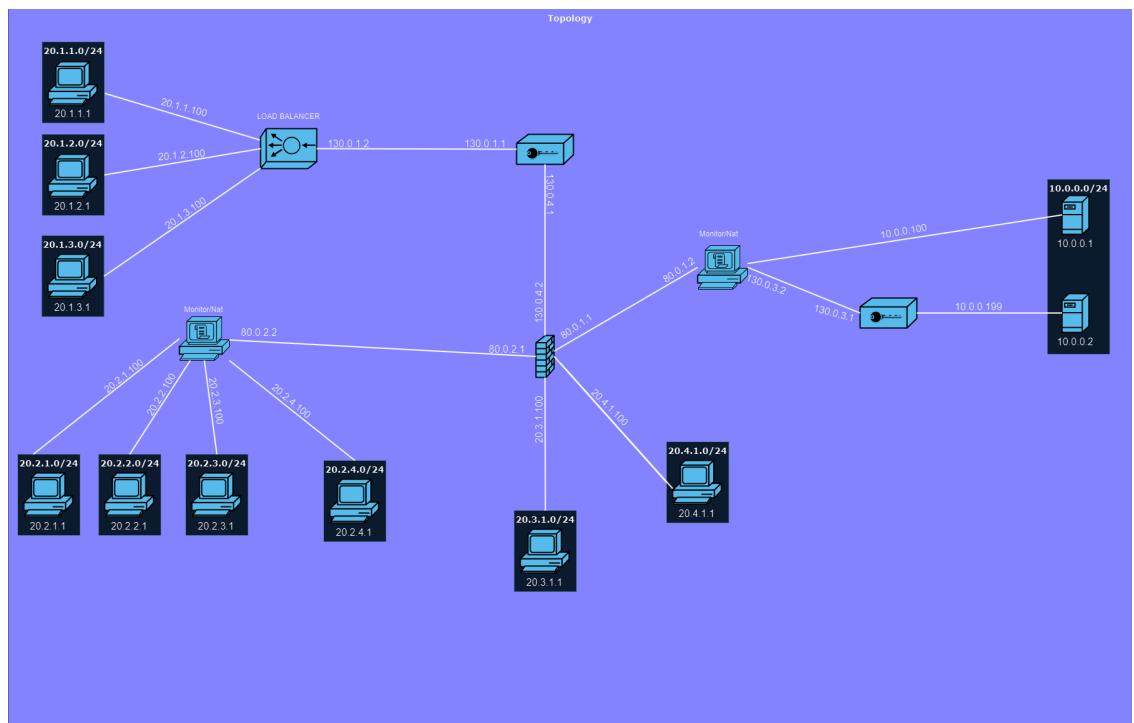


Figura 7.2. Verefoo Output

Come ci si poteva aspettare, dati i requisiti di sicurezza definiti sono state allocate tre Network Security Functions. Al centro della topologia è stato inserito

un Firewall, che a prima vista sembra essere posizionato correttamente perché in quella posizione riesce a scartare il traffico proveniente sia dalla rete 20.1.*.*/16 che quello dalla rete 20.2.*.*/16. Ai lati sinistro e destro sono invece presenti i due VPN Gateway, occorre portare particolare attenzione a quello situato a sinistra in quanto il suo allocation places non era presente in input (figura 7.1), di conseguenza è possibile dedurre che dopo aver istanziato una Network Security Function la successiva è stata posizionata in uno degli allocation places aggiuntivi formati nella nuova versione del framework.

7.3.2 Configurazioni Network Security Functions

Data la posizione che sembra definire una soluzione plausibilmente corretta, è possibile analizzare anche le configurazioni fornite per ogni funzione di sicurezza. La seguente è la tabella che descrive la configurazione dei due gateway VPN:

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

Tabella 7.3. Configurazione dei due VPN gateway Demo B

Come si può notare, dato l'unico requisito di protezione che era stato definito, solo due gateway sono stati allocati, rispettivamente di accesso per quanto riguarda il gateway presente a sinistra, e di uscita per quello di destra. In entrambe le configurazioni si può notare che il singolo host dal quale comunicazioni dovranno essere cifrate è la sorgente 20.1.1.1, mentre il nodo di destinazione è il server S2 definito dall'IP 10.0.0.2. Anche i requisiti sul protocollo di trasporto sono verificati in quanto la porta di destinazione che il server deve ricevere è la 22 che è quella indicata nei requisiti ed inoltre anche nella configurazione non c'è nessun vincolo di protocollo tra TCP e UDP.

Per quanto riguarda invece la configurazione del firewall, Verefoo fornisce il seguente output:

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

Tabella 7.4. Configurazione Firewall Demo B

Anche in questo caso il framework sembra produrre una soluzione plausibile, in quanto il Firewall è stato configurato in blacklist, permettendo a qualsiasi traffico il transito tranne a quello definito nelle regole specifiche del firewall. In questo modo è possibile far comunicare tranquillamente le sottoreti 20.3.1.0/24 e 20.4.1.0/24 in quanto qualsiasi pacchetto da e per i due server S1 ed S2 non verrà mai scartato dal firewall. Viceversa per quanto riguarda le sottoreti 20.1.0.0/16 e 20.2.0.0/16 il

framework non solo ha prodotto delle regole corrette, ma è riuscito anche a raggruppare tutte le definizioni singole sulle reti più piccole in una definizione più generale grazie alla notazione con “*”. Tramite ciò quindi ogni elemento appartenente alla sottorete 20.1.0.0/16 non potrà comunicare con il server S1 ed ogni elemento nella rete 20.2.0.0/16 non potrà comunicare con il server S2.

7.3.3 Configurazioni Strongswan

Nonostante le configurazioni della topologia e delle network security function siano corrette, il framework prodotto allo stato attuale non fornisce una implementazione aggiornata del Translator di Strongswan, non è quindi possibile ottenere automaticamente una versione aggiornata dei file di configurazione “swanctl.conf”. Per aggirare questo problema, durante lo sviluppo di questa Demo i file di configurazione dei due Gateway è stato prodotto manualmente a partire dalle configurazioni fornite da Verefoo.

Di conseguenza, il seguente snippet di codice rappresenta uno dei due file di configurazione scritti per testare l’ambiente virtuale:

```

connections {
    site-site {
        local_addrs = 130.0.4.1
        remote_addrs = 130.0.3.1
        local {
            auth = pubkey
            certs = VpnConfig1Cert.pem
            id = VpnConfig1.strongswan.org
        }
        remote {
            auth = pubkey
            id = VpnConfig2.strongswan.org
        }
        children {
            net-net {
                local_ts = 20.1.1.1/32
                remote_ts = 10.0.0.2/32
                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
    }
}

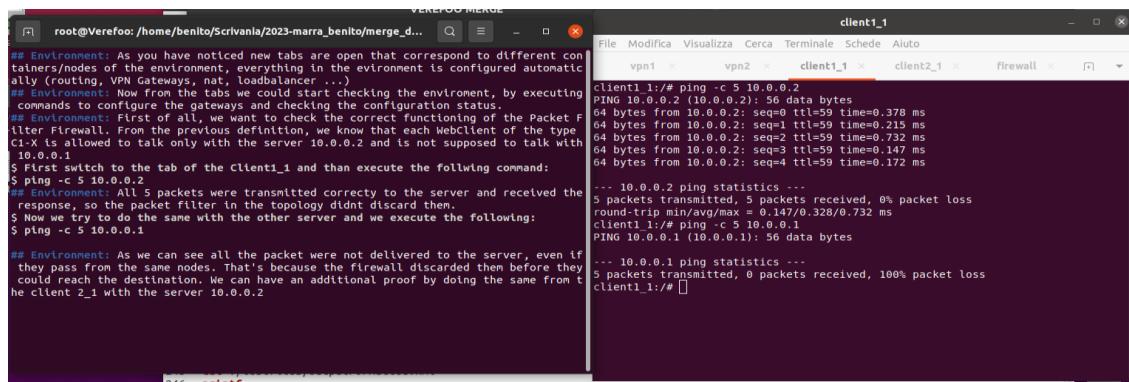
```

All'interno di questo esempio la connessione site-to-site è definita dalle interfacce dei due VPN gateway con IP 130.0.4.1 e 130.0.3.1. In locale vengono caricati i certificati attraverso il file VpnConfig1Cert.pem e l'autenticazione viene effettuata tramite chiave pubblica asimmetrica, in maniera simile viene definito l'id e il metodo di autenticazione del gateway remoto. Infine viene dichiarata una connessione end-to-end tra il client C1-1 definito da 20.1.1.1/32 e il Web Server S2 definito da 10.0.0.2/32. All'interno di questa connessione viene definita anche la proposta da effettuare per l'incapsulamento in IPSec tramite il protocollo ESP, utilizzando gli algoritmi di aes256-cbc e sha2-256.

7.4 Verifiche e Test

Grazie alle configurazioni prodotte in output dalla nuova versione del framework è possibile verificare che le soluzioni prodotte siano efficaci e corrette. Per fare ciò, come nella demo precedente è stato predisposto un ambiente virtuale composto da container Docker che simuleranno il comportamento di ogni elemento di rete descritto nella topologia. In questo caso le verifiche da effettuare all'interno della topologia saranno diverse perché non sarà necessario dimostrare solo che le comunicazioni fra l'host C1-1 ed il server S2 siano protette crittografia ma è anche necessario controllare se le comunicazioni fra i client ed i vari host vengono filtrate dal firewall.

Partendo dalla verifica del firewall sarà necessario utilizzare due terminali differenti, il primo corrispondente al client C1-1 che come da requisiti dovrebbe essere in grado di comunicare solo con il server S2, isolando tutte le comunicazioni con S1. Per testare ciò utilizzeremo i seguenti comandi Linux:



The screenshot shows two terminal windows side-by-side. The left window, titled 'VERFOO MERGE', is a root shell on 'root@Verefoo'. It contains a series of comments and commands related to environment setup and packet filtering. The right window, titled 'client1_1', is a user shell on 'client1_1'. It shows two ping operations: one to '10.0.0.2' (server S2) and another to '10.0.0.1' (server S1). The first ping to S2 shows 0% packet loss, while the second ping to S1 shows 100% packet loss, indicating successful filtering by the firewall.

```

## Environment: As you have noticed new tabs are open that correspond to different containers/nodes of the environment, everything in the environment is configured automatically (routing, VPN Gateways, nat, loadbalancer ...)
## Environment: Now from the tabs we could start checking the configuration status.
## Environment: First of all we want to check the correct functioning of the Packet Filter Firewall. From the previous definition, we know that each WebClient of the type C1-X is allowed to talk only with the server 10.0.0.2 and is not supposed to talk with 10.0.0.1
$ First switch to the tab of the Client1_1 and then execute the following command:
$ ping -c 5 10.0.0.2
## Environment: All 5 packets were transmitted correctly to the server and received the response, so the packet filter in the topology didnt discard them.
$ Now we try to do the same with the other server and we execute the following:
$ ping -c 5 10.0.0.1

## Environment: As we can see all the packet were not delivered to the server, even if they pass from the same nodes. That's because the firewall discarded them before they could reach the destination. We can have an additional proof by doing the same from the client 2_1 with the server 10.0.0.2

```

Figura 7.3. Verifica pacchetti scartati dal Firewall per rete 20.1.*.*

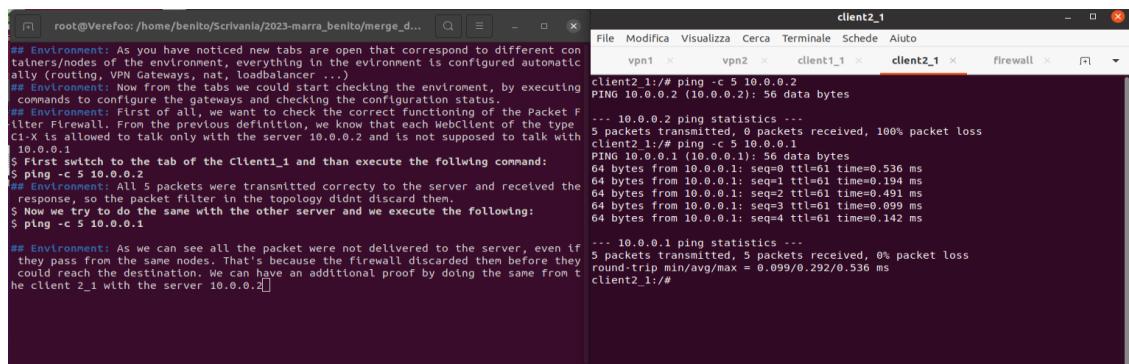
All'interno del terminale C1-1 eseguiamo il comando `"ping -c 5 [ipaddr]"` per verificare le connessioni con i due server presi in esame. Nel primo caso testiamo sul server dall'IP "10.0.0.2" ovvero S2: come si può notare vengono trasferiti 5 pacchetti col protocollo ICMP e contemporaneamente vengono ricevuti 5 pacchetti con una statistica di 0% packet loss. Se invece eseguiamo la stessa operazione con il server dall'IP "10.0.0.1" il risultato sarà opposto, ovvero tutti i pacchetti inviati

vengono scartati non ricevendo alcuna risposta ed il numero di packet loss sarà uguale al 100%.

Considerando il percorso che eseguono i pacchetti, ovvero

C1-1 → Lb → VPNGateway1 → Firewall → Monitor → VPNGateway2 → Server è possibile dedurre che l'elemento che scarta i pacchetti e impedisce le comunicazioni è proprio il firewall con la configurazione prodotta da Verefoo.

Tuttavia verificare le comunicazioni fra il client C1-1 ed il Server S2 non è sufficiente a verificare il corretto funzionamento del firewall, è infatti necessario assicurarsi che anche le comunicazioni all'interno della rete 20.2.0.0/16 vengano filtrate quando si prova a comunicare con il server S1.



```

root@Verefoo:/home/benito/Scrivana/2023-marra_benito/merge_d...
File Modifica Visualizza Cerca Terminale Schede Aiuto
client2_1
vpn1 < vpn2 < client1_1 < client2_1 < firewall < F1 <
client2_1:/# ping -c 5 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56 data bytes
... 10.0.0.2 ping statistics ...
5 packets transmitted, 5 packets received, 100% packet loss
client2_1:/# ping -c 5 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56 data bytes
64 bytes from 10.0.0.1: seq=0 ttl=61 time=0.536 ms
64 bytes from 10.0.0.1: seq=1 ttl=61 time=0.194 ms
64 bytes from 10.0.0.1: seq=2 ttl=61 time=0.491 ms
64 bytes from 10.0.0.1: seq=3 ttl=61 time=0.099 ms
64 bytes from 10.0.0.1: seq=4 ttl=61 time=0.142 ms
... 10.0.0.1 ping statistics ...
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.099/0.292/0.536 ms
client2_1:/#

```

Figura 7.4. Verifica pacchetti scartati dal Firewall per rete 20.2.*.*

Per verificare anche questa condizione vengono effettuati gli stessi comandi sul terminale del client C2-1, avendo un IP appartenente alla rete 20.2.0.0/16. Il risultato che viene visualizzato in output è coerente con quello che ci si aspetta, essendo opposto all'output del client C1-1. Le comunicazioni con il server 10.0.0.2 non risultano possibili in quanto ogni pacchetto è stato scartato dal Firewall ed è presente anche qua il 100% di packet loss, viceversa quando si fa un ping sul server 10.0.0.1 corrispondente a S1 ogni pacchetto viene correttamente trasmesso e ricevuto dal server, passando quindi i controlli del firewall.

È quindi possibile affermare che le configurazioni riguardanti i requisiti di raggiungibilità e di isolamento prodotte dal framework risultano corretti e con il minimo uso di risorse possibili, in quanto grazie ad un unico firewall sono state soddisfatte tutti le caratteristiche fornite in input.

La successiva verifica che deve essere effettuata sulla bontà della soluzione prodotta riguarda l'istanziazione e la configurazione dei VPN Gateway nella topologia di output. In questa istanza specifica solo 2 gateway sono stati allocati, quindi è sufficiente verificare che i pacchetti in transito tra il nodo C1-1 ed il server S2 risultano cifrati correttamente. Prendendo spunto dai test effettuati nella demo precedente utilizziamo tcpdump nel seguente modo:

The screenshot shows a terminal window with two tabs: 'root@Verefoo:/home/benito/Scrivania/2023-marra_benito/m...' and 'firewall'. The terminal tab displays the configuration of a VPN connection ('site-site') using swanctl commands. It includes steps for loading certificates, initiating the connection, and checking its status. The firewall tab shows a packet capture session ('tcpdump -i eth4') running on the interface 'eth4' (EN10MB Ethernet). The output of the capture shows several ESP (Encapsulated Security Payload) packets being transmitted between the local host and a remote IP address (130.0.3.1).

```

root@Verefoo:/home/benito/Scrivania/2023-marra_benito/m...
$ successfully loaded 1 connections, 0 unloaded
$ Do the same for vpn2 (the following command is executed automatically into the container)
$ sudo docker exec vpn2 swanctl -q
loaded certificate from '/etc/swanctl/x509/vpnConfig3cert.pem'
loaded certificate from '/etc/swanctl/x509ca/strongswanCert.pem'
loaded ED25519 key from '/etc/swanctl/ecdsa/vpn3key.pem'
no authorities found, 0 unloaded
no pool found, 0 unloaded
loaded certificate 'site-site'
successfully loaded 1 connections, 0 unloaded
$ swanctl --list-conns --> type this command on vpn1 or 2 to check the correct configuration file loading
$ swanctl --list-certs --> type this command on vpn1 or 2 to check the correct certificates loading
$ Now that the configurations are loaded we can initiate the connection by: (the following command is executed automatically into the container)
$ sudo gnome-terminal --tab --title="Initiate" -- bash -c 'docker exec vpn1 swanctl --initiate --child net-net'
$ ipsec status --> type this command on vpn1 and vpn1 to check the connection setup
$ swanctl -l--> type this command on vpn1 and vpn1 to check the connection setup
## Testing: As you can see the previous commands ensure that the connection between the vpn gateways is up and running.
## Testing: Now we can test if the connection is really encrypted and secure.
$ Go on the firewall tab and execute the following:
$ tcpdump -i eth4

```

Figura 7.5. Setup tcpdump per verifiche di sicurezza

Il nodo preso in analisi è lo stesso in cui viene instanziato il firewall, utilizzando tcpdump è possibile osservare e monitorare i pacchetti in transito attraverso una delle interfacce del nodo. L’interfaccia scelta per questo test è ”eth4” corrispondente alla connessione fra il firewall ed il primo VPN Gateway. Il risultato aspettato è quindi, come per la prima demo, osservare il transito dei pacchetti non come semplici ICMP packets ma come ESP packets, ovvero il protocollo utilizzato per l’incapsulamento in IPsec.

Per controllare l’output è quindi necessario, una volta impostato il monitoraggio dell’interfaccia, eseguire il comando di ping come fatto per il firewall precedentemente e controllare l’output nel terminale del firewall. Di seguito viene proposto un possibile output effettuato:

The screenshot shows a terminal window with two tabs: 'root@Verefoo:/home/benito/Scrivania/2023-marra_benito/m...' and 'firewall'. The terminal tab displays the configuration of a VPN connection ('site-site') using swanctl commands. The firewall tab shows a packet capture session ('tcpdump -i eth4') running on the interface 'eth4' (EN10MB Ethernet). The output of the capture shows several ESP (Encapsulated Security Payload) packets being transmitted between the local host and a remote IP address (130.0.3.1). Below the capture, a ping command is shown being run from the terminal, with the output indicating successful communication through the tunnel.

```

root@Verefoo:/home/benito/Scrivania/2023-marra_benito/m...
$ successfully loaded 1 connections, 0 unloaded
$ Do the same for vpn2 (the following command is executed automatically into the container)
$ sudo docker exec vpn2 swanctl -q
loaded certificate from '/etc/swanctl/x509/vpnConfig3cert.pem'
loaded certificate from '/etc/swanctl/x509ca/strongswanCert.pem'
loaded ED25519 key from '/etc/swanctl/ecdsa/vpn3key.pem'
no authorities found, 0 unloaded
no pool found, 0 unloaded
loaded certificate 'site-site'
successfully loaded 1 connections, 0 unloaded
$ swanctl --list-conns --> type this command on vpn1 or 2 to check the correct configuration file loading
$ swanctl --list-certs --> type this command on vpn1 or 2 to check the correct certificates loading
$ Now that the configurations are loaded we can initiate the connection by: (the following command is executed automatically into the container)
$ sudo gnome-terminal --tab --title="Initiate" -- bash -c 'docker exec vpn1 swanctl --initiate --child net-net'
$ ipsec status --> type this command on vpn1 and vpn1 to check the connection setup
$ swanctl -l--> type this command on vpn1 and vpn1 to check the connection setup
## Testing: As you can see the previous commands ensure that the connection between the vpn gateways is up and running.
## Testing: Now we can test if the connection is really encrypted and secure.
$ Go on the firewall tab and execute the following:
$ tcpdump -i eth4
## Testing: Thanks to tcpdump we are able to check every packet that passes into the firewall. $ Return to the client 1 and try to ping the server 10.0.0.2 again with the following:
$ ping -c 5 10.0.0.2
## Testing: If we come back to the firewall tab we can see that all the packets that passed are encrypted using the ESP protocol. This means that the tunnel is up and working correctly. ## Testing: Now that we checked everything is ok, lets shut down the environment.
$ ping -c 5 10.0.0.2

```

Figura 7.6. Verifica cifratura dei pacchetti in transito

Il risultato ottenuto corrisponde alle aspettative descritte precedentemente. È infatti possibile notare come ogni pacchetto venga codificato con il protocollo ESP e viene monitorato con il path specifico del tunnel VPN. Di conseguenza è possibile affermare che il risultato prodotto da Verefoo è corretto in quanto garantisce la sicurezza in tutto il traffico partente dal nodo C1-1 al server S2. Grazie a questa ennesima verifica anche i requisiti di sicurezza sono rispettati, confermando la correttezza del framework e della demo prodotta. Di conseguenza si può affermare che

anche il terzo ed ultimo obiettivo della tesi è stato portato a termine, producendo una demo funzionante che mettesse in risalto le caratteristiche della nuova versione del framework.

Capitolo 8

Conclusioni e Lavori Futuri

Durante lo sviluppo dei lavori di tesi numerosi passi avanti nello sviluppo di Verefoo sono stati effettuati. Inizialmente, dopo aver studiato approfonditamente il funzionamento e le limitazioni del framework è stata prodotta una demo consultabile in maniera open source per eventuali utenti che vorrebbero testare e verificare la funzionalità di allocazione di tunnel VPN Gateway all'interno di una topologia di rete.

In seconda battuta, diverse modifiche sono state implementate al codice sorgente del software che precedentemente era in grado di allocare solamente Firewall o solamente tunnel VPN dipendentemente dalla versione. La soluzione prodotta invece è mista e consente di fornire all'utente topologie di rete che siano in grado di allocare contemporaneamente entrambe le Network Security Functions.

Dopo aver testato e corretto eventuali problemi della nuova versione del framework è stata prodotta una demo conclusiva del lavoro svolto, in grado di mostrare in maniera evidente le nuove funzionalità del merge prodotto. All'interno di quest'ultima viene inoltre mostrato un test della correttezza del lavoro prodotto, che è stato successivamente documentato e reso disponibile per eventuali pubblicazioni all'interno dei repository del framework.

Nonostante il lavoro prodotto risulta correttamente testato e completo, è possibile effettuare ulteriori innovazioni per rendere il prodotto disponibile all'utente ancora più efficiente. Per quanto riguarda le due demo prodotte i nodi configurati come Network Address Translator all'interno della topologia di rete in input al framework non vengono istanziati come tali all'interno dell'ambiente virtuale, ma sono stati instanziati invece dei nodi placeholder che agiscono come dei semplici forwarder dei pacchetti (con le rispettive static routes). Di conseguenza utilizzando software come iptables è possibile configurarli come dei NAT veri e propri per simulare in maniera quanto più affidabile possibile la situazione reale descritta in input.

Considerando invece il lavoro svolto sul codice sorgente di Verefoo il framework sembra funzionare nella maggior parte dei casi correttamente, tuttavia una parte ancora non sviluppata all'interno del framework è la traduzione e configurazione

automatica dei file di StrongSwan da produrre per le soluzioni prodotte. Sarebbe necessario automatizzare anche il meccanismo di configurazione di questi file, così da velocizzare eventuali test su ambienti virtualizzati e fornire all'utente una primitiva ma corretta configurazione dei tunnel VPN.

Inoltre nonostante la versione prodotta durante questo lavoro se ha risposto efficacemente a diversi input ci sono stati alcuni output più complessi di quelli presentati in esempio in questa tesi che hanno portato a delle soluzioni che erano incomplete o completamente errate, probabilmente dovuto ai vincoli definiti ed imposti su Z3 che conterranno qualche errore di configurazione.

Infine l'ultima innovazione che è consigliabile implementare è quella di aggiungere ulteriori funzioni di sicurezza di rete come ad esempio un IDS (Intrusion Detection System) o Web Application Firewall, al fine di garantire più flessibilità e scelta all'utente nel definire le funzionalità che la rete da configurare deve avere.

Bibliografia

- [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 1 marzo 2024. [Online]. Available: <https://www.vmware.com/topics/glossary/content/virtual-machine.html>
- [7] Docker, Inc. (2024) What is a container? Accessed on 1 marzo 2024. [Online]. Available: <https://www.docker.com/resources/what-container/>
- [8] (2024) What is a hypervisor? VMware, Inc. Accessed on 1 marzo 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 1 marzo 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>