Alma Mater Studiorum — Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Microsegmentazione, un approccio scalabile alla sicurezza basato su overlay network

Presentata da:
Giorgia Rondinini

Relatore:
Prof. Michele Colajanni

Correlatore:
Ing. Claudio Zanasi

Anno Accademico 2021–2022

Parole chiave

ZTA

Zero Trust

Microsegmentazione

Sicurezza

Scalabilità

Overlay network

Abstract

Il crescente numero di attacchi condotti contro sistemi e servizi informatici richiede nuove strategie per la cybersicurezza. In questa tesi si prende in considerazione uno degli approcci più moderni per questa attività, basato su architetture Zero Trust, che deperimetrizzano i sistemi e mirano a verificare ogni tentativo di accesso alle risorse indipendentemente dalla provenienza locale o remota della richiesta. In tale ambito, la tesi propone una nuova forma di microsegmentazione agent-based basata su overlay network, con l'obiettivo di migliorare la scalabilità e la robustezza delle soluzioni esistenti, ad oggi messe in secondo piano in favore della facilità di configurazione. Una consistente serie di test dimostra che l'approccio descritto, attuabile in molteplici tipologie di sistemi cloud, è in grado di garantire, oltre alla sicurezza, scalabilità al crescere dei nodi partecipanti, robustezza evitando punti unici di fallimento e semplicità di configurazione.

Indice

1 Introduzione

Gli ultimi anni hanno visto un continuo aumento degli attacchi informatici rivolti ad ogni genere di sistema. Le classiche tecniche di sicurezza informatica, che prevedono la difesa del perimetro della rete aziendale e assumono che gli elementi interni della rete siano fidati, si stanno rivelando inefficaci visti i cambiamenti dei sistemi informatici e dell'uso che ne viene fatto. Lavoro in mobilità, politiche di "Bring Your Own Device", cloud ibridi e Internet of Things sono tutti cambiamenti che hanno reso le vecchie tecniche di cybersicurezza inadeguate alla protezione di sistemi informatici moderni.

In questo panorama è emerso il *paradigma Zero Trust*, che afferma che tutto il traffico di rete non è attendibile [20], non solo quello proveniente dall'esterno della rete. Ciò implica la necessità di non limitarsi a controllare il perimetro della rete, ma di estendere le verifiche di sicurezza a tutte le richieste rivolte ai propri server, indipendentemente dalla loro provenienza.

L'accesso ad ogni risorsa deve essere prima autenticato e autorizzato [11], con le modalità che risultano più convenienti nella specifica situazione. Esistono infatti diverse tipologie di architetture che implementano il paradigma Zero Trust e numerosi modelli che descrivono come debbano interagire i componenti incaricati di autenticazione, autorizzazione e gestione degli accessi alle risorse. Una tipologia di architettura Zero Trust di particolare interesse è la microsegmentazione, che prevede l'assegnazione delle risorse a segmenti di rete isolati in base ai loro requisiti di accesso. Implementare un'architettura Zero Trust basata su microsegmentazione può essere fatto in vari modi, dando la possibilità di costruire questa architettura in ambienti di varia natura, da cloud pubblici a cloud ibridi a reti con dispositivi IoT.

Una delle forme più elastiche di microsegmentazione è quella agent-based, utilizzabile anche in cloud ibridi e multicloud. Esistono numerosi strumenti per la sua implementazione, che offrono configurazione e gestione centralizzata della rete. In

questo modo gli strumenti garantiscono un buon livello di comodità d'uso.

Il problema degli approcci seguiti da questi strumenti è che introducono un punto unico di fallimento e vulnerabilità: problemi al controller centralizzato hanno conseguenze sull'intera rete, con il rischio che sicurezza, scalabilità e robustezza non siano garantite. In questa tesi viene proposto un approccio alla microsegmentazione che si differenzia da quelli diffusi ad oggi per l'uguale importanza che dà a scalabilità, robustezza e facilità di configurazione. Abbandonando l'idea di usare un controller centralizzato diventa infatti possibile aumentare la scalabilità e la robustezza dei sistemi usati per la microsegmentazione. Sorgono però complessità in merito alla facilità di configurazione, che l'approccio proposto mira a risolvere. Questo approccio si basa sull'uso di overlay network, per permetterne l'applicazione anche in cloud ibridi e multicloud, che sono ambienti sempre più diffusi. L'usabilità di questo approccio verrà valutata tramite la realizzazione di una *proof of concept*, mentre scalabilità e robustezza saranno oggetto di una estesa serie di test.

Nel Capitolo 2 vengono fornite le basi teoriche, riguardanti architetture Zero Trust e il principio Policy as Code, necessarie alla comprensione del lavoro svolto per questa tesi. Nel Capitolo 3 viene introdotto l'approccio tema di questa tesi, dopo aver presentato le caratteristiche generali della microsegmentazione. Il Capitolo 4 approfondisce gli approcci attualmente applicati dai principali strumenti per la microsegmentazione, confrontandoli con l'approccio proposto. Viene inoltre descritta con maggiore dettaglio l'architettura dello strumento che si desidera realizzare per implementare l'approccio alla microsegmentazione basato su overlay network. Gli strumenti utilizzati per l'implementazione sono presentati nel Capitolo 5, mentre come essa è stata realizzata è oggetto del Capitolo 6. I test svolti per valutare quanto prodotto sono descritti nel Capitolo 7, i loro risultati sono riportati e analizzati nel Capitolo 8. Il Capitolo 9 conclude questa tesi con una valutazione complessiva del lavoro svolto e delle proposte di possibili lavori futuri.

2 Moderni approcci alla cybersicurezza

Le minacce informatiche sono aumentate negli ultimi anni, sia in numero sia in complessità, spingendo diversi enti ed aziende a cercare di migliorare la sicurezza informatica delle proprie infrastrutture. Un impulso particolare alla ricerca di nuove tecniche di sicurezza è stato dato dal governo degli Stati Uniti d'America, che nel 2013 diede ordine al National Institute of Standards and Technology (NIST) di creare un insieme di politiche e linee guida volontarie per aiutare a sviluppare il framework di sicurezza informatica degli Stati Uniti [28]. La proposta del NIST fu il cosiddetto *modello Zero Trust*, in base a cui sono sviluppate le *architetture Zero Trust*. Questo modello è stato formalizzato dal NIST, ma la proposta iniziale del concetto di Zero Trust è da attribuirsi a John Kindervag.

Contemporaneamente alle innovazioni più tecniche, la necessità di migliorare la sicurezza non solo dei sistemi informatici ma anche dei loro processi di creazione ha portato all'introduzione di nuove metodologie di sviluppo, come DevSecOps. Principi come Configuration as Code e Policy as Code, esistenti da anni, hanno acquisito particolare popolarità per i vantaggi che portano in combinazione con tali metodologie.

Questo capitolo presenterà una panoramica di questi argomenti, in modo da fornire un'idea di cosa siano architetture Zero Trust, Configuration as Code e Policy as Code.

2.1 Architetture Zero Trust

Le architetture Zero Trust, o ZTA, sono, secondo il NIST, architetture di sicurezza informatica aziendale basate sui principi Zero Trust e modellate per prevenire furti

di dati e limitare movimenti laterali interni [26]. La rapida crescita dell'Internet of Things e delle piattaforme di cloud computing sta rivelando l'inefficacia delle classiche infrastrutture di sicurezza, sia per la protezione di infrastrutture critiche sia per la protezione di comuni aziende, portando a vedere le ZTA come l'architettura di sicurezza da scegliere per tali infrastrutture [29]. Il cloud computing e il lavoro da remoto sono esempi del perché le aziende debbano espandere il loro perimetro digitale e adattarsi alle tendenze contemporanee [2].

Non esiste una singola ZTA, bensì una serie di architetture, realizzate con tecnologie più o meno diverse, che vengono definite tali per le loro caratteristiche; la cosa che le accomuna è seguire lo stesso insieme di principi guida per flusso di lavoro, design del sistema ed operazioni.

2.1.1 Zero Trust

Il concetto di Zero Trust e i principi ad esso associati si sono evoluti dal concetto di deperimetrizzazione [26], allontanandosi dall'idea di fiducia basata sulla posizione all'interno della rete dell'utente (o più in generale del dispositivo) e verso l'idea di valutare il grado di fiducia per ogni singola transazione effettuata. La definizione data dal NIST di Zero Trust è:

> *Zero Trust è un paradigma di sicurezza informatica focalizzato sulla protezione delle risorse e sul presupposto che la fiducia non è mai concessa implicitamente ma deve continuamente essere valutata.*

In base a questa definizione un prerequisito dell'applicazione del paradigma Zero Trust è l'individuazione di tutte le risorse che si desidera difendere. Questo compito è in realtà fondamentale nella progettazione di qualsiasi architettura di sicurezza, anche non Zero Trust. Il fatto che la fiducia che il sistema ha nei confronti di una richiesta di accesso alle risorse, e quindi nei confronti dell'utente che effettua la richiesta, debba essere continuamente valutata ha conseguenze sull'invasività delle componenti di sicurezza nei confronti del sistema aziendale. Per poter valutare le richieste con origine dentro la rete aziendale, infatti, gli elementi che svolgono i controlli non potranno essere posizionati ai margini della rete. I componenti di controllo devono così essere posizionati all'interno della rete aziendale, vicino alla risorse da proteggere, in modo da avere visibilità su tutte le richieste effettuate nei

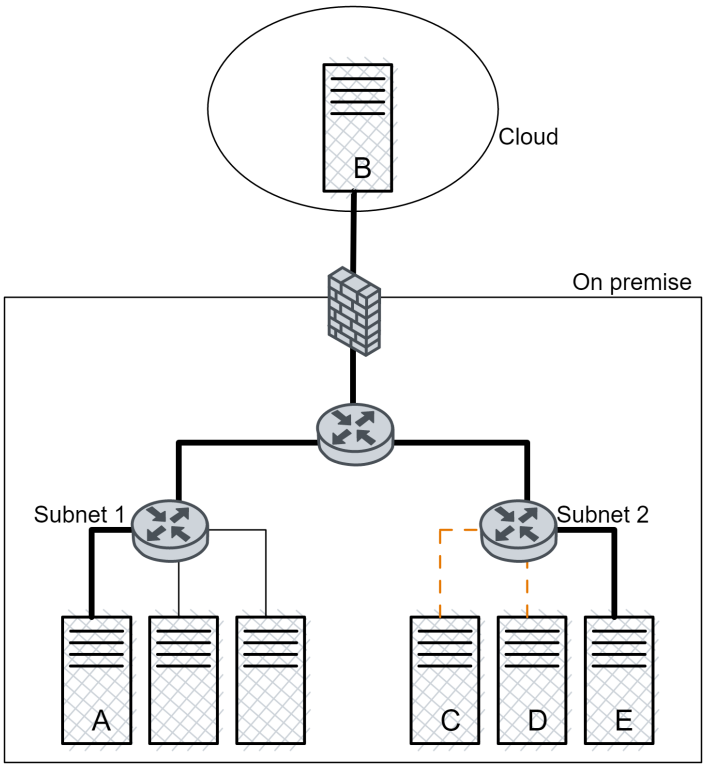


Figura 2.1: Rappresentazione delle comunicazioni tra server in cloud e server on premise, mettendo in evidenza il traffico nord-sud (in grassetto) e il traffico est-ovest (tratteggiato).

confronti di tali risorse. Può sorgere l'idea che in questo modo i sistemi di sicurezza diventino molto invasivi, ma ciò è una conseguenza del fatto che i progetti attuali partono da reti insicure e si limitano a sovrapporre alle reti esistenti un numero sempre maggiore di controlli nel tentativo di creare una parvenza di rete sicura [20]. Se la progettazione delle reti aziendali partisse già con l'idea di inserire questi componenti di sicurezza al loro interno non li si considererebbe aggiunte invasive ma componenti naturali della rete.

Nelle architetture di sicurezza tradizionali il traffico che viene controllato è il cosiddetto *traffico nord-sud*, che in Figura 2.1 consiste nel traffico che si svolge tra i server A e B e i server A ed E. Un flusso nord-sud di traffico fa riferimento a traffico che attraversa uno o più punti di aggregazione per raggiungere la destinazione [24], quindi il traffico tra datacenter ma anche il traffico che viaggia nello stesso datacenter ma tra sottoreti diverse. Nelle architetture Zero Trust invece oltre al traffico nord-sud viene controllato il *traffico est-ovest*, ovvero il traffico tra dispositivi della stessa sottorete, come i server C e D in Figura 2.1.

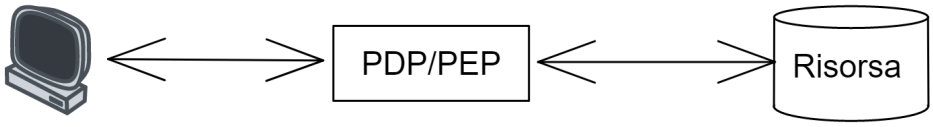


Figura 2.2: Gestione degli accessi Zero Trust.

Il funzionamento logico di un'architettura che segua il paradigma Zero Trust è rappresentato in Figura 2.2: ogni richiesta verso le risorse da proteggere, che sia parte del traffico est-ovest o del traffico nord-sud, passa attraverso un *policy enforcement point*, che inoltra o blocca la richiesta in base alla valutazione di un *policy decision point*. Il policy decision point effettuerà la sua valutazione in base a una policy di sicurezza e a delle informazioni sul contesto in cui è effettuata la richiesta; ad esempio, richieste per le informazioni sui clienti di un'azienda potrebbero essere autorizzate dal policy decision point solo se fatte da un dipendente del reparto commerciale da un computer aziendale. Ne consegue che sia sempre necessario autenticare le richieste, essendo le informazioni sulla loro provenienza importanti per la valutazione del policy decision point. Al cuore di una ZTA ci sono quindi autenticazione e controllo degli accessi [29].

I principi Zero Trust base che caratterizzano una ZTA sono:

- Tutte le sorgenti di dati e i servizi di computazione sono considerati risorse: ogni dispositivo della rete aziendale, o in grado di connettersi alla rete aziendale, è considerato una risorsa e come tale deve essere gestito; ciò implica controllarne gli accessi e monitorarne il comportamento, in accordo con gli altri principi Zero Trust. Questo principio evidenzia la necessità di definire un elenco di tutte le risorse aziendali, per poter definire quali sia necessario difendere e quali no.

- Tutte le comunicazioni vengono rese sicure indipendentemente dalla posizione nella rete: richieste di accesso provenienti dall'esterno e richieste di accesso provenienti dall'interno del perimetro aziendale sono trattate allo stesso modo, ponendo gli stessi requisiti di sicurezza. Prendendo ad esempio la rete con difesa perimetrale rappresentata in Figura 2.3, le uniche richieste di cui viene valutata l'autorizzazione sono quelle rappresentate con linee non tratteggiate; se essa fosse una ZTA le richieste da autenticare e autorizzare sarebbero tutte

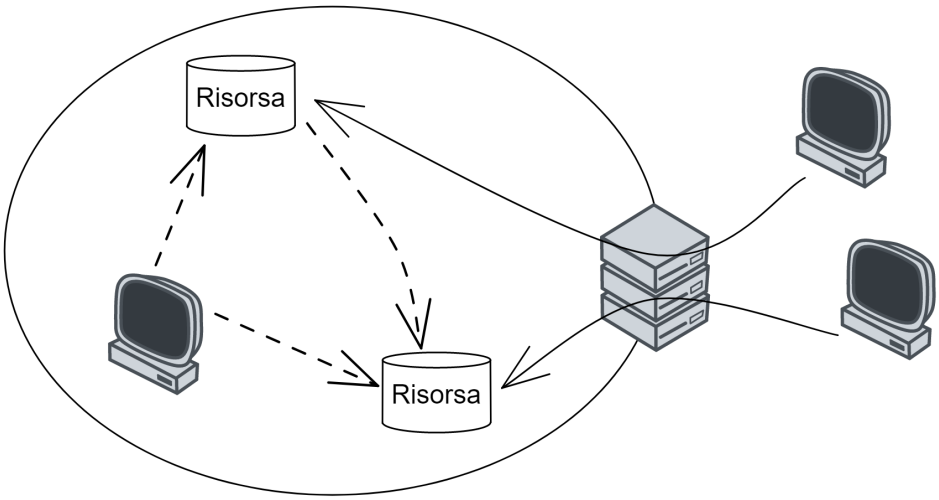


Figura 2.3: Schema di rete con difesa perimetrale in cui le comunicazioni rappresentate con linee non tratteggiate vengono sempre controllate, mentre le comunicazioni rappresentate con linee tratteggiate non vengono controllate.

quelle mostrate, quindi anche quelle rappresentate da linee tratteggiate. Una richiesta quindi non è automaticamente considerata attendibile se proviene da una posizione considerata sicura: l'origine della richiesta, che deve essere autenticata, potrà essere parte del processo di autorizzazione, ma non dovrà esserne l'unico elemento. Tutte le comunicazioni inoltre dovrebbero essere svolte nella maniera più sicura possibile, garantendo confidenzialità e integrità: un'azione da intraprendere a questo fine è cifrare tutte le comunicazioni. Si assume che tutte le comunicazioni sulla rete siano una minaccia, finché non sono autenticate, autorizzate e protette [28]. Una conseguenza di questo principio è che, rispetto ad un'architettura di sicurezza tradizionale basata sulla difesa del perimetro della rete, in una ZTA molte più risorse di calcolo dovranno essere spese sull'autenticazione e autorizzazione delle richieste.

- L'accesso alle singole risorse aziendali è concesso sessione per sessione: ogni richiesta viene autenticata e autorizzata, in modo da concedere i privilegi minimi necessari al completamento dell'operazione. Una sessione di autenticazione e autorizzazione avviene per ogni singola risorsa. L'autenticazione deve essere rinnovata periodicamente, per essere certi che le prove dell'identità rilasciate dal gestore delle identità non siano state compromesse; l'autorizzazione deve essere ripetuta per ogni singola risorsa, in quanto l'accesso a una specifica

risorsa in un dato momento non garantisce l'accesso a un'altra risorsa in un altro momento.

- L'accesso alle risorse è determinato da una policy dinamica, che considera lo stato della risorsa e degli attributi ambientali e comportamentali: l'autorizzazione per l'accesso a una risorsa viene negata o concessa considerando l'identità del richiedente, il suo comportamento, la risorsa in questione, l'ambiente e il contesto in cui è effettuata la richiesta. Gli attributi ambientali includono, tra gli altri, la posizione dell'utente nella rete, l'orario, se siano in corso attacchi o meno. Alcuni esempi di attributi comportamentali sono statistiche sul comportamento dell'utente, sul suo dispositivo e le variazioni rispetto al comportamento registrato in sessioni precedenti. La raccolta di questi dati, che diventano ulteriori risorse da proteggere, è anch'essa un'attività che contribuisce ai maggiori costi, in termini di risorse di calcolo, di una ZTA rispetto a un'architettura di sicurezza perimetrale.

- L'impresa monitora e misura l'integrità e la posizione di sicurezza di tutti i beni posseduti e associati: monitorare continuamente lo stato di risorse e client permette di individuare elementi con vulnerabilità non corrette ed elementi compromessi, consentendo di trattarli diversamente dagli elementi senza apparenti criticità. È buona norma applicare questo principio anche al di fuori di una ZTA, per poter individuare le vulnerabilità e risolverle prontamente.

- Autenticazione e autorizzazione per tutte le risorse sono dinamiche e strettamente applicate prima che l'accesso sia permesso: la fiducia da concedere ad ogni richiesta è continuamente rivalutata, richiedendo di effettuare ogni volta autenticazione e autorizzazione. La frequenza con cui ciò avviene dipende dalla policy di sicurezza applicata: potrebbe essere necessario, per esempio, ad ogni nuova richiesta, dopo una certa quantità di tempo o in seguito alla modifica della risorsa. Per poter gestire al meglio questo processo occorre che la ZTA si appoggi a un sistema di gestione di identità, credenziali e accessi (ICAM).

- L'impresa raccoglie quante più informazioni possibili sullo stato corrente dei beni, dell'infrastruttura di rete e delle comunicazioni, usandole per migliorare la propria posizione di sicurezza: la raccolta di informazioni sullo stato del sistema aziendale è utile per individuare possibili problemi di sicurezza, per migliorare le policy di sicurezza e la loro applicazione, per fornire un contesto alle richieste di accesso alle risorse.

Questi principi sono agnostici da specifiche tecnologie, permettendo quindi di implementare una ZTA nei modi che risultano più vantaggiosi alle specifiche aziende. Come già accennato, seguire questi principi nell'implementazione di un'architettura di sicurezza porta a dover dedicare molte più risorse ad autenticazione, autorizzazione e monitoraggio rispetto a un'architettura che si limiti a difendere il confine della rete aziendale, dato che l'area su cui agire è molto più ampia. Il consumo di risorse può quindi diventare un fattore importante nella scelta delle tecnologie da utilizzare per la realizzazione di una ZTA. Questo potrebbe implicare costi maggiori per le aziende nel caso esse implementino una ZTA, ma per il momento il punto di vista economico nella ricerca sul paradigma Zero Trust è stato trascurato [4], quindi non ci sono valutazioni in merito. È bene però ricordare che un sistema di sicurezza più costoso ma anche più efficace potrebbe portare a una spesa minore per il recupero dagli incidenti informatici e quindi a un risparmio sul lungo periodo.

2.1.2 Composizione di una ZTA

La struttura logica di una ZTA è rappresentata in Figura 2.4, in maniera più ricca rispetto alla Figura 2.2. In questa immagine sono chiaramente indicati il piano di controllo e il piano dei dati, che è buona pratica siano separati se non fisicamente almeno logicamente. Il piano dei dati è usato dalle applicazioni aziendali per comunicare, il piano di controllo è usato dai componenti logici della ZTA e ha accesso illimitato al piano dei dati. I componenti logici principali sono:

- Policy engine, o PE: componente che decide se una richiesta debba essere soddisfatta o meno, valutando la policy di sicurezza del sistema e prendendo in considerazione informazioni come l'identità del richiedente e i risultati del monitoraggio dell'infrastruttura. È buona norma che tutte le decisioni prese vengano registrate dal policy engine, per poter effettuare auditing ed

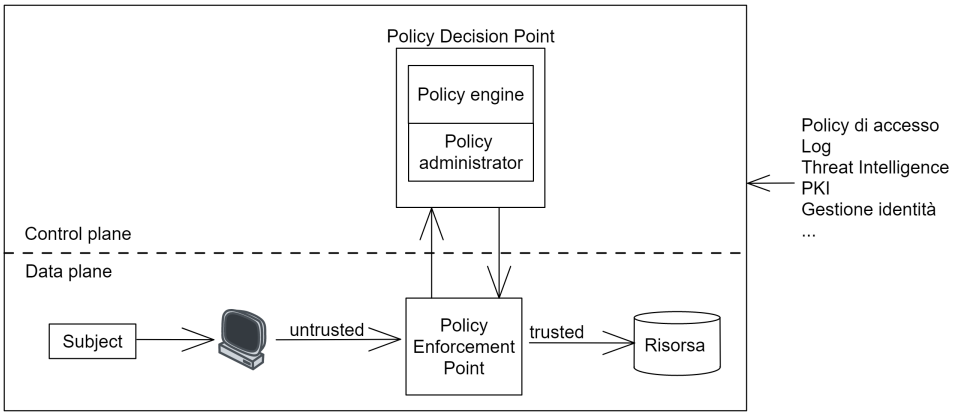


Figura 2.4: Schema logico di una ZTA.

indagini in caso di incidenti. La valutazione avviene tramite un *trust algorithm*, che può essere singolare, ovvero valutare la singola richiesta, o contestuale, ovvero valutare la storia delle richieste ricevute, e il cui risultato può essere un semplice sì o no (algoritmo basato su criteri) oppure un livello di fiducia da assegnare alla richiesta (algoritmo basato su punteggio).

- Policy administrator, o PA: componente che si occupa di controllare l'applicazione della decisione del policy engine, generando token o credenziali di sessione se necessario. Non applica direttamente la decisione, che è invece compito dei policy enforcement point. Il suo ruolo è quello di dare ordini ai policy enforcement point in modo che applichino correttamente le decisioni del policy engine.

- Policy enforcement point, o PEP: componente che media la comunicazione tra richiedente e risorse, monitorandola e terminandola se necessario. Esegue i comandi del policy administrator. È spesso suddiviso in due sottocomponenti, uno in esecuzione sul dispositivo richiedente e uno in esecuzione lato risorse. Definisce il confine della *trust zone* che ospita le risorse.

Policy engine e policy administrator insieme compongono il policy decision point, o PDP. Durante il processo di autorizzazione è essenziale che policy enforcement point e policy decision point possano comunicare.

Questa struttura logica generale può essere implementata in modi diversi in base alla situazione e alle risorse disponibili. I componenti logici potrebbero essere aggregati insieme, o le loro funzionalità suddivise tra più componenti concreti. In

base a come questi componenti sono aggregati, a quali tecniche sono utilizzate per realizzarli e a cosa si utilizza come fonte principale delle regole della policy di sicurezza si distinguono tre principali approcci all'implementazione di una ZTA:

- ZTA basata su gestione avanzata dell'identità.
- ZTA basata su microsegmentazione.
- ZTA basata su infrastruttura di rete e perimetri definiti tramite software.

Per poter definire un'architettura "ZTA" non è necessario che presenti tutti i componenti sopra citati o che segua precisamente uno di questi tre approcci: l'importante è che vengano seguiti i principi Zero Trust.

ZTA basata su gestione avanzata dell'identità

La caratteristica principale di una ZTA basata sulla gestione avanzata dell'identità è l'uso dell'identità degli attori come componente chiave della creazione delle policy di sicurezza, combinata con altri attributi ambientali e comportamentali. Questo approccio è spesso usato in situazioni in cui persone (o dispositivi) non appartenenti all'azienda devono accedere alla rete aziendale; la rete viene lasciata aperta, in modo che tutti possano connettersi, come mostrato in Figura 2.5, ma l'accesso alle risorse viene riservato alle identità con i permessi necessari. Il lato negativo di questo approccio è che la rete è liberamente accessibile, con il rischio che attori malintenzionati possano analizzarne la struttura o sfruttarla per lanciare attacchi di tipo Denial of Service. Si possono prevedere dei meccanismi di autenticazione e autorizzazione anche per l'accesso alla rete, ma se si ha la necessità di permettere a degli ospiti di accedere alla rete ciò potrebbe risultare complicato o sconveniente.

Questo approccio è anche adeguato per applicazioni e servizi cloud-based, come molte offerte SaaS, che non permettono di utilizzare componenti di sicurezza operati dall'azienda.

Il focus di questo approccio è sulle risorse, con la rete che rappresenta un mezzo di collegamento tra esse piuttosto che l'elemento centrale da difendere. Resta comunque importante monitorare la rete per individuare situazioni anomale.

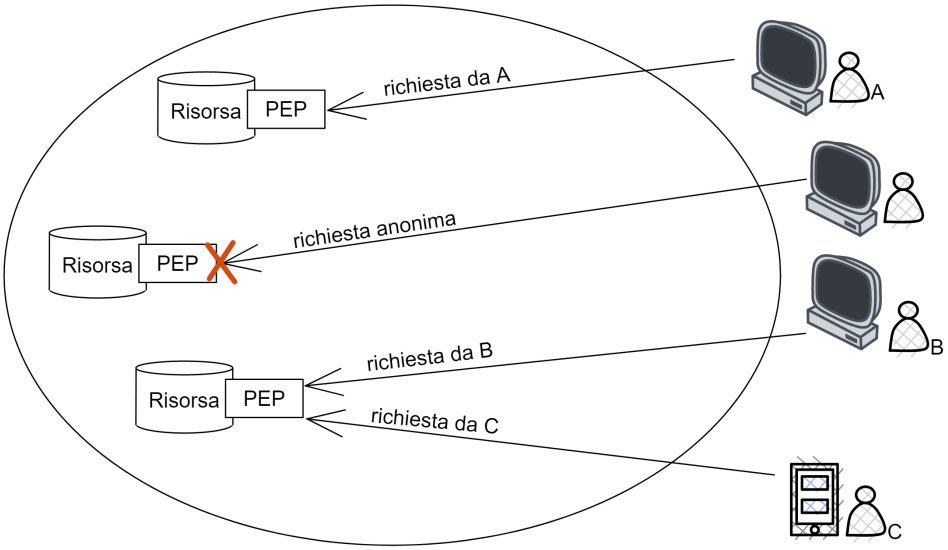


Figura 2.5: Schema di una ZTA basata su gestione avanzata dell'identità.

ZTA basata su microsegmentazione

La *microsegmentazione* è una tecnica di sicurezza emergente che separa delle reti fisiche in microsegmenti logici isolati [3]. Questo concetto può essere esteso anche a reti virtuali, dato che nel cloud moderno quello che appare all'utente come una rete fisica potrebbe in realtà essere una rete virtuale. Implementare una ZTA basata su microsegmentazione, come quella rappresentata in Figura 2.6, significa quindi posizionare risorse, singolarmente o in gruppo, su segmenti di rete separati e protetti da un gateway o simile strumento di sicurezza, come un *next generation firewall*. La microsegmentazione può essere anche agent-based, realizzata usando agenti software o firewall eseguiti sugli endpoint stessi.

Nella microsegmentazione i gateway o gli strumenti alternativi utilizzati fungono da PEP, a volte venendo associati a un agente client side. La segmentazione della rete dovrebbe essere definita in base alle criticità e ai requisiti aziendali [28]: due risorse con gli stessi requisiti di accesso e frequentemente contattate insieme potrebbero essere posizionate nello stesso microsegmento, mentre risorse particolarmente sensibili potrebbero essere isolate in un microsegmento a loro dedicato. L'isolamento delle risorse deve essere realizzato applicando delle policy di sicurezza basate sul principio del minimo privilegio, che possano limitare l'abilità di un attaccante di eseguire dei movimenti laterali. La granularità della policy di sicurezza dipenderà dalla quantità di risorse contenute in ogni microsegmento: più risorse si troveranno

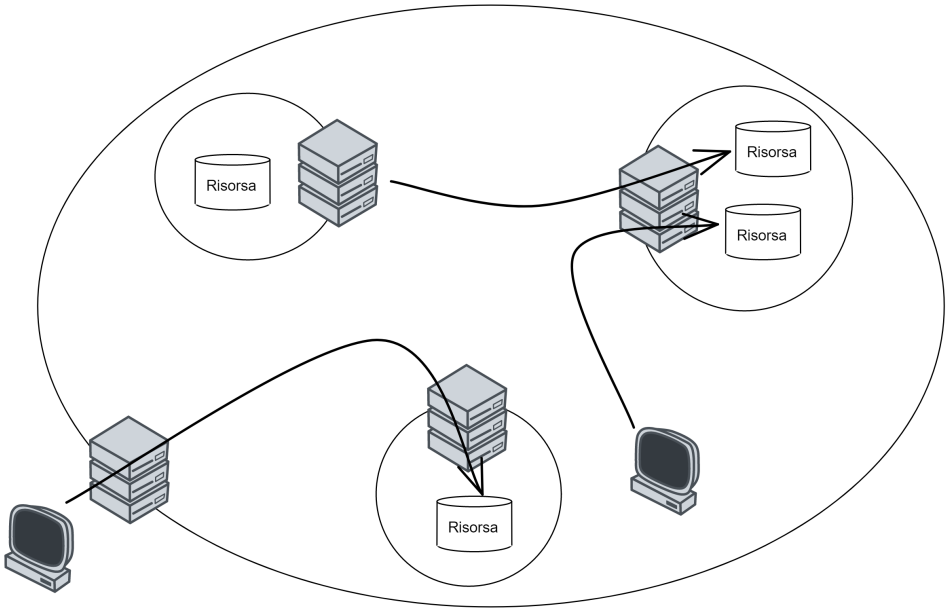


Figura 2.6: Schema di una ZTA basata su microsegmentazione.

in un singolo segmento, maggiore sarà la granularità.

Una necessità fondamentale da considerare nel definire una ZTA è che il comportamento dei PEP, ovvero quale traffico sia permesso e quale no, possa variare rapidamente per rispondere a minacce o cambi di flusso di lavoro. Per realizzare ZTA basate su microsegmentazione è possibile utilizzare gateway non avanzati e firewall stateless, ma l'impossibilità di riconfigurarli rapidamente incrementa i costi di gestione, oltre a rendere difficile l'adattamento ai cambiamenti. Nel caso di microsegmentazione agent-based però su alcuni dispositivi potrebbe non essere disponibile la potenza di calcolo necessaria a supportare queste funzioni; nel caso si voglia realizzare questo approccio implementativo occorre quindi tenere in considerazione le tipologie di dispositivi presenti nella propria rete.

ZTA basata su infrastruttura di rete e Software Defined Perimeter

Una possibile implementazione di ZTA prevede l'uso di un *overlay network*, ovvero di una rete virtuale o logica costruita sopra un'altra rete [25], fisica o virtuale. Per questa tipologia di ZTA si parla anche di *software defined perimeter*, o SDP.

Il controller della rete funge da policy administrator, riconfigurando la rete in base alle decisioni prese dal policy engine. L'accesso diretto alle risorse continua a essere gestito dai PEP, che sono controllati dal policy administrator. In Figura 2.7

viene mostrato come il controller comunichi con gli elementi, ad esempio client e gateway, che vanno a comporre l'overlay network, indicandogli da chi debbano accettare traffico o meno.

Si può dire che sulla rete virtuale siano possibili solo le comunicazioni permesse dalla policy di sicurezza e che siano i PEP a imporre questo comportamento, bloccando il traffico che non soddisfa i requisiti. In questo modo dal punto di vista degli utenti sarà come se le comunicazioni non permesse non siano proprio possibili.

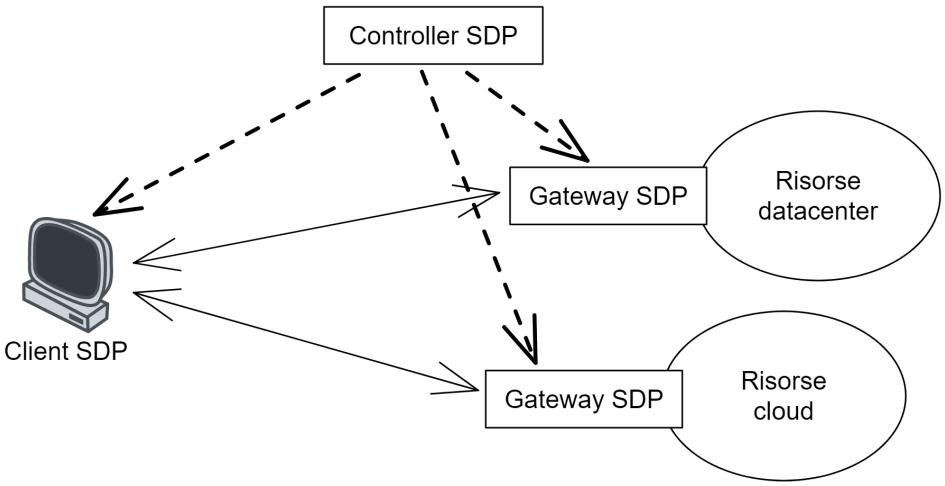


Figura 2.7: Schema di una ZTA basata su SDP.

2.1.3 Possibili variazioni di implementazione di una ZTA

Le tipologie di ZTA descritte in Sezione 2.1.2 sono contraddistinte dal livello a cui si applicano le restrizioni sul traffico: nel caso di ZTA basata su gestione avanzata dell'identità si applicano a livello di singole risorse, mentre nel caso di ZTA basata su SDP si applicano a livello di rete. In base a come invece i singoli componenti, ovvero PA, PE e PDP, sono aggregati e implementati si distinguono variazioni dell'implementazione dell'architettura astratta della ZTA. I possibili modelli di implementazione identificati dal NIST sono quattro, uno basato su dispositivo agente e gateway, uno basato su enclave, uno basato su portale delle risorse, uno basato su sandboxing.

Come per le tipologie di ZTA in Sezione 2.1.2, non sempre la distinzione tra i vari modelli è chiara: ci potrebbero essere implementazioni che richiamano più di

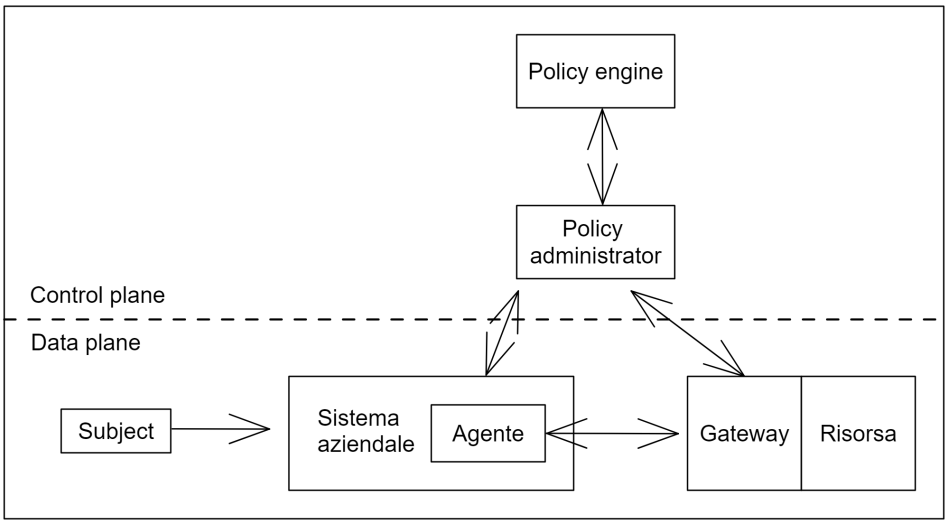


Figura 2.8: Schema di una ZTA basata su dispositivo agente e gateway.

un modello, rendendo difficile classificarle. All'interno di una stessa ZTA inoltre possono essere utilizzati più modelli, in base alle specifiche necessità dell'azienda.

Modello basato su dispositivo agente e gateway

Nel modello basato su dispositivo agente e gateway il policy enforcement point è suddiviso in due parti, come mostrato in Figura 2.8. Una parte si trova in esecuzione sul dispositivo che richiede di accedere alle risorse, mentre l'altra, solitamente un gateway, si trova direttamente davanti alla risorsa da difendere. Il gateway funge da proxy per la risorsa e comunica con il policy administrator. L'agente software in esecuzione sul dispositivo client è responsabile di indirizzare il traffico al giusto gateway, dopo aver creato una connessione con esso all'inizio della sessione di utilizzo della risorsa. In caso di cambiamento delle autorizzazioni, la connessione viene distrutta. Questo tipo di modello è quello che garantisce il maggior controllo delle risorse, ma funziona bene solo quando l'azienda ha il pieno controllo dei dispositivi che dovranno accedere alla risorse.

Un esempio di implementazione concreta di questo modello è BeyondCorp[1] di Google, che utilizza il browser Google Chrome come agente lato client.

[1]https://cloud.google.com/beyondcorp

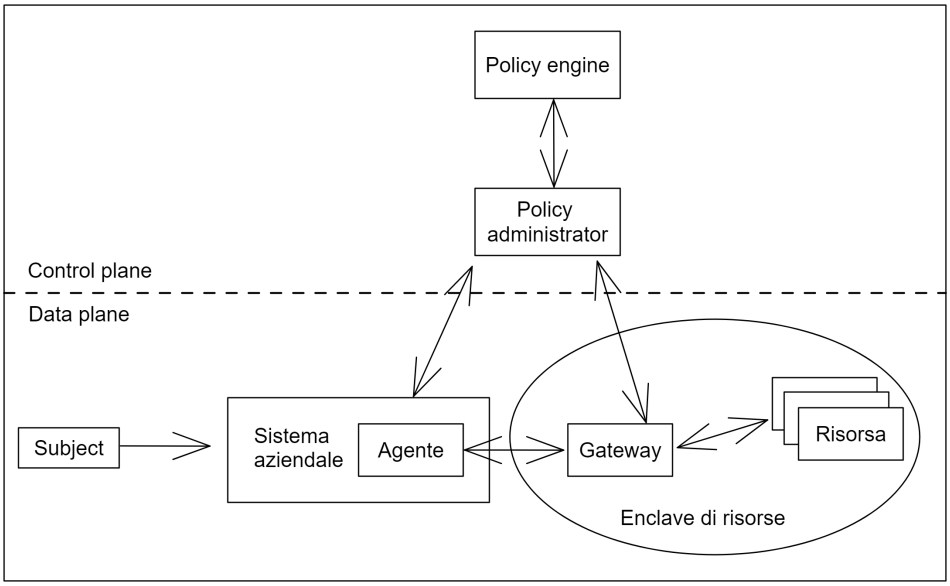


Figura 2.9: Schema di una ZTA basata su enclave.

Modello basato su enclave

Il modello basato su enclave funziona in modo simile al modello basato su dispositivo agente e gateway, ma con una diversa granularità: la differenza tra questi due modelli è che, mentre nel modello con agente e gateway ogni gateway gestisce una singola risorsa, nel modello basato su enclave ogni gateway difende risorse multiple. Confrontando Figura 2.8 e Figura 2.9 si può vedere la somiglianza tra i modelli, che si distinguono solo per il numero di risorse che il gateway deve gestire. Il gateway è posizionato ai margini dell'enclave di risorse, quindi la granularità della policy è più grossa rispetto a quella del precedente modello. Un caso in cui questo modello è utile è quando ci sono risorse che operano come una singola entità. Talvolta le risorse di un intero cloud o datacenter on premise possono andare a formare una singola enclave; questo però contrasta con la necessità di definire policy a granularità fine.

Modello basato su portale delle risorse

Il modello basato su portale delle risorse è logicamente molto simile ai modelli già descritti, ma il policy enforcement point è un singolo componente e non è spezzato in due parti. Come raffigurato in Figura 2.10 esso è comunque un gateway che si pone come intermediario tra origine delle richieste e risorse. Il vantaggio di questo

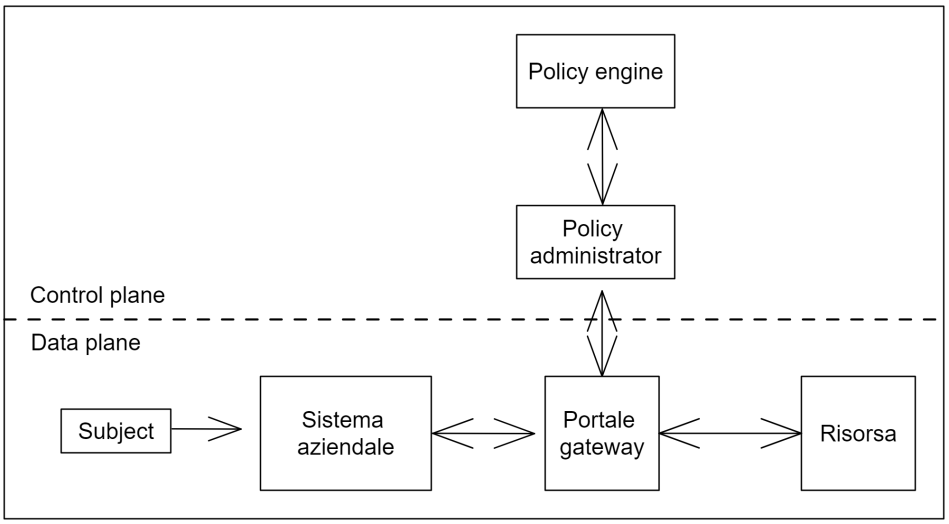


Figura 2.10: Schema di una ZTA basata su portale delle risorse.

modello è che non richiede particolari configurazioni o software in esecuzione sul dispositivo client, permettendo quindi politiche di tipo "Bring Your Own Device", o BYOD; ha però il difetto di poter utilizzare informazioni ridotte in merito al dispositivo che sta richiedendo l'accesso al momento della valutazione della richiesta. Non necessitando di software particolari per accedere alle risorse, inoltre, questo modello permette ad eventuali attaccanti di testare l'accesso alle risorse o di tentare attacchi DoS. In alcune situazioni questo modello potrebbe essere confuso con il modello basato su dispositivo agente e gateway. Nel caso si abbia una web app che gestisce autenticazione e autorizzazione nei confronti del gateway si potrebbe pensare di avere un'implementazione del modello basato su dispositivo agente, in quanto si avrebbero due componenti del PEP: la web app, ovvero l'agente, e il gateway. In realtà però la web app potrebbe non essere strettamente necessaria per l'accesso risorse, inoltre essa non può avere il controllo del dispositivo su cui è in esecuzione: queste caratteristiche spingerebbero a identificare il modello seguito nell'implementazione in quello basato su portale delle risorse. La differenza tra i vari modelli quindi è abbastanza sottile.

Modello basato su sandboxing delle applicazioni del dispositivo

Nel modello basato su sandboxing delle applicazioni del dispositivo ogni applicazione autorizzata a contattare il policy enforcement point è contenuta in una sandbox. Le applicazioni non contenute in una sandbox, quindi non fidate, non

riusciranno a contattare il policy enforcement point, come si può osservare in Figura 2.11, anche se sullo stesso dispositivo di un'applicazione fidata. In questo modo le applicazioni che potrebbero effettivamente avere accesso alle risorse sono isolate dal resto del dispositivo, evitando di essere influenzate dalle sue vulnerabilità. Il problema di questo modello è che occorre gestire tutte le sandbox e garantire che esse siano sicure.

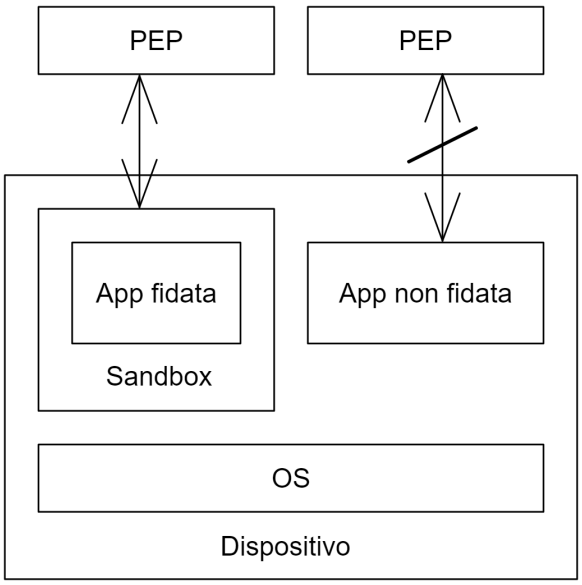


Figura 2.11: Schema di una ZTA basata su sandboxing delle applicazioni del dispositivo.

2.1.4 Debolezze di una ZTA

Una ZTA adeguatamente implementata e mantenuta, indipendentemente dal modello su cui si basa, risulta, se associata a buone pratiche e comportamenti, più sicura di altre architetture, ma non completamente esente da rischi. Alcune minacce assumono connotazioni particolari quando riguardano una ZTA. Tra le principali minacce e problematiche a cui prestare attenzione quando si implementa una ZTA ci sono:

- Sovversione del processo decisionale di policy engine e policy administrator, il cui corretto funzionamento è essenziale per la sicurezza del sistema. Anomalie nel loro comportamento determinano l'insicurezza dell'intera rete. Policy

engine e policy administrator sono quindi dei punti critici dell'architettura, a cui va riservata particolare attenzione, sia perché potrebbero essere singoli punti di vulnerabilità, sia perché potrebbero essere singoli punti di fallimento.

- Interruzione di rete, che potrebbe rendere impossibili le comunicazioni tra policy administrator e policy enforcement point, bloccando i processi di autorizzazione degli accessi e di conseguenza tutte le attività dell'impresa. Naturalmente il blocco delle comunicazioni tra altre applicazioni del sistema aziendale potrebbe avere conseguenze simili, ma come già accennato nel caso del policy administrator il suo isolamento è sicuro di portare a un blocco di tutte le attività.

- Furto di credenziali, che in una ZTA ben implementata però ha probabilità minori di permettere movimenti laterali. L'uso di un trust algorithm contestuale permette inoltre di riconoscere rapidamente pattern di accesso anomali e bloccare gli accessi per il client sospetto. I trust algorithm contestuali però sono più costosi dei trust algorithm singolari, quindi occorre effettuare una valutazione dei costi, dei rischi e dei benefici delle varie tecniche e protezioni che si desidera applicare.

- Opacità del traffico sulla rete, che potrebbe rendere problematico analizzare il traffico e individuare la presenza di un attaccante. In una ZTA cifrare il traffico è buona norma, in quanto aiuta a preservare la confidenzialità delle comunicazioni nel caso ci sia un attaccante infiltrato nella rete aziendale, ma ne complica il monitoraggio e l'analisi. In una rete con protezione perimetrale la difficoltà di analizzare il traffico si presenta solo per le richieste cifrate provenienti dall'esterno della rete; in una ZTA questa difficoltà si presenta per ogni singola richiesta.

- Protezione delle informazioni sul sistema e sulla rete, generate dai processi di monitoraggio. Queste informazioni, essenziali per le corrette operazioni della ZTA, possono essere di interesse per un attaccante, in quanto permettono di ottenere informazioni sull'architettura del sistema e sui beni dell'azienda. Inoltre, inquinare queste informazioni permetterebbe a un attaccante di alterare il comportamento di eventuali sistemi che usino tali dati per l'apprendimento

automatico. Ad oggi le policy di sicurezza sono spesso generate da simili sistemi automatici, rendendo la loro integrità fondamentale.

- Uso di soluzioni e formati di dati proprietari, che non è un problema unicamente per le ZTA ma che in esse assume un'importanza particolare, visto come i processi di autorizzazione raccolgono dati da molteplici fonti. Dati in formati proprietari possono portare a un *vendor lock-in*, rendendo impossibile sostituire lo strumento che li utilizza in caso di necessità, ad esempio cambi di prezzo delle licenze o attacchi alla catena di approvvigionamento. Un vendor lock-in quindi non è solo un problema di natura economica e di qualità, ma è anche un problema di sicurezza.

- Uso di *Non-person Entities* (NPE), ovvero di intelligenze artificiali o agenti software, nell'amministrazione delle ZTA. Dall'uso di queste entità sorge il rischio che il loro comportamento possa essere manipolato da un ipotetico attaccante. Le NPE inoltre devono autenticarsi nei confronti dei componenti della ZTA per poter svolgere le proprie funzioni amministrative, ma come fare è un problema aperto [26]: spesso questi agenti hanno requisiti di autenticazione meno sicuri che un utente umano, con il rischio che un attaccante possa ottenere le loro credenziali e impersonarli.

- Compromissione degli endpoint, che pur non permettendo movimenti laterali potrebbe comunque causare danni. Le sessioni già autenticate e autorizzate di un endpoint compromesso possono essere sfruttate per effettuare attività limitate, ma comunque dannose [2]. I permessi a disposizione dell'attaccante in queste situazioni sono limitati e non necessariamente quelli di cui esso ha bisogno per raggiungere i propri scopi, ma forniscono comunque un accesso al sistema che può risultare pericoloso per la sua integrità. Ignorare il rischio di compromissione degli endpoint, in particolare se si applicano politiche aziendali di tipo BYOD, può portare a un falso senso di sicurezza.

Nel valutare i miglioramenti che il modello Zero Trust può portare nel panorama della sicurezza informatica attuale occorre considerare che esso è stato pensato con gli attacchi più diffusi ad oggi in mente. In futuro le tipologie di attacchi più diffuse potrebbero cambiare, per prendere di mira proprio le debolezze appena elencate.

2.1.5 Considerazioni sullo stato delle ZTA

Il concetto di ZTA, pur non essendo esattamente nuovo, si sta diffondendo solo negli ultimi anni, anche se ci si aspetta che il mercato per le tecnologie Zero Trust raddoppi entro il 2024 [2]. Diverse aziende hanno già adottato o stanno adottando questo paradigma di sicurezza: Google già da anni ha presentato il suo strumento BeyondCorp, mentre Microsoft ha dichiarato di star implementando un modello di sicurezza Zero Trust per i propri ambienti [18]. Nonostante ciò la ricerca scientifica sulle ZTA, agnostica da specifici venditori e tecnologie, è scarsa; in compenso le offerte commerciali sono molto variegate: ci sono offerte legate a specifiche piattaforme, come BeyondCorp Enterprise di Google che offre di gestire gli accessi al proprio cloud Google in modalità Zero Trust, ci sono offerte generiche come l'offerta di IBM, che propone soluzioni Zero Trust ad-hoc realizzate con i loro strumenti, infine ci sono offerte di specifici strumenti non legati ad alcuna piattaforma, come Illumio[2]. Il mercato è ampio, ma non esente da proposte di dubbia qualità, come ad esempio semplici VPN pubblicizzate come sistemi zero trust completi. Le VPN possono sì essere parte di una strategia Zero Trust, ma non come componente principale. A prova di ciò si prevede che entro il 2023 il 60% delle VPN in uso sarà sostituito da strumenti più propriamente Zero Trust [5]. Non esiste comunque nessun venditore che offra di implementare il paradigma Zero Trust con una singola soluzione; ciò non è in realtà desiderabile, in quanto espone al rischio di vendor lock-in.

Per poter definire la propria architettura Zero Trust però non basta utilizzare determinate tecnologie: molte organizzazioni hanno già degli elementi Zero Trust nella loro infrastruttura [26], quello che serve in questi casi per implementare una completa ZTA sono non solo nuovi strumenti ma anche nuove competenze, un approccio diverso alla sicurezza informatica e una maggiore consapevolezza delle proprie risorse. In questi casi il processo di implementazione avviene gradatamente, portando una ZTA e un'architettura basata sulla protezione del perimetro della rete a coesistere per un periodo di tempo più o meno lungo. Il paradigma Zero Trust dovrebbe essere visto come un'evoluzione delle correnti strategie di sicurezza informatica [26], piuttosto che come un approccio totalmente separato.

[2]https://www.illumio.com/

Le tecnologie e tecniche disponibili per realizzare una ZTA sono numerose, ognuna con i propri punti di forza, debolezze e vulnerabilità. Talvolta, nell'implementazione di una ZTA le aziende sono limitate nella scelta degli strumenti dalla necessità di usare soluzioni compatibili con i sistemi già in uso. Non esistono ancora standard riconosciuti per i protocolli e gli strumenti per ZTA, ma proposte in tal senso sono state avanzate dall'Internet Engineering Task Force e dalla Cloud Security Alliance. Fino a che non ci saranno standard diffusi e ampiamente accettati il rischio di vendor lock-in sarà sempre presente, insieme alla possibilità di dover mantenere strumenti realizzati ad-hoc per rendere compatibili soluzioni proprietarie.

Per l'autenticazione, sia di dispositivi sia di utenti, esistono molteplici tecniche e strumenti, più o meno sicuri. Ormai uno standard per l'autenticazione degli utenti è l'autenticazione a più fattori, che però presenta problemi di usabilità, in quanto spesso richiede un secondo dispositivo. L'autenticazione dei dispositivi vede l'uso di metodi crittografici, che però sono a rischio di sovversione per via dei progressi fatti nel calcolo quantistico [29]. Autenticazione continua e context-aware sono già in uso in molti sistemi, ma presentano alcuni problemi relativi alla loro precisione, ai requisiti che hanno in termini di dispositivi necessari e alla privacy degli utenti. In generale si può dire che l'autenticazione sia un problema risolto ma con margini di miglioramento. Un rischio nel definire schemi di autenticazione sempre più complessi è che ci si trovi a fare i conti con il fenomeno dell'affaticamento da sicurezza, in cui gli utenti finali devono confrontarsi con così tante policy di sicurezza e sfide che la loro produttività è influenzata negativamente [26]. Questo potrebbe essere un ostacolo alla diffusione di meccanismi di autenticazione sicuri e di conseguenza delle ZTA che ne fanno uso. Il rischio è anche che i criteri di sicurezza vengano rilassati per evitare questo affaticamento, riducendo la sicurezza del sistema.

La maggior parte dei framework disponibili ad oggi per il controllo degli accessi sono pensati per l'accesso di utenti umani e se applicati ad ambiti IoT non risultano efficaci. Un sistema di controllo degli accessi idoneo ad essere utilizzato all'interno di una ZTA deve invece essere in grado di usare grandi quantità di dati per valutare se autorizzare una richiesta, deve poter operare in ambienti eterogenei che spaziano da cloud pubblici a cloud privati a reti di dispositivi IoT e deve inoltre essere resistente ad attacchi che mirino a interromperne le funzioni. Usare più framework in combinazione tra loro può aiutare ad ottenere queste caratteristiche, complicando

però la gestione del sistema, in quanto i vari framework avranno meccaniche di funzionamento, formato dei dati e requisiti diversi. La presenza di standard in merito potrebbe facilitare questo compito.

L’uso di microsegmentazione per l’implementazione di una ZTA è concettualmente semplice, ma in pratica presenta numerose sfide [29], che riguardano principalmente la difficoltà del microsegmentare una rete la cui struttura e le cui applicazioni non sono particolarmente idonee alle microsegmentazione. Un esempio di ciò sono le applicazioni monolitiche, che per loro natura non possono essere microsegmentate in maniera efficace.

Il concetto di SDP è recente, ma oggetto di forte interesse. Le offerte disponibili sul mercato che lo applicano non sono ancora completamente mature, ma possono essere usate in ambito industriale senza troppi problemi o complicazioni [2].

Gli strumenti e le tecniche disponibili per l’implementazione di una ZTA sono quindi numerosi, a volte anche già in uso da tempo, ma ciò non semplifica l’opera: combinare questi strumenti in maniera efficace è complesso. Un’ulteriore difficoltà è data dalla necessità poi di gestire e mantenere tutti questi elementi. Occorre poi tenere in considerazione come i lavoratori possano essere impattati da questi cambiamenti nell’infrastruttura aziendale: in mancanza di esperienza e addestramento adeguato l’applicazione del modello Zero Trust potrebbe non portare alcun miglioramento.

2.2 Principi “as Code”

Con *Policy as Code* si intende un approccio alla gestione delle policy in cui le policy sono definite, aggiornate, condivise e applicate usando del codice [34]. Con *Configuration as Code* invece si intende la pratica di gestire i file di configurazione in una repository [8], come appunto se fossero semplice codice.

Nel caso di una ZTA con microsegmentazion agent-based questi due concetti potrebbero sovrapporsi, dato che la configurazione degli agenti software, una questione quindi di Configuration as Code, ha implicazione sugli accessi alle risorse permessi, che è invece una questione di Policy as Code. Inoltre, dato che in alcune forme di microsegmentazione la configurazione di sicurezza è dipendente dalla struttura di rete, si ha che il concetto di Policy as Code si può sovrapporre con quello di

Infrastructure as Code. Con Infrastructure as Code si intende la gestione dell'infrastruttura (reti, macchine virtuali, load balancers e topologia delle connessioni) in un modello descrittivo, che usi lo stesso sistema di versionamento che i team DevOps usano per il codice [32].

Questi tre concetti quindi, seppur con ambiti di applicazione ben separati, nella complessità dei sistemi software moderni tendono a confondersi tra loro. Anche i vantaggi che la loro applicazione porta sono molto simili, ma con piccole variazioni.

2.2.1 Configuration as Code

Da un certo punto di vista è facile confondere Configuration as Code con Policy as Code e Infrastructure as Code: in tutti e tre i casi l'idea di base è di gestire della configurazione come codice. Ciò che differenzia i tre concetti è l'area di applicazione: policy di sicurezza, infrastruttura e applicazioni. Nelle pipeline di sviluppo queste tre tipologie di configurazione vengono gestite in maniera diversa e applicate in momenti diversi.

Nel caso di applicazioni sviluppate in proprio, seguendo il principio di Configuration as Code è consigliato conservare la configurazione in una repository separata da quella del codice dell'applicazione. Questo permette di ottenere diversi vantaggi come:

- Sicurezza: conservare la configurazione in una repository permette di gestire meglio chi vi ha accesso e con che permessi.
- Tracciabilità: usare una repository specifica per le configurazioni rende più facile tracciare i cambiamenti e gli aggiornamenti.
- Gestibilità: i processi di build e deployment specifici per le configurazioni possono essere organizzati in modo completamente indipendente da quelli della relativa applicazione, introducendo anche passi aggiuntivi di validazione e testing.

Anche non riservando una repository per la sola configurazione, ma conservandola nella stessa repository del codice applicativo, si hanno comunque dei vantaggi in termini di tracciabilità e possibilità di verifica della sua correttezza al momento della build e del deploy.

2.2.2 Policy as Code

Il concetto di Policy as Code non è un concetto così recente, dato che si trovano articoli e blog su di esso risalenti ad almeno una decina d'anni fa. Nonostante ciò sembra non essere un concetto così noto al di fuori del suo ambito di utilizzo. Negli ultimi anni comunque la sua popolarità sta crescendo e ci sono molti strumenti che lo applicano. Un esempio particolarmente significativo di applicazione del concetto di Policy as Code è l'Open Policy Agent[3], che utilizza policy scritte in linguaggio Rego. Si tratta di un sistema open source general purpose e che per questo si differenzia da altri sistemi che offrono nativamente la possibilità di definire policy sotto forma di codice, ma risultando compatibili solo con i propri formati proprietari.

Indipendentemente dallo strumento che li applica, i principali vantaggi del seguire i principi di Policy as Code sono:

- Efficienza: quando le policy, di sicurezza o meno, sono scritte sotto forma di codice possono essere distribuite e applicate automaticamente, senza bisogno di una persona che le configuri manualmente nel sistema. Possono essere inoltre definite in un linguaggio meno ambiguo rispetto al linguaggio naturale.

- Velocità: una volta definite, le policy sono applicabili automaticamente, senza dover attendere che qualcuno le applichi manualmente. Per ottenere appieno questo vantaggio è necessario l'uso di sistemi di deploy automatici.

- Visibilità: quando le policy sono definite sotto forma di codice è più semplice controllare quali regole sono applicate in un dato momento, anche da parte di chi non è addetto alla loro gestione e definizione. Se la policy non fosse definita come codice in una repository, infatti, le effettive regole in uso sarebbero visibili solo tramite il software che le applica. È più sicuro e semplice concedere l'accesso in lettura a una repository di codice piuttosto che alla console di controllo di un software.

- Collaborazione: fornire un sistema uniforme e sistematico di gestione delle policy rende più semplice collaborare sulla loro scrittura e validazione. Ciò vale sia per i membri del team addetti alla loro scrittura, sia per tutti gli altri

[3]https://www.openpolicyagent.org/

team che potrebbero avere un interesse ad esaminare le policy, ad esempio gli sviluppatori.

- Accuratezza: definendo le policy come codice e applicandole automaticamente si evita il rischio di configurare il sistema in modo indesiderato, come invece può accadere quando la gestione del sistema è manuale.
- Versionamento: le varie versioni della policy sono conservate nel tempo e in caso di errori o problemi è sempre possibile tornare ad applicare una versione precedente.
- Testing e validazione: come per il normale codice, è possibile testare, validare e fare auditing delle policy scritte come codice, riducendo il rischio di introdurre errori in un ambiente di produzione. Questa possibilità aiuta nel migliorare la posizione di sicurezza del sistema a cui sarà applicata la policy.

Molti di questi vantaggi sono una conseguenza della possibilità di automatizzare le operazioni riguardanti le policy. Vista la crescente applicazione dei principi DevOps nello sviluppo dei sistemi software questo aspetto ha una grande importanza.

2.2.3 Infrastructure as Code

Il concetto di Infrastructure as Code è più diffuso del concetto di Policy as Code, probabilmente perché è un concetto più vecchio e con un più ampio campo di applicazione. Lo scopo di Infrastructure as Code è di poter replicare, in modo consistente, un determinato ambiente, senza dover effettuare modifiche manuali. Gli strumenti di Infrastructure as Code, come Terraform[4] e Puppet[5], prendono come input una configurazione che descrive l'ambiente che si vuole ottenere e, dato il giusto comando di deploy, configurano l'ambiente obiettivo nel modo desiderato. Un principio fondamentale di Infrastructure as Code è l'*idempotenza*, ovvero la proprietà secondo cui il comando di deploy imposta sempre l'ambiente obiettivo nella stessa configurazione, indipendentemente dallo stato di partenza dell'ambiente [32]. Ciò avviene o modificando un ambiente esistente o distruggendo l'ambiente esistente e creandone uno nuovo da zero.

[4]https://www.terraform.io/
[5]https://puppet.com/

L'approccio più adeguato all'Infrastructure as Code è tramite l'uso di una definizione dichiarativa della configurazione, piuttosto che di una dichiarazione imperativa. Questa astrazione, che separa come deve essere l'ambiente da come lo si deve creare, permette al fornitore dell'infrastruttura di effettuare ottimizzazioni durante la creazione dell'ambiente. L'uso di una definizione dichiarativa, inoltre, riduce il debito tecnico legato alla manutenzione di codice imperativo, come sarebbero eventuali script di deploy. Il formato usato per definire l'infrastruttura può essere o un formato proprietario o un formato generico come JSON o YAML.

I vantaggi dell'uso di Infrastracture as Code sono molto simili a quelli dell'uso di Policy as Code: si può fare versionamento dell'infrastruttura, la procedura di creazione dell'ambiente è più accurata, efficiente e rapida, definire l'infrastruttura in modo collaborativo è più facile e si può testare e validare la configurazione prima di applicarla. L'Open Policy Agent, ad esempio, offre delle integrazioni con Terraform per verificare che la configurazione dell'infrastruttura rispetti determinate regole prima di applicarla. Definire un ambiente secondo i principi di Infrastructure as Code ha inoltre il grande vantaggio di renderlo facilmente riproducibile. Questo è un vantaggio sia nel caso si debba ricreare un ambiente di produzione sia nel caso si voglia creare un ambiente di testing identico a quello di produzione.

3 Microsegmentazione

La microsgmentazione, descritta a grandi linee in Sezione 2.1.2, è una delle strategie più efficaci per proteggersi da minacce informatiche [3]: se implementata correttamente, può tenere il passo con l'aumento della natura eterogenea, ibrida e dinamica dell'informatica di oggi [6].

Essa quindi merita di essere una delle opzioni considerate qualora ci si trovi a dover organizzare le difese di una infrastruttura aziendale. In questa tesi si vuole proporre un nuovo approccio alla microsegmentazione che migliori scalabilità e resistenza rispetto alle soluzioni attuali.

Per fare ciò è necessario comprendere appieno le caratteristiche della microsegmentazione, i suoi vantaggi e i suoi problemi. Questo capitolo descrive i principali aspetti della microsegmentazione, a partire dalle molteplici forme che essa può assumere, dato che, come per le ZTA più in generale, essa può essere implementata in diversi modi.

3.1 Tipologie di microsegmentazione

La microsegmentazione può essere definita come un metodo di sicurezza per la gestione degli accessi alla rete tra carichi di lavoro [33], termine che in questa specifica situazione indica le risorse e i processi necessari ad eseguire un'applicazione [33]. Alcuni esempi di carichi di lavoro secondo questa definizione sono macchine virtuali e container.

Di fatto, la microsegmentazione è un'implementazione di un firewall virtuale distribuito, che regola l'accesso alla risorse della rete in base a delle regole di sicurezza definite per ogni carico di lavoro [3]. Per implementare il firewall distribuito possono essere utilizzate diverse tecniche; in base alla tecnica utilizzata si distinguono quattro

tipologie di microsegmentazione: *hypervisor-based*, *cloud native*, *network-based* ed *agent-based*.

3.1.1 Microsegmentazione hypervisor-based

La microsegmentazione *hypervisor-based* prevede che l'elemento software che costituisce il firewall distribuito sia posizionato concettualmente tra l'hypervisor e la macchina virtuale, come mostrato in Figura 3.1. Dato che il traffico di rete passa sempre attraverso l'hypervisor questo garantisce il massimo controllo su di esso, anche in caso di spostamento della macchina virtuale. L'hypervisor non cambia mai e quindi continuano ad essere applicate le stesse policy di sicurezza che erano applicate nella posizione precedente della macchina.

Questo approccio alla microsegmentazione è però molto limitato, perché è fortemente dipendente dall'hypervisor [22] scelto; per questo motivo è scarsamente utilizzato. È inoltre applicabile solo in ambienti virtualizzati, ma ciò non è particolarmente problematico, visto che più del 92% delle aziende informatiche fa uso di virtualizzazione [30].

Una società che propone questo tipo di microsegmentazione è VMWare.

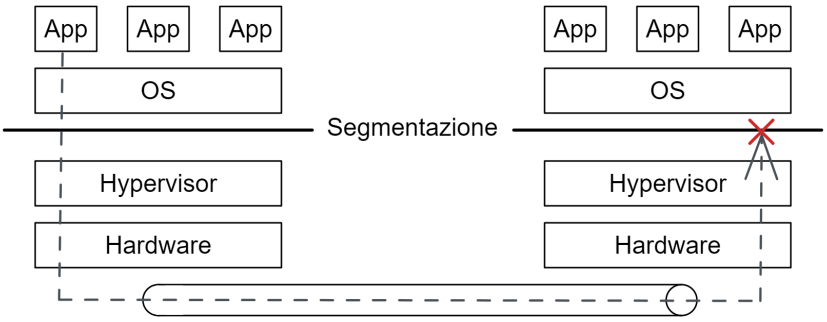


Figura 3.1: Esempio di microsegmentazione hypervisor-based in cui un tentativo di comunicazione da una macchina virtuale all'altra viene bloccato.

3.1.2 Microsegmentazione cloud native

La microsegmentazione *cloud native* è una tipologia di microsegmentazione in cui vengono sfruttate funzionalità del fornitore di servizio cloud come i gruppi

di sicurezza Amazon o il firewall Azure. L'elasticità e il livello di precisione di questo approccio nella gestione del traffico dipendono totalmente dallo strumento utilizzato.

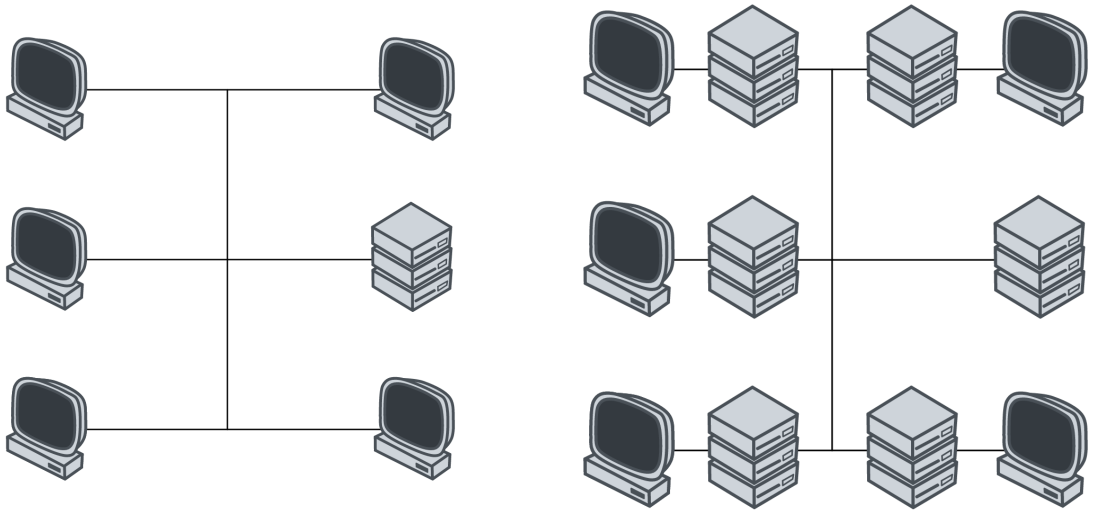
La microsegmentazione cloud native non è particolarmente idonea nel caso si lavori in cloud ibridi o multicloud, in quanto utilizza strumenti specifici di un determinato fornitore di servizio. Nel caso però il sistema aziendale sia completamente contenuto nel cloud di un singolo fornitore questa tipologia di microsegmentazione potrebbe garantire il massimo sfruttamento delle risorse cloud disponibili.

3.1.3 Microsegmentazione network-based

Quando il controllo della segmentazione si basa sull'infrastruttura di rete si parla di microsegmentazione network-based. Vengono utilizzati i dispositivi di rete, virtuali e non, per applicare la policy di sicurezza. Alcuni di questi dispositivi sono switch, load balancers, software defined network (SDN).

La microsegmentazione network-based è talvolta più facile da implementare rispetto alle alternative perché si appoggia a tecniche e strumenti noti da tempo per la segmentazione della rete. Questo vale però solo nei casi in cui si debba microsegmentare una rete fisica, perché nel caso le risorse da proteggere si trovino in una rete virtuale che si espande su multiple reti private e cloud sarà necessario utilizzare una SDN per la realizzazione. Questi strumenti, pur rimanendo tutto sommato ben conosciuti, non lo sono necessariamente al livello degli strumenti di rete classici. Quando si valuta quindi questa tipologia di microsegmentazione occorre distinguere la microsegmentazione network-based realizzata con strumenti hardware e la microsegmentazione network-based realizzata con strumenti software.

La microsegmentazione network-based basata su strumenti hardware è in generale a grana grossa, a causa della difficoltà di mappare le necessità aziendali di segmentazione ai costrutti di rete [1]. In Figura 3.2a è rappresentato un esempio di rete con difesa perimetrale, in Figura 3.2b è rappresentata la stessa rete ma microsegmentata con strumenti hardware: nella situazione rappresentata la grana della microsegmentazione è abbastanza grossa perché tutte le risorse su una stessa macchina saranno soggette alle stesse regole di accesso. Più gli strumenti usati per creare i microsegmenti sono avanzati, meglio sarà possibile rifinire le regole di accesso. Ad esempio, se la microsegmentazione in Figura 3.2b fosse effettuata con



(a) Rete non microsegmentata

(b) Rete microsegmentata

Figura 3.2: Rappresentazione di una rete in versione non microsegmentata e in versione microsegmentata con microsegmentazione network-based.

dei firewall in grado di analizzare i pacchetti al livello 7 dello stack ISO/OSI si avrebbe una grana molto più fine che se i firewall operassero al livello 3 dello stack.

Un ulteriore problema di questo modello è che si appoggia ad hardware specializzato, che deve essere mantenuto aggiornato e sostituito nel caso non riesca più a sostenere la quantità di traffico che viaggia nella rete; ciò implica costi non indifferenti.

La microsegmentazione network-based basata su strumenti software è probabilmente più complessa a causa di una minore disponibilità di conoscenze in merito, ma è più elastica rispetto alla versione che usa strumenti hardware. Bisogna però riconoscere che utilizzarla per la realizzazione di una ZTA va a sfumare i confini tra questa tipologia di ZTA e le ZTA basate su SDP.

3.1.4 Microsegmentazione agent-based

La microsegmentazione *agent-based*, anche detta *host-based*, sfrutta un agente software eseguito all'interno della risorsa per controllare le comunicazioni in entrata (e in uscita) su di essa. Le soluzioni per microsegmentazione host-based possono sfruttare i firewall naturalmente presenti sugli host. Anche per questa tipologia di microsegmentazione si possono riconoscere forti somiglianze con le ZTA basate su

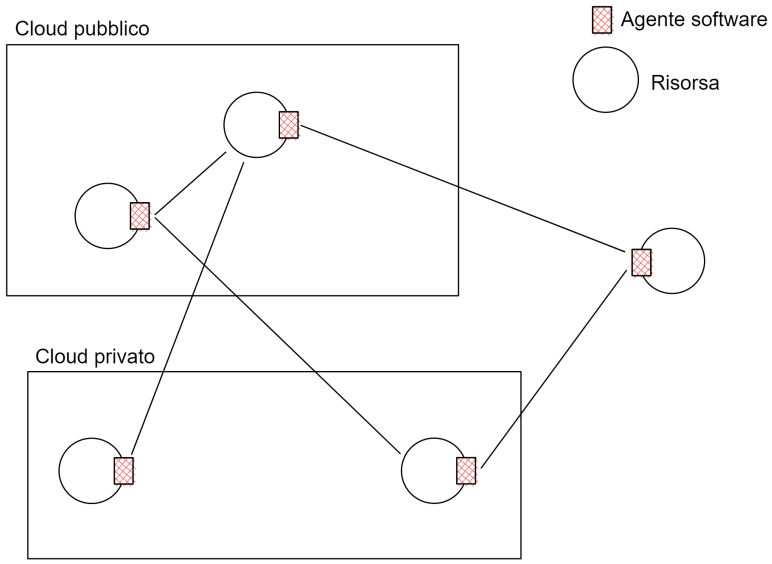


Figura 3.3: Esempio di microsegmentazione agent-based con carichi di lavoro distribuiti su più cloud.

SDP.

Questa tipologia di microsegmentazione è la tipologia più elastica, dato che non richiede nessuna modifica alla rete ed è applicabile anche se i carichi di lavoro si trovano su piattaforme diverse, come mostrato in Figura 3.3. È in grado inoltre di applicare policy di sicurezza a grana molto più fine della microsegmentazione network-based: ad esempio, si potrebbe segmentare ogni Pod in esecuzione su un cluster Kubernetes utilizzando un cosiddetto container sidecar che esegua l'agente software. I problemi di questo approccio sono che richiede di installare agenti software su ogni host e che le conoscenze su queste tecnologie sembrano essere meno diffuse rispetto a quelle necessarie all'implementazione della microsegmentazione network-based. Un rischio legato a questo tipo di microsegmentazione è che un attaccante riesca ad ottenere un accesso privilegiato ad un host e sovvertire il comportamento dell'agente. Un altro rischio, condiviso in realtà da tutte le tipologie di microsegmentazione, è che l'attaccante possa usare connessioni stabilite legittimamente dall'agente per i propri fini.

3.2 Vantaggi dell'uso della microsegmentazione

L'uso della microsegmentazione presenta diversi vantaggi, infatti esso riduce la superficie di attacco, migliora il contenimento delle violazioni e rafforza la conformità

alle normative [33].

Le conseguenze che l'uso della microsegmentazione ha sulla sicurezza della rete, in particolare, sono:

- Migliore visibilità della rete, come conseguenza del fatto che i carichi di lavoro possono comunicare tra loro solo seguendo regole molto restrittive. In questo modo diventa molto più chiaro cosa possa e non possa accadere all'interno della rete. Inoltre, per applicare la microsegmentazione è necessario avere una chiara idea di quali sono i carichi di lavoro, migliorando di conseguenza la consapevolezza dell'azienda in merito a quali risorse siano nella rete.

- Miglioramento del contenimento di una violazione [33], in virtù della migliore abilità di monitorare il traffico e confrontarlo con le policy di sicurezza. Per valutare la presenza o meno di violazioni in una rete microsegmentata è sufficiente confrontare il traffico che coinvolge ogni specifico microsegmento con le sole policy che lo interessano, permettendo quindi un confronto più rapido che in una rete non microsegmentata. Porre rimedio alla violazione implica agire su uno specifico microsegmento (o più di uno, nel caso di un attacco con movimento laterale) invece che sull'intera rete.

- Incremento del numero di risorse che un attaccante deve violare per raggiungere il proprio obiettivo, dato che, avendo minimizzato le risorse raggiungibili da ogni microsegmento seguendo il principio di minimo privilegio, le probabilità che un attaccante debba attraversare molti microsegmenti per raggiungere il proprio obiettivo aumenta. In alcuni casi la lunghezza della catena di risorse da violare in un ambiente microsegmentato può essere anche doppia che in un ambiente non microsegmentato [3].

- Riduzione della possibilità di sfruttare vulnerabilità, dato che, come nel caso dell'incremento del numero di risorse da violare per raggiungere un obiettivo, la probabilità che un attaccante riesca ad accedere alle risorse con delle vulnerabilità e sfruttarle è ridotta in una rete microsegmentata.

I miglioramenti di sicurezza che la microsegmentazione porta sono quindi dovuti a una maggiore chiarezza della struttura della rete e a una maggiore difficoltà ad effettuare movimenti laterali. Prendendo ad esempio la rete rappresentata in

Figura 3.4, se essa non fosse microsegmentata una volta ottenuto accesso a un qualsiasi host un attaccante raggiungerebbe facilmente l'obiettivo E. Essendo però la rete microsegmentata ciò non è più vero: l'unico modo per poter contattare l'host E è tramite host non vulnerabili, verso cui quindi l'attaccante ha accesso limitato. Anche se esso riuscisse ad accedere ad uno degli host vulnerabili della rete (rappresentati in rosso) non potrebbe effettuare movimenti laterali utili al suo scopo con successo.

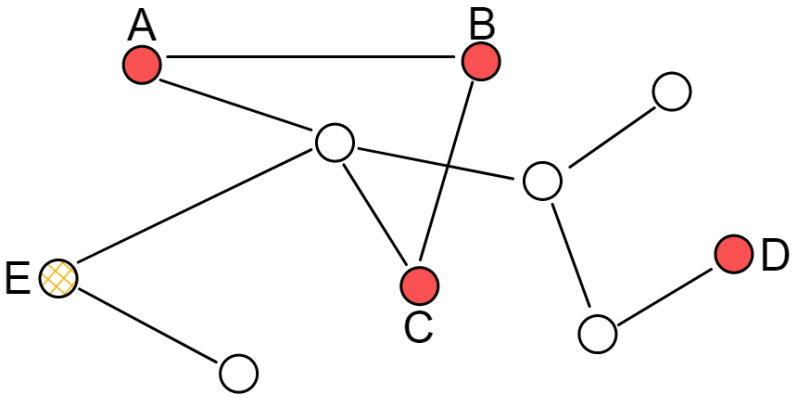


Figura 3.4: Esempio di rete con microsegmenti, con evidenziati alcuni host vulnerabili in rosso e un host obiettivo per un attaccante in giallo.

Oltre a questi vantaggi la microsegmentazione permette una maggiore conformità alle normative, dato che le risorse la cui gestione deve seguire regole apposite possono essere isolate in un loro microsegmento. Le comunicazioni permesse sono così definite in modo granulare, riducendo il rischio di uso delle risorse non conforme alla normativa.

3.3 Sfide della microsegmentazione

Come già accennato in Sezione 2.1.5, la microsegmentazione presenta diverse sfide, in particolare:

- In una rete ampia segmentare efficacemente il traffico è difficile, a causa della complessità delle interazioni e delle dipendenze tra applicazioni.

- Applicazioni monolitiche e *legacy* non sono idonee alla microsegmentazione.
- Tradurre le necessità di accesso di un flusso di lavoro in policy di controllo degli accessi a livello di rete è un compito complesso, il cui risultato potrebbe non prevenire attività dannose. Una possibilità per ridurre questo rischio è l'uso di microservizi, potendo così definire in modo più preciso il traffico permesso per ogni elemento.
- I continui cambiamenti delle applicazioni possono rendere difficile mantenere le policy di accesso, portando ad errori di configurazione. Questo problema può essere mitigato tramite l'utilizzo di strumenti che identificano questi cambiamenti e automaticamente aggiornano le policy.

Una delle ragioni per usare la microsegmentazione è il poter definire policy di sicurezza con granularità molto fine, ma ciò mal si abbina alle applicazioni monolitiche: essendo indivisibile, l'intero monolite deve essere posizionato in un unico microsegmento, ma in tal modo si è costretti a permettere per quel microsegmento diverse categorie di traffico, aumentando la granularità delle policy di sicurezza, come mostrato in Figura 3.5. In realtà, in base al tipo di microsegmentazione applicata questo problema può essere mitigato: se il componente che si occupa di effettuare la microsegmentazione è in grado di esaminare il traffico al livello 7 dello stack ISO/OSI è possibile comunque applicare policy con una granularità fine. Ciò però pone dei limiti non solo sul tipo di strumento da utilizzare per la microsegmentazione ma anche sulla possibilità di cifrare il traffico. L'analisi dei pacchetti al livello 7 dello stack ISO/OSI rimane comunque un problema aperto anche al di fuori della microsegmentazione, per le implicazioni che ha sulla confidenzialità delle comunicazioni.

Anche architetture con una granularità più fine delle architetture monolitiche potrebbero comunque creare problemi nella definizione delle policy, impedendo di definire regole sufficientemente precise e mirate. Da questo punto di vista le applicazioni basate su un'architettura a microservizi risultano quelle più semplici da proteggere usando la microsegmentazione in quanto, come mostrato in Figura 3.6, è possibile definire regole più specifiche. Questo è vero se per ogni microservizio si prevede un microsegmento: inserire più microservizi in uno stesso microsegmento porta a dover definire policy a grana più grossa. L'avere un numero elevato di com-

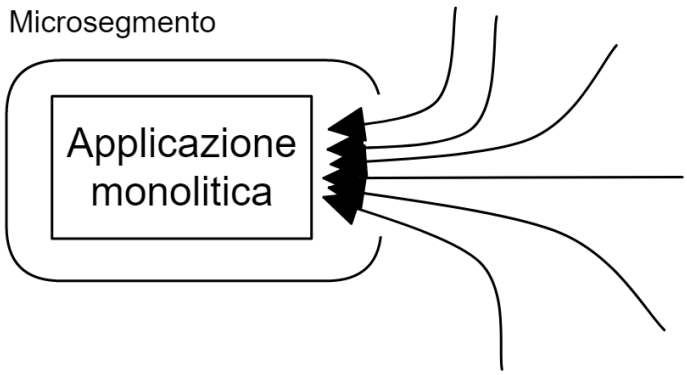


Figura 3.5: Applicazione monolitica in un microsegmento, con policy di sicurezza a grana grossa.

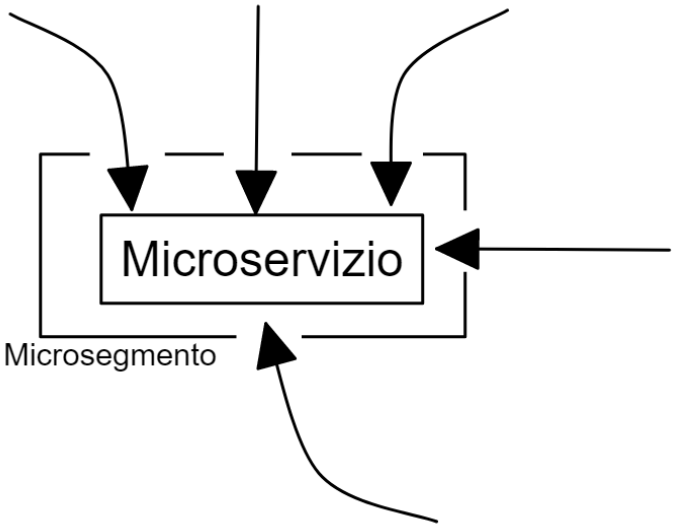


Figura 3.6: Microservizio in un microsegmento, con policy di sicurezza a grana fine.

ponenti dell'applicazione da proteggere, però, come può avvenire nelle applicazioni a microservizi, rende difficile capire a fondo le interazioni e le dipendenze tra ogni componente.

Questo problema, già presente a livello di singole applicazioni non monolitiche, è particolarmente significativo in reti ampie con molte applicazioni. Per poter microsegmentare efficacemente una rete occorre innanzitutto comprendere bene la topologia della rete, compito che però si complica tanto più è ampia la rete. Oltre alla topologia della rete è necessario tracciare i flussi di comunicazione tra le varie applicazioni, operazione che risulta complessa tante più sono le applicazioni in gioco. Per tracciare questi flussi di comunicazione è essenziale conoscere tutte le

applicazioni in esecuzione all'interno della rete.

Usare queste informazioni sulla struttura della rete per definire microsegmenti e policy di sicurezza che rispettino il principio di minimo privilegio non sempre porta a bloccare tutto il possibile traffico malevolo. Ciò è dovuto in parte alla difficoltà di comprendere appieno tutte le interazioni tra le applicazioni, che porta a fare errori nella definizione delle policy, e in parte all'incompatibilità tra la struttura delle applicazioni e la microsegmentazione. Un esempio di ciò è, come già accennato, la presenza di applicazioni monolitiche.

Un aiuto nella definizione delle policy di sicurezza viene dato da sistemi automatici che esaminano la struttura della rete e il traffico per poi definire le policy di sicurezza in base a quanto appreso. Questi sistemi possono anche utilizzare l'intelligenza artificiale per migliorare la policy di sicurezza esaminando continuamente la rete. In caso di aggiornamenti alle applicazioni l'uso di questi strumenti permette un rapido aggiornamento delle policy di sicurezza. Gli strumenti di definizione automatica delle policy però non sono esenti da rischi: nel caso gli esempi di traffico utilizzati comprendano traffico malevolo questo potrebbe essere permesso dalle policy generate o comunque potrebbe influire sul processo di apprendimento dell'intelligenza artificiale.

Le policy generate dagli strumenti automatici possono essere considerate anche solo dei consigli, da controllare prima di applicarle al sistema. Questo risulta più sicuro che applicare direttamente le policy, in quanto c'è un livello di controllo in più, ma è costoso in termini di tempo e impegno. In reti molto ampie un controllo manuale potrebbe essere impossibile da svolgere in tempo utile, costringendo ad affidarsi alla policy generata automaticamente.

3.4 Approccio alla microsegmentazione basato su overlay network

Come indicato nelle precedenti sezioni, la microsegmentazione è un'ottima scelta come forma di implementazione di una ZTA. Le sue controindicazioni riguardano principalmente le difficoltà nella definizione delle policy di sicurezza e la scarsa compatibilità con sistemi legacy, ma questi sono problemi che, in maniera più o meno

diversa, riguardano tutte le tipologie di ZTA; non sono quindi motivi sufficienti per favorire altre forme di ZTA.

In compenso la microsegmentazione è più elastica rispetto alle altre tipologie di architetture Zero Trust, permettendo una gestione più semplice dei dispositivi IoT rispetto alle ZTA basate sulla gestione avanzata dell'identità e offrendo più possibilità di implementazione rispetto alle ZTA basate su SDP. Scegliere la microsegmentazione come tecnica per la realizzazione di una ZTA garantisce, almeno a livello teorico, la possibilità di applicarla a molteplici ambienti con meno complicazioni rispetto alle altre tipologie.

Gli overlay network sono uno strumento particolarmente interessante per l'implementazione della microsegmentazione, soprattutto quelli realizzati tramite l'uso di agenti software. Questa tipologia di overlay network permette di collegare dispositivi di reti diverse e anche, se il software è dotato di capacità di firewall, di controllare il traffico tra essi. Gli overlay network con queste caratteristiche in pratica rendono possibile attuare una forma di microsegmentazione agent-based.

La proposta di questa tesi è un approccio per la microsegmentazione basato su overlay network scalabile e robusto, ma anche di comoda configurazione e utilizzo. Gli attuali strumenti per la microsegmentazione tendono a favorire la comodità d'uso a scapito di scalabilità e robustezza, ma vista la continua evoluzione dei moderni sistemi software e i sempre più frequenti attacchi informatici questa scelta non è ottimale.

L'approccio proposto evita punti unici di fallimento per incrementare la robustezza, propone una gestione degli accessi distribuita per favorire scalabilità e robustezza e prevede una definizione centralizzata della policy di sicurezza per garantire un buon livello di comodità d'uso.

4 Microsegmentazione agent-based con configurazione centralizzata

L'approccio alla microsegmentazione che si sta proponendo, già descritto brevemente in Sezione 3.4, si differenzia da quelli attualmente applicati per l'uguale importanza data a scalabilità, robustezza e comodità d'uso. La tendenza al momento è infatti di favorire la comodità di configurazione e di utilizzo degli strumenti per la microsegmentazione facendo concessioni sulla scalabilità e la robustezza. Questo capitolo elabora in maniera più approfondita l'approccio alla microsegmentazione proposto, confrontandolo con gli approcci che compongono il panorama attuale della microsegmentazione.

4.1 Tendenze attuali della microsegmentazione

La tendenza attuale delle infrastrutture è verso l'uso di cloud ibridi e multicloud, per questo motivo la tendenza della microsegmentazione è verso la tipologia agent-based, dato che è in pratica l'unica versione di segmentazione utilizzabile in questi ambienti. È comunque possibile trovare soluzioni per microsegmentazione hypervisor-based e network-based in vendita, in quanto in specifiche situazioni possono risultare una scelta migliore.

Ci sono diversi strumenti commerciali, come ditno[1] e Illumio, che si propongono di realizzare microsegmentazione secondo i principi Zero Trust. Questi strumenti, per la maggior parte agent-based, offrono capacità di networking per connettere i vari dispositivi, microsegmentazione per isolare i carichi di lavoro della rete, gestione centralizzata delle policy di sicurezza, automazione della scoperta della

[1] https://www.ditno.com/

struttura della rete. Quello che si propongono di fare è migliorare la sicurezza dell'infrastruttura aziendale e allo stesso tempo ridurre i costi.

Anche se il concetto di ZTA rimane una novità, la microsegmentazione è utilizzata da abbastanza tempo da non poter essere più considerata una tecnologia nuova. È possibile trovare venditori di soluzioni per la microsegmentazione con esperienza, i cui prodotti sono in grado di lavorare in sistemi molto ampi. Tutti i venditori credibili dovrebbero essere in grado di soddisfare ridondanza, alta disponibilità, distribuzione delle policy, logging e altre preoccupazioni su vasta scala [19]. Ridondanza e alta disponibilità sono due elementi essenziali di un sistema basato sulla microsegmentazione, a cui diventa molto importante prestare attenzione se lo strumento utilizzato per realizzarla ha un controller centralizzato.

Un aspetto che accomuna la maggior parte degli strumenti disponibili per la microsegmentazione è appunto che offrono un controllo centralizzato della policy di sicurezza, tramite interfacce web o REST API. Ciò è generalmente una buona cosa ed infatti strumenti come ditno ed Illumio si vantano di offrire queste funzionalità. Ci sono però due problemi con questo approccio alla configurazione della rete: la centralizzazione porta ad avere un potenziale punto unico di fallimento e vulnerabilità, mentre l'uso di un'interfaccia web porta a non poter seguire il principio di Policy as Code. Le tecniche per evitare che il punto da cui si configura la policy sia resiliente e la sua disponibilità garantita sono ben note, quindi quello della centralizzazione del controllo della policy è un problema relativo. Più importanti sono la presenza di un unico punto di vulnerabilità e il non poter definire la policy secondo il principio di Policy as Code.

Le principali proposte di ZTA assumono che, mentre il piano dei dati è sempre insicuro, il piano di controllo è sempre fidato. Ciò però risulta incoerente con i principi Zero Trust, in quanto è irrealistico assumere che gli attaccanti non possano mai compromettere i componenti del piano di controllo che forniscono accesso alle risorse [11]. Nel caso la policy di sicurezza venga gestita e valutata in un unico punto la compromissione di tale componente implicherebbe la compromissione dell'intera rete; una simile architettura sarebbe quindi da evitare per aumentare la sicurezza del sistema.

Il non poter definire la policy secondo il principio di Policy as Code è un problema parzialmente aggirabile nel caso siano disponibili delle REST API per configurare

il sistema: si potrebbero definire e memorizzare le richieste HTTP necessarie a configurare il sistema. Alcune interfacce web offrono la possibilità di importare ed esportare le policy, ma spesso introducono limitazioni su quali regole siano esportabili. Illumio, ad esempio, non permette di esportare regole legate a uno specifico carico di lavoro o server.

In base alla tipologia di microsegmentazione che si sta realizzando parlare di Policy as Code potrebbe non essere totalmente corretto: nel caso di microsegmentazione hypervisor-based o cloud native il definire i segmenti di rete può essere inquadrato nell'ambito dell'Infrastructure as Code, mentre per la microsegmentazione agent-based si potrebbe parlare di Configuration as Code. Indipendentemente però da quale termine sia più accurato utilizzare è importante poter definire l'architettura della propria rete sotto forma di codice, ottenendo tutti i vantaggi di efficienza, tracciabilità e gestibilità che accomunano questi concetti. Purtroppo, come dimostra il caso di Illumio, non molti strumenti offrono questa possibilità e ancora meno strumenti la offrono in modo completo.

Un altro problema noto delle ZTA in generale è la necessità che policy administrator e policy enforcement point possano sempre comunicare; questo problema è presente anche in un sistema microsegmentato. Ogni microsegmento deve avere il proprio policy enforcement point, in quanto esso è il componente con il ruolo di bloccare il traffico indesiderato, mentre il policy administrator è solitamente un componente centralizzato. Ci sono diversi vantaggi nel centralizzare questo componente:

- È possibile fornire al policy engine una maggiore potenza di calcolo, permettendogli di valutare policy più complesse. Un policy decision point distribuito richiederebbe di replicare questa potenza di calcolo per ogni istanza, potenzialmente risultando troppo costoso.

- Aggiornare la policy è semplice, in quanto basta modificarla in un solo punto affinché il cambiamento abbia effetto sull'intero sistema.

- È possibile implementare trust algorithm contestuali senza particolari impedimenti, in quanto lo storico delle richieste potrà essere memorizzata in un unico punto.

Nonostante questi vantaggi centralizzare il policy administrator non è sempre una buona scelta, in quanto esso diventa un unico punto di fallimento. Anche replicandolo si potrebbe comunque essere soggetti al rischio che interruzioni di rete rendano impossibile per i policy enforcement point contattarlo. In una situazione come quella in Figura 4.1 infatti i policy enforcement point del datacenter on premise non sono in grado di contattare nessun policy administrator, portando al blocco delle operazioni in tale zona. L'unica soluzione che permetterebbe di evitare ciò sarebbe replicare il policy administrator in ogni area; questo però risulterebbe costoso e di difficile gestione, soprattutto se lo strumento scelto non supporta nativamente questo tipo di funzionamento.

Un ulteriore problema di avere un policy administrator centralizzato è la sua scalabilità. In caso di aumento delle richieste di autorizzazione si rende necessario scalare o verticalmente o orizzontalmente il sistema, ma nessuna delle due opzioni è ottimale. La scalabilità verticale porta ad avere risorse inutilizzate nei momenti con meno richieste, la scalabilità orizzontale va a creare problemi qualora si usino trust algorithm contestuali perché sarà impossibile per un'istanza del policy administrator sapere che richieste stanno esaminando le altre istanze. Per permettere quindi a un policy administrator centralizzato di essere pienamente scalabile si va a perdere uno dei vantaggi di questa centralizzazione.

Sono disponibili strumenti che non operano in maniera centralizzata, anche se essi sono la minoranza. Essi presentano comunque problematiche sotto alcuni aspetti. DxOdissey[2], ad esempio, non prevede controller centralizzati, in modo da essere più resistente a eventuali interruzioni di rete. Esso utilizza una serie di gateway, indipendenti tra loro, per permettere l'accesso alle risorse da parte dei client tramite tunnel criptati. Il problema principale dell'approccio di DxOdissey è che un gateway gestisce un'enclave di risorse e quindi suoi malfunzionamenti impedirebbero l'accesso all'intero gruppo di risorse. Un altro problema è che i gateway, per conoscersi tra loro, fanno uso al momento del loro avvio di un servizio proprietario offerto da DH2i, l'azienda produttrice di DxOdissey. Questo può rivelarsi un unico punto di fallimento; considerando poi che la sua gestione non è in mano all'azienda che usa DxOdissey ma a DH2i si potrebbero avere problemi legati alla catena di approvvigionamento e quindi non risolvibili in autonomia.

[2]https://dh2i.com/dxodyssey/

Escluse comunque poche eccezioni la maggior parte degli strumenti correntemente disponibili per la microsegmentazione favoriscono la comodità d'uso a scapito della resistenza e scalabilità del sistema.

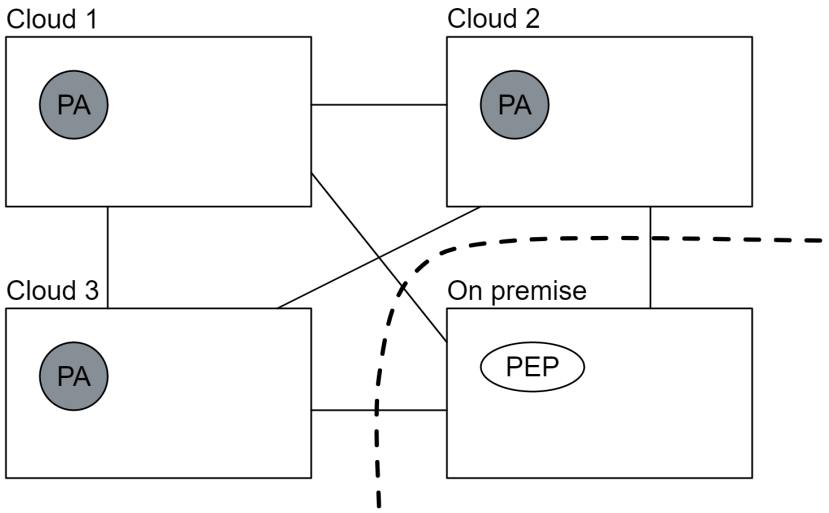


Figura 4.1: Esempio di situazione in cui un PEP non può contattare nessuna replica del policy administrator, a causa di un'interruzione della rete rappresentata dalla riga tratteggiata.

4.2 ZTA in cloud ibridi e multicloud

Prendendo in considerazione i vari approcci all'implementazione di una ZTA il più idoneo a essere utilizzato in cloud ibridi e multicloud, pur continuando ad essere utilizzabile in qualsiasi tipologia di ambiente, è la microsegmentazione. L'approccio basato sulla gestione avanzata dell'identità potrebbe essere efficace, ma in presenza di NPE rischierebbe di rivelarsi controproducente, a causa dei problemi nella gestione delle loro credenziali. L'approccio basato su SDP prevede un controller di rete centralizzato, che invece si desidera evitare in quanto punto unico di fallimento.

La microsegmentazione risulta quindi l'approccio più idoneo in questa situazione, anche se la tipologia di microsegmentazione scelta è importante: l'unica utilizzabile in cloud ibridi e multicloud è infatti la microsegmentazione agent-based.

Come evidenziato in Sezione 4.1 però la tendenza della microsegmentazione agent-based è l'uso di un controller centralizzato, abbinato a interfacce web e REST API per la configurazione del sistema. Con queste modalità scalabilità e resistenza non sono garantite, inoltre anche la sicurezza è potenzialmente inferiore a quella di un sistema con controller distribuito.

Il problema della resistenza del sistema nasce dalla centralizzazione del policy administrator (e del policy decision point): la soluzione è non centralizzarlo. Avere un policy decision point distribuito rende più resistente il sistema a interruzioni di rete e lo rende anche maggiormente scalabile, a scapito però della comodità d'uso del sistema.

Un policy decision point distribuito potrebbe prevedere che le varie istanze del PDP collaborino per raggiungere una decisione o che ogni istanza agisca in completa autonomia. Nel primo caso, la fiducia è distribuita sulle varie istanza, rendendo il sistema più sicuro, ma non necessariamente più resistente. Una possibilità per l'implementazione di un PDP definito in questo modo è l'uso di *threshold signatures* [27]: data una serie di PDP, il policy enforcement point inoltra la richiesta solo se ha ricevuto l'autorizzazione da un sottoinsieme dei PDP di cardinalità sufficiente. Questo meccanismo è più sicuro di un PDP centralizzato in quanto un attaccante per alterare il processo di autorizzazione dovrebbe compromettere numerose istanze del PDP invece di una sola; in situazioni estreme come quella in Figura 4.1 però il sistema smetterebbe di funzionare per l'impossibilità di contattare sufficienti istanze del PDP. Quando invece le istanze del policy decision point lavorano in completa autonomia comprometterne una non ha influenze di alcun tipo sulle altre, mentre la robustezza del sistema dipende dalla quantità di istanze e dalla loro posizione. La massima robustezza si raggiunge quando policy enforcement point e policy decision point sono aggregati: non sarà mai possibile che non possano comunicare per problemi di rete. Inoltre la compromissione di un PDP avrà conseguenze su un unico PEP. Nel caso della microsegmentazione agent-based occorre quindi che l'agente software valuti localmente la policy di sicurezza e applichi la decisione raggiunta. La valutazione locale della policy può anche portare a un miglioramento delle performance del sistema, in quanto il policy enforcement point non dovrà inviare messaggi sulla rete per contattare il policy administrator e aspettare la sua risposta.

In ambienti in cui si trovano numerosi dispositivi con risorse limitate delegare ai singoli dispositivi la valutazione della policy di sicurezza potrebbe non essere possibile, proprio per motivi legati alle loro capacità di calcolo. In questi casi le opzioni sono due: o rendere più efficiente la valutazione della policy in questi dispositivi, magari limitando il numero di regole che devono essere esaminate a quelle strettamente di interesse, o delegare la valutazione a un altro dispositivo. Con questa seconda opzione, rappresentata in Figura 4.2, si potrebbero avere problemi legati alle interruzioni di rete, ma non ai livelli che si avrebbero se tutte le richieste della rete fossero gestite in un unico punto o se si dovessero contattare multiple istanze del PDP. Questo fatto è chiaro se si confrontano Figura 4.2 e Figura 4.1: in Figura 4.2 se l'interruzione di rete rappresentata dalla linea tratteggiata si verificasse solo alcune delle risorse del datacenter on premise, ovvero quelle rappresentate da cerchi bianchi, sarebbero irraggiungibili; in Figura 4.1 se si verificasse l'interruzione di rete rappresentata tutte le risorse del datacenter on premise non sarebbero raggiungibili. Il dispositivo delegato alla valutazione potrebbe fungere anche da gateway per l'accesso ai dispositivi, portando ad avere una microsegmentazione a grana grossa. Per limitare il numero di regole che ogni agente deve valutare si dovrebbe strutturare la policy in maniera adeguata: con una policy di sicurezza di tipo "deny by default" le regole da valutare su ogni agente sarebbero solo quelle che definiscono il traffico permesso; nel caso di microsegmentazione molto fine queste regole non dovrebbero essere in numero eccessivo.

In ogni caso distribuire il policy decision point, indipendentemente dalla granularità del controllo degli accessi che verrà poi attuato, porterebbe a un aumento della resistenza del sistema. Anche la scalabilità aumenta, in quanto ogni policy decision point potrà essere dimensionato in base alle sue specifiche necessità, evitando inoltre che un sovraccarico di richieste per una specifica risorsa blocchi la valutazione delle richieste per altre risorse.

In mancanza di un punto unico da cui gestire la rete e le relative policy la configurazione del sistema diventa più complicata. La necessità quindi è di poter definire la configurazione del sistema e le policy di sicurezza in maniera centralizzata, seguendo i principi di Policy e Configuration as Code per una maggiore gestibilità, per poi applicare queste impostazioni in maniera distribuita.

Un miglioramento della scalabilità, resistenza e usabilità degli strumenti di

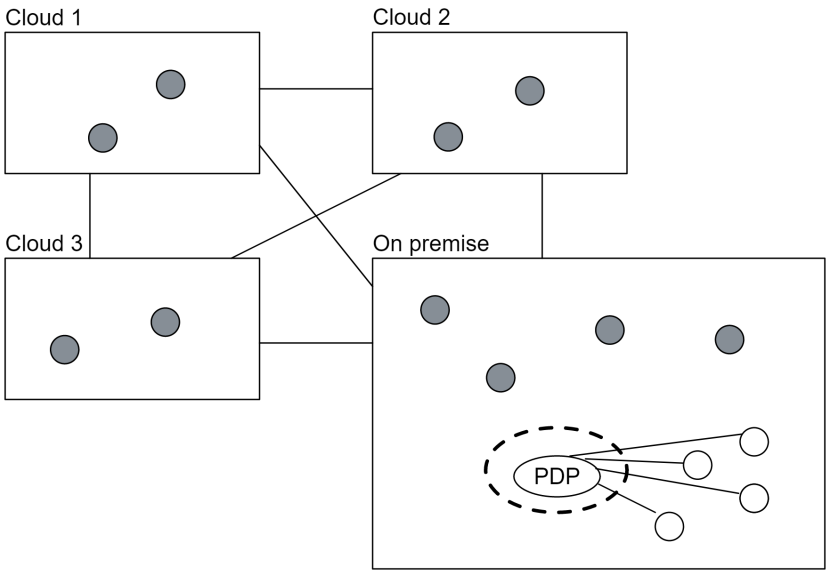


Figura 4.2: Sistema in esecuzione su più cloud e on premise, con due tipi di risorse rappresentate da dei cerchi. Le risorse grigie hanno sia PEP che PDP integrato. Le risorse bianche fanno riferimento a un PDP esterno.

microsegmentazione è quindi possibile, ma non seguendo i principali approcci diffusi attualmente.

4.3 Approccio scalabile alla microsegmentazione con configurazione centralizzata

Prendendo in considerazione le problematiche evidenziate in Sezione 4.2, uno strumento per la microsegmentazione che ambisca a essere più scalabile e robusto delle alternative dovrebbe avere le seguenti caratteristiche:

- Essere agent-based, in quanto lavorando su cloud ibridi e multicloud le altre possibilità, cloud native, network-based e hypervisor-based, sono irrealizzabili o molto più complesse da implementare.
- Offrire funzionalità di overlay networking, per poter connettere i carichi di lavoro dei vari cloud in un'unica rete senza dover ricorrere ad ulteriori strumenti, che sarebbero da studiare, configurare e gestire.

- Permettere di definire la policy di sicurezza sotto forma di codice, seguendo i principi di Policy as Code, in forma centralizzata, ovvero definendo l'intera policy in un unico punto (e non separatamente per ogni carico di lavoro).
- Permettere di definire la configurazione degli agenti software sotto forma di codice, seguendo i principi di Configuration as Code.
- Applicare la policy di sicurezza in base a decisioni prese localmente, da parte dell'agente software, piuttosto che da un componente centralizzato.

Ognuno di questi elementi va a risolvere uno specifico problema dei classici approcci alla microsegmentazione: l'essere agent-based e con funzionalità di overlay networking permette l'applicazione della microsegmentazione in ambienti eterogenei; la definizione di policy e configurazione sotto forma di codice in maniera centralizzata permette di mantenere la comodità d'uso dei sistemi centralizzati e di migliorare la gestione di queste impostazioni; la valutazione locale della policy di sicurezza incrementa la resistenza e scalabilità del sistema. Integrare le capacità di overlay networking nell'agente software che controlla gli accessi alle risorse riduce inoltre il rischio che un attaccante possa sfruttare la connessione aggirando il controllo degli accessi. Ciò è comunque possibile, ma è più complesso: si può dire che il controllo del traffico sia integrato nella connessione, quindi aggirarlo è più difficile rispetto al caso in cui controllo e connessione sono due elementi separati.

L'approccio descritto presenta evidenti vantaggi rispetto agli approcci alternativi più diffusi; valutarne più a fondo l'applicazione è quindi una strada degna di essere esplorata. A tal fine si intende implementare nella pratica questo approccio, realizzando una *proof of concept* che ne dimostri la fattibilità. Si andranno poi a valutare scalabilità, resistenza e comodità d'uso di quanto realizzato.

4.4 Piano di sviluppo e metriche di valutazione

Gli attuali strumenti disponibili non offrono, in autonomia, tutte le caratteristiche necessarie per implementare l'approccio appena descritto. Occorre quindi capire come sia possibile realizzare una soluzione che lo implementi. È anche necessario definire con maggiore precisione secondo quali criteri sarà valutata la soluzione.

4.4.1 Piano di sviluppo della soluzione

Gli aspetti principali da considerare nella realizzazione di uno strumento che implementi l'approccio proposto sono:

- Necessità di connettere carichi di lavoro in cloud e reti distinti.
- Necessità di filtrare il traffico tra i vari carichi di lavoro.
- Necessità di definire centralmente la policy di sicurezza.

Le prime due necessità possono essere soddisfatte da uno strumento di overlay networking, con la condizione che esso non presenti un singolo punto di valutazione delle regole di filtraggio del traffico. La terza necessità può essere gestita separatamente dalle altre due: essa è rilevante al momento della configurazione del sistema, mentre le altre sono rilevanti al momento del suo funzionamento.

La proof of concept potrebbe quindi essere realizzata o implementando un singolo strumento ex novo o combinando strumenti già esistenti e strumenti realizzati appositamente se necessario. Visto il livello di performance degli strumenti di overlay networking già esistenti, si può escludere l'idea di implementare un singolo strumento ex novo, impegno che comunque si spingerebbe al di fuori dello scopo di questa tesi.

Occorre quindi selezionare un software per l'overlay networking con le caratteristiche necessarie, ovvero:

- Capacità di firewall con valutazione locale della policy di sicurezza.
- Assenza di un controller di rete centralizzato.

Altre caratteristiche, necessarie per il rispetto dei principi Zero Trust ma non essenziali per valutare la validità dell'approccio alla microsegmentazione proposto, sono che il traffico dell'overlay network sia cifrato e che la configurazione dell'overlay network possa essere modificata durante il funzionamento. La possibilità di identificare ogni nodo della rete potrebbe poi essere utile nel definire una policy di sicurezza con un'adeguata granularità. L'identità di un nodo non può basarsi sul semplice indirizzo IP, vista la facilità con cui esso può essere falsificato. Ogni nodo dell'overlay network dovrà eseguire una copia del software con la propria configurazione, agendo in modo totalmente indipendente dagli altri nodi.

Il modello di ZTA che viene seguito è quindi quello basato su enclave: un singolo agente potrebbe proteggere risorse multiple e la comunicazione avviene necessariamente da agente ad agente, con un'istanza che funge da client e una che funge da gateway. Il software di overlay networking scelto dovrebbe però offrire sufficienti modalità di esecuzione da permettere di definire microsegmenti di dimensioni minime, riducendo quindi le risorse gestite da un singolo agente. In questo modo problemi ad un agente influenzerebbero l'accesso a meno risorse possibili.

Una volta selezionato uno strumento per l'overlay networking adeguato ci si può concentrare sul soddisfare la terza necessità precedentemente elencata, ovvero definire centralmente la policy di sicurezza, che coincide con la configurazione dell'overlay network. A questo fine occorre, nell'ordine:

1. Specificare un formato per descrivere la configurazione dell'intero overlay network, inclusa la policy di sicurezza. Questo permette di definire centralmente la configurazione.

2. Sviluppare uno strumento che a partire dalla configurazione centralizzata generi la configurazione dei singoli nodi dell'overlay network.

Questo strumento deve essere sviluppato ex novo, vista l'elevata specificità della sua funzione.

Con questi due strumenti è possibile implementare l'approccio proposto in Sezione 4.3 e valutarne scalabilità, robustezza e semplicità d'uso.

4.4.2 Metriche di valutazione

L'applicabilità dell'approccio alla microsegmentazione che viene proposto in questa tesi verrà valutata tramite la realizzazione di una proof of concept. La sua qualità in termini di scalabilità e robustezza sarà invece valutata tramite un'estesa serie di test. Complessivamente, gli aspetti che si mirano a valutare sono:

- Semplicità di definizione della policy di sicurezza.

- Impatto dell'uso dello strumento sui tempi di comunicazione tra dispositivi.

- Scalabilità dello strumento, ovvero quanto le performance dello strumento si riducono all'aumentare del numero di dispositivi sulla rete e del traffico da essi generato.
- Resistenza dello strumento a problemi di rete, come perdita di pacchetti durante la comunicazione.
- Richiesta di risorse da parte dello strumento, in termini di CPU e memoria RAM.

5 Strumenti utilizzati

Come indicato in Sezione 4.4.1, la realizzazione di una proof of concept che dimostri la viabilità dell'approccio proposto richiede l'uso di uno strumento di overlay networking e di uno strumento per la definizione centralizzata della policy.

Lo strumento di overlay networking scelto è stato Nebula, mentre lo strumento per la definizione delle policy è stato implementato usando Dhall, un linguaggio di configurazione programmabile, ed Haskell, un linguaggio di programmazione scelto unicamente perché offre diverse librerie per operare sui file in linguaggio Dhall. In questo capitolo verranno presentate le caratteristiche principali di Nebula e Dhall, insieme alle motivazioni per la loro scelta.

5.1 Nebula

Nebula[1] è uno strumento di overlay networking progettato per essere veloce, sicuro e scalabile [23], sviluppato originariamente da Slack per uso interno, poi reso open source. Al momento il suo sviluppo è guidato da Defined Networking Inc., una società fondata da due ex dipendenti di Slack appositamente per questo scopo.

La funzionalità principale di Nebula è la creazione di una rete virtuale che colleghi un numero variabile di host, appartenenti a reti IP diverse, su richiesta, tramite tunnel criptati, senza necessità di aprire porte del firewall. A ciò si aggiunge la possibilità di definire regole firewall in merito al traffico dell'overlay network per ogni host. Gli host possono appartenere o meno a dei gruppi, su cui può basarsi la definizione delle regole del firewall.

In questa tesi i termini host e nodo verranno usati, quando in riferimento alla rete Nebula, in modo intercambiabile.

[1] https://www.defined.net/nebula/

5.1.1 Overlay networking

Una rete Nebula è formata da host di due tipi: *lighthouse* e normali host. I normali host comunicano tra loro in modalità peer-to-peer, con le lighthouse che fungono da supporto nella scoperta delle route e nell'attraversamento NAT. Le lighthouse devono essere raggiungibili da tutti gli host e il loro indirizzo IP dovrebbe essere statico. In particolare, è il traffico Nebula, ovvero traffico UDP di default sulla porta 4242, che deve poter raggiungere la lighthouse dagli altri nodi della rete. Per ogni rete, come mostrato in Figura 5.1, si dovrebbe avere almeno una lighthouse.

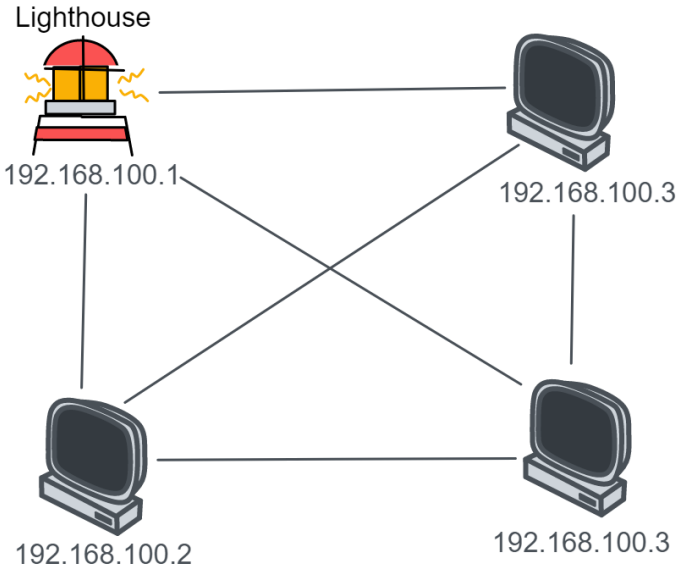


Figura 5.1: Esempio di rete Nebula con una lighthouse.

Presi due nodi, a meno che non siano entrambi dietro un NAT simmetrico, utilizzando *UDP hole punching* e *spoofing* dell'indirizzo IP di una lighthouse sarà sempre possibile per loro comunicare. L'ultima versione di Nebula, la 1.6.0, propone la funzionalità ancora sperimentale "relay", che permetterebbe di superare i limiti imposti dai NAT simmetrici, usando un nodo raggiungibile dai nodi dietro i NAT simmetrici come intermediario nella comunicazione. I ruoli di relay e lighthouse sono distinti e possono essere assegnati a host diversi.

Una volta che due nodi hanno stabilito la connessione non hanno più bisogno di contattare la lighthouse, salvo problemi di connessione tra essi che li costringano a

stabilire un nuovo tunnel criptato. Maggiore è il numero di lighthouse sulla rete, quindi, più forte è la garanzia che due nodi riusciranno a stabilire una connessione, ma una volta fatto ciò la presenza delle lighthouse diventa ininfluente per loro. Una rete Nebula è quindi parzialmente resistente alla perdita di lighthouse.

Il protocollo crittografico utilizzato da Nebula è basato sul Noise Protocol Framework[2]. I nodi si autenticano mutualmente usando il protocollo di Diffie-Hellman basato sulle curve ellittiche, mentre la comunicazione viene cifrata utilizzando o AES o ChaChaPoly. In una rete tutti i nodi devono usare lo stesso cifrario. Nebula usa un'infrastruttura a chiave pubblica per garantire l'identità dei nodi della rete: per ogni nodo deve essere rilasciato un certificato che indichi l'indirizzo IP, il nome e i gruppi a cui appartiene il nodo in questione. I certificati dei nodi della stessa rete possono essere firmati con chiavi diverse; ogni nodo può avere a disposizione più di una chiave pubblica per controllore la validità dei certificati. Questa funzionalità è utile anche per facilitare la rotazione delle chiavi delle Certification Authority, o CA, in quanto è possibile distribuire sui nodi la nuova chiave pubblica della CA prima che la vecchia scada.

Per garantire la massima sicurezza dei nodi della rete Nebula è teoricamente possibile configurare i firewall che li separano da Internet in modo che solo il traffico Nebula sia permesso: la comunicazione tra i vari host sarà così possibile solo sulla rete Nebula. La possibilità che un malintenzionato cerchi di contattare il nodo tramite Internet viene così ridotta, se non persino eliminata.

5.1.2 Regole firewall group-based

Integrata in Nebula c'è la possibilità di filtrare il traffico della rete; in particolare per ogni host è possibile definire il traffico permesso in uscita (regole *outbound*) e quello permesso in entrata (regole *inbound*), di default tutto il traffico è bloccato. Il traffico a cui applicare una determinata regola è identificato tramite protocollo, porta e host di origine o destinazione. Le regole da applicare a un determinato host vanno inserite nella configurazione di quello specifico host. La valutazione della policy di sicurezza avviene localmente, senza contattare un policy administrator centrale. Il trust algorithm utilizzato è quindi singolare e basato su criteri. Ogni host deve conoscere solo le regole che lo riguardano.

[2] `https://noiseprotocol.org/`

Dato che Nebula lavora al livello 3 dello stack di Internet i protocolli supportati nella definizione delle regole sono TCP, UDP e ICMP. La porta può essere una porta specifica, un range di porte o "fragment" per pacchetti frammentati. Gli host di origine (o destinazione) possono essere indicati in diversi modi:

- Tramite il nome, se la regola va applicata solo al traffico di uno specifico host.
- Tramite il gruppo, se la regola va applicata al traffico di tutti gli host di un gruppo.
- Tramite un elenco di gruppi, se la regola va applicata al traffico degli host appartenenti esattamente a tutti i gruppi della lista.
- Un CIDR, se la regola va applicata al traffico degli host appartenenti a un determinato CIDR.

Tutte le opzioni, ovvero protocollo, porta e qualsiasi modalità venga usata per selezionare gli host, prevedono un valore col significato di "qualunque". Si può inoltre richiedere che il traffico permesso sia stato cifrato utilizzando chiavi certificate da una specifica CA.

Definire regole *host-based*, ovvero che selezionano il traffico in base allo specifico host di origine o destinazione, può essere fattibile in una rete di dimensioni limitate, ma risulta troppo complicato in una rete di medie o grandi dimensioni. Si può quindi dire che la maniera più sensata di definire le regole del firewall è in modo che siano *group-based*, ovvero che selezionino il traffico in base al gruppo (o i gruppi) a cui appartiene l'host di provenienza o destinazione.

Prendendo come riferimento altri strumenti simili a Nebula, come Netmaker[3], ZeroTier[4], Tailscale[5], il filtraggio che Nebula può fare del traffico è forse meno ricco e a grana più grossa, ma risulta anche più semplice da comprendere e definire.

5.1.3 Assenza di un controller centrale

La configurazione di Nebula viene definita separatamente per ogni host e non c'è un controller centrale da cui poter visualizzare lo stato generale della rete. Questa

[3]https://www.netmaker.org/
[4]https://www.zerotier.com/
[5]https://tailscale.com/

è una scelta di design di Nebula che evita che si crei un singolo punto di fallimento, oltre ad evitare che la compromissione di tale controller pregiudichi la sicurezza dell'intera rete.

Per permettere la connessione tra due nodi, infatti, il traffico deve essere permesso in entrata in un nodo e in uscita nell'altro. Anche se un attaccante riuscisse ad avere accesso ad uno dei nodi della rete, ipotizzando quindi di poterne modificare la configurazione per i propri scopi, non potrebbe automaticamente modificare la configurazione degli altri nodi: supponendo che il movimento laterale dell'attaccante tra i due nodi richieda una certa tipologia di traffico, a meno che questa non sia già permessa in entrata nel nodo di destinazione l'attaccante non avrà modo di completare il movimento. Al contrario, in presenza di un controller centrale basterebbe avere accesso ad esso per modificare la configurazione dell'intera rete.

L'assenza di una modalità per la configurazione centralizzata della rete, quindi, è sia un vantaggio in termini di sicurezza sia uno svantaggio in termini di comodità e rapidità del cambiamento della configurazione dei nodi.

Le lighthouse non fungono da controller ma solo da servizio per far conoscere i nodi tra loro.

5.1.4 Motivazioni per l'uso di Nebula

Tra gli strumenti di overlay networking disponibili in commercio se ne trovano sia di proprietari, come Tailscale, sia di open source, come Netmaker, Zerotier, Tinc[6], Nebula. Tra questi ultimi strumenti open source e gratuiti, Nebula è probabilmente quello che permette la messa in esecuzione più rapida e semplice. I requisiti di esecuzione di Nebula, inoltre, sono minimi, anche se non chiaramente specificati.

A differenza di Zerotier e Netmaker, ad esempio, una rete Nebula non ha nessun controller centralizzato ed è totalmente autonoma, gestita unicamente dall'utilizzatore. In questo senso Nebula garantisce il totale controllo dei propri dati e potenzialmente una maggiore resilienza, se nella rete sono presenti abbastanza lighthouse.

Un'ulteriore elemento che differenzia Nebula da alcuni degli altri strumenti di overlay networking è la gestione del traffico tramite le regole del firewall: mentre per definire le regole di una rete Nebula occorre ragionare in termini di host al

[6]https://www.tinc-vpn.org/

livello 3 dello stack di Internet, gli altri strumenti citati richiedono di ragionare o a livelli diversi dello stack di Internet, o introducendo il concetto di utenti, o senza il concetto di gruppi di host.

Rispetto ad altri strumenti, quindi, Nebula risulta sì meno potente nella definizione delle regole di gestione del traffico, ma è di più semplice utilizzo e gestione, senza ridurre la sicurezza della rete. Un ulteriore vantaggio che Nebula ha rispetto ad alcuni degli strumenti simili esistenti è che è disponibile anche per processori non x86/64, rendendo quindi possibile il suo utilizzo anche su dispositivi utilizzati in ambito IoT, come i Raspberry Pi. Sono inoltre disponibili client Nebula per Android e iOS, che permettono di dare accesso a una rete Nebula sia a microservizi eseguiti in cloud sia a utenti che desidirino accedere a tali microservizi da tablet o smartphone.

Facendo riferimento alle necessità elencate in Sezione 4.4.1, l'unica non soddisfatta da Nebula è quella relativa alla modifica della configurazione durante l'esecuzione: l'hot reloading è supportato solo per alcune impostazioni. Per il resto Nebula offre capacità di firewall con valutazione locale della policy di sicurezza, non presenta un controller centralizzato, cifra il traffico e permette di identificare ogni singolo nodo tramite l'uso di certificati e chiavi private. Dato inoltre che Nebula può essere eseguito in un container si ha anche una certa elasticità su che risorse una singola istanza possa proteggere: essa potrebbe essere eseguita su una macchina virtuale, per difendere tutte le risorse di quella macchina, o anche come container sidecar di un Pod di Kubernetes, per gestire le connessioni al singolo Pod. La granularità delle policy di sicurezza che potranno essere definite è quindi abbastanza fine. È inoltre degno di nota come, a differenza di software come DxOdissey, Nebula permetta di avere più lighthouse e di gestirle in autonomia, garantendo una maggiore resistenza a problemi di rete. Il livello di robustezza può essere scelto decidendo il numero di lighthouse della rete: la rete in Figura 5.1 avendo una sola lighthouse non sopporta problemi a questa tipologia di nodi; la rete in Figura 5.2 è in grado di continuare a funzionare anche con un numero maggiore di lighthouse irragiungibili o malfunzionanti perché esse sono replicate. Essendoci quattro lighthouse nella rete di Figura 5.2, in qualsiasi momento in cui tre o meno di esse hanno problemi gli altri host continuano ad essere in grado di conoscersi tramite la quarta lighthouse, quella non affetta da problemi. In generale, supponendo che tutti gli host siano a conoscenza di tutte le lighthouse, più sono le lighthouse di una rete Nebula maggiore

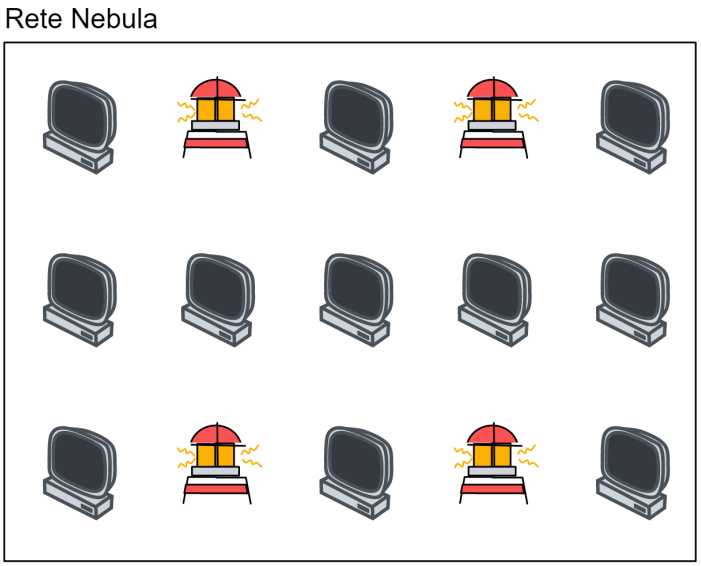


Figura 5.2: Esempio di rete Nebula con multiple lighthouse.

è la sua resistenza.

Viste queste caratteristiche, Nebula è idoneo ad essere utilizzato come base per realizzare una ZTA basata su microsegmentazione agent-based.

5.2 Linguaggio Dhall

Il linguaggio Dhall è un linguaggio di configurazione programmabile [31], utilizzato per definire *file di configurazione programmabili*. Come descritto dalla documentazione del linguaggio, un file di configurazione programmabile è un file di configurazione che è anche un'espressione in un linguaggio di programmazione [10], la cui valutazione porta a ridurre il file a una forma normale inerte. I file di configurazione programmabili possono essere usati così come sono o possono essere tradotti in un altro linguaggio di configurazione.

5.2.1 Struttura del linguaggio

Essendo Dhall un linguaggio di configurazione programmabile esso presenta caratteristiche tipiche di un linguaggio di programmazione, come funzioni e tipi di dati. La documentazione del linguaggio descrive Dhall come JSON con l'aggiunta di funzioni [10].

In Dhall le funzioni possono essere o meno anonime, con la possibilità di definirle sotto forma di *lambda*. Tutte le funzioni, anonime e non, possono essere utilizzate come argomenti di altre funzioni. Un esempio di ciò, visibile nel frammento 5.1, è il passaggio di una funzione, definita sotto forma di lambda, come argomento alla funzione `List/map`.

Frammento 5.1: Esempio di uso di `List/map` in Dhall.

```
let users : List Text = [ "fabio", "lisa", "matteo" ]
in List/map
    Text
    Text
    (\(name : Text) -> "/home/${name}")
    list
-- [ "/home/fabio", "/home/lisa", "/home/matteo" ]
```

Dhall è stato pensato per incoraggiare uno stile di programmazione fortemente tipizzato, per cui ogni elemento della configurazione ha un tipo. I tipi possono essere tipi base (built-in nel linguaggio), unioni (*sum type*) o record (*product type*). Per i campi di un product type possono essere definiti dei valori di default, che possono poi essere utilizzati per autocompletare i valori di tale tipo. L'elemento che definisce i valori di default dei campi di un record è detto *schema*.

Una configurazione Dhall può essere suddivisa in più file, che possono fare riferimento l'uno all'altro, evitando riferimenti circolari. Ciò permette di organizzare al meglio la configurazione, rendendo più facile modificarla e mantenerla nel tempo.

5.2.2 Principio DRY

La possibilità di utilizzare funzioni durante la definizione di una configurazione permette di automatizzare la generazione di elementi ripetuti della configurazione, nel rispetto del principio DRY, "Don't repeat yourself". Un esempio di ciò è in frammento 5.1, in cui per definire le directory home dei vari utenti non è necessario indicarle una per una: in caso di cambio del path delle directory home sarà sufficiente modificare la funzione che genera il path a partire dal nome utente, mentre in caso di

aggiunta o rimozione di un utente sarà sufficiente aggiungerne o rimuoverne il nome dalla lista degli utenti. Il vantaggio di ciò è soprattutto in termini di manutenibilità e leggibilità, oltre che in termini di riduzione degli errori dovuti a disattenzione.

Anche la possibilità di definire valori di default per i campi dei record aiuta nel rispetto del principio DRY: invece di indicare ripetutamente in più punti della configurazione gli stessi valori è possibile indicarli come default in un unico punto; in questo modo modifiche a tali valori dovranno essere apportate una sola volta invece che in multiple posizioni.

In generale, il fatto che Dhall permetta di definire una configurazione come se fosse del classico codice permette di attuare tutte quelle buone pratiche ormai riconosciute come standard nella programmazione "classica", come l'evitare l'uso di *magic numbers* e la limitazione delle ripetizioni. Questo va oltre il principio di Configuration as Code: la configurazione non è solo trattata come codice, è effettivamente codice.

5.2.3 Validazione

Un'altra importante caratteristica del linguaggio Dhall è la possibilità di definire delle funzioni di validazione della configurazione stessa, in modo che la riduzione dei file a una forma normale fallisca se il loro contenuto non soddisfa determinate condizioni.

Per effettuare la validazione occorre innanzitutto definire una funzione che confronti i risultati dei vari controlli sui valori con i risultati attesi; ciò è solitamente fatto definendo dei record (anche innestati) con campi di tipo booleano, con ogni campo che corrisponde a un controllo, e confrontando il record con i risultati effettivi e il record con i risultati attesi. L'effettiva validazione avviene a type-checking time [10], tramite la keyword `assert`, che verifica che l'equivalenza tra espressioni indicata dalla funzione di validazione precedentemente definita sia vera. Nel frammento 5.2 è possibile osservare un esempio di validazione.

La possibilità di validare la configurazione a type-checking time, quindi anche prima del momento dell'utilizzo, permette di non dover eseguire controlli di validità nei propri programmi a runtime, in quanto essi vengono effettuati prima. Inoltre, evita di dover eseguire il software che utilizza la configurazione ogni volta che se ne desidera controllare la validità, anche durante la sua definizione. Mentre è

sicuramente possibile definire tool che verifichino la correttezza di una configurazione JSON o YAML, la possibilità di integrare questa funzione nella configurazione stessa ne semplifica lo sviluppo.

Frammento 5.2: Esempio di validazione in Dhall.

```
let nCPUs = 3
let validate = \(n : Natural) ->
    let expected = { nCheck = True }
    let actual = { nCheck = Natural/lessThan n 5 }
    in expected === actual
let _ = assert : validate nCPUs
in nCPUs
```

Definire le funzioni di valutazione della configurazione è un compito in più che richiede tempo e risorse, ma nel tempo porta vantaggi in termini sia di sicurezza sia di manutenibilità.

5.2.4 Limitazioni del linguaggio

Le principali limitazioni del linguaggio Dhall sono legate a specifiche scelte di design, alcune fatte per ragioni di sicurezza, altre per obbligare a seguire determinate pratiche di programmazione.

Una delle maggiori limitazioni è che Dhall non è un linguaggio Turing-completo, ma un linguaggio funzionale totale. Per poter essere un linguaggio funzionale totale Dhall non supporta la ricorsione (risultando quindi non Turing-completo), ottenendo in compenso di poter sempre provare che i suoi "programmi" terminano in un tempo finito (anche se non necessariamente breve). Il fatto che Dhall non sia Turing-completo è una scelta di design fatta per avere garanzie di sicurezza al livello di quelle di altri linguaggi di configurazione non programmabili [10].

Un'altra grande limitazione di Dhall è l'impossibilità di controllare l'uguaglianza di due valori testuali, o anche solo di calcolarne la lunghezza: l'unica operazione permessa sui valori di tipo `Text` è la concatenazione. Questa limitazione è data dalla volontà di incoraggiare l'uso di rappresentazioni più strutturate che catturino gli

errori a type-checking time invece di fallire silenziosamente a runtime [10]. In alcuni casi questa limitazione è conveniente, in altri no. Volendo ad esempio rappresentare i protocolli che Nebula usa nella definizione delle proprie regole firewall, si potrebbe definire un tipo unione i cui valori sono i tre protocolli e un valore che indica un qualsiasi protocollo. Ciò permetterebbe di essere certi che nella configurazione non siano presenti protocolli non supportati, perché in virtù della tipizzazione forte di Dhall si dovrà per forza usare uno dei valori del tipo unione quando richiesto. I nomi invece degli host Nebula devono per forza essere rappresentati con dei valori di tipo `Text`, essi non possono essere rappresentati con delle unioni dato che non è possibile stabilire a priori l'insieme dei possibili nomi. È però necessario verificare la loro univocità all'interno della rete, operazioni che Dhall non permette a causa delle limitazioni poste sul tipo `Text`.

Come per le stringhe, per molti altri tipi di dato non è possibile controllare l'uguaglianza dei valori. Un esempio di ciò sono i valori `Double`, data l'imprecisione dell'aritmetica floating point. Di conseguenza, Dhall non supporta né mappe né set, in quanto non sarebbe possibile controllare se due chiavi o due elementi sono uguali tra loro. Esiste un tipo di dato `Map`, equivalente a una lista di record con i campi `entryKey` ed `entryValue`, che però non verifica la presenza di chiavi duplicate.

5.2.5 Ragioni per l'uso di Dhall

Nonostante le sue limitazioni, il linguaggio Dhall risulta comunque una scelta ideale per definire la configurazione di un overlay network da cui poi generare le configurazioni YAML per Nebula per ogni singolo host.

Rispetto alle alternative come JSON e YAML, Dhall offre maggiori certezze sulla correttezza della configurazione, grazie alla possibilità di definire come validarla e alla forte tipizzazione del linguaggio. Per JSON e YAML una funzione di validazione andrebbe definita ed eseguita come script a parte, mentre in Dhall la validazione può essere integrata direttamene nella configurazione, assicurandone una più semplice esecuzione. La tipizzazione forte aiuta nell'individuare errori nella configurazione prima del suo uso a runtime.

Oltre alle assicurazioni sulla correttezza della configurazione, l'uso di Dhall presenta vantaggi rispetto a JSON e YAML per la possibilità di ridurre le ripetizioni (ad esempio, permettendo di definire una sola volta la configurazione base di ogni

nodo della rete) e per la possibilità di generare automaticamente certi aspetti della configurazione.

Scegliendo di definire la configurazione della rete in JSON o YAML sarebbe necessario definire uno strumento che estragga da essa la configurazione dei singoli host, mentre utilizzando Dhall tale strumento può essere direttamente integrato nella configurazione sotto forma di funzione definita appositamente. Ciò porta vantaggi in termini di manutenibilità. Un ulteriore vantaggio di Dhall è che, usando gli strumenti già esistenti, è possibile generare direttamente una configurazione YAML a partire da una configurazione Dhall strutturata in modo complesso, mentre al contrario convertire una configurazione YAML in Dhall avrebbe come risultato un semplice cambio di sintassi. Ad esempio, in Dhall si potrebbe definire una serie di valori, raggruppandoli in diverse liste, contenute in un record, senza ripetizioni. La traduzione di questa configurazione in YAML vedrebbe esplicitati tutti i valori ripetuti. Volendo generare la configurazione Dhall a partire dalla configurazione YAML queste ripetizioni continuerebbero ad esistere. Prendere come punto di partenza una configurazione Dhall quindi permette di avere una definizione più chiara e di alto livello di ciò che si desidera rappresentare, pur mantenendo la possibilità di esprimere gli stessi concetti in YAML; non è vero il contrario: prendendo come punto di partenza la configurazione YAML non sarebbe possibile in alcun modo ottenere una rappresentazione in Dhall che non sia una esatta trasposizione di quella in formato YAML.

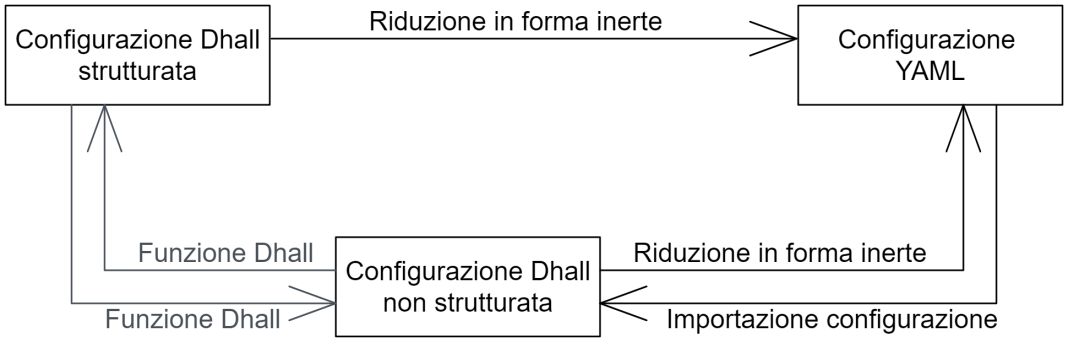


Figura 5.3: Schema delle conversioni possibili tra configurazioni definite in Dhall e configurazioni definite in YAML.

In realtà, come si può vedere in Figura 5.3 dove sono rappresentate le conversioni possibili tra configurazioni Dhall e configurazioni YAML, sarebbe possibile passare da una configurazione YAML a una configurazione Dhall più strutturata passando attraverso una configurazione Dhall non strutturata e usando delle funzioni definite appositamente. Questo passaggio però richiede di implementare funzioni in Dhall che descrivano la struttura della configurazione Dhall: è molto più rapido ed efficace definire direttamente la configurazione in Dhall con la struttura voluta.

Considerando quindi le sue funzionalità di validazione, la possibilità di definire funzioni e la disponibilità di strumenti per la conversione della configurazione Dhall in JSON o YAML, il linguaggio Dhall risulta una scelta più comoda e sicura delle alternative nell'implementazione dell'approccio alla microsegmentazione proposto in questa tesi.

6 Creazione di una rete con microsegmentazione agent-based e configurazione centralizzata

Come descritto in Sezione 4.4.1, l'implementazione di un approccio scalabile alla microsegmentazione richiede l'uso di uno strumento agent-based che permetta la creazione di un overlay network con funzionalità di controllo del traffico e di uno strumento che permetta di definire in modo centralizzato la configurazione per i vari agenti.

Nebula, selezionato come strumento per l'overlay networking, richiede che per ogni nodo della rete vengano generati la configurazione e la coppia di chiavi pubblica e privata, firmate dalla Certification Authority; il punto di partenza deve però essere la configurazione della rete definita in maniera centralizzata.

È stato scelto di definire la configurazione centralizzata in Dhall e a partire da essa generare tutti i file necessari, tramite uno strumento implementato ad hoc. Lo sviluppo di questo strumento è stato svolto in due fasi: come prima cosa è stato definito come descrivere la rete Nebula in un file di configurazione Dhall, in un secondo tempo è stato implementato il software che presa la configurazione della rete genera i file necessari al funzionamento delle varie istanze di Nebula.

La valutazione dell'approccio che questa tesi propone verterà quindi sulla combinazione di Nebula e dello strumento sviluppato.

6.1 Configurazione centralizzata per Nebula

Nebula, come spiegato in Sezione 5.1.3, purtroppo non offre la possibilità di configurare una rete in maniera centralizzata, con l'applicazione automatica dei

cambiamenti, ma obbliga a configurare ogni host singolarmente.

Una configurazione centralizzata non può limitarsi ad essere l'aggregazione delle configurazioni dei singoli nodi, soprattutto in caso di reti molto grandi. Definire la configurazione per ogni nodo può risultare complicato, perchè per ogni configurazione punto a punto che si desidera permettere occorre indicare le relative regole inbound e outbound, su entrambi gli host. Nel caso si parli di regole che coinvolgono tutti gli host di un gruppo sarà necessario inserire tali regole nella configurazione di tutti gli host del gruppo. Con un numero elevato di host o gruppi nella rete questo processo è soggetto ad errori e dimenticanze, che potrebbero essere notati troppo tardi. Per evitare questi errori è necessario poter definire la configurazione della rete in termini di più facile comprensione e gestione per un essere umano.

La configurazione centrale quindi deve essere composta da due componenti: le configurazioni degli host, che descrivono caratteristiche specifiche dei singoli nodi, e la configurazione delle comunicazioni, che descrive quali comunicazioni devono essere permesse sulla rete Nebula. Non è possibile evitare di descrivere ogni singolo host, perché ci sono impostazioni come la dimensione della coda di trasmissione dell'interfaccia di rete usata che cambiano per ogni nodo. Le impostazioni delle comunicazioni invece riguardano più nodi per volta, quindi è necessario astrarre questa configurazione: invece di definire le regole firewall dei singoli nodi si andranno a definire le comunicazioni permesse sulla rete. Al posto di dire quale sia il traffico permesso in entrata su un nodo e in uscita su un altro, nella configurazione centrale sarà possibile specificare che una certa tipologia di traffico è permessa da un certo nodo a un altro.

Per gli esseri umani è più facile ragionare in termini di singole comunicazioni tra gruppi di host che in termini di multiple comunicazioni punto a punto. Ad esempio, supponendo di avere tre host che devono tutti essere in grado di comunicare l'uno con l'altro, è più semplice pensare "tutti gli host del gruppo alfa devono poter comunicare tra loro", piuttosto che "l'host A deve poter comunicare con B e C, l'host B deve poter comunicare con A e C, l'host C deve poter comunicare con A e B". Per questo motivo la definizione delle comunicazioni nella configurazione centrale sfrutterà il concetto di gruppi di nodi.

Si può dire che la parte di configurazione riguardante i singoli host sia una questione di Configuration as Code, mentre la parte di configurazione riguardante

le comunicazioni è una questione di Policy as Code.

6.2 Configurazione centralizzata in linguaggio Dhall

La configurazione centralizzata di Nebula, sia per gli host che per le comunicazioni, è stata definita in linguaggio Dhall. La configurazione Dhall può essere suddivisa in quattro parti:

- Definizione dei tipi.
- Definizione degli *schema* relativi ai tipi.
- Funzioni, in particolare funzioni per la conversione della configurazione da Dhall a YAML e funzioni per la validazione della configurazione.
- Configurazione della rete, definita utilizzando le componenti precedenti.

Le definizioni dei tipi, degli schema e delle funzioni vengono riutilizzate per ogni rete di cui occorre definire la configurazione; la configurazione della rete viene definita in uno o più file separati e fa riferimento ad esse. Le definizioni dei tipi, degli schema e delle funzioni sono esportate da un unico file Dhall, "package.dhall", a cui si fa riferimento dai file che contengono la configurazione della rete. L'organizzazione dei file, mostrata in Figura 6.1, mette in evidenza come non solo il codice Dhall sia stato suddiviso in base alla sua funzione, ovvero definizione dei tipi, degli schema e delle funzioni, ma anche in base ai concetti trattati, ovvero nodi, comunicazioni, configurazione YAML, validazione.

In particolare si può notare come nella cartella "utils", contenente i file che definiscono le funzioni Dhall, siano presenti più file che nelle altre cartelle: questo accade perché nella configurazione Dhall si è fatto ampio uso di funzioni, rendendo necessario suddividerle in molti file in base al loro scopo per una maggiore comprensibilità del codice.

Questa suddivisione in diversi file e directory garantisce una maggiore manutenibilità del codice. Ciò è stato verificato già durante la definizione di tipi e funzioni, avendo scelto di svolgere questa operazione in maniera incrementale. Inizialmente in questa definizione erano state incluse solo una parte delle impostazioni relative ai nodi Nebula, poi per aggiungere quelle mancanti è stato sufficiente modificare i

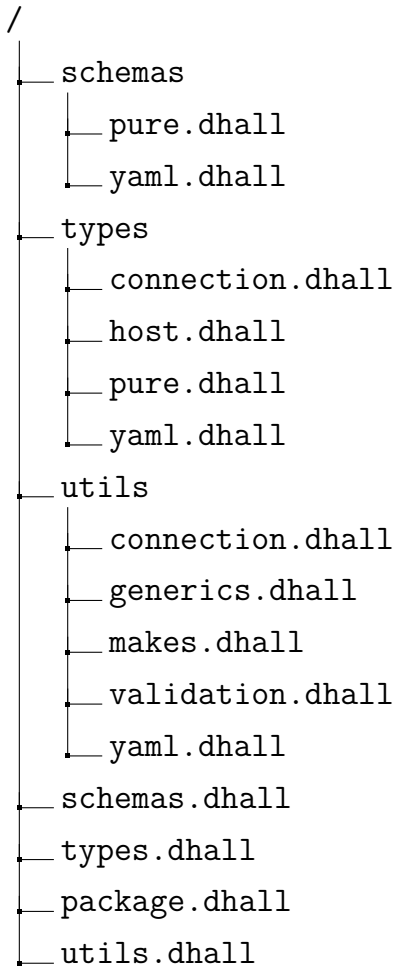


Figura 6.1: Struttura dei file Dhall

file relativi agli host nelle varie directory. Il fatto che le funzioni siano fortemente tipizzate ha inoltre permesso di identificare facilmente le modifiche da attuare in seguito alla modifica dei tipi: il nuovo tipo non risultava compatibile con le operazioni effettuate su di esso e quindi il processo di type checking dava errore, permettendo di individuare facilmente e correggere gli errori prima che le definizioni fossero effettivamente usate nella configurazione di una rete. Questo ha dimostrato la facilità di manutenzione della configurazione definita in Dhall.

6.2.1 Configurazione host

All'interno della configurazione della rete Nebula è necessario prevedere le configurazioni per ogni nodo, in quanto alcune impostazioni per l'esecuzione di Nebula non sono legate alla rete bensì al nodo stesso. Queste impostazioni sono relative, ad esempio, a dove nel file system l'eseguibile di Nebula dovrà cercare certificati e

chiavi, a come dovrà essere effettuato il logging, su quale interfaccia di rete dovrà rimanere in ascolto l'eseguibile, a quali altre reti sono raggiungibili dall'host. Il tipo di dato relativo a queste configurazioni è `Host`, rappresentato nel frammento 6.1.

Frammento 6.1: Definizione del tipo Dhall relativo agli host.

```
let Host
    : Type
    = { name : HostName
      , ip : IPv4
      , lighthouse_config : Optional IsLighthouseConfig
      , pki : PkiInfo
      , lighthouse : LighthouseInfo
      , static_ips : List IPv4WithPort
      , listen_interface : ListenInfo
      , punchy : PunchyInfo
      , logging : LogInfo
      , tun : TunInfo
      , local_range : Optional Text
      , sshd : Optional SSHDInfo
      , am_relay : Bool
      , use_relays : Bool
      , relays : List IPv4
      }
```

Oltre a queste impostazioni che riguardando unicamente l'host ci sono una serie di parametri che sono specifici dell'host ma che vanno ad influenzare la configurazione finale degli altri host, come gli indirizzi IP statici, il fatto che l'host sia o meno una lighthouse, il fatto che esso possa fungere da relay o meno. Questi parametri vengono definiti per ogni host e poi quando viene generata la configurazione YAML dei singoli essi vengono utilizzati dove è necessario.

La configurazione degli host in generale è soggetta a validazione, ma per alcuni dei parametri che influenzano la configurazione di altri host è necessario verificare

non solo che essi siano validi ma che gli host influenzati vi facciano riferimento in maniera corretta. Un esempio di ciò sono le impostazioni relative alla modalità relay: nella configurazione di un host è indicato da che relay sono accettati pacchetti, ma occorre controllare che questi relay abbiano effettivamente il valore `am_relay` della configurazione uguale a `True`.

Alcune degli elementi della configurazione degli host non trovano un corrispondente nella configurazione YAML di Nebula: questi elementi, `name` e `ip`, sono rilevanti a livello di rete e necessari al momento della generazione, firma e validazione dei certificati. Altri elementi trovano invece un corrispondente diretto e sono usati per inferire il valore di altri campi della configurazione; un esempio di ciò è il campo `lighthouse_config`, che ha tipo `Optional IsLighthouseConfig`. Questo tipo indica che il valore del campo potrebbe essere o un elemento di tipo `IsLighthouseConfig` o un elemento vuoto. Se l'host è una lighthouse il valore del campo è la configurazione della stessa lighthouse, se l'host non è una lighthouse il valore del campo è nullo.

Aver definito il tipo `Host` non direttamente corrispondente alla configurazione di un host Nebula porta ad avere una maggiore complessità nella parte riutilizzabile della configurazione Dhall, dato che è stato ncessario definire delle funzioni di conversione. Allo stesso tempo questa separazione tra `Host` e la configurazione Nebula facilità il ragionamento sulla e la definizione della configurazione dell'intera rete.

6.2.2 Configurazione delle comunicazioni permesse

Ragionando su quali comunicazioni debbano essere permesse uno dei modi in cui è più naturale pensare è in termini di comunicazioni punto a punto e in termini di comunicazioni all'interno di un gruppo di host. Permettere però la definizione solo di questi due tipi di connessione risulterebbe limitante, dato che ci potrebbero casi in cui la comunicazione permessa deve essere solo unidirezionale o in cui tutti gli host devono poter comunicare liberamente.

Si può quindi dire che una comunicazione generica avvenga tra due entità e che possa essere unidirezionale o bidirezionale. Le entità in questione potrebbero essere gruppi di host, o singoli host. Una comunicazione bidirezionale può essere

considerata l'unione di due connessioni unidirezionali in cui origine e destinazione vengono invertite.

Seguendo questo ragionamento, la configurazione Dhall relativa alle comunicazioni permesse sulla rete definisce il tipo `Connection` come equivalente ad una lista di `UnidirectionalConnection`, ovvero di connessioni unidirezionali. L'ideale sarebbe stato definire una connessione come o una connessione unidirezionale o una coppia di connessioni unidirezionali, controllando in quest'ultimo caso che esse fossero definite tra gli stessi estremi, ma invertiti. Purtroppo Dhall non permette, senza andare a rendere eccessivamente complesso il codice, la definizione di tali tipi di dato. È per tale motivo che è stato scelto di definire una connessione come lista di connessioni unidirezionali, come mostrato nel frammento 6.2. L'alternativa sarebbe stata definire un tipo unione, ma ciò avrebbe reso più complessa la funzione che a partire da un valore di tipo `Connection` genera le regole del firewall di un nodo. Si è preferito implementare una funzione più semplice, piuttosto che definire un tipo di dati più preciso ma più complesso, non essendoci particolari controindicazioni all'uso di liste di `UnidirectionalConnection` per rappresentare una connessione.

Frammento 6.2: Definizione del tipo Dhall relativo alle connessioni di rete.

```
let ConnectionTarget
    : Type
    = < CTGroup : Group
      | CTHost : host.Host
      | CTCidr : host.IPv4Network
      | AnyNebulaHost
      | AnyExternalHost
      >
let UnidirectionalConnection
    : Type
    = { uc_port : Port
      , uc_proto : Proto
      , from : ConnectionTarget
      , to : ConnectionTarget
      , ca_name : Optional Text
```

```
      , ca_sha : Optional Text
      }
let Connection
    : Type
    = List UnidirectionalConnection
```

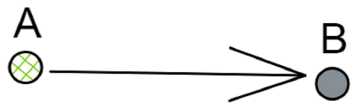
Una connessione unidirezionale è caratterizzata da porta, protocollo, origine e destinazione. Inoltre, visto che le regole firewall di Nebula lo supportano, per ogni connessione unidirezionale può essere indicato il nome o l'hash della CA che deve aver firmato i certificati dei pacchetti che compongono il traffico della connessione. In Figura 6.2 sono rappresentate i possibili tipi di comunicazione unidirezionale che vedono come destinazione un singolo host: l'origine può essere un singolo host, oppure tutti gli host di un determinato gruppo, tutti gli host della rete oppure gli host di una determinata sottorete IP. La destinazione di una comunicazione viene definita allo stesso modo dell'origine e può quindi essere uno specifico host, gli host appartenenti a un determinato gruppo, qualunque host della rete, oppure un host di una certa sottorete IP.

Non è stata considerata la possibilità che una delle estremità della comunicazione debba poter appartenere a più gruppi: pur essendo un'opzione ragionevole, è stato valutato che dover definire ogni gruppo singolarmente vada a creare maggiore consapevolezza di che host saranno coinvolti nella comunicazione. Questa maggiore consapevolezza si traduce in un minor rischio di errori e quindi una maggiore sicurezza.

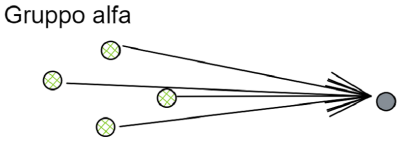
6.2.3 Configurazione della rete

Una configurazione Dhall può direttamente essere convertita in YAML, quindi è possibile definire un tipo di dato Dhall che corrisponda esattamente alla configurazione YAML di un nodo Nebula.

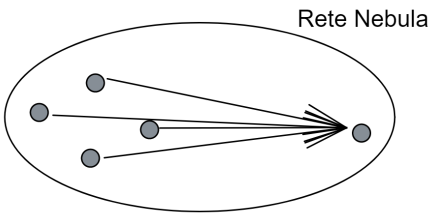
La configurazione Dhall di un nodo Nebula comprende tutte le informazioni della corrispondente definizione di tipo `Host`, più le regole del firewall ricavate a partire dalle connessioni permesse, più altre impostazioni che devono essere definite a livello della rete Nebula.



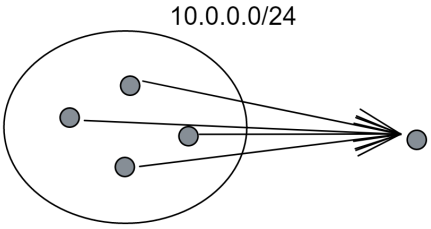
(a) Da singolo host a singolo host



(b) Da gruppo di host a singolo host



(c) Da tutti gli host a singolo host



(d) Da host di una sottorete a singolo host

Figura 6.2: Esempi di comunicazioni unidirezionali da origini di vario tipo verso un singolo host.

È quindi conveniente andare a definire un tipo di dato che descriva la rete Nebula, più una funzione che permetta di generare la configurazione Dhall direttamente convertibile in YAML di un nodo, a partire dalla configurazione della rete e indicando l'host per cui la si vuole generare. Il processo logico di generazione della configurazione YAML a partire dalla descrizione di connessioni permesse e host diventa quindi quello rappresentato in Figura 6.3: la configurazione della rete è composta dalla configurazione dei nodi e delle connessioni e viene usata per generare la configurazione Dhall di un host direttamente corrispondente alla configurazione YAML di un nodo Nebula; a partire da questa viene generata la configurazione YAML del nodo.

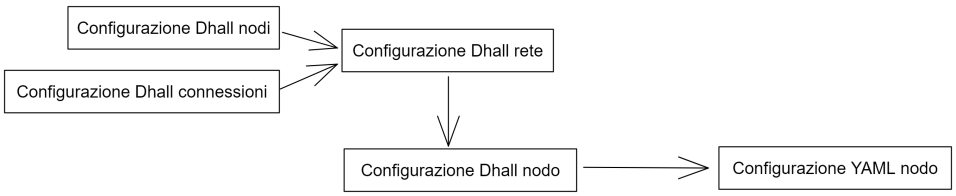


Figura 6.3: Schema logico della generazione della configurazione YAML di un nodo Nebula.

Descrizione della rete Nebula

La configurazione della rete consiste nella definizione di un elemento di tipo `Network` contenente la lista delle configurazioni degli host della rete, la lista dei gruppi di host, la lista delle comunicazioni permesse, la lista dei certificati revocati, il cifrario da utilizzare per cifrare le comunicazioni e il numero di bit della maschera di rete da applicare agli indirizzi IP degli host.

La lista dei certificati revocati è da utilizzare nel caso ci siano degli host i cui certificati non debbano più essere considerati validi. La lista contiene le *fingerprint* dei certificati da considerare invalidi, ma la cui CA rimane ancora fidata. Questo elenco è definito a livello di rete in quanto è stato reputato che se un certificato deve essere invalidato per un host, deve essere invalidato anche per tutti gli altri. La principale ragione per invalidare un certificato è che la chiave privata ad esso associata sia compromessa: in tale situazione nessuno può più fidarsi del certificato, quindi la lista dei certificati revocati deve essere aggiornata in tutti gli host della rete.

Il cifrario da utilizzare per cifrare le comunicazioni viene definito a livello di rete, dato che tutti gli host devono usare lo stesso cifrario. È escluso che in futuro possa essere supportato l'uso di cifrari diversi all'interno della stessa rete [9], quindi inserire il cifrario come elemento della configurazione dei singoli host, prevedendo poi di controllare che sia sempre lo stesso, sarebbe inutile.

In modo analogo al cifrario, la maschera di rete deve essere la stessa per tutti gli host, quindi è più conveniente definirla una volta nella configurazione della rete piuttosto che definirla multiple volte nelle configurazioni degli host.

In una rete Nebula con un elevato numero di nodi potrebbe essere scomodo dover definire manualmente la configurazione di ogni singolo nodo: un modo per evitare questa operazione ripetitiva è usare delle funzioni che definiscono la configurazione dei nodi in base al loro indirizzo IP. Allo stesso modo si potrebbero definire i gruppi di nodi.

Generazione della configurazione YAML

L'applicativo Haskell richiede in input un file Dhall che esporti unicamente un elemento di tipo `Network`. Dato questo file, e tutti gli altri file che definiscono tipi

e funzioni della configurazione, è possibile generare la configurazione YAML per un nodo, dato il suo indirizzo IP, utilizzando le funzioni Dhall. Viene usato l'indirizzo IP piuttosto che il nome poiché Dhall non permette il controllo di uguaglianza tra stringhe: non sarebbe stato possibile per una funzione ricevere come input il nome del nodo e cercare nell'elenco degli host il valore di tipo `Host` con il valore del campo `name` uguale all'input. Per rappresentare l'indirizzo IP è stato definito un apposito tipo di dato, in pratica equivalente ad una quadrupla di numeri interi. Per valutare l'uguaglianza tra due indirizzi IP viene semplicemente controllato che gli elementi corrispondenti delle due quadruple siano uguali.

Il codice Dhall usato per generare la configurazione direttamente convertibile in YAML è rappresentato nel frammento 6.3.

Frammento 6.3: Codice Dhall per la generazione della configurazione YAML di un determinato nodo.

```
let network = ./vm-laptop-azure.dhall
let nebula = ./package.dhall
in  nebula.configFromIP
      network
      (nebula.mkIPv4 192 168 100 2)
```

È importante notare come in questo frammento di codice le uniche componenti variabili siano il nome del file contente la configurazione della rete e l'indirizzo IP: generare questo frammento di codice, a partire da tali valori, è molto semplice e può essere fatto automaticamente concatenando delle stringhe.

In una prima versione della configurazione Dhall insieme all'elemento di tipo `Network` era necessario esportare esplicitamente ogni singolo host definito, rendendo così complicato e prono ad errori il processo di definizione della configurazione. In questa versione si assumeva inoltre che il nome di un host fosse lo stesso nome della variabile che ne conteneva la configurazione. Nella versione mostrata nel frammento 6.3 invece ciò non è necessario, eliminando quindi il rischio che la generazione del file YAML fallisca a causa di una esportazione dimenticata. Il problema di questa versione è che potrebbe non essere trovato nessun host all'interno della rete con quell'indirizzo IP, evento che viene gestito usando il tipo built-in di

Dhall `Optional`: il risultato della normalizzazione della configurazione potrebbe essere la configurazione YAML o una configurazione vuota.

Questo comportamento è discutibile: in entrambi i casi si ha un fallimento della generazione della configurazione YAML, ma in un caso tale fallimento è silenzioso, mentre nell'altro no. Un fallimento esplicito è sicuramente meglio che un fallimento silenzioso, ma l'esportare singolarmente ogni host rimane comunque un compito sgradito. La soluzione a tale problema è arricchire il frammento 6.3 con delle funzionalità di validazione, che controllino che la configurazione generata non sia nulla. Questa modifica può essere osservata nel frammento 6.4; si può anche osservare come gli unici elementi variabili del codice rimangono il nome del file da cui è esportata la configurazione della rete e l'indirizzo IP dell'host per cui generare la configurazione.

Frammento 6.4: Codice Dhall per la generazione della configurazione YAML di un determinato nodo con controllo per risultati nulli.

```
let network = ./vm-laptop-azure.dhall
let nebula = ./package.dhall
let Optional/null =
      https://prelude.dhall-lang.org/v21.1.0/Optional/null
let yaml =
      nebula.configFromIP
        cnetwork
        (nebula.mkIPv4 192 168 100 2)
let isNone = Optional/null nebula.HostConfig yaml
let _ = assert : nebula.validate network
let _ = assert : False === isNone
in  yaml
```

Il codice in frammento 6.4 non fa parte della parte di configurazione che viene esportata dal file "package.dhall": si tratta di un esempio dell'utilizzo che si dovrebbe fare di tale configurazione.

A grandi linee per poter generare la configurazione YAML di un nodo occorrono il file "package.dhall" (con tutti gli altri file a cui fa riferimento), un file "network.dhall"

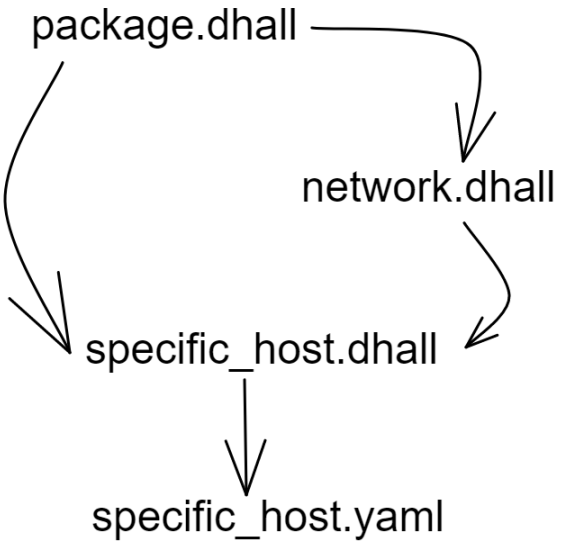


Figura 6.4: Schema ad alto livello delle dipendenze tra i vari file di Dhall ai fini della generazione della configurazione YAML di un nodo.

che esporta la configurazione della rete e un file "specific_host.dhall" che contiene il codice in frammento 6.4. In Figura 6.4 sono rappresentate le dipendenze tra questi file: sia "network.dhall" sia "specific_host.dhall" hanno bisogno dei tipi e delle funzioni esportate in "package.dhall", in più "specific_host.dhall" ha bisogno della configurazione della rete esportata da "network.dhall". A partire da "specific_host.dhall" è possibile generare direttamente la configurazione YAML dello specifico nodo scelto.

Tutte le lighthouse della rete sono inserite nella configurazione di un nodo, mentre i relay da cui un nodo accetta pacchetti devono essere indicati manualmente dallo sviluppatore. Questa scelta è stata fatta considerando eventuali costi della comunicazione tra host in reti e cloud diversi e le necessità di robustezza della rete.

Le lighthouse sono scarsamente coinvolte nella comunicazione tra gli host, il loro ruolo è unicamente di far "conoscere" i vari nodi. Più lighthouse un nodo conosce, più possibilità ci sono che esso sia in grado di contattare almeno una di esse per creare un tunnel criptato verso un altro nodo, quindi più sono le lighthouse note a tutti i nodi maggiore è la robustezza della rete. Considerando che i messaggi scambiati tra le lighthouse e i nodi sono pochi le spese per la comunicazione tra loro non risulterebbero eccessive. Seguendo questo ragionamento la generazione della configurazione YAML è fatta in modo che tutti i nodi conoscano tutte le lighthouse:

l'aumento della robustezza della rete è netto con un aumento di spesa minimo.

Per i relay invece la questione è diversa: essi devono sostenere, potenzialmente, sia le proprie comunicazioni sia le comunicazioni di altri nodi che li usano come relay. In questo caso i costi delle comunicazioni potrebbero crescere rapidamente, in maniera dipendente dal tipo di applicazioni eseguite sulla rete Nebula e dalla disposizione degli host nei vari cloud e reti private. È stato giudicato che per evitare costi eccessivi dovessero essere gli sviluppatori a decidere dove posizionare i vari relay e chi dovesse utilizzarli, anche in considerazione della scalabilità della rete. Il rischio potrebbe essere infatti di sovraccaricare un nodo relay con il traffico di altri nodi, rendendogli impossibile gestire il proprio traffico.

6.2.4 Validazione configurazione

La validazione della configurazione avviene richiamando la funzione `validate` nel file Dhall che esporta la configurazione della rete, come rappresentato nel frammento 6.5. Al momento in Dhall non è possibile definire delle regole che i valori dei vari tipi di dato devono rispettare, quindi le funzioni di validazione devono essere richiamate manualmente.

Frammento 6.5: Codice Dhall per la validazione della configurazione di un elemento `network` di tipo `Network`.

```
let nebula = ./package.dhall
-- definizione della configurazione
let _ = assert : nebula.validate network
in network
```

Questa caratteristica pone la responsabilità di validare la configurazione della rete in mano a chi la definisce, che quindi può scegliere se eseguirla o meno. Eseguirla è sicuramente un vantaggio in termini di sicurezza e identificazione degli errori in fasi precoci dello sviluppo, ma pigrizia, distrazione o intenti malevoli potrebbero spingere a non eseguirla. Per ovviare a questo problema la validazione della configurazione viene eseguita dall'applicativo Haskell al momento della generazione

delle configurazioni dei singoli host, usando codice simile a quello rappresentato nel frammento 6.4, dove la validazione è già presente.

Pur essendo il più completa possibile, la validazione della configurazione della rete presenta alcune mancanze, dovute alle limitazioni del linguaggio Dhall. Queste mancanze, in particolare il controllo sulla duplicazione dei nomi degli host sulla rete, sono risolte dall'applicativo Haskell.

6.3 Applicativo per la generazione delle configurazioni dei nodi Nebula

Uno strumento per convertire una configurazione Dhall in una configurazione YAML è distribuito su GitHub. Il problema di tale strumento è che prende in input un singolo file Dhall, quindi per utilizzarlo servirebbe un altro strumento che:

1. A partire dalla configurazione della rete identifichi ogni host per cui occorre generare la configurazione.

2. Per ogni host identificato generi la configurazione, usando il codice in frammento 6.4 adattato allo specifico host.

Per leggere una configurazione Dhall sono disponibili librerie per diversi linguaggi di programmazione; per gli stessi linguaggi è anche disponibile la libreria che permette di generare dei file YAML a partire da dei file Dhall. Vista la situazione è più comodo implementare da zero uno strumento che svolga da sé tutte le operazioni necessarie piuttosto che implementare uno strumento che richiami un altro strumento esterno. Il linguaggio utilizzato per l'implementazione è stato Haskell.

6.3.1 Funzionalità dell'applicativo

Lo strumento sviluppato, utilizzabile da riga di comando, come prima operazione al momento dell'esecuzione legge la configurazione Dhall indicata, che come già detto in Sezione 6.2.3 deve esportare unicamente un valore di tipo `Network`. Lo strumento è così in grado di "esplorare" la configurazione, esaminando i singoli elementi che la compongono ed effettuando le dovute verifiche sulla sua validità.

Basandosi sulla configurazione letta, lo strumento offre le seguenti funzioni:

- `config`: genera le configurazioni YAML per ogni host della rete, secondo quanto indicato dalla configurazione Dhall della rete Nebula.
- `certificates`: genera le chiavi private e i certificati per ogni host della rete, secondo quanto indicato dalla configurazione Dhall della rete Nebula.
- `sign`: firma una chiave pubblica per un determinato host, indicando nel certificato generato le informazioni relative all'host ricavate dalla configurazione Dhall della rete Nebula.
- `autosign`: firma le chiavi pubbliche per tutti gli host della rete, come indicato nella relativa configurazione Dhall, supponendo che esse si trovino in una data directory, organizzate in una specifica struttura di sottodirectory.
- `verify`: verifica che un certificato sia valido e compatibile con la configurazione Dhall della rete Nebula.

La funzionalità `certificates` è una funzionalità di comodo per la fase di testing del progetto: in un ambiente di produzione, infatti, generare la coppia di chiavi pubbliche e private su una macchina e poi trasferirle su un'altra non è ideale. La procedura corretta da svolgere sarebbe generare le chiavi sul futuro nodo della rete, trasferire la sola chiave pubblica, generare il certificato (usando o la funzionalità `sign` o la funzionalità `autosign`) e copiarlo sulla giusta macchina; in questo modo la chiave privata non esiste mai al di fuori dal nodo su cui verrà utilizzata. La firma della chiave pubblica deve avvenire sulla macchina su cui è conservata la chiave privata della Certification Authority, sempre per evitare di copiare la chiave privata su altri host. La conservazione della chiave privata della CA è un punto critico della sicurezza della rete, la compromissione della chiave privata della Certification Authority renderebbe l'intera rete inaffidabile.

La funzione `verify` è pensata per controllare se un certificato già generato sia compatibile con la corrente configurazione della rete. Un caso d'uso per questa funzionalità potrebbe essere quando la configurazione della rete viene modificata: invece di rigenerare tutti i certificati si può controllare se i certificati siano compatibili o meno e solo se necessario riemetterli.

Tutte le funzionalità si affidano alla validazione di Dhall per controllare che la configurazione sia valida; in più viene effettuato un controllo sui nomi degli host,

per verificare che siano univoci. Questo controllo non è possibile in Dhall, ma è necessario per essere sicuri che le regole del firewall che permettono traffico verso o da uno specifico host non possano riferirsi a due host diversi.

Le varie funzionalità di generazione, firma e verifica dei certificati sono realizzate appoggiandosi all'applicativo `nebula-cert`, che fa parte della distribuzione di Nebula. La sua posizione nel file system è richiesta come parametro dallo strumento realizzato.

L'applicativo `nebula` offre la possibilità di verificare la validità di una configurazione YAML, ma i risultati del controllo sono corretti solo se viene eseguito sullo stesso dispositivo su cui verrà usata la configurazione. La verifica infatti riguarda anche la presenza dei certificati e della chiave privata nella posizione corretta del file system. Per questo motivo questa funzionalità non è stata sfruttata all'interno dell'applicativo sviluppato.

6.3.2 Uso di Template Haskell

Essendo Haskell un linguaggio fortemente tipizzato, per lavorare con i valori della configurazione Dhall occorre che essi abbiano dei tipi definiti all'interno del codice Haskell. La cosa più semplice sarebbe definire manualmente i vari tipi, in modo analogo a come sono definiti in Dhall, ma questo approccio presenta delle criticità.

I tipi di dato Dhall sono soggetti a variazioni, per il semplice fatto che Nebula è uno strumento in fase di sviluppo che riceve aggiornamenti piò o meno regolari. Di conseguenza potrebbe essere necessario in futuro aggiungere nuovi campi o rimuoverne. Se i tipi di dato Haskell corrispondenti fossero definiti manualmente essi andrebbero modificati ogni volta che i tipi di dato Dhall vengono modificati. Nel caso questi cambiamenti siano frequenti, il compito di mantenere il codice diventerebbe oneroso.

Una soluzione migliore, anche se più complessa, è l'uso di *Template Haskell*, che permette di generare automaticamente i tipi di dato Haskell a partire dal codice Dhall al momento della compilazione del codice Haskell. Con questo approccio è necessario definire soltanto quali tipi di dato generare, quale nome dargli e la loro tipologia, se *sum type* o *product type*. Fatto ciò sarà il compilatore a generare le definizioni dei tipi di dato e a controllare che il loro utilizzo nel codice sia corretto.

Un problema di questo approccio è che non si ha una rappresentazione scritta dei tipi di dato, costringendo il programmatore ad affidarsi al codice Dhall, i cui tipi elementari però non corrispondono a quelli di Haskell. Questo è un problema inevitabile, sia che si definiscano manualmente i tipi di dato Haskell, sia che si usi Template Haskell. In un qualche modo si sarà sempre costretti a controllare le equivalenze tra i vari tipi, nel caso si usi Template Haskell solo al momento dell'uso dei tipi di dato, nel caso si definiscano manualmente i tipi ogni volta che li si modifica e li si usa.

L'uso di Template Haskell porta a una maggiore manutenibilità del codice: a ogni aggiornamento di Nebula e quindi a ogni cambiamento dei tipi Dhall sarà sufficiente ricompilare l'applicativo Haskell; eventuali errori nel codice dovuti a incompatibilità tra i nuovi e i vecchi tipi saranno segnalati dal compilatore. Nel caso invece i tipi fossero definiti esplicitamente sarebbe necessario correggerli manualmente ad ogni aggiornamento, insieme agli eventuali errori.

L'applicativo Haskell è stato implementato usando Template Haskell proprio per la maggiore manutenibilità derivante da tale approccio.

6.4 Realizzazione della proof of concept

Per verificare la validità di quanto sviluppato come implementazione di un approccio scalabile alla microsegmentazione è stata realizzata una *proof of concept*: tre host in reti diverse potevano comunicare come membri della stessa rete Nebula.

La configurazione della rete è stata scritta in Dhall e prevedeva tre host: una lighthouse, con indirizzo pubblico e posizionata sul cloud Azure, e due host comuni posizionati su due diverse reti private domestiche, come si può vedere in Figura 6.5. I due host comuni, "laptop1" e "laptop2" appartenevano allo stesso gruppo, "laptops". Le comunicazioni permesse erano quelle caratterizzate dal protocollo ICMP, tra tutti gli host, con qualsiasi porta, in tutte le direzioni e quelle caratterizzate dal protocollo TCP, tra gli host del gruppo "laptops", in entrata sulla porta 8080 e in uscita su qualsiasi porta.

Le configurazioni di firewall e router sulle reti private utilizzate erano quelle standard normalmente in uso, mentre per la lighthouse sul cloud Azure l'unica tipologia di traffico permessa dal firewall era il traffico UDP sulla porta 4242.

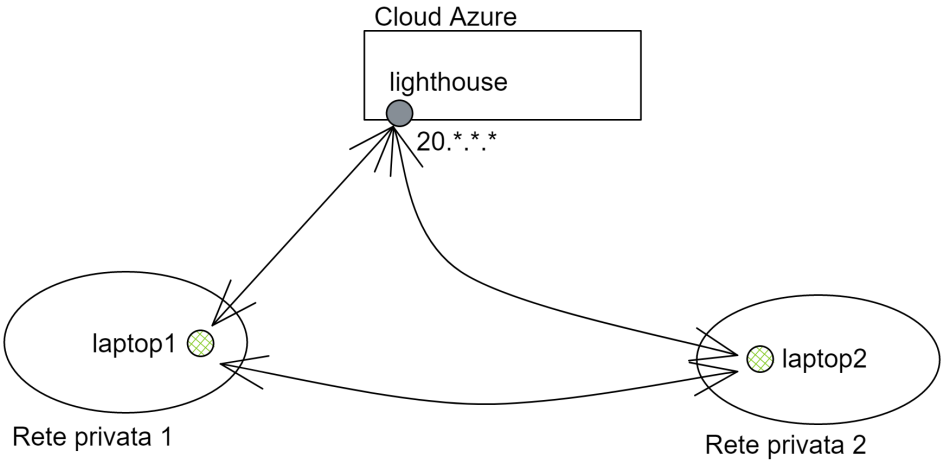


Figura 6.5: Schema della rete Nebula realizzata come proof of concept.

Una volta definita la configurazione in Dhall è stato utilizzato lo strumento sviluppato per generare le necessarie configurazioni YAML, poi le coppie di chiavi pubbliche e private sono state generate su ogni host. Tutte le chiavi pubbliche sono state trasferite sulla macchina su cui era conservata la chiave privata della Certification Authority e lo strumento realizzato è stato utilizzato per generare i relativi certificati. A questo punto i certificati generati e i file YAML delle configurazioni sono stati copiati sugli host a cui facevano riferimento.

Dopo aver avviato Nebula su ciascuno degli host è stato possibile contattare dall'host "laptop1" un web server in esecuzione sull'host "laptop2" sulla porta 8080, ma non un web server sempre in esecuzione sull'host "laptop2" ma sulla porta 8090. In questo modo è stato dimostrato che è teoricamente possibile utilizzare Nebula e lo strumento realizzato per implementare una rete microsegmentata con configurazione definita centralmente ma senza controller centrale.

7 Testing delle performance

La proof of concept realizzata dimostra la possibilità di utilizzare Nebula per la creazione di una rete microsegmentata, ma in una situazione reale il numero di host della rete, esclusa la lighthouse, è ampiamente maggiore di due. Per questo motivo è necessario capire se Nebula possa gestire un numero molto grande di host e un numero molto elevato di comunicazioni concorrenti sulla rete. Inoltre è necessario osservare come si comporti Nebula in casi in cui le condizioni della rete sottostante non sono ottimali. I test descritti in questo capitolo sono stati svolti per valutare la scalabilità, la resistenza e l'efficienza dello strumento sviluppato.

7.1 Descrizione dei test svolti

Le performance di Nebula come overlay network sono state valutate in termini di performance assolute, ovvero di quanto l'uso di Nebula influisca sulla velocità di comunicazione, e in termini di scalabilità, cioè di quanto le performance di Nebula degradino al crescere del carico di lavoro, ovvero al crescere del numero di host che comunicano contemporaneamente. Dato che il principio di funzionamento di Nebula è peer-to-peer quello che serve capire è quante comunicazioni concorrenti da host diversi possa gestire un singolo nodo Nebula senza che le performance calino.

Per valutare questi due aspetti sono stati eseguiti diversi test, con un numero crescente di client che effettuavano richieste verso uno stesso server. In ogni test dei client, in esecuzione su un cluster Kubernetes, inviavano richieste GET HTTP a un web server ogni 250 millisecondi; dopo 15 minuti i risultati e i tempi impiegati per le richieste erano inviati a un altro web server, che aveva il compito di memorizzare questi dati per poterli successivamente elaborare. In caso di tempi di risposta superiori ai 10 secondi la richiesta era considerata fallita. I due web server si trovavano in una macchina virtuale esterna al cluster Kubernetes.

Per valutare invece la resistenza di Nebula al degrado della rete sottostante sono stati svolti dei test, tutti con lo stesso numero di client, in cui veniva variato il numero di pacchetti persi dalla rete durante la comunicazione. È stato così possibile osservare in che modo la perdita di pacchetti da parte della rete influisse sulla percentuale di fallimenti e sui tempi di soddisfazione delle richieste. Questi valori dipendono da come il protocollo di comunicazione di Nebula, basato su UDP, gestisce l'identificazione e la ritrasmissione dei pacchetti perduti.

I test sono stati svolti senza utilizzare la modalità relay di Nebula e scegliendo AES come cifrario. È stato scelto di non effettuare valutazioni di performance usando la modalità relay in quanto essa è ancora sperimentale. Il cifrario AES è stato scelto in quanto quello che, teoricamente, garantisce le performance migliori sulla tipologia di macchine utilizzate per i test. Per verificare questa affermazione uno dei test è stato svolto anche usando ChaChaPoly, con lo scopo di confrontare i risultati con quelli del test svolto con lo stesso numero di client e il cifrario AES.

7.2 Descrizione dell'infrastruttura di testing

I test sono stati eseguiti sulla piattaforma cloud Azure, utilizzando:

- Una macchina virtuale "Standard B1s", con una vCPU e 1GiB di memoria RAM, con indirizzo IP pubblico, che fungesse da lighthouse per la rete Nebula.
- Una macchina virtuale "Standard B2ms", con due vCPU e 8GiB di memoria RAM, su cui erano eseguiti i due web server.
- Due cluster Kubernetes con tre nodi; due nodi erano macchine virtuali di tipo "Standard E2s v3" con due vCPU e 16GiB di memoria RAM, un nodo era una macchina virtuale di tipo "Standard E2 v3" sempre con due vCPU e 16GiB di memoria RAM.

Le due macchine virtuali erano in esecuzione nella regione Japan East, mentre i due cluster si trovavano uno in US East e uno in US East 2, come rappresentato in Figura 7.1.

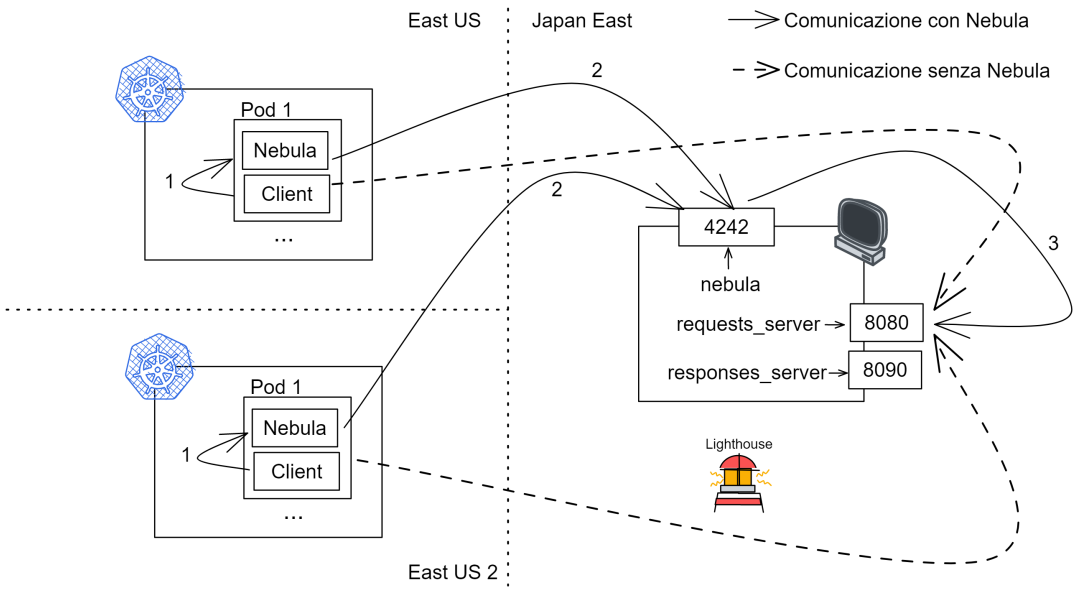


Figura 7.1: Schema dell'infrastruttura usata per il testing e delle comunicazioni tra client e server.

Il posizionamento geografico delle macchine e il loro tipo sono stati dettati in parte dalle disponibilità del cloud Azure e in parte dalle necessità oggettive dei test. Azure pone infatti un limite al numero di vCPU utilizzabili in ogni regione, rendendo quindi necessario posizionare in tre regioni diverse i cluster e le macchine virtuali; sempre per lo stesso motivo è stato necessario utilizzare due cluster invece che uno, in quanto un singolo cluster non aveva sufficiente disponibilità di CPU per eseguire tutti i client.

Le tipologie di macchine virtuali sono state scelte cercando di limitare i costi, pur soddisfando le richieste di memoria e CPU del sistema di testing.

7.3 Organizzazione dei test di valutazione della performance

Per valutare le performance di Nebula in termini di scalabilità sono stati eseguiti complessivamente dieci test, cinque utilizzando Nebula per mediare le comunicazioni tra client e server, cinque senza utilizzare Nebula. I test di ogni serie sono stati effettuati con un numero crescente di client, da 50 a 300, con un aumento di 50 client ogni test. Tutti i test hanno coinvolto i due web server, i test con Nebula

coinvolgevano inoltre la lighthouse. I test senza Nebula sono stati necessari per definire una *baseline* di performance dell'infrastruttura usata per il testing e poter quindi capire quanto l'uso di Nebula influisse sui tempi di comunicazione tra client e server.

7.3.1 Web server

I web server utilizzati per i test sono stati entrambi definiti in Go, usando il framework Gin. La scelta di questi strumenti è stata dettata dalla loro semplicità di utilizzo e dalla necessità di avere un web server performante, i cui eventuali sovraccarichi e rallentamenti influissero il meno possibile sui risultati dei test. Secondo i benchmark forniti dagli sviluppatori di Gin le richieste a route statiche sono soddisfatte in alcune decine di microsecondi, rendendolo un ottimo candidato per l'implementazione dei web server.

Con il web server la cui definizione è nel frammento 7.1 la gestione della richiesta e la preparazione della risposta avvenivano effettivamente nell'ordine dei microsecondi, anche durante i test con il maggior numero di client. In virtù di ciò eventuali variazioni di performance durante i test sono state ricondotte interamente a Nebula e non a cali di performance del web server.

Frammento 7.1: Web server in Go

```
func main() {
    r := gin.Default()
    r.SetTrustedProxies(nil)
    r.GET("/", func(c *gin.Context) {
        c.String(http.StatusOK, "")
    })
    r.Run()
}
```

Per quanto riguarda il web server di raccolta dei risultati le performance non erano essenziali, in quanto esso opera terminati i test. Una volta finito di effettuare le richieste infatti i vari client inviavano dei file di testo contenenti i risultati a

questo web server, tramite una richiesta HTTP POST. I file erano salvati su file system, poi tutte le informazioni contenute in essi erano aggregate per poter essere analizzate.

7.3.2 Client

I client che effettuavano richieste ai web server erano eseguiti su Kubernetes e gestiti tramite uno StatefulSet. L'uso di uno StatefulSet è stato necessario in quanto ogni client doveva essere distinto dall'altro, dato che ognuno doveva avere un indirizzo IP Nebula diverso.

Ogni client doveva avere due funzionalità: doveva essere in grado di fare richieste al web server e doveva essere in grado di connettersi alla rete Nebula. Dato che in Kubernetes i container dello stesso Pod condividono le interfacce di rete, per una maggiore comprensibilità e facilità di implementazione sono stati definiti due container per Pod, come mostrato in Figura 7.2.

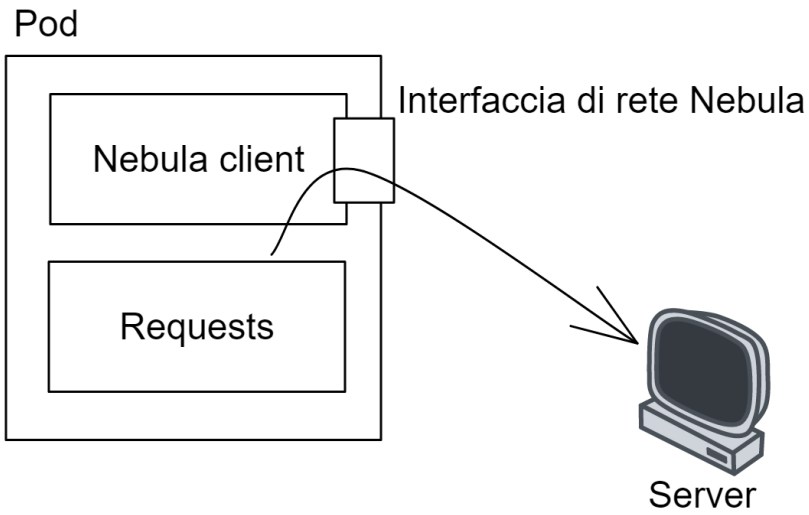


Figura 7.2: Rappresentazione di un pod dello StatefulSet che gestisce i client Nebula durante i test.

Un container, la cui definizione è nel frammento 7.2, esegue l'applicativo Nebula, fornendo connettività all'altro container. All'avvio questo container, che già include la configurazione Nebula, prepara i certificati necessari al funzionamento. La ge-

nerazione dei certificati in questo modo, assolutamente da evitare in ambienti di produzione dato che richiede che certificato e chiave privata della CA siano inclusi nell'immagine del container, è stata impostata così per velocizzare e semplificare la preparazione dei test. In un ambiente di produzione i certificati e le chiavi private dovrebbero essere generati a parte, per poi essere iniettati nel container usando i Secret e le ConfigMap di Kubernetes. Dopo aver preparato i certificati viene avviato l'eseguibile di Nebula.

L'indirizzo IP del Pod sulla rete Nebula dipende dal nome del Pod stesso, che nel caso di un Pod gestito da uno StatefulSet contiene un numero che lo distingue da tutti gli altri, risultando quindi adatto ad essere utilizzato per generare l'IP univoco del client sulla rete Nebula. Per generare l'indirizzo IP a partire dal numero di Pod sono stati usati degli applicativi forniti con la console Bash su Ubuntu; per questo motivo l'immagine base da cui sono state definite le immagini dei due container utilizzati nei test è l'immagine di Ubuntu 20.04.

Frammento 7.2: Definizione del container che gestisce la connessione alla rete Nebula

```
- name: nebula
  image: gior/nebula-client:3.0_no_relay
  command:
    - "/bin/bash"
    - "-c"
    - "./run-nebula.sh $((${HOSTNAME##*-} + 3))"
  resources:
    limits:
      memory: 400M
    requests:
      memory: 80M
  securityContext:
    privileged: true
    capabilities:
      add:
        - NET_ADMIN
```

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 60
  periodSeconds: 20
```

L'altro container, la cui definizione è nel frammento 7.3, si occupa di effettuare richieste HTTP GET ogni 250ms, tramite l'applicativo cURL, verso un indirizzo IP configurabile, per un periodo di tempo anch'esso configurabile. Per ogni richiesta erano salvati in un file di testo il timestamp, il codice HTTP della risposta alla richiesta e il tempo impiegato a ricevere la risposta. Al termine di questo periodo di tempo il container invia il file dei risultati a un altro indirizzo configurabile. Lo script utilizzato si aspetta il web server che attende le richieste GET in esecuzione sulla porta 8080 e il web server che raccoglie i risultati in esecuzione sulla porta 8090.

Anche l'esecuzione del container che effettua le richieste richiede come parametro un identificatore univoco, per poter inviare i risultati con un nome univoco e permettere al web server dei risultati di distinguerli e salvarli in percorsi distinti del file system.

Frammento 7.3: Definizione del container che effettua le richieste al web server

```
- name: requests
  image: gior/requests:1.10
  command:
    - "/bin/bash"
    - "-c"
    - >-
      ./make-requests.sh $((${HOSTNAME##*-} + 3)) 15
      \"192.168.0.2\" \"192.168.0.2\"
```

La configurazione della rete Nebula utilizzata per i test prevedeva tre host: lighthouse, server e client. La configurazione dei client era identica per ogni singolo client, quindi è stata generata una volta, integrata nell'immagine del container e riutilizzata per tutti i client. L'identificatore univoco usato dai vari Pod era il numero del Pod più tre: alla lighthouse era assegnato il primo indirizzo della rete, al server il secondo, quindi al Pod numero zero doveva essere assegnato il terzo e così via. Lavorando su due cluster diversi in realtà gli identificatori dei Pod erano ripetuti, quindi in un cluster l'indirizzo Nebula di un Pod era generato sommando tre all'identificatore del Pod, nell'altro cluster era ottenuto sommando tre più il numero di client eseguiti nel primo cluster.

7.4 Organizzazione dei test di valutazione della resistenza

I test per la valutazione della resistenza di Nebula al degrado della rete sono stati svolti utilizzando la stessa infrastruttura dei test volti alla valutazione delle performance. Sono stati effettuati nove test, ognuno con 100 client, suddivisi sui due cluster, che effettuavano richieste verso il web server, sempre ogni 250 millisecondi e con un timeout di 10 secondi, sfruttando la rete Nebula. La durata di ogni test è stata dieci minuti. Usando l'applicativo `tc` sulla macchina virtuale dove era eseguito il web server è stato fatto in modo che l'interfaccia di rete di Nebula perdesse una certa percentuale di pacchetti, diversa per ogni test. Le percentuali andavano dal 10% al 90%, con intervalli del 10%.

Avendo escluso l'uso della funzionalità relay dai test, le comunicazioni tra i nodi della rete Nebula sono tutte punto a punto: per lo scopo di questo test non è quindi necessario che i pacchetti vengano persi uniformemente durante il tragitto sulla rete Internet, ma basta che essi non arrivino a destinazione. L'uso di `tc` sulla macchina virtuale che ospita il web server è stato quindi sufficiente per simulare una rete degradata ai fini di questi test: come si può notare dalla Figura 7.3, che il pacchetto venga perso all'inizio del percorso in Internet (punto A), in un tratto intermedio (punto B) o alla fine del percorso (punto C) non cambia il fatto che esso non arrivi mai al server. Si possono quindi approssimare i vari casi imponendo che tutti i pacchetti siano persi nel punto C, ovvero nell'ultimo tratto del percorso, subito

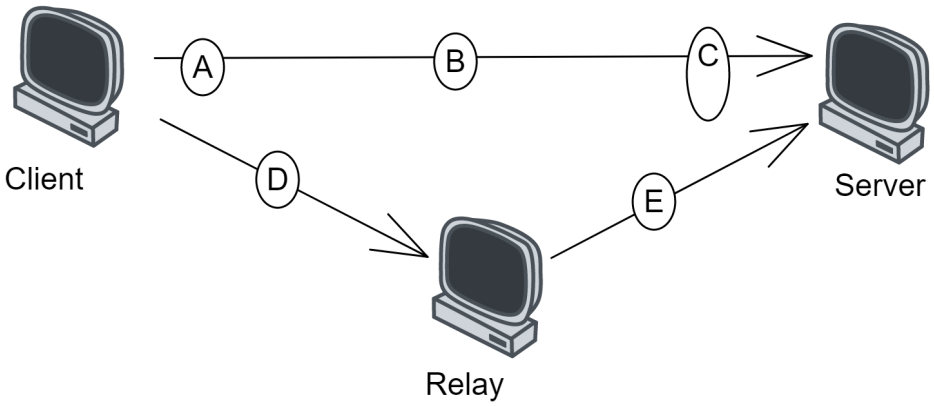


Figura 7.3: Schema dei punti in cui potrebbero venir persi i pacchetti durante la comunicazione tra client e server, passando o meno attraverso un relay.

prima di essere consegnati all'interfaccia di rete di Nebula. Se fosse stata testata anche la funzionalità relay sarebbe stato necessario che i pacchetti venissero persi anche nelle comunicazioni interne al cluster. Usando i relay è infatti possibile che i pacchetti vengano persi o prima o dopo il passaggio attraverso essi, quindi o nel punto D o nel punto E di Figura 7.3. Perdere i pacchetti in un punto potrebbe non avere le stesse conseguenze che perderli in un altro, quindi in caso di perdita di pacchetti da parte della rete sottostante il comportamento della rete Nebula usando i relay potrebbe non essere uguale a quello della rete senza l'uso di relay.

Per questi test è fondamentale che il web server di raccolta dei risultati sia raggiungibile attraverso un'interfaccia di rete che non sia quella di Nebula, in quanto in alcuni casi la quantità di pacchetti persi potrebbe impedire che i risultati siano correttamente inviati.

7.5 Problematiche incontrate

Durante la preparazione e l'esecuzione dei test sono stati incontrati dei problemi relativi alla capacità, in termini di memoria e CPU, delle macchine utilizzate. Non avendo infatti dei riferimenti precisi sui requisiti necessari all'esecuzione di Nebula è stato necessario muoversi per tentativi. Da un lato è stato necessario potenziare la macchina su cui erano erano seguiti i web server, perché Nebula non era in grado di gestire il numero di connessioni concorrenti con le risorse inizialmente assegnate; dall'altro è stato necessario aumentare memoria e CPU dei cluster (passando inoltre

da uno a due cluster) dato che i client non riuscivano ad essere creati ed eseguiti con le giuste tempistiche, a causa di una quantità di risorse insufficienti.

Questo errato dimensionamento iniziale è stato fastidioso, in quanto ha richiesto del tempo per essere diagnosticato e risolto, ma ha permesso di verificare anche, a grandi linee, i requisiti di esecuzione di Nebula.

8 Analisi dei risultati dei test

Le performance di Nebula sono state valutate prendendo in considerazione i risultati dei singoli test e poi confrontando i risultati dei vari test tra loro. Sono inoltre state fatte delle considerazioni in merito ai requisiti di esecuzione di Nebula. In questo capitolo sono riportati i risultati ottenuti nei test, già in parte elaborati sotto forma di grafici e tabelle, insieme alle relative analisi.

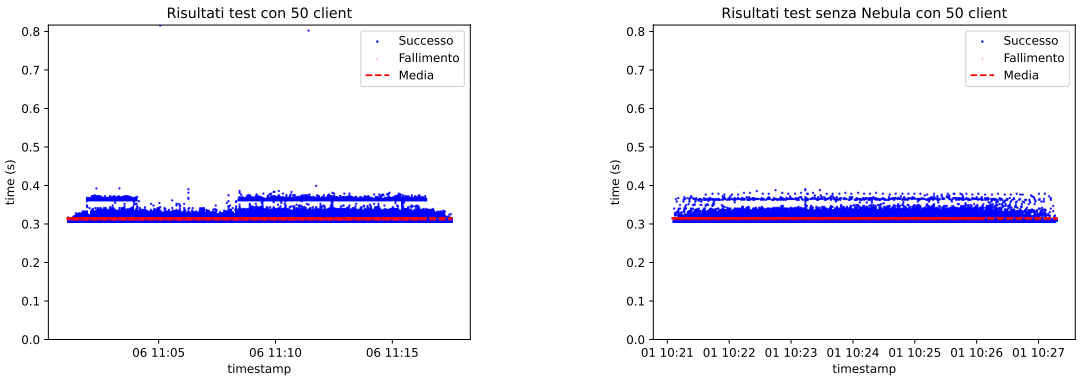
8.1 Performance generali

Le performance di Nebula in termini assoluti e in termini di scalabilità che risultano dai test effettuati sono buone, anche se il confronto tra performance del sistema usando e non usando Nebula è probabilmente influenzato dall'infrastruttura su cui sono stati svolti i test.

8.1.1 Performance assolute

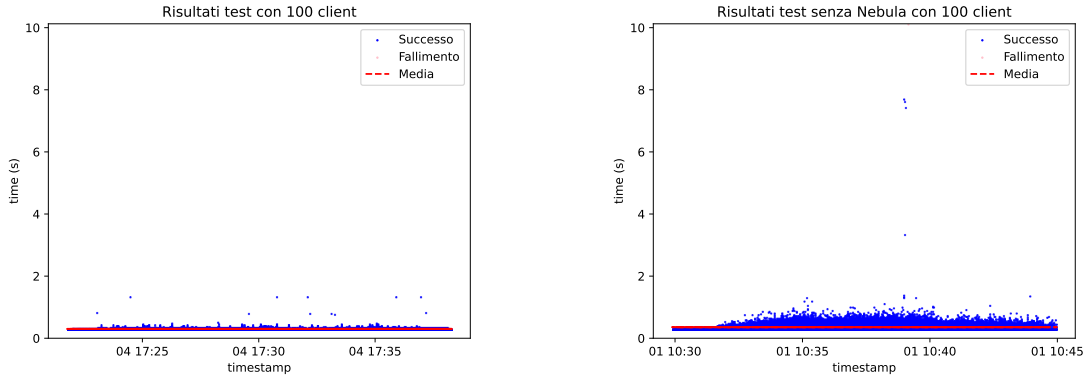
Come si può vedere in Figura 8.1 e Figura 8.2 con 50 e 100 client non ci sono notabili differenze tra le performance del sistema usando Nebula e non usandolo. La maggior parte delle richieste viene soddisfatta in qualche decimo di secondo e solo un numero trascurabile di richieste impiega un tempo maggiore. Si può però notare come mentre con 50 client usando Nebula si ottengano tempi molto più vari che non usandolo, nel test con 100 client è l'inverso.

Nei test effettuati con 150 client, i cui risultati sono visibili in Figura 8.3, inizia a vedersi, in modo inaspettato, che l'uso di Nebula porti a performance decisamente migliori rispetto a quelle del sistema senza Nebula. Potrebbe sorgere il sospetto che ciò sia un caso dovuto al momento in cui sono stati eseguiti i test, che comunque potrebbero essere in parte influenzati dalla situazione delle rete Internet e della rete interna del cloud Azure del momento. Andando però a osservare i risultati dei test



(a) Con Nebula (b) Senza Nebula

Figura 8.1: Risultati del test con 50 client.



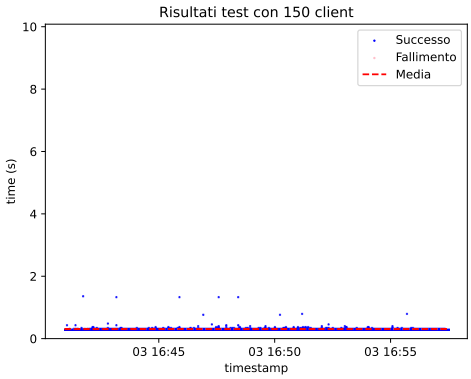
(a) Con Nebula (b) Senza Nebula

Figura 8.2: Risultati del test con 100 client.

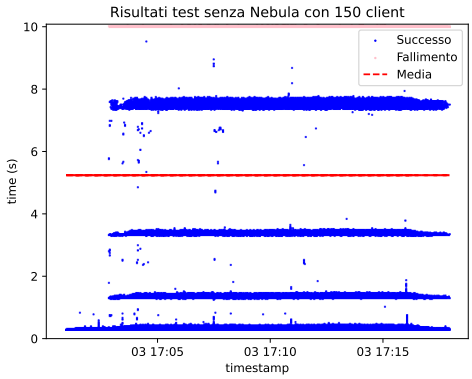
con 200, 250, 300 client, rappresentati rispettivamente in Figura 8.4, Figura 8.5, Figura 8.6, si può capire non sia questo il caso, in quanto anche questi test riportano risultati simili a quelli del test da 150 client.

Questi risultati possono essere ricondotti al fatto che le richieste senza Nebula avvengano usando TCP, mentre Nebula utilizza UDP per l'invio dei pacchetti. Nello specifico, la causa di queste scarse performance di TCP potrebbe essere trovata negli algoritmi di controllo della congestione della rete. È infatti ragionevole pensare che, vista la grande quantità di pacchetti inviata in questi test, questi algoritmi intervengano per cercare di ridurre la congestione della rete, rallentando però la comunicazione tra client e server.

Dato che esistono diversi algoritmi di congestione è possibile che ripetendo i test senza Nebula con un'infrastruttura diversa si possa avere un miglioramento, o anche un peggioramento, delle performance.

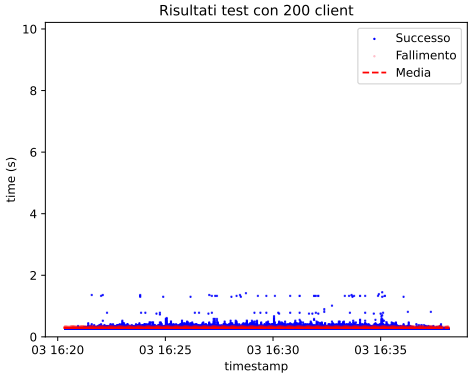


(a) Con Nebula

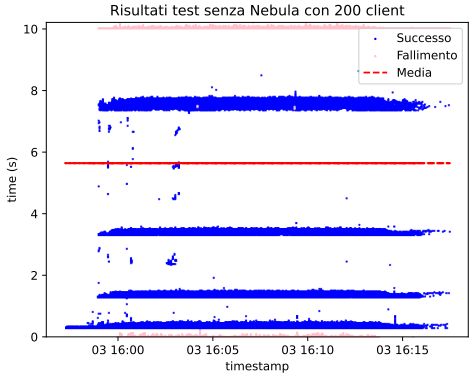


(b) Senza Nebula

Figura 8.3: Risultati del test con 150 client.

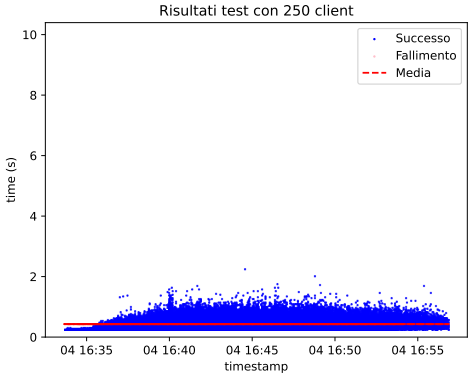


(a) Con Nebula

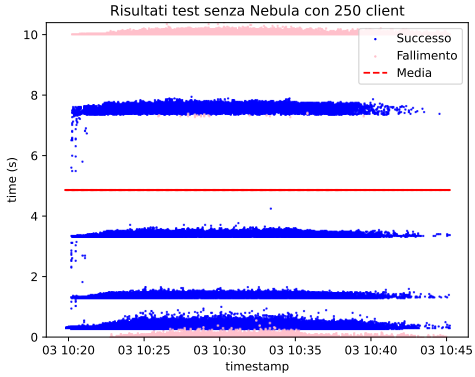


(b) Senza Nebula

Figura 8.4: Risultati del test con 200 client.

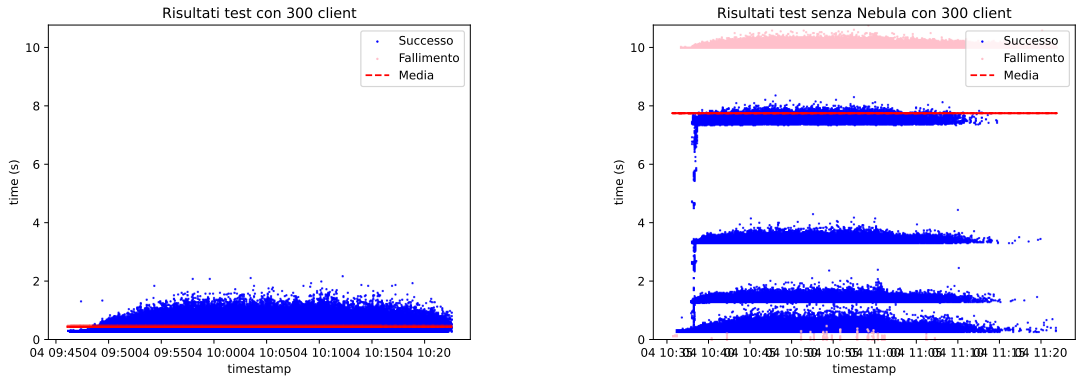


(a) Con Nebula



(b) Senza Nebula

Figura 8.5: Risultati del test con 250 client.



(a) Con Nebula (b) Senza Nebula

Figura 8.6: Risultati del test con 300 client.

È necessario notare che i risultati dei test svolti senza Nebula presentano anche un elevato numero di richieste rimaste senza risposta. Ciò è meglio rappresentato in Figura 8.7, in cui si vede chiaramente che al crescere del numero di client cresce la percentuale di richieste senza risposta (o con risposta inviata dopo più di dieci secondi) nei test svolti senza Nebula. Ciò invece non avviene nei test con Nebula, in cui la percentuale di successi rimane sempre uguale a 100%, come mostrato in Tabella 8.1.

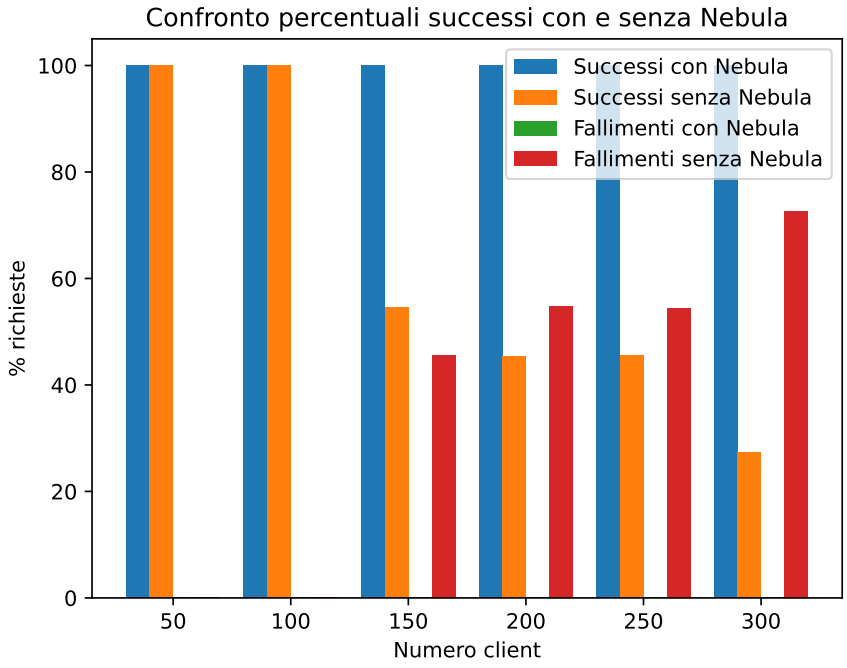


Figura 8.7: Confronto tra percentuali di successi e insuccessi nei vari test.

Client	% successi	T. medio (s)	T. medio successi (s)	σ (s)	σ successi (s)
50	100	0.314	0.314	0.008	0.008
100	100	0.309	0.309	0.011	0.011
150	100	0.304	0.304	0.009	0.009
200	100	0.313	0.313	0.016	0.016
250	100	0.425	0.425	0.110	0.110
300	100	0.447	0.447	0.129	0.129

Tabella 8.1: Risultati dei test con Nebula.

Se si osservano i dati in Tabella 8.2, confrontandoli con quelli di Tabella 8.1, si può notare come gli insuccessi (per cui il tempo impiegato risulterà in linea di massima pari a 10s) abbiano una forte influenza sulla media del tempo impiegato per le richieste senza Nebula: il tempo medio dei successi ottenuti senza l'uso di Nebula risulta sensibilmente più basso del tempo medio calcolato considerando tutte le richieste. Superati i 150 client il tempo medio dei successi non va incontro a variazioni particolarmente grandi; anche osservando la deviazione standard si nota che la distribuzione dei tempi rimane simile. Prendendo in esame tutte le richieste senza Nebula invece il tempo medio aumenta all'aumentare del numero di client, con il test da 300 client in cui si può osservare un netto aumento del tempo medio di risposta e una diminuzione della deviazione standard. La deviazione standard cala in questo modo perché nel test da 300 client un numero decisamente maggiore di richieste fallisce. Tutte le richieste fallite riporteranno tempi di risposta prossimi ai dieci secondi, quindi la deviazione standard cala: la maggior parte delle richieste riportano tempi di risposta simili. Per le richieste che hanno successo, invece, la media e la deviazione standard rimangono simili ai test precedenti, in quanto la distribuzione dei tempi di risposta rimane simile. Queste osservazioni sono sostenute dai grafici mostrati in precedenza nelle Figure da 8.3 a 8.6.

Va notato che anche considerando le sole richieste eseguite con successo il sistema senza Nebula risulta meno performante di quello con Nebula.

Nel valutare le performance di Nebula occorre comunque tenere in considerazione il cifrario usato: AES offre buone performance quando è usato su macchine con processori x86/64, ma non quando è usato su dispositivi ARM, in quanto su questa famiglia di processori manca il supporto hardware a questo cifrario. Nel caso quindi la rete Nebula sia formata da tipologie diverse di dispositivi l'uso di AES non sarebbe una buona scelta, portando a preferire il cifrario ChaChaPoly. L'uso di ChaChaPoly

Client	% successi	T. medio (s)	T. medio successi (s)	σ (s)	σ successi (s)
50	100	0.314	0.314	0.008	0.008
100	99.99	0.358	0.358	0.074	0.070
150	54.53	5.237	1.266	4.600	2.027
200	46.31	5.638	1.913	4.560	2.660
250	45.48	4.871	0.926	4.745	1.716
300	27.39	7.749	1.783	3.871	2.220

Tabella 8.2: Risultati dei test senza Nebula.

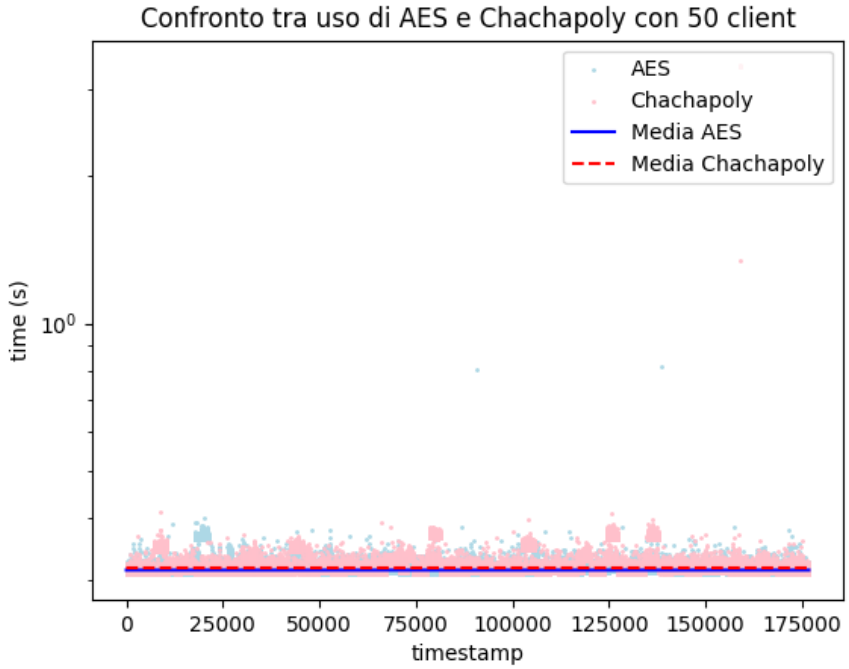


Figura 8.8: Confronto tra i tempi di risposta con 50 client usando AES e usando ChaChaPoly come cifrario.

però potrebbe portare a un peggioramento delle performance di Nebula su macchine x86/64 rispetto all'uso di AES. Come si può vedere in Figura 8.8, con l'infrastruttura di testing utilizzata le performance di Nebula usando ChaChaPoly come cifrario sono molto simili a quelle ottenute usando AES, ma comunque leggermente inferiori.

8.1.2 Scalabilità

Le performance di Nebula al crescere del carico si mantengono buone, con una crescita dei tempi di risposta minimi, come si può osservare in Figura 8.9 e Figura 8.10. Nelle performance del sistema senza Nebula, invece, si notano peggioramenti della performance significativi al crescere del numero di client, probabilmente per

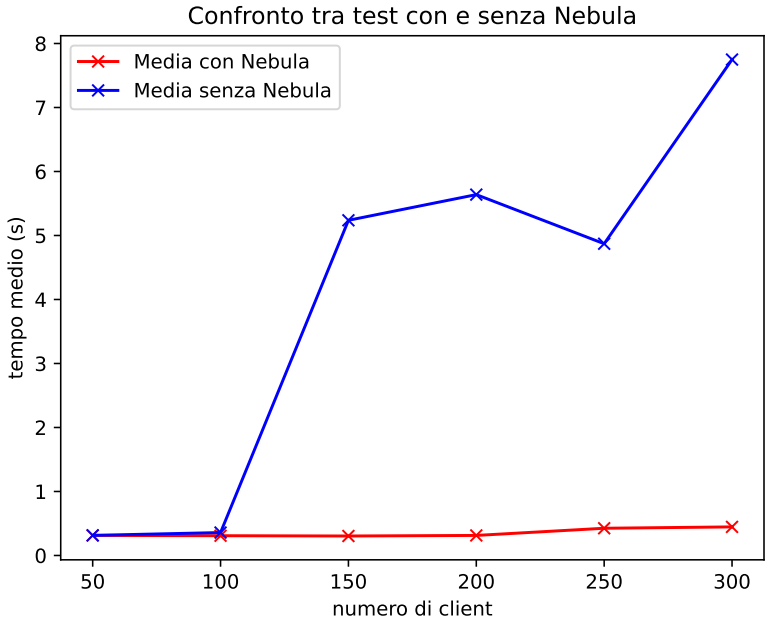


Figura 8.9: Confronto tra i tempi medi di risposta dei vari test.

le ragioni già spiegate in Sezione 8.1.1. La minore regolarità dei tempi relativi al sistema che non usa Nebula possono essere ricondotti a comportamenti della rete leggermente diversi nello specifico momento del test.

Il comportamento di Nebula può essere confrontato con quello di altre VPN e overlay network. Un confronto tra questa tipologia di strumenti è già stato fatto in [13], seppur con un'infrastruttura e delle modalità leggermente diverse ed escludendo Nebula. È possibile osservare in modo qualitativo l'andamento delle performance delle varie VPN in Figura 8.11. Si può dire che Nebula si comporta in maniera migliore di Tinc e SoftEther in quanto, a parità di numero di client, il calo delle perfomance di Nebula è minore del calo delle performance di Tinc e SoftEther. Non è possibile osservare differenze con gli altri software di VPN utilizzati nei test dell'articolo citato in quanto, per le quantità di client utilizzati durante i test di Nebula, l'andamento dei grafici pare simile e un confronto quantitativo non è fattibile a causa delle diverse infrastrutture di testing usate.

8.2 Resistenza al degrado della rete

Il livello di resistenza di Nebula al degrado della rete è buono, anche se non eccezionale. Al crescere della percentuale di pacchetti persi i tempi medi di soddisfazione

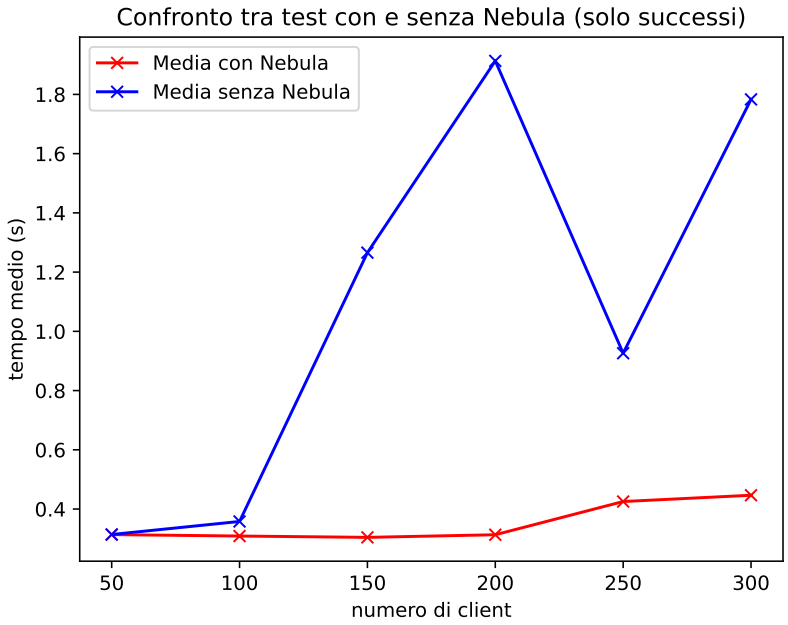


Figura 8.10: Confronto tra i tempi medi di risposta delle richieste aventi successo dei vari test

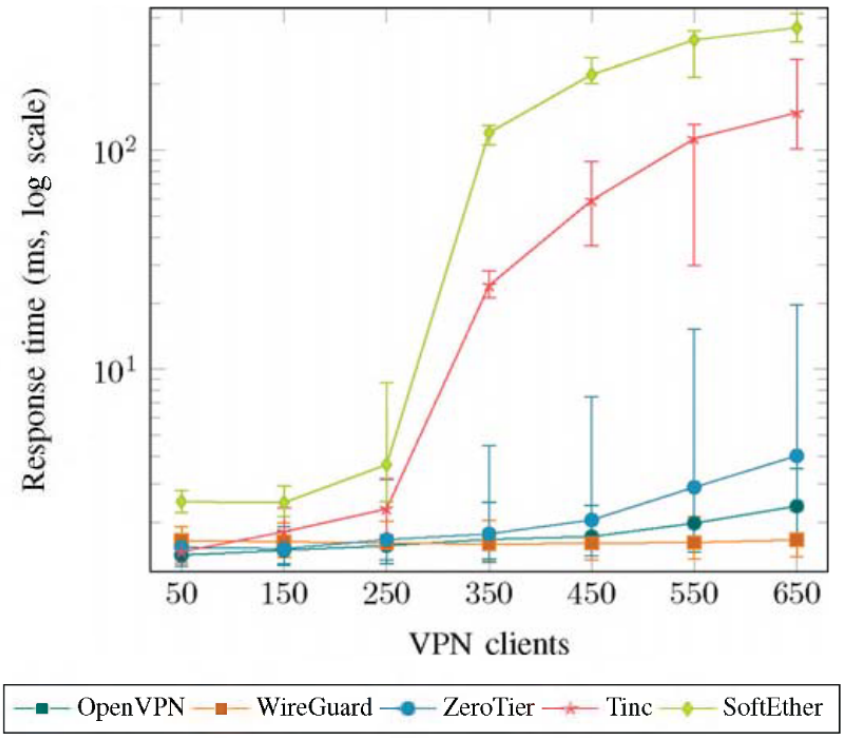


Figura 8.11: Confronto tra le performance al crescere del numero di client di OpenVPN, Wireguard, ZeroTier, Tinc e SoftEther, tratto da [13].

Figura 8.12: Tempi medi necessari a soddisfare le richieste di 100 client Nebula al crescere del numero di pacchetti persi sulla rete.

delle richieste crescono con regolarità, come si può vedere in Figura 8.12. Purtroppo non è possibile un confronto quantitativo con i risultati, mostrati in Figura 8.13, ottenuti in un test simile in [13], non avendo svolto i test sulla stessa infrastruttura. Si può notare però come l'andamento del grafico dei tempi medi di Nebula sia più simile all'andamento del grafico riferito a Wireguard che a quelli dei grafici riferiti agli software VPN e di overlay network. Questo porterebbe a classificare Nebula tra i software peggiori da questo punto di vista.

Considerare però solo i tempi medi di risposta risulta ingannevole nel valutare le capacità di Nebula: se infatti si osserva il grafico in Figura 8.14 si può vedere che anche con il 90% di pacchetti persi una certa percentuale di richieste riesce ancora ad essere soddisfatta. Questo è un risultato migliore di quello di software come Tinc e OpenVPN, che già con l'80% di pacchetti persi hanno una percentuale di fallimento delle richieste vicina al 100%. Anche in questo caso i risultati in Figura 8.15 non possono essere direttamente confrontati con quelli ottenuti durante i test riguardanti Nebula, ma è si può comunque affermare che, almeno in base a un confronto qualitativo, sotto questo aspetto Nebula è migliore degli altri software.

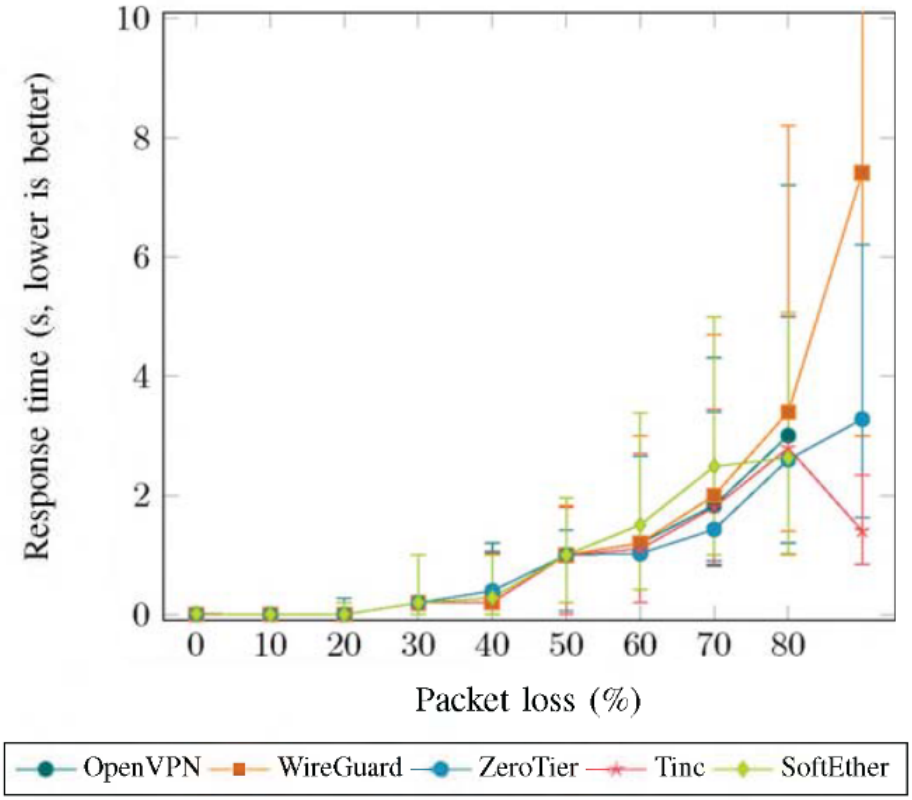


Figura 8.13: Tempi medi necessari a soddisfare le richieste di 100 client al crescere del numero di pacchetti persi sulla rete per i software OpenVPN, Wireguard, ZeroTier, Tinc e SoftEther, tratto da [13].

8.3 Utilizzo delle risorse da parte di Nebula

Dai tentativi effettuati per l'esecuzione dei test si possono ricavare informazioni interessanti sui requisiti di memoria e CPU di Nebula, che purtroppo non sono specificati nella sua documentazione.

Per quanto riguarda i client in esecuzione su Kubernetes, si è notato che:

- Fino a un centinaio di client è sufficiente che il cluster su cui sono in esecuzione abbia a disposizione 4 vCPU per creare e gestire i Pod in maniera idonea.
- Oltre il centinaio e fino a due centinaia circa di client i test potevano essere eseguiti con 6 vCPU totali a disposizione.
- Per i test effettuati con 250 e 300 client è stato necessario eseguire i Pod suddivisi in due gruppi su due cluster diversi, ognuno con 6 vCPU a disposizione.

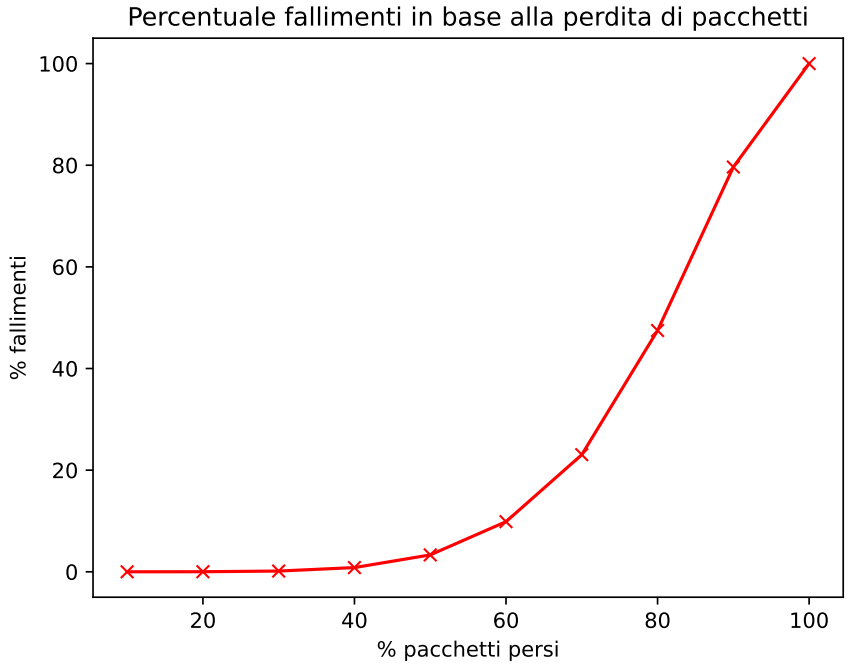


Figura 8.14: Percentuali di fallimento delle richieste di 100 client Nebula al crescere del numero di pacchetti persi sulla rete.

Purtroppo questi dati, seppur utili per avere una vaga idea dei requisiti di CPU di Nebula, non forniscono informazioni definitive in merito ai requisiti di Nebula, in quanto sulle macchine virtuali del cluster e all'interno del cluster stesso sono presenti i vari processi di gestione di Kubernetes. Questi processi consumano risorse che non è possibile quantificare esattamente.

Per quanto riguarda invece il consumo di memoria si è visto che ogni Pod richiedeva circa 50MiB di RAM durante la sua esecuzione.

Indicazioni sui requisiti di memoria di Nebula sono state ottenute anche osservando lighthouse e server. La macchina virtuale usata come lighthouse aveva solo 1GiB di RAM e non ha mai presentato problemi o cali di performance. Questo è dovuto al fatto che il compito della lighthouse è solo facilitare la conoscenza tra due host della rete Nebula, quindi una volta fatto ciò all'avvio dei client, che comunque avviene in maniera graduale per come funziona Kubernetes, la lighthouse rimane semplicemente in attesa. Probabilmente si avrebbe una situazione diversa se la lighthouse dovesse fungere anche da relay.

La situazione cambia nel caso del server, in cui Nebula deve gestire molte connessioni concorrenti in modo continuo. Con 1 vCPU e 1GiB di memoria RAM

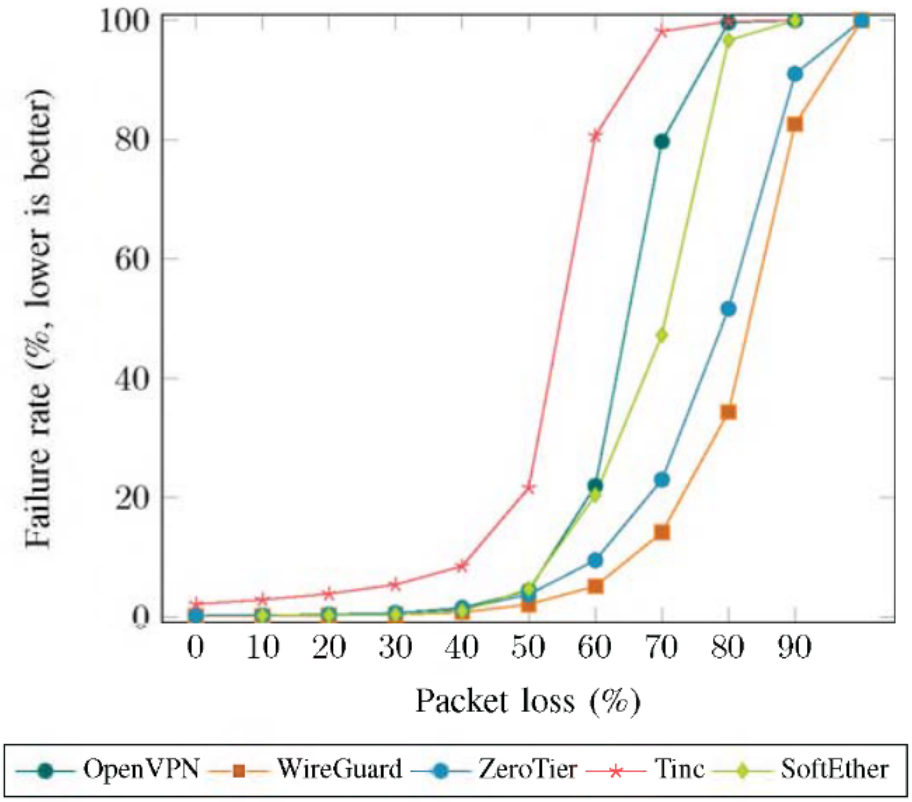


Figura 8.15: Percentuali di fallimento delle richieste di 100 client al crescere del numero di pacchetti persi sulla rete per i software OpenVPN, Wireguard, ZeroTier, Tinc e SoftEther, tratto da [13].

Nebula non riusciva a gestire neanche 50 client: quando cercava di connettersi un client intorno al cinquantesimo Nebula iniziava a segnalare errori. Il messaggio di errore indicava che l'indirizzo da cui il client aveva aperto la comunicazione non era raggiungibile, ma tale indirizzo era segnalato come vuoto. Questo problema si è risolto andando a incrementare le capacità della macchina virtuale, a 2 vCPU e 4GiB di memoria RAM. Questa potenza è stata sufficiente per i test fino a 200 client, superati i quali è stato necessario incrementare ulteriormente le capacità del server per evitare errori. Per i test successivi la capacità è stata incrementata a 2 vCPU e 8GiB di memoria.

Questi fatti farebbero pensare che Nebula richieda 1GiB di memoria ogni 50 connessioni concorrenti da gestire, anche se di fatto osservando l'utilizzo di memoria della macchina virtuale ciò non risulta: mentre al web server arrivavano richieste da più di 300 client in contemporanea, infatti, il processo di Nebula segnalava l'occupazione della CPU al 18.4% e l'uso della memoria all'1.5%. Si può invece

escludere che sia un problema di quantità di CPU, in quanto sia la macchina usata inizialmente per i test da 200 client e quella usata poi per i test da 300 client avevano la stessa quantità di vCPU a disposizione. È comunque indicato l'uso di almeno 2 vCPU, in quanto le operazioni di cifratura dei pacchetti possono essere costose in termini di computazione [21].

8.4 Valutazione complessiva

Volendo fare una valutazione complessiva delle performance di Nebula si può dire che esso risulti un ottimo strumento di overlay networking, vista la sua scarsa influenza sui tempi di comunicazione tra applicazioni e le sue capacità di superare i NAT. Il fatto che usi UDP e che quindi il suo traffico non sia soggetto alla gestione della congestione della rete si rivela inoltre, fino a un certo punto, un vantaggio nelle situazioni in cui il traffico di rete è intenso.

Anche nella gestione di elevate quantità di traffico Nebula risulta un buono strumento, riuscendo a gestire numerose connessioni concorrenti senza problemi. Occorrerebbero valutazioni più approfondite, effettuando test con un numero ancora maggiore di client, ma già dai test effettuati si vede che anche al crescere del carico di lavoro le performance di Nebula non peggiorano in modo rilevante. In caso di degrado della rete, inoltre, Nebula dimostra di poter gestire meglio la perdita di pacchetti rispetto ad alti software simili.

L'unico punto oscuro che emerge dai risultati dei test effettuati sono i requisiti di esecuzione di Nebula: mentre per i client essi non appaiono troppo elevati, facendo quindi pensare che Nebula possa anche essere utilizzata su dispositivi IoT con scarse risorse, per i server le risorse richieste sono state più di quello che ci si aspettava. Va comunque riconosciuto che, al giorno d'oggi, le risorse di cui si è avuto bisogno per permettere a Nebula di gestire 300 connessioni concorrenti, ovvero due vCPU e 8GiB di memoria, sono risorse facilmente ottenibili a prezzi non elevati.

Combinate con le sue funzionalità di firewall e nonostante i suoi requisiti di esecuzione poco chiari, le performance di Nebula lo rendono idoneo come strumento scalabile e robusto per la microsegmentazione di un'architettura aziendale distribuita su più cloud.

9 Conclusione e lavori futuri

Il panorama dei sistemi informatici ha subito negli ultimi anni una forte diversificazione: con l'avvento del cloud è iniziato un periodo di transizione che vede i sistemi aziendali spostarsi da datacenter privati a cloud pubblici, passando per cloud ibridi e multicloud. Un completo passaggio ai cloud pubblici è per il momento impossibile: normative, timori, mancanza di conoscenze impediscono il completamento di questa transizione. Ci si trova quindi ad avere a che fare con infrastrutture complesse di cui è necessario garantire la sicurezza.

Contemporaneamente a questa transizione si è assistito a un incremento degli attacchi informatici e a un aumento della loro complessità. I cambiamenti nelle modalità di utilizzo dei sistemi aziendali non hanno fatto altro che facilitare questa crescita: la presenza di utenti in mobilità o in lavoro a distanza, soprattutto in seguito alla pandemia di Covid-19, hanno richiesto di cambiare le modalità di accesso ai sistemi, spesso lasciandoli vulnerabili.

Proposte per la sicurezza dei sistemi informatici, anche esistenti da anni, hanno iniziato ad accumulare popolarità, in quanto più idonee delle precedenti alternative. Tra tutte il paradigma Zero Trust ha ricevuto particolare attenzione: esso prevede che ogni richiesta venga considerata insicura, indipendentemente dalla sua provenienza, abbandonando l'idea che richieste provenienti dall'interno della rete aziendale siano automaticamente affidabili. Ogni richiesta deve essere esaminata da un policy decision point, che indica a dei policy enforcement point se la richiesta debba essere inoltrata alla sua destinazione o meno.

Le tecniche per applicare i principi Zero Trust nella pratica sono numerose, con diversi gradi di novità. Una tecnica di particolare interesse visto il crescente uso di cloud ibridi e multicloud è la microsegmentazione agent-based, in cui gli accessi a singole o multiple risorse sono controllati da agenti software eseguiti su macchine virtuali o container.

I principali approcci alla microsegmentazione al giorno d'oggi vedono l'uso di controller centralizzati per la gestione degli agenti: mentre questo aumenta la comodità d'uso degli strumenti in questione, la scalabilità e la resistenza di queste soluzioni possono essere messe in discussione. Interruzioni di rete o un numero eccessivo di richieste potrebbero mandare in crisi il singolo elemento centrale che gestisce l'intera rete. Questi controller centrali inoltre sono dei singoli punti di vulnerabilità, in quanto la loro compromissione ha conseguenze sull'intera rete.

Scalabilità e resistenza sono fondamentali nel panorama informatico attuale, quindi rivedere l'approccio con cui si applica la microsegmentazione nei propri sistemi è necessario per godere di queste proprietà.

Eliminare il controller centrale e rendere indipendente ogni agente, facendo in modo che essi possano essere configurati individualmente e che possano valutare la policy di sicurezza in autonomia, aumenta la scalabilità e la resistenza dei sistemi, a scapito però della comodità di utilizzo.

Per mantenere quindi sia scalabilità e resistenza sia comodità di utilizzo in uno strumento per la microsegmentazione occorre definire un approccio che permetta la definizione centralizzata della configurazione degli agenti, ma che li lasci indipendenti nel loro funzionamento.

In questa tesi è stato proposto un approccio alla microsegmentazione che abbandona l'idea di un controller centralizzato, a favore di scalabilità e robustezza, mantenendo però la possibilità di definire in maniera centralizzata la configurazione della rete, per garantire un buon livello di comodità d'uso. Come base di questo approccio è stato usato un overlay network, per renderne possibile l'applicazione anche ad ambienti quali cloud ibridi e multicloud.

Per implementare l'approccio proposto è stato usato uno strumento di overlay networking, Nebula, ed è stato realizzato uno strumento, in Haskell, che permette di definire la configurazione della rete in Dhall per poi convertirla nelle configurazioni dei singoli agenti Nebula. La configurazione della rete consiste nella configurazione degli agenti software e nella policy di sicurezza che essi devono applicare.

La configurazione è definita centralmente, quindi in maniera semplice e comoda, permettendo all'amministratore di avere sott'occhio tutta la configurazione in un dato momento. L'uso di Dhall come linguaggio permette, in virtù delle sue caratteristiche, di integrare la validazione della configurazione nella configurazione

stessa, aumentando il suo livello di sicurezza. Il fatto che la configurazione venga definita testualmente, come codice, permette di gestirne i cambiamenti tramite sistemi di versionamento e di testarne la correttezza prima di applicarla.

Ogni agente Nebula lavora indipendentemente dagli altri: la valutazione se una richiesta debba essere permessa o meno avviene localmente, in base alla configurazione dell'agente. In questo modo non c'è più un unico punto di fallimento e la manipolazione di un agente da parte di un attaccante non influisce su tutti gli altri. Le connessioni sono inoltre punto a punto, altro aspetto che migliora la robustezza e la scalabilità della soluzione.

La configurazione di ogni agente viene generata a partire dalla configurazione generale della rete, sfruttando una combinazione di funzioni Dhall integrate nella configurazione stessa e dello strumento sviluppato ad hoc in Haskell. Questo strumento si occupa, data la configurazione di rete, di generare le configurazioni degli agenti e di firmare le loro chiavi pubbliche, inserendo nel certificato generato i valori necessari in base alla configurazione ricevuta in input. Nebula si basa infatti su un'architettura a chiave pubblica per garantire l'identità dei client.

Il sistema implementato usando Nebula e il linguaggio Dhall aveva lo scopo di dimostrare che fosse possibile realizzare un approccio alla microsegmentazione che portasse ad avere una rete semplice da configurare, robusta e senza problemi di scalabilità.

La semplicità di configurazione è stata ottenuta permettendo la definizione della configurazione di rete in maniera centralizzata e automatizzando la generazione delle configurazione dei vari nodi della rete a partire da essa.

La robustezza dell'approccio proposto è data dal fatto che la policy di sicurezza sia validata localmente dagli agenti, senza interrogare un controller centrale: è stato così eliminato un punto unico di fallimento. Nella forma in cui l'approccio è stato implementato, inoltre, si ha che il sistema è anche resistente alla perdita di pacchetti da parte della rete su cui viene costruito l'overlay network. Questo è stato dimostrato svolgendo una serie di test in cui un centinaio client inviava richieste HTTP verso un server; una percentuale crescente di pacchetti per ogni test veniva persa durante il percorso da client a server. I risultati di questo test hanno mostrato che anche con alte percentuali di pacchetti persi una buona parte delle richieste era soddisfatta, dimostrando come Nebula sia resistente al degrado della rete su cui

viene costruito l'overlay network.

La scalabilità della rete microsegmentata creata dipende dalla scalabilità dei singoli nodi, in virtù del fatto che i processi di autenticazione e autorizzazione avvengono localmente su essi. Meglio i singoli nodi riescono a gestire quantità crescenti di richieste, meglio la rete in generale riesce a gestire grandi quantità di traffico. Mancando un controller centrale non c'è un punto che possa diventare un collo di bottiglia per l'intero overlay network. Essendo la comunicazione tra i nodi di Nebula peer-to-peer, i test svolti per valutare la scalabilità del sistema avevano l'obiettivo di misurare quanto le performance di un singolo nodo diminuissero al crescere delle richieste rivolte ad esso. I risultati ottenuti hanno mostrato che al crescere del traffico il calo delle performance di Nebula è ridotto.

I test svolti per valutare la scalabilità del sistema hanno anche mostrato come l'impatto generale dell'uso di Nebula sui tempi di comunicazione tra dispositivi sia molto ridotto. Durante lo svolgimento dei test sono state anche raccolte informazioni sulla richiesta di risorse del sistema microsegmentato: i requisiti di CPU e memoria RAM di Nebula, pur non essendo perfettamente chiari, non sono eccessivamente alti.

Tenendo conto dei risultati test svolti, la valutazione di quanto realizzato è stata positiva, portando a considerare l'approccio alla microsegmentazione realizzato una possibile alternativa agli approcci ora più diffusi. L'originalità della proposta individuata e la dimostrazione della sua funzionalità sono tali da permettere di redigere un articolo, "Scalable Zero Trust microsegmentation based on overlay network" di Giorgia Rondinini, Claudio Zanasi e Michele Colajanni, che sarà sottoposto all'attenzione di una conferenza internazionale.

Dal punto di vista teorico l'approccio definito è idoneo ad essere applicato nella pratica. Anche l'implementazione proposta è effettivamente utilizzabile, ma sono possibili dei miglioramenti futuri che ne renderebbero ancora più comodo l'uso e ne aumenterebbero la sicurezza.

Un punto critico di quanto realizzato è la distribuzione della configurazione sui vari nodi della rete e il suo aggiornamento. Queste operazioni sono facilmente automatizzabili con degli script, ma sono fortemente dipendenti dalla tipologia di nodo a cui è destinata la configurazione: macchine virtuali, Pod di Kubernetes, dispositivi fisici. Un'estensione interessante di quanto svolto potrebbe essere la

modifica della configurazione della rete per includere le informazioni sulla tipologia di nodo e l'aggiunta di funzionalità all'applicativo Haskell per semplificare la distribuzione dei file di configurazione. La creazione di una interfaccia grafica renderebbe la definizione della configurazione ancora più facile e comoda di com'è ora.

L'aggiornamento della configurazione richiede, oltre alla distribuzione dei file di configurazione, il riavvio del software Nebula, in quanto quest'ultimo non supporta ancora l'hot reloading di tutte le impostazioni. Per permettere questa operazione sarebbe quindi necessario o estendere le capacità di Nebula o creare uno strumento che si occupi di riavviare Nebula se necessario.

Un'ulteriore estensione di Nebula che sarebbe utile implementare è la verifica dell'autenticità della configurazione fornita: durante l'avvio Nebula potrebbe controllare che la configurazione sia firmata dalla stessa Certification Authority che ha rilasciato il certificato delle sue chiavi. Questo impedirebbe la manipolazione della configurazione da parte di un attaccante che riuscisse ad avere accesso, anche privilegiato, ai nodi.

Bibliografia

[1] *3 Approaches to Micro-Segmentation and Their Pros and Cons*. URL: https://colortokens.com/blog/approaches-micro-segmentation-pros-and-cons/ (visitato il 09/08/2022).

[2] Lampis Alevizos, Vinh Thong Ta e Max Hashem Eiza. «Augmenting Zero Trust Architecture to Endpoints Using Blockchain: A State-of-The-Art Review». In: *SECURITY AND PRIVACY* 5.1 (gen. 2022). ISSN: 2475-6725, 2475-6725. DOI: 10.1002/spy2.191. URL: http://arxiv.org/abs/2104.00460 (visitato il 16/08/2022).

[3] Nardine Basta et al. «Towards a Zero-Trust Micro-segmentation Network Security Strategy: An Evaluation Framework». In: *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. ISSN: 2374-9709. Apr. 2022, pp. 1–7. DOI: 10.1109/NOMS54207.2022.9789888.

[4] Cristoph Buck et al. «Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust». In: (2021). DOI: 10.1016/j.cose.2021.102436.

[5] Mark Campbell. «Beyond Zero Trust: Trust Is a Vulnerability». In: *Computer* 53.10 (ott. 2020). Conference Name: Computer, pp. 110–113. ISSN: 1558-0814. DOI: 10.1109/MC.2020.3011081.

[6] Alan S. Cohen. *The Truth About Micro-Segmentation*. 2018.

[7] *Config as Code: What is it and how is it beneficial?* URL: https://octopus.com/blog/config-as-code-what-is-it-how-is-it-beneficial (visitato il 15/08/2022).

[8] *Configuration as Code: How to Streamline Your Pipeline*. URL: https://www.perforce.com/blog/vcs/configuration-as-code (visitato il 15/08/2022).

[9] *Configuration reference guide for Nebula*. URL: `https://www.defined.net/nebula/config` (visitato il 25/07/2022).

[10] *Dhall Documentation*. URL: `https://docs.dhall-lang.org` (visitato il 14/07/2022).

[11] Luca Ferretti et al. «Survivable zero trust for cloud computing environments». In: *Computers & Security* 110 (nov. 2021), p. 102419. ISSN: 0167-4048. DOI: `10.1016/j.cose.2021.102419`. URL: `https://www.sciencedirect.com/science/article/pii/S0167404821002431` (visitato il 30/08/2022).

[12] *gin/BENCHMARKS*. URL: `https://github.com/gin-gonic/gin/blob/master/BENCHMARKS.md` (visitato il 13/08/2022).

[13] Tom Goethals et al. «Scalability evaluation of VPN technologies for secure container networking». In: ISSN: 2165-963X. Ott. 2019, pp. 1–7. DOI: `10.23919/CNSM46954.2019.9012673`.

[14] Nicole Henderson e Eric Hanselman. *2022 Global Hybrid Cloud Trends Report*. Mag. 2022.

[15] Ryan Huber. *Introducing Nebula, the open source global overlay network from Slack*. URL: `https://slack.engineering/introducing-nebula-the-open-source-global-overlay-network-from-slack/` (visitato il 20/07/2022).

[16] Ryan Huber. *Nebula is officially a thing, and today it is available on iOS and Android!* Ott. 2020. URL: `https://medium.com/definednet/mobile-nebula-ios-android-mesh-vpn-1088a7c536ee`.

[17] *Illumio Technical Documentation*. URL: `https://docs.illumio.com/core/22.2/Content/Home.htm` (visitato il 14/08/2022).

[18] *Implementing a Zero Trust security model at Microsoft*. URL: `https://www.microsoft.com/en-us/insidetrack/implementing-a-zero-trust-security-model-at-microsoft` (visitato il 05/09/2022).

[19] Nathanael Iversen. *High Watermarks for Micro-Segmentation in 2021*. URL: `https://www.illumio.com/blog/micro-segmentation-2021` (visitato il 11/08/2022).

[20] John Kindervag. «Build Security Into Your Network's DNA: The Zero Trust Network Architecture». en. In: (2010).

[21] *Low transmission efficiency on moderate/low host*. URL: https://github.com/slackhq/nebula/issues/136 (visitato il 06/08/2022).

[22] *Micro-Segmentation Deployment Models*. URL: https://www.zentera.net/knowledge/micro-segmentation-types (visitato il 09/08/2022).

[23] *Nebula Project*. URL: https://www.defined.net/nebula/ (visitato il 20/07/2022).

[24] *North-South and East-West*. URL: https://www.zentera.net/knowledge/micro-segmentation-nsew (visitato il 13/08/2022).

[25] *overlay network*. URL: https://www.techtarget.com/searchnetworking/definition/overlay-network (visitato il 03/08/2022).

[26] Scott Rose et al. *Zero Trust Architecture*. Rapp. tecn. Ago. 2020. DOI: 10.6028/NIST.SP.800-207. URL: https://doi.org/10.6028/NIST.SP.800-207.

[27] Binanda Sengupta e Anantharaman Lakshminarayanan. «DistriTrust: Distributed and low-latency access validation in zero-trust architecture». In: (2021). DOI: 10.1016/j.jisa.2021.103023.

[28] Nabeel Sheikh, Mayur Pawar e Victor Lawrence. «Zero trust using Network Micro Segmentation». In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Mag. 2021, pp. 1–6. DOI: 10.1109/INFOCOMWKSHPS51825.2021.9484645.

[29] Naeem Firdous Syed et al. *Zero Trust Architecture (ZTA): A Comprehensive Survey*. Mag. 2022.

[30] *The 2020 State of Virtualization Technology*. URL: https://www.spiceworks.com/marketing/reports/state-of-virtualization/ (visitato il 05/09/2022).

[31] *The Dhall configuration language*. URL: https://dhall-lang.org/ (visitato il 14/07/2022).

[32] *What is Infrastructure as Code?* URL: https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code (visitato il 14/08/2022).

[33] *What is Microsegmentation?* URL: `https://www.paloaltonetworks.com/cyberpedia/what-is-microsegmentation` (visitato il 08/08/2022).

[34] *What Is Policy-as-Code?* URL: `https://www.paloaltonetworks.com/cyberpedia/what-is-policy-as-code` (visitato il 14/08/2022).