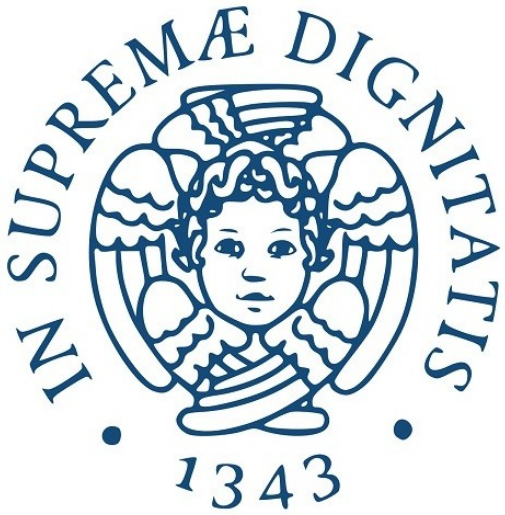
Transcompilazione fra Linguaggi di Firewall

sintesi e generazione di configurazioni



Relatore : Prof. Pierpaolo Degano

Candidato : Lorenzo Ceragioli

Dipartimento di Informatica

Università di Pisa

Laurea Magistrale in Informatica

Indice

Sommario

Il porting delle configurazioni da un sistema di firewall a un altro è un procedimento difficile e costoso. Le configurazioni consistono in centinaia di regole scritte in linguaggi di basso livello, specifici della piattaforma ed in cui l'ordine delle regole influenza la semantica della configurazione. Senza una procedura automatica per il porting, un amministratore è tenuto a conoscere i dettagli delle politiche di sicurezza implementate e a progettare da capo la configurazione per il nuovo sistema. Nel caso in cui le politiche di sicurezza non siano state documentate accuratamente è necessario analizzare la configurazione iniziale e tentare di creare una configurazione equivalente per il sistema target: questo è un procedimento rischioso perché è possibile tralasciare dettagli significativi e produrre un firewall non equivalente a quello di partenza. È possibile in questo modo compromettere la sicurezza della rete in quanto si implementa senza accorgersene una politica diversa da quella originale e gli asset potrebbero non essere protetti in modo corretto. In un lavoro recente è stata proposta una pipeline di transcompilazione fra linguaggi di configurazione di firewall, composta da tre fasi: (*i*) decompilazione della configurazione dal linguaggio di origine ad un linguaggio intermedio; (*ii*) estrazione del significato della configurazione come insieme minimale di regole dichiarative che descrivono i pacchetti accettati e le traduzioni in termini logici; (*iii*) compilazione delle regole dichiarative nel linguaggio target. Lo strumento *firewall synthesizer* rappresenta l'implementazione delle prime due fasi in quanto permette, facendo uso di un SAT solver, di derivare una rappresentazione ad alto livello della semantica di un firewall. Per la fase (*iii*) è stato proposto un algoritmo che tuttavia non ha garanzie di conservare la traduzione degli indirizzi (Network Address Translation o `NAT`) e che si basa sull'operazione di marking dei pacchetti, la quale è soggetta a restrizioni differenti nei vari sistemi firewall.

In questa tesi presentiamo un nuovo algoritmo per la sintesi della semantica nella fase (*ii*) che non necessita di un SAT solver e analizziamo formalmente la generazione della configurazione target nella fase (*iii*), tenendo in considerazione il problema di `NAT`. A questo scopo studiamo la differente espressività dei sistemi firewall riguardo la traduzione degli indirizzi dei pacchetti. Nel linguaggio intermedio, che è dotato di una semantica formale, ogni sistema firewall è modellato tramite un diagramma di controllo e ogni configurazione come un assegnamento di ruleset ai nodi del diagramma. Per ogni linguaggio di configurazione individuiamo dei vincoli che caratterizzano quali assegnamenti di ruleset ai nodi del diagramma di controllo possono essere espressi e sfruttiamo questi vincoli sia per studiare l'espressività dei linguaggi di configurazione, sia per definire un algoritmo per la generazione della configurazione finale.

Capitolo 1

Introduzione

I firewall sono uno dei meccanismi standard per la protezione di reti di computer, la loro funzione è quella di ispezionare il traffico della rete, filtrandolo in accordo con la propria configurazione. Oltre a fungere da filtro, i firewall contribuiscono a realizzare l'instradamento, effettuando delle traduzioni sugli indirizzi dei pacchetti attraverso il NAT. Per decidere se consentire o meno il transito di un pacchetto, e quali trasformazioni effettuare, il firewall si basa su un insieme di regole stabilite dall'amministratore. Il linguaggio per la definizione delle regole e l'ordine nel quale queste sono valutate dipendono dal sistema scelto.

Le configurazioni consistono comunemente in centinaia di regole scritte in linguaggi di basso livello, in cui le regole interagiscono fra loro e il loro ordine influenza la semantica della configurazione. Spesso l'ordine di valutazione delle regole è non banale, e può essere modificato da istruzioni dedicate, simili alle chiamate di procedura dei linguaggi di programmazione imperativi. Inoltre questi linguaggi sono del tutto privi di semantica formale, e in genere anche la caratterizzazione informale è piuttosto lasca.

Le regole possono essere statiche, se dipendono unicamente dalle proprietà del pacchetto, o dinamiche se dipendono dallo stato del firewall stesso, che tiene traccia delle connessioni attive, permettendo filtri e modifiche dipendenti dai pacchetti precedentemente osservati.

Il porting di un firewall è un compito oneroso e rischioso anche per amministratori esperti; è infatti necessario conoscere a fondo sia il sistema di origine, sia quello di destinazione ed è possibile introdurre accidentalmente delle vulnerabilità senza accorgersene. Attualmente infatti l'approccio comune è quello di ripartire dalle politiche di sicurezza che si era deciso di implementare nel sistema precedente e codificarle da capo nel linguaggio del sistema target, verificando infine con dei test l'aderenza del comportamento del firewall rispetto a quello atteso. Questo procedimento è molto dispendioso in termini di tempo e non si hanno garanzie che i due sistemi si comporteranno nella stessa identica maniera di fronte agli stessi pacchetti. Presentiamo un metodo automatico per il porting delle configurazioni, basato sulla transcompilazione, che garantisce la conservazione della semantica del firewall, sia per quanto riguarda il filtro, sia per le trasformazioni.

Basiamo il nostro approccio su IFCL, un linguaggio formale per la definizione di firewall originariamente presentato in [4, 5], che usiamo come linguaggio intermedio. IFCL evidenzia la struttura bipartita tipica dei sistemi firewall, composta dalle regole in sé e dal meccanismo che stabilisce l'ordine di valutazione e la modalità di applicazione. La prima parte è modellata attraverso un insieme di liste di regole scritte in un linguaggio formale che è stato progettato in modo tale da consentire una compilazione relativamente facile da ogni sistema firewall, in quanto incorpora tutte le feature dei linguaggi di firewall come il NAT, i salti, le invocazioni e l'accesso allo stato delle connessioni. La seconda parte è modellata attraverso un *diagramma di controllo*, un grafo in cui ai nodi sono assegnate

liste di regole, e gli archi sono etichettati da predicati e rappresentano la possibilità di passare da uno stato ad un altro, nel processo di valutazione del pacchetto.

Il procedimento di transcompilazione è una pipeline composta da quattro stadi: (*i*) il firewall iniziale viene rappresentato attraverso IFCL; (*ii*) si calcola una rappresentazione astratta e sintetica della semantica della configurazione come funzione dall'insieme dei pacchetti alle possibili trasformazioni (compreso l'essere scartato); (*iii*) si genera un firewall IFCL del tipo target avente semantica corrispondente a quella calcolata al punto precedente; (*iv*) si compila la configurazione IFCL nel linguaggio target.

Valutiamo inoltre l'espressività dei linguaggi di configurazione, intesa come l'insieme delle funzioni dall'insieme dei pacchetti alle possibili trasformazioni che possono essere espresse da una configurazione per il sistema in esame. Dimostriamo che non tutti i sistemi firewall sono capaci di esprimere le stesse funzioni su pacchetti. Questo studio ci fornisce dei limiti entro i quali il porting può essere effettuato in maniera corretta.

La soluzione proposta supporta i sistemi `iptables`, `pf` e `ipfw`, che sono sfruttati per fornire esempi concreti; tuttavia la procedura di transcompilazione è definita in modo tale da essere adattabile, abbastanza semplicemente, a nuovi sistemi. Infatti, la parte centrale della pipeline è basata su IFCL, e gli algoritmi impiegati sono parametrici rispetto alle specifiche del firewall, anziché limitarsi ai soli sistemi supportati attualmente. Per estendere il supporto ad un nuovo sistema è quindi sufficiente fornire una sua caratterizzazione attraverso IFCL. Il funzionamento dell'algoritmo che implementa la fase (*iii*) della pipeline, per il momento, è garantito unicamente quando il sistema target non consente di effettuare lo stesso tipo di trasformazione (SNAT o DNAT) in più momenti diversi della valutazione di un pacchetto.

1.1 Background

Presentiamo i lavori correlati a quello esposto in questa tesi, facendo particolare attenzione al linguaggio di configurazione intermedio IFCL, presentato in [5] e [4], sul quale basiamo il nostro approccio di transcompilazione, e che definiremo in dettaglio nel capitolo 2 (in una versione lievemente modificata). Per ognuno dei sistemi attualmente supportati diamo una piccola introduzione che permetta al lettore di comprendere gli esempi presentati nel resto della tesi. Concludiamo, infine, presentando alcune definizioni che useremo in seguito.

1.1.1 Lavori correlati

I metodi formali sono stati impiegati per modellare il controllo degli accessi dei firewall seguendo diversi metodi, si vedano ad esempio [6, 11, 9]; qui noi restringiamo la nostra attenzione agli approcci che, come il nostro, si basano sui linguaggi.

Recentemente c'è stata una crescita di interesse riguardo linguaggi di alto livello per la programmazione di reti di computer come un tutt'uno. Il paradigma delle Software Defined Network (SDN) separa il controllo della rete dalle funzioni di inoltro, astraendo dalle applicazioni e dai servizi di rete i dettagli legati all'infrastruttura sottostante [16]. Un approccio unificato e di alto livello alla configurazione delle reti e dei firewall è attraente e potrebbe rendere le configurazioni più semplici e meno soggette ad errori. Tuttavia le SDN richiedono una infrastruttura adeguata e, nonostante la rapidità di diffusione, non possiamo aspettarci che le "vecchie" tecnologie siano dismesse troppo rapidamente, anche per questioni legate alla conservazione di sistemi legacy. Negli anni a venire avremmo

ancora bisogno di affrontare i problemi legati alla configurazione, verifica e porting di sistemi firewall tradizionali.

In questo lavoro seguiamo un approccio antitetico rispetto a quello di SDN, consideriamo i linguaggi di configurazione specifici delle diverse piattaforme dei *linguaggi macchina*, con i quali interagiamo attraverso compilazioni e decompilazioni verso linguaggi di alto livello, che nel caso della pipeline di transcompilazione fungono da linguaggi intermedi.

La transcompilazione è una tecnica ben nota nell'ambito del code refactoring, della parallelizzazione automatica e del porting di codice legacy in nuovi linguaggi di programmazione. Recentemente questa tecnica è stata ampiamente usata nel campo della programmazione web per implementare dei linguaggi di programmazione di alto livello in JavaScript, si vedano ad esempio [2, 19]. Al meglio delle nostre conoscenze, oltre a [4], non sono presenti in letteratura approcci per il porting automatico delle configurazioni dei firewall. Esistono invece approcci che attraverso la definizione di una semantica formale dei linguaggi di configurazione esistenti, permettono di fare refactoring e di verificare la presenza di eventuali errori. Alcuni di questi approcci sono basati sulla traduzione delle configurazioni analizzate in linguaggi ad alto livello, nei quali risulta più facile controllare l'aderenza alle policy di sicurezza. Sono stati proposti anche approcci che a partire dalle specifiche di sicurezza espresse attraverso linguaggi formali ad alto livello, permettono la generazione di configurazioni per sistemi reali.

La proposta di [8] "pulisce" le regole, poi le analizza con uno strumento automatico; usa una semantica formale di `iptables` (senza NAT) per descrivere delle trasformazioni per la semplificazione che preservano la semantica. Il tool FIREMAN [24] localizza inconsistenze e inefficienze delle configurazioni firewall (senza NAT). Margrave [15] è un analizzatore di configurazioni per firewall IOS: è estensibile ad altri linguaggi, ma si concentra sulla ricerca di errori specifici piuttosto che nella sintesi di una specifica di alto livello del significato della configurazione. Un altro strumento per la ricerca di anomalie è Fang [12, 13], che sintetizza anche una politica astratta. Mignis [14, 1] è un tool che permette di definire policy ad alto livello, con un linguaggio dotato di semantica formale, e di compilarle in configurazioni `iptables` (senza possibilità immediata di generalizzazione). Fra gli articoli che formalizzano la semantica dei linguaggi di firewall menzioniamo anche [3] che permette di specificare delle politiche di filtro astratte da compilare nel sistema firewall reale.

Molti di questi approcci definisce il proprio linguaggio formale, ma la compilazione viene effettuata sempre in una sola direzione: dal linguaggio di configurazione reale a quello ad alto livello per analisi e correzione; oppure nel verso opposto per la generazione di configurazioni a partire dalle specifiche. Inoltre molti dei lavori citati non supportano componenti essenziali del comportamento reale dei firewall, come NAT e lo stato interno; e spesso si concentrano su un sistema firewall specifico. Nel nostro approccio invece, IFCL funge da linguaggio intermedio fra i linguaggi reali source e target. Pertanto la traduzione fra IFCL e i linguaggi di configurazione è definita in entrambe le direzioni. Gestiamo inoltre sia i costrutti per la modifica dei pacchetti, sia lo stato interno del firewall.

Il nostro approccio differisce da quelli proposti precedentemente quindi in quanto *allo stesso tempo*: (*i*) è indipendente dal linguaggio; (*ii*) definisce una semantica formale capace di esprimere il comportamento dei firewall; (*iii*) supporta NAT e stato interno; (*iv*) consente di derivare una rappresentazione ad alto livello, funzionale e concisa del comportamento del firewall; (*v*) permette di generare una configurazione per un sistema dato, il cui comportamento sia coerente con la descrizione ad alto livello.

Il nostro lavoro si basa soprattutto su [4], nel quale si propone una pipeline di transcompilazione fra linguaggi di configurazione di firewall, e di cui questa tesi può essere considerata una prosecuzione.

L'articolo [5] descrive la progettazione di un tool automatico per la sintesi della semantica di una configurazione firewall, e la sua applicazione a casi reali. Il tool implementa i primi due stadi della pipeline di transcompilazione proposta in [4] e supporta l'amministratore di sistema nella verifica delle politiche data una configurazione. In particolare l'utente può verificare implicazione, equivalenza e differenze fra configurazioni, e la raggiungibilità fra host. Il tool usa la stessa sintassi di [4] in cui però IFCL viene presentato in maniera più approfondita, definendone la semantica operazionale e spiegando in maniera approfondita e formale le fasi di sintesi della rappresentazione dichiarativa della semantica del firewall, e dimostrandone la correttezza. Inoltre in [4] si affronta il problema della compilazione della specifica dichiarativa nel linguaggio target, permettendo la transcompilazione. Per risolvere automaticamente i problemi di porting e refactoring delle configurazioni, si propone una pipeline di transcompilazione composta dalle seguenti fasi:

1. traduzione in IFCL della configurazione espressa nel linguaggio source;
2. estrazione della semantica della configurazione come insieme di regole dichiarative non sovrapposte che descrivono i pacchetti accettati e le loro trasformazioni in termini logici;
3. compilazione delle regole dichiarative nel linguaggio target.

La prima fase della pipeline è invariata rispetto alla versione che discutiamo in questa tesi. La seconda fase si basa su una caratterizzazione logica della semantica di IFCL; la rappresentazione sintetica consiste in una descrizione concisa, basata su multicubi, dell'insieme dei modelli del predicato che rappresenta la semantica, ed è calcolata attraverso un SAT solver. Per quanto relativamente efficiente, questa fase rappresenta il collo di bottiglia della transcompilazione; da qui il desiderio di studiare un algoritmo alternativo.

Per l'ultima fase della pipeline viene proposto un algoritmo basato sui tag, che produce una configurazione IFCL per il sistema target. In [4] si garantisce che il firewall prodotto accetta tutti e soli i pacchetti accettati dal firewall di partenza. Tuttavia non non si danno garanzie sulla conservazione delle trasformazioni NAT e non viene studiata la possibilità effettiva di ricompilare il firewall ottenuto nel linguaggio target. La ricompilazione da IFCL al linguaggio reale potrebbe in effetti essere problematica proprio a causa del largo uso che viene fatto dei tag, i quali sono soggetti a limitazioni diverse nei vari sistemi, in contraddizione con l'intento di delineare un approccio il più generale possibile.

Nel capitolo 2, definiamo una versione lievemente modificata di IFCL e in generale presentiamo i risultati di [4] che ci servono come base per impostare la nostra rivisitazione della pipeline di transcompilazione.

1.1.2 Sistemi supportati

Per il momento i sistemi supportati sono `iptables`, `pf` e `ipfw`. Dato che una volta tradotta la configurazione in IFCL, la procedura di transcompilazione è indipendente dal sistema di origine, è possibile supportare nuovi sistemi fornendo un compilatore da e per il linguaggio di configurazione desiderato.

iptables

Si tratta del sistema firewall di default delle distribuzioni Linux [21]. Opera grazie a *Netfilter*, il framework standard per la gestione di pacchetti implementato nel kernel di Linux [22]. Ogni regola di `iptables` è assegnata ad una tabella e ad una catena. Intuitivamente, una catena è una lista ordinata di regole, una tabella è una collezione di catene.

Le tabelle più comunemente usate sono:

`FILTER` : per il filtro di pacchetti, nella quale è possibile scartare i pacchetti che non soddisfano determinati vincoli;

`NAT` : per la traduzione degli indirizzi in accordo con un protocollo di NAT (Network Address Translation), nella quale è possibile modificare gli indirizzi dei pacchetti;

`MANGLE` : per l'alterazione dei pacchetti, dove si possono associare etichette ai pacchetti, aggiornare contatori etc.

Ci sono cinque catene built-in che sono ispezionate in momenti specifici del ciclo di vita di un pacchetto, e sulla base dell'interfaccia di destinazione e origine [23]:

`prerouting` : quando il pacchetto raggiunge l'host;

`forward` : quando il pacchetto viene instradato dall'host;

`postrouting` : quando il pacchetto sta per lasciare l'host;

`input` : quando il pacchetto è instradato verso l'host;

`output` : quando il pacchetto è generato dell'host.

Non tutte le tabelle contengono necessariamente tutte le catene. L'utente può inoltre definire delle proprie catene, che saranno ispezionate solo se chiamate espressamente da quelle built-in. Ogni regola è divisa in due parti: una condizione e un target. Se un pacchetto verifica la condizione allora viene gestito in accordo con il target. I target possono essere quelli predefiniti oppure una catena user-defined. I target predefiniti più comuni sono:

`ACCEPT` : permette al pacchetto di passare, continuando la valutazione delle altre catene;

`DROP` : scarta il pacchetto;

`RETURN` : interrompe la valutazione della catena corrente e ritorna alla valutazione della catena precedente;

`DNAT` : permette di modificare l'indirizzo di destinazione del pacchetto (destination NAT), il pacchetto viene immediatamente accettato dalla ruleset;

`SNAT` : permette di modificare l'indirizzo di origine del pacchetto (source NAT), il pacchetto viene immediatamente accettato dalla ruleset.

Quando invece il target è un catena user-defined occorre specificare un metodo di invocazione fra:

`call` : che esegue la catena chiamata fino alla fine o al primo `RETURN`, e che successivamente ritorna alla catena chiamante come se si fosse invocata una procedura in un linguaggio imperativo;

`jump` : che passa ad eseguire la nuova catena in modo definitivo, senza tornare alla catena chiamante, come nei salti dei linguaggi assembly.

Le catene built-in hanno una policy di default configurabile, corrispondente ad `ACCEPT` o `DROP`: se un pacchetto raggiunge la fine della catena built-in o di una catena invocata con metodo `jump`, senza che il pacchetto sia accettato o scartato, allora si applica la policy di default.

pf

Si tratta del firewall standard di *OpenBSD* [17], supportato anche da *FreeBSD* [10]. Differentemente dagli altri firewall, l'azione applicata ad un pacchetto dipende dall'ultima regola in cui la condizione è verificata dal pacchetto, non dalla prima, tranne dove specificato diversamente attraverso l'etichetta `quick`.

`pf` ha una singola ruleset che viene ispezionata sia all'arrivo sull'host, sia alla partenza da esso. Nella versione di *FreeBSD* le regole di traduzione, quelle che implementano NAT, sono applicate prima di quelle di filtro, nella versione di *OpenBSD* si segue rigidamente l'ordine di definizione.

I pacchetti appartenenti a connessioni stabilite sono accettati di default, aggirando le regole di filtro.

ipfw

Si tratta del sistema firewall standard per *FreeBSD* [20]. Come in `pf`, le regole sono inserite in un'unica lista che viene valutata due volte, quando il pacchetto arriva all'host e quando lo lascia (un'etichetta `in` o `out` può essere applicata alla regola se vogliamo che sia applicata solo in uno dei due casi); e come in `iptables` queste sono valutate in ordine e la prima di cui è verificata la condizione viene applicata.

Il pacchetto viene scartato se nessuna condizione è verificata. L'ordine di valutazione sequenziale può essere alterato da regole contenenti `skipto` e `goto`. `goto` è simile all'istruzione `jump` di `iptables`, ma la destinazione invece di essere una catena separata, è una regola all'interno dell'unica lista di regole. `skipto` è equivalente a `goto`, ma è valido solo se la destinazione è successiva alla regola chiamante.

I pacchetti che appartengono a connessioni stabilite possono essere accettati usando regole apposite.

1.1.3 Definizioni preliminari

Chiamiamo **IP** l'insieme degli indirizzi IPv4, cioè i numeri interi da 0 a $2^{32}-1$, che rappresenteremo con la classica notazione composta da quattro numeri fra 0 e 255 separati da punti; **Port** l'insieme delle porte, cioè numeri interi da 0 a $2^{16}-1$; e **Tag** l'insieme dei possibili tag associabili ad un pacchetto dal firewall. Sulla forma di **Tag** non facciamo assunzioni, in alcuni sistemi i tag sono stringhe arbitrarie, in altri sono numeri interi (talvolta confrontabili per intervallo e usando maschere).

Definiamo $\mathbb{P}$, l'insieme di tutti i pacchetti, come il prodotto cartesiano dei possibili valori per: IP di origine, porta di origine, IP di destinazione, porta di destinazione, tag del pacchetto.

$$\mathbb{P} = \mathbf{IP} \times \mathbf{Port} \times \mathbf{IP} \times \mathbf{Port} \times \mathbf{Tag}$$

Per alcuni protocolli, come ad esempio *ICMP*, non sono definiti i campi relativi alle porte, per questo assumiamo la presenza di un valore speciale $\diamond \in \mathbf{Port}$ corrispondente alla mancanza del campo stesso. Ogni predicato $\phi(port)$ diverso da $true$, presente nella condizione di una regola di firewall, dove $port \in \mathbf{Port}$, è falsificato da un assegnamento di valore che associa $\diamond$ a $port$.

Usiamo una notazione "ad oggetti" per accedere ai campi di un pacchetto p, scrivendo $p.sIP$, $p.sPort$, $p.dIP$, $p.dPort$ e $p.tag$. Inoltre, per leggibilità, anziché usare la classica notazione per le tuple, ci riferiamo al pacchetto $(sIP, sPort, dIP, dPort, tag)$ come $(sIP : sPort,\ dIP : dPort,\ tag)$.

I firewall possono modificare i pacchetti, ad esempio attraverso il NAT. Scriviamo $p[da \mapsto a]$ e $p[sa \mapsto a]$, con $a = (ip, port) \in \mathbf{IP} \times \mathbf{Port}$ (rappresentato a sua volta come $ip : port$), per denotare un pacchetto identico a p, tranne per l'IP e la porta relativamente di destinazione da e di origine sa, che sono uguali ad ip e $port$. Allo stesso modo, $p[tag \mapsto m]$ denota un pacchetto identico a p, con il campo tag uguale a m.

1.2 Ipotesi di lavoro

Nonostante l'intento di non tralasciare aspetti fondamentali del comportamento dei firewall reali, come NAT e stato interno, facciamo comunque delle semplificazioni rispetto al reale funzionamento dei sistemi supportati. Delineiamo, attraverso queste assunzioni, un contesto abbastanza contenuto da poter essere trattato in modo sufficientemente completo e in una forma adeguata, supportando comunque la maggior parte dei casi di studio reali.

Imponiamo che le trasformazioni NAT applicate dai firewall possano modificare i campi stabiliti solo in modo deterministico, escludendo quindi la possibilità di specificare per i campi del pacchetto, trasformazioni con valori scelti non deterministicamente da intervalli o insiemi arbitrari. Questo serve a garantire che il comportamento dei firewall stessi sia deterministico. Ad esempio in `iptables`, normalmente, è possibile definire una trasformazione NAT secondo la quale l'IP di destinazione viene modificato in uno fra i possibili indirizzi della sottorete 192.168.0.0/24; l'indirizzo viene selezionato con una politica round robin (a meno di configurazioni differenti).

Assumiamo che questo genere di configurazione non siano legali, anche per evitare conseguenze apparentemente contraddittorie, come il fatto che un pacchetto possa essere accettato o scartato non deterministicamente. Alcune versioni di questa generalizzazione del NAT potrebbero essere definite sfruttando lo stato interno, senza bisogno di estendere il modello per gestire il non determinismo; in effetti per implementare una politica round robin, tutto quello che ci serve è la memoria riguardo l'ultimo indirizzo assegnato.

È bene segnalare che, a meno di approssimazioni (come quelle fatte in [4], vedi 2.5), la modellazione delle trasformazioni dovute allo stato, ad esempio in un contesto di NAT dinamico, non comportano problemi in quanto non si tratta di trasformazioni intrinsecamente non deterministiche, ma semplicemente dipendenti dallo stato interno del firewall.

Nella modellazione dei linguaggi di configurazione attraverso IFCL, per quanto riguarda le azioni su pacchetti che dipendono dallo stato interno del firewall, assumiamo l'esistenza di una funzione che, dato lo stato interno e il pacchetto attuale, restituisce l'azione prescritta per il pacchetto. Questo viene fatto senza dare dettagli relativi alla forma di questa funzione e senza descrivere espressamente l'insieme delle azioni che possono essere applicate ai pacchetti. Il formalismo è quindi molto generale, permettendo in teoria di modellare ogni genere di azione che dipenda dallo stato interno come NAT dinamico, politiche di bilanciamento del carico e rate limit; ma demanda la formalizzazione della caratteristica in sé all'utente interessato.

Per lo studio dell'espressività e per l'implementazione della pipeline di transcompilazione ci occupiamo unicamente dei pacchetti appartenenti a nuove connessioni, assumendo che il comportamento di default del sistema target, per pacchetti appartenenti a connessioni stabilite, sia soddisfacente. Infatti ogni sistema ha un modo lievemente diverso di gestire le connessioni stabilite, e non è detto che configurare il sistema target perché simuli quello di origine sia in assoluto desiderabile.

Inoltre, per lo studio dell'espressività e la generazione di firewall tralasciamo l'uso dei tag. Questo perché essi sono soggetti a vincoli diversi nei diversi sistemi di firewall ed è difficile tenere traccia di queste limitazioni quando si parla della versione IFCL del firewall. Per permettere di implementare i diversi sistemi reali in IFCL, infatti, abbiamo definito un sistema di tag il più permissivo possibile; e questo chiaramente è un problema quando la configurazione IFCL deve essere compilata nel linguaggio target, che potrebbe non permettere alcune operazioni adoperate.

1.3 Contributo

Il contributo originale di questo lavoro, soprattutto rispetto alla versione precedente della pipeline di transcompilazione, presentata in [4], può essere riassunto nei seguenti punti:

- definizione di una semantica denotazionale per i firewall IFCL;
- presentazione di un nuovo algoritmo di sintesi che si basa sulla semantica denotazionale e che non richiede l'uso di un SAT solver;
- sviluppo di una teoria per lo studio dell'espressività di un sistema firewall, data la sua caratterizzazione IFCL;
- definizione di una procedura di compilazione che a partire dalla semantica astratta genera un nuovo firewall per il sistema target senza usare i tag;
- valutazione delle condizioni di correttezza dei firewall prodotti dall'algoritmo basato sui tag presentato in [4];
- definizione formale della rappresentazione sintetica usata e dimostrazione di correttezza.

1.4 Schema della tesi

Nel capitolo 2 presentiamo IFCL, il linguaggio formale sul quale basiamo il nostro lavoro: mostriamo come compilare i sistemi supportati in IFCL, ne definiamo la semantica operazionale, forniamo un algoritmo di riscrittura che rimuove i costrutti che influenzano il flusso di controllo (mantenendo invariata la semantica) e forniamo una caratterizzazione dichiarativa della semantica mettendo in luce anche l'approssimazione dello stato interno sulla quale è basata. Nel capitolo 3 definiamo l'insieme delle possibili trasformazioni su pacchetti e diamo la semantica denotazionale di IFCL come funzione da pacchetti a trasformazioni. Nel capitolo 4 ridefiniamo la pipeline di transcompilazione, modificando quella originariamente proposta in [4] ed evidenziando i passi intermedi e le condizioni necessarie affinché il firewall prodotto sia semanticamente equivalente a quello di partenza. Qui introduciamo anche la rappresentazione sintetica usata ed enunciamo le condizioni sotto le quali il comportamento degli algoritmi, che operano sulla rappresentazione sintetica, rispetta le condizioni formali di correttezza espresse. Il capitolo 5 è dedicato al nuovo algoritmo per il calcolo della semantica astratta di una configurazione; si presenta anche un algoritmo per la rimozione dei cicli dal diagramma di controllo di un firewall. Nel capitolo 6 studiamo l'espressività dei diversi sistemi firewall basandoci sulla loro formalizzazione IFCL, prima definiamo le configurazioni IFCL legali in un dato sistema, poi attraverso queste deduciamo quali funzioni da pacchetti a trasformazioni sono esprimibili in quel sistema. Il capitolo 7 presenta l'algoritmo per la generazione della configurazione target, specificando in quali casi abbiamo garanzie di successo e confrontandolo con quello presentato in [4]; per entrambi gli algoritmi analizziamo anche la possibilità di compilare il firewall ottenuto da IFCL al linguaggio target. Infine, nel capitolo 8 concludiamo tirando le somme sul lavoro svolto e parlando dei possibili sviluppi futuri che ci aspettiamo, sia per quanto riguarda la produzione di uno strumento software che implementi quanto descritto, sia rispetto a possibili estensioni della teoria e rilassamenti delle ipotesi di lavoro.

Capitolo 2

IFCL

In questo capitolo presentiamo IFCL (*Intermediate Firewall Configuration Language*), un formalismo per la definizione di firewall proposto originariamente in [5] e riportato con maggiore dettaglio in [4]. I firewall modellati in IFCL sono composti da due elementi: il diagramma di controllo, che rappresenta un'astrazione dell'algoritmo di controllo del sistema firewall, e la configurazione, un assegnamento di ruleset espresse in un linguaggio comune ai nodi del diagramma di controllo.

Mostreremo come modellare `iptables`, `pf` e `ipfw` in IFCL e daremo una definizione formale della semantica operazionale di un firewall espresso in IFCL. Presenteremo anche una procedura di normalizzazione delle configurazioni e una caratterizzazione logica del linguaggio, entrambe utili per la realizzazione della pipeline di transcompilazione.

I contenuti presentati in questo capitolo sono una rivisitazione del materiale di [4], che è stato adattato alle ipotesi di questa tesi, in particolare all'ipotesi secondo la quale il NAT possa essere effettuato unicamente verso indirizzi singoli e non verso intervalli. Abbiamo inoltre cercato di rendere più chiaro il ruolo dello stato interno del firewall e la differenza fra le proprietà di un firewall legate al sistema firewall impiegato e quelle legate alla sua configurazione.

2.1 Fondamenti di IFCL

Presentiamo il linguaggio di configurazione intermedio IFCL. Non forniamo specifiche esatte del comportamento dello stato, preferendo rimanere parametrici rispetto a esso.

Assumiamo che una qualche nozione di stato $s \in S$ sia definita, dove S è l'insieme di tutti gli stati possibili, e che il sistema tenga traccia di quali pacchetti appartengono a una connessione stabilita e quali no. Il comportamento del firewall rispetto allo stato interno è parametrizzato attraverso il predicato astratto $p \vdash_s \alpha$, vero se il pacchetto p appartiene a una connessione stabilita secondo lo stato s, che prescrive un'azione α da applicare al pacchetto p. Se un pacchetto p non appartiene ad alcuna connessione stabilita secondo lo stato s scriviamo $p \nvdash_s$.

Per permettere la modifica di un pacchetto secondo lo stato della connessione introduciamo un'azione dedicata nel linguaggio delle ruleset, il target `CHECK-STATE`.

Una regola firewall è composta di due parti: un predicato ϕ che esprime una condizione sui pacchetti, e un *target* che definisce l'azione da applicare al pacchetto. Le regole possono predicare su proprietà intrinseche del pacchetto, come indirizzo di origine o destinazione, ed eventualmente anche sull'appartenenza o meno del pacchetto a una connessione stabilita. Pertanto il predicato ϕ ha come parametri sia il pacchetto da valutare che lo stato s del firewall. Non diamo una specifica completa dei predicati che possono comparire come condizione di una regola, ma imponiamo che questi possano

Target	**Effetto**
ACCEPT	Il pacchetto viene accettato dalla ruleset, la valutazione della ruleset termina
DROP	Il pacchetto viene scartato dal firewall, la valutazione del pacchetto termina completamente
CALL(R)	Invoca la ruleset R, questa verrà valutata al massimo fino all'invocazione di un RETURN o alla fine della ruleset R, successivamente si ritornerà alla ruleset chiamante
GOTO(R)	Salta alla ruleset R, il destino del pacchetto sarà quello assegnatogli da R
RETURN	Ritorna dalla ruleset corrente a quella chiamante
NAT(n_d, n_s)	Effettua una trasformazione sul pacchetto e lo accetta, la valutazione della ruleset termina
MARK(m)	Marca il pacchetto col tag m, la valutazione prosegue dalla regola successiva
CHECK-STATE(X)	Esamina lo stato: se il pacchetto appartiene a una connessione stabilita effettua le trasformazioni previste dallo stato, il pacchetto viene accettato e la valutazione della ruleset termina; altrimenti la valutazione prosegue dalla regola successiva

Tabella 2.1: I target IFCL e loro semantica informale.

essere scomposti in una congiunzione di predicati tali che ciascuno di essi predica unicamente su uno dei campi del pacchetto, più uno per lo stato. Questo vincolo è coerente con i linguaggi di configurazione dei sistemi firewall reali e ci serve a garantire alcune proprietà sull'insieme dei pacchetti che verificano la condizione di una regola. Alcune azioni comportano la fine della valutazione della ruleset da parte del pacchetto, altre prevedono che il pacchetto prosegua la valutazione della ruleset dalla regola successiva, altre ancora comportano la fine della valutazione del pacchetto da parte dell'intero firewall.

Consideriamo un insieme di azioni incluso in molti dei firewall reali. Queste azioni non determinano solo il destino del pacchetto, ma influenzano anche il flusso di controllo in cui le regole sono applicate. I target e la loro semantica informale sono presentati in Tabella 2.1.

I target CALL(_) e RETURN implementano un comportamento simile a una chiamata a procedura; GOTO(_) ha un comportamento simile a quello dei jump in assembly. Nell'azione NAT, n_d e n_s sono indirizzi usati per tradurre rispettivamente la destinazione e l'origine.

Usiamo il simbolo $\star$ per indicare la trasformazione identità, ad esempio $n : \star$ rappresenta una trasformazione in cui l'indirizzo IP è trasformato in n e la porta è lasciata inalterata. L'argomento $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ dell'azione CHECK-STATE indica i campi del pacchetto che possono essere modificati dallo stato. Più precisamente, $\rightarrow$ riscrive solo la destinazione, $\leftarrow$ solo l'origine e $\leftrightarrow$ entrambi. Formalmente:

Definizione 1 (Regola di firewall)**.** *Una regola di firewall r è una coppia (ϕ, t) dove t è l'azione target della regola e ϕ è una formula logica su un pacchetto e sullo stato tale che:*

$$\phi(p, s) = \phi_{sIP}(p.sIP) \wedge \phi_{sPort}(p.sPort) \wedge \phi_{dIP}(p.dIP) \wedge \phi_{dPort}(p.dPort) \wedge \phi_{tag}(p.tag) \wedge \phi_s(p, s)$$

per qualche predicato ϕ_{sIP}, ϕ_{sPort}, ϕ_{dIP}, ϕ_{dPort} e ϕ_{tag}, e per un predicato $\phi_s(p, s)$ che può essere solo $\exists\alpha.\ p \vdash_s \alpha$ oppure $\neg\exists\alpha.\ p \vdash_s \alpha$.

Un pacchetto p si abbina a una regola r con target t, in uno stato s se ϕ vale.

Definizione 2 (Abbinamento di una regola)**.** *Data una regola $r = (\phi, t)$, diciamo che p si abbina a r con target t, nello stato s, scritto $p, s \Vdash_r t$, se e solo se $\phi(p, s)$. Scriviamo $p, s \nVdash_r$ quando p non si abbina a r nello stato s.*

Per semplificare la trattazione assumeremo sempre che, per ogni regola del tipo $(\phi, \text{CHECK-STATE}(X))$, valga che

$$\phi(p, s) \Rightarrow \exists\alpha\ .\ p \vdash_s \alpha$$

Nella pratica questo può essere ottenuto imponendo che le condizioni ϕ delle regole con target $\text{CHECK-STATE}(X)$ siano sempre della forma $\phi' \wedge (state = \texttt{ENSTABLISHED})$ per un qualche predicato ϕ' (eventualmente *true*). Si noti che l'applicazione di $\text{CHECK-STATE}(X)$ in uno stato in cui la connessione non è stabilita produce lo stesso risultato di quando la condizione non è verificata, quindi le due regole $(\phi, \text{CHECK-STATE}(X))$ e $(\phi', \text{CHECK-STATE}(X))$ hanno semantica equivalente. In pratica l'assunzione non limita affatto l'espressività del formalismo, si tratta unicamente di una questione di forma.

Definiamo ora come un pacchetto viene elaborato da una lista di regole, da qui in poi chiamata *ruleset*.

Definizione 3 (Ruleset)**.** *Un ruleset R è una lista di regole di firewall (anche vuota) corredata da un target di default indicato come $t_d \in \{\text{ACCEPT}, \text{DROP}\}$, che specifica l'operazione da compiere quando un pacchetto raggiunge la fine della ruleset e non ci sono ruleset chiamanti a cui tornare.*

Quando non specificato diversamente assumeremo che il target di default sia sempre ACCEPT, inoltre chiamiamo per convenienza R_ϵ la ruleset vuota con default policy ACCEPT.

Le regole sono ispezionate una dopo l'altra, viene eseguita l'azione specificata dal target della prima regola che si abbina al pacchetto e la valutazione della ruleset può proseguire o meno, e in modi diversi, sulla base del target. Come abbiamo detto infatti, alcune azioni fanno sì che l'ispezione termini, altre prevedono che si continui a ispezionare la ruleset dalla regola immediatamente successiva, e in altri casi ancora la valutazione delle regole prosegue in maniera diversa dal normale. Per sanità non permettiamo che per mezzo di azioni GOTO(R) o CALL(R) si creino dei cicli nella catena delle ruleset applicate. Il sistema tiene traccia delle ruleset invocate e nel caso si creino delle chiamate circolari fra le ruleset scarta il pacchetto incriminato. Questo comportamento è implementato nella semantica operazionale di IFCL e modella fedelmente il comportamento dei firewall reali supportati.

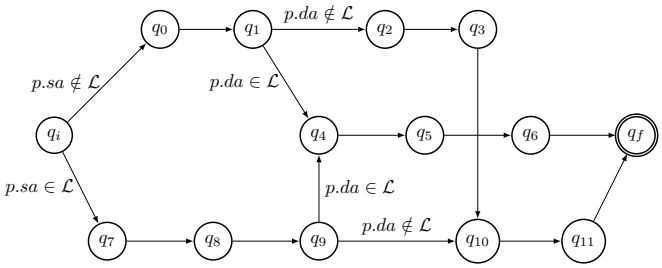
Definizione 4 (Abbinamento in una ruleset)**.** *Data una ruleset $R = [r_1, \ldots, r_n]$, diciamo che R abbina p alla i-esima regola con target t, nello stato s, scritto $p, s \Vdash_R (t, i)$, se e solo se*

$$i \leq n \ \wedge\ p, s \Vdash_{r_i} t \ \wedge\ \forall j < i\ .\ p, s \nVdash_{r_j}$$

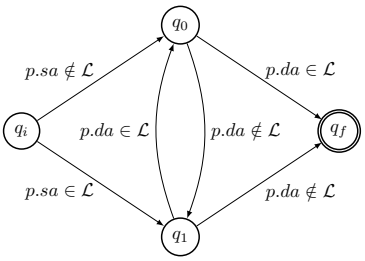
Dove $r_i = (\phi, t)$.

Scriviamo anche $p, s \nVdash_R$ se p non si abbina a nessuna regola in R, nello stato s, formalmente se $\forall r \in R\,.\, p, s \nVdash_r$. Ci sentiremo liberi di omettere l'indice i quando non necessario, scrivendo semplicemente $p, s \Vdash_R t$.

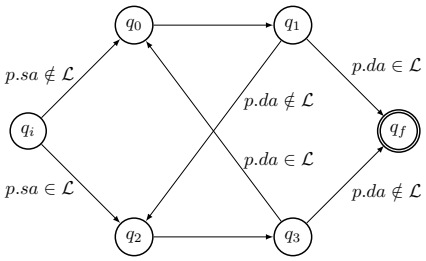
Nel nostro modello, astraendo dalle specifiche operazioni compiute dal sistema operativo per processare un pacchetto, rappresentiamo l'algoritmo che valuta il pacchetto secondo le ruleset attraverso un *diagramma di controllo*, cioè un grafo in cui i nodi rappresentano differenti fasi di valutazione e gli



(a) Diagramma di controllo di `iptables`



(b) Diagramma di controllo di `ipfw`



(c) Diagramma di controllo di `pf`

Figura 2.1: Diagrammi di controllo dei sistemi supportati.

archi i possibili passaggi da una fase all'altra. Gli archi sono etichettati da un predicato che descrive i requisiti che un pacchetto deve soddisfare per poter passare alla fase di valutazione successiva. Assumiamo che i diagrammi di controllo siano deterministici, cioè che ogni pacchetto possa attraversa uno ed uno solo degli archi uscenti da ogni nodo.

Definizione 5 (Diagramma di controllo)**.** *Sia Ψ un insieme di predicati sui pacchetti. Un* diagramma di controllo *$\mathcal{C}$ è una tupla (Q, A, q_i, q_f), dove*

- *Q è un insieme di nodi;*
- *$A \subseteq Q \times \Psi \times Q$ è un insieme di archi tale che $\forall p, q \neq q_f \;.\; \#\{q' \mid (q, \psi, q') \in A \wedge \psi(p)\} = 1$;*
- *q_i,$q_f \in Q$ sono nodi speciali che denotano l'inizio e la fine dell'elaborazione.*

Dato che il diagramma di controllo è deterministico possiamo definire una funzione di transizione che dato lo stato attuale e un pacchetto restituisce il prossimo stato della valutazione.

Definizione 6 (Funzione di transizione)**.** *Sia (Q, A, q_i, q_f) un diagramma di controllo, la funzione di transizione $\delta \colon Q \times \mathbb{P} \mapsto Q$ è definita come*

$$\delta(q, p) = q' \quad sse \quad \exists (q, \psi, q') \in A.\ \psi(p)$$

Possiamo ora definire un firewall IFCL.

Definizione 7 (Firewall IFCL)**.** *Un firewall IFCL $\mathcal{F}$ è una coppia $(\mathcal{C}, \Sigma)$, dove $\mathcal{C}$ è un diagramma di controllo; $\Sigma = (\rho, c)$ è una configurazione in cui ρ è un insieme di ruleset e $c \colon Q \mapsto \rho$ è una* corrispondenza *che mappa ogni nodo di $\mathcal{C}$ in una rulesets in ρ.*

2.2 Modellazione dei sistemi reali supportati

Mostriamo come codificare tre sistemi di firewall standard in IFCL: `iptables`, `pf`[1] e `ipfw`. Una conseguenza immediata è che attraverso la codifica definiremo implicitamente una semantica formale

[1] Nella versione supportata da *freeBSD* [10].

per i tre linguaggi, in termini della semantica formale di IFCL. Per ognuno di questi forniamo un diagramma di controllo (si veda la Figura 4.1), mostriamo come tradurre il file di configurazione in un insieme di ruleset e come assegnare queste ruleset ai nodi del diagramma. La traduzione della configurazione dal linguaggio source è presentata in due fasi, per prima cosa spieghiamo come tradurre le singole regole del linguaggio in IFCL e infine come distribuire le regole ottenute nelle varie ruleset e come assegnarle ai nodi del diagramma.

Per economia di spazio e tempo, non specifichiamo completamente la traduzione da regole del linguaggio di configurazione a IFCL, tralasciando alcuni dettagli non particolarmente delicati e che richiederebbero la presentazione della sintassi completa dei linguaggi di configurazione supportati. In particolare non descriviamo come tradurre le condizioni delle regole in predicati ϕ e diamo solo alcuni accenni su come tradurre i parametri dei target. Crediamo che i dettagli tralasciati siano di immediata comprensione per il lettore e che possano essere ricavati autonomamente senza particolari difficoltà.

2.2.1 Modellazione di iptables

La figura 4.1a mostra il diagramma di controllo $\mathcal{C}_{iptables}$ di `iptables`. In questo grafo e in tutti quelli che seguono, assumiamo che gli archi senza etichetta siano implicitamente etichettati del predicato "*true*". Idealmente ogni nodo corrisponde a una chain built-in e il diagramma ricalca il modo in cui un pacchetto viene valutato attraversando chain differenti a seconda se sia proveniente e diretto verso indirizzi locali o non locali.

Ogni regola di `iptables` è immediatamente traducibile in una equivalente in IFCL da una semplice elaborazione sintattica, si noti che il target `CALL(_)` di IFCL corrisponde al target `jump` di `iptables` per l'invocazione di una chain user-defined. In `iptables` ogni regola è associata a una tabella e a una chain built-in oppure è associata a una chain user-defined. Ogni coppia tabella chain built-in e ogni chain user-defined corrisponde a una diversa ruleset. Ogni nodo del diagramma di controllo corrisponde a una coppia tabella chain built-in.

Data la funzione di traduzione delle singole regole, la ruleset relativa a una coppia tabella chain built-in o a una chain user-defined consiste semplicemente nella concatenazione della traduzione delle regole a essa assegnate. L'ordine delle regole nelle ruleset deve essere lo stesso del file di configurazione source in quanto l'ordine di valutazione delle regole è lo steso per IFCL e `iptables` (vengono applicate dalla prima all'ultima).

In `iptables` ci sono dodici chain built-in, ognuna delle quali corrisponde a una singola ruleset. Possiamo definire un insieme $\rho_p \subseteq \rho$ di ruleset, che sono la traduzione delle ruleset built-in di `iptables`. Chiamiamo queste ruleset R^{man}_{INP}, R^{nat}_{INP}, R^{fil}_{INP}, R^{man}_{OUT}, R^{nat}_{OUT}, R^{fil}_{OUT}, R^{man}_{PRE}, R^{nat}_{PRE}, R^{man}_{FOR}, R^{fil}_{FOR}, R^{man}_{POST} e R^{nat}_{POST}, dove l'apice indica il nome della chain e il pedice il nome della tabella. Si noti che $\rho \setminus \rho_p$ contiene le ruleset definite dall'utente, che entrano in gioco solo per mezzo di istruzioni `CALL(_)` o `GOTO(_)` .

Le ruleset R^{nat}_{INP}, R^{nat}_{OUT}, R^{nat}_{PRE}, e R^{nat}_{POST} contengono come prima regola ($state = \texttt{ENSTABLISHED}, \texttt{CHECK-STATE}(\leftrightarrow)$), questo permette di modellare il comportamento reale di `iptables`, in cui se un pacchetto appartiene a una connessione stabilita questo non passa per le chain della tabella `NAT`, ma subisce invece le trasformazioni previste dallo stato.

La funzione di associazione delle ruleset ai nodi $c_{iptables} \colon Q \mapsto \rho$ è definita come segue:

$$
\begin{array}{lll}
c_{iptables}(q_i) = R_\epsilon & c_{iptables}(q_f) = R_\epsilon & c_{iptables}(q_0) = R^{man}_{\text{PRE}} \\
c_{iptables}(q_1) = R^{nat}_{\text{PRE}} & c_{iptables}(q_4) = R^{man}_{\text{INP}} & c_{iptables}(q_3) = R^{fil}_{\text{FOR}} \\
c_{iptables}(q_5) = R^{nat}_{\text{INP}} & c_{iptables}(q_6) = R^{fil}_{\text{INP}} & c_{iptables}(q_7) = R^{man}_{\text{OUT}} \\
c_{iptables}(q_8) = R^{nat}_{\text{OUT}} & c_{iptables}(q_9) = R^{fil}_{\text{OUT}} & c_{iptables}(q_2) = R^{man}_{\text{FOR}}
\end{array}
$$

$$c_{iptables}(q_3) = R^{fil}_{\text{FOR}} \qquad c_{iptables}(q_{10}) = R^{man}_{\text{POST}} \qquad c_{iptables}(q_{11}) = R^{nat}_{\text{POST}}$$

I target di default delle ruleset sono quelli delle default policy specificati dalla configurazione `iptables`.

2.2.2 Modellazione di pf

Il diagramma di controllo $\mathcal{C}_{pf}$ di `pf`, è mostrato in figura 2.1c. I nodi q_0 e q_2 rappresentano la fase di valutazione delle regole di trasformazione, i rimanenti nodi la fase di filtro.

Differentemente da `iptables`, `pf` non prevede la divisione delle regole in tabelle o chain differente.

Le regole sono valutate nell'ordine in cui compaiono. Quelle di trasformazione sono valutate prima di quelle di filtro e l'algoritmo che stabilisce quale regola applicare è differente per i due tipi di regole. Fra le regole di trasformazione viene applicata la prima regola che verifica la condizione ϕ. Fra le regole di filtro invece viene applicata l'ultima regola fra quelle di cui le condizioni sono verificate, tranne le regole contenenti l'opzione `quick`, che sono applicate immediatamente.

La traduzione di una singola regola `pf` in IFCL non comporta particolari difficoltà in quanto le regole esprimibili in `pf` sono piuttosto semplici (nessun salto o chiamata).

La traduzione delle regole del file di configurazione sono divise nelle seguenti ruleset, che compongono ρ_{pf}:

- nella ruleset R_{dnat} sono inserite le regole `rdr`;
- nella ruleset R_{snat} sono inserite le regole `nat`;
- nella ruleset R_{finp} sono inserite le regole di filtro `quick` che non hanno modificatore `out`;
- nella ruleset R_{finpr} sono inserite, in ordine inverso rispetto a quello del file di configurazione, le regole di filtro non `quick` che non hanno modificatore `out`;
- nella ruleset R_{fout} sono inserite le regole di filtro `quick` che non hanno modificatore `in`;
- nella ruleset R_{foutr} sono inserite, in ordine inverso rispetto a quello del file di configurazione, le regole di filtro non `quick` che non hanno modificatore `in`;

In aggiunta la ruleset R_{dnat} contiene come prima regola ($state =$ `ENSTABLISHED`, `CHECK-STATE`$(\rightarrow)$), R_{snat} contiene come prima regola ($state =$ `ENSTABLISHED`, `CHECK-STATE`$(\leftarrow)$), e le ruleset R_{finp} e R_{fout} contengono sempre come prima regola ($state =$ `ENSTABLISHED`, `ACCEPT`). Questo permette di modellare il comportamento di `pf` rispetto ai pacchetti appartenenti a una connessione stabilita, che sono accettati automaticamente e che subiscono solo le traduzioni prescritte dallo stato.

Le ruleset di filtro sono collegate infine dall'inserimento della regola ($true$, `GOTO`(R_{finpr})) alla fine di R_{fout} e ($true$, `GOTO`(R_{foutr})) alla fine di R_{fout}.

L'assegnamento delle ruleset ai nodi c_{pf} è il seguente:

$$c_{pf}(q_i) = R_\epsilon \qquad c_{pf}(q_0) = R_{dnat} \qquad c_{pf}(q_2) = R_{snat}$$
$$c_{pf}(q_f) = R_\epsilon \qquad c_{pf}(q_1) = R_{finp} \qquad c_{pf}(q_3) = R_{fout}$$

Il target di deafault delle ruleset è sempre `ACCEPT` in `pf`.

2.2.3 Modellazione di ipfw

Il diagramma di controllo $\mathcal{C}_{\texttt{ipfw}}$ di `ipfw`, mostrato in figura 4.1b, è più semplice di quelli analizzati precedentemente. Abbiamo sostanzialmente un nodo per le operazioni effettuate sui pacchetti in input (q_0) ed uno per i pacchetti in output (q_1).

La configurazione di un firewall `ipfw` consiste in una serie di regole etichettate con un numero identificativo che ne indica l'ordine di valutazione. L'idea è quella di costruire una coppia di ruleset per ogni regola della configurazione: una per i pacchetti in input, che sarà "vuota" per regole etichettate con la keyword `out`; una per i pacchetti in output, che sarà "vuota" per regole etichettate con la keyword `in`.

Più precisamente, siano $r_{id_1}, \ldots, r_{id_k}$ le regole del file di configurazione `ipfw`, dove id_i è l'identificativo numerico assegnato a ciascuna delle regole. $\rho_{\texttt{ipfw}}$ contiene $2k$ differenti ruleset: k ruleset denominate R_i^I, una per ogni regola r_{id_i} e k denominate R_i^O, sempre una per ogni regola r_{id_i}. Se la regola r_{id_i} contiene la keyword `out`, allora sarà $R_i^I = [(true, \texttt{GOTO}(R_{i+1}^I))]$. Altrimenti, definiamo $R_i^I = (trs(r_{id_i}), (true, \texttt{GOTO}(R_{i+1}^I)))$, dove trs è definita come:

$$trs(r) = \begin{cases} (trs'(\phi), \texttt{GOTO}(R_n^I)) & \text{se } r \text{ è } \texttt{skipto } id_n\ \phi \\ (trs'(\phi), \texttt{CALL}(R_n^I)) & \text{se } r \text{ è } \texttt{call } id_n\ \phi \\ (trs'(\phi), trs'(\texttt{t})) & \text{se } r \text{ è } \texttt{t}\ \phi \end{cases}$$

In cui la funzione trs' realizza una semplice traduzione sintattica per target e condizioni.

La costruzione delle rulesets R_i^O è identica, ma in questo caso sono le regole etichettate con `in` a essere tradotte in un ruleset "vuote".

La funzione di assegnamento delle ruleset ai nodi c_{ipfw} è definita come segue:

$$c_{ipfw}(q_i) = R_\epsilon \qquad c_{ipfw}(q_0) = R_1^I$$
$$c_{ipfw}(q_1) = R_1^O \qquad c_{ipfw}(q_f) = R_\epsilon$$

`ipfw` permette di definire una policy di default, questa determinerà il target di default di tutte le ruleset.

2.3 Semantica operazionale di IFCL

Esprimiamo la semantica di un firewall IFCL attraverso due sistemi di transizione operanti in un modalità master-slave. Il sistema master definisce una relazione della forma $s \xrightarrow{p,p'} s'$, intuitivamente significa che un firewall nello stato s che riceve un pacchetto p lo trasforma in p' e modifica il suo stato interno in s'.

Le configurazioni del sistema slave sono triple (q, s, p) dove: (i) $q \in Q$ è un nodo del diagramma di controllo; (ii) s è lo stato del firewall; (iii) p è il pacchetto. Una transizione $(q, s, p) \to (q', s, p')$ descrive come il nodo q del firewall nello stato s gestisce un pacchetto p trasformandolo in p', e passandolo al nodo q' per la prosecuzione della valutazione. Lo stato non cambia in questo sistema in quanto viene aggiornato solo alla fine della valutazione del pacchetto, e solo se non viene scartato.

Usiamo il predicato $p, s \models_R^S (t, p')$ che definisce il risultato della valutazione (pacchetto risultante p' e destino associato $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$) di un pacchetto p da parte di una ruleset R nello stato s con stack delle chiamate S. Il predicato cerca nella ruleset R una regola che si abbini al pacchetto p attraverso $p, s \Vdash_R (t, i)$. Se la trova, il target t è applicato a p per ottenere il nuovo pacchetto p' (potenzialmente identico a p).

$$(1)\ \frac{p, s \Vdash_R (t, i) \quad t \in \{\texttt{ACCEPT}, \texttt{DROP}\}}{p, s \models^S_R (t, p)} \qquad (2)\ \frac{p, s \Vdash_R (\texttt{CHECK-STATE}(X), i) \quad p \vdash_s \alpha \quad p' = \mathit{establ}(\alpha, X, p)}{p, s \models^S_R (\texttt{ACCEPT}, p')}$$

$$(3)\ \frac{p, s \Vdash_R (\texttt{CHECK-STATE}(X), i) \quad p \nvdash_s \quad p, s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')} \qquad (4)\ \frac{p, s \Vdash_R (\texttt{NAT}(d_n, s_n), i)}{p, s \models^S_R (\texttt{ACCEPT}, \mathit{nat}(p, d_n, s_n))}$$

$$(5)\ \frac{p, s \Vdash_R (\texttt{GOTO(R')}, i) \quad R' \notin S \quad p, s \models^{\overline{R}.S}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (6)\ \frac{p, s \Vdash_R (\texttt{GOTO(R')}, i) \quad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)}$$

$$(7)\ \frac{p, s \Vdash_R (\texttt{CALL(R')}, i) \quad R' \notin S \quad p, s \models^{R_{i+1}.S}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (8)\ \frac{p, s \Vdash_R (\texttt{CALL(R')}, i) \quad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)}$$

$$(9)\ \frac{p, s \Vdash_R (\texttt{RETURN}, i) \quad \mathit{pop}^*(S) = (R', S') \quad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (10)\ \frac{p, s \Vdash_R (\texttt{RETURN}, i) \quad \mathit{pop}^*(S) = \dagger}{p, s \models^S_R (t_d, p)}$$

$$(11)\ \frac{p, s \nVdash_R \quad S \neq \epsilon \quad \mathit{pop}^*(S) = (R', S') \quad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (12)\ \frac{p, s \nVdash_R \quad (S = \epsilon \ \vee \ \mathit{pop}^*(S) = \dagger)}{p, s \models^S_R (t_d, p)}$$

$$(13)\ \frac{p, s \Vdash_R (\texttt{MARK}(m), i) \qquad p[\mathit{tag} \mapsto m], s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')}$$

Tabella 2.2: Predicato $p, s \models^S_R (t, p')$.

Lo stack S serve a gestire le azioni CALL(_), RETURN e GOTO(_), e supplisce a una duplice funzione: controllare che non ci siano chiamate cicliche e tenere traccia delle ruleset chiamanti nel caso si incontri un'azione RETURN. Si noti che nel momento di dover tornare alla ruleset chiamante è necessario ignorare le ruleset che sono state inserite in S da un GOTO(_), limitandosi a quelle inserite attraverso una CALL(_). Distinguiamo dunque le ruleset inserite nella pila da un GOTO(_) sopralineandole: le ruleset sopralineate sono ignorate quando si ritorna da una chiamata.

Chiamiamo ϵ la pila vuota e indichiamo con $\cdot$ la concatenazione di elementi sullo stack. Definiamo la funzione $\mathit{pop}^\star$ per trattare il ritorno da una ruleset invocata († segnala che non c'è nessuna ruleset a cui tornare)

$$\mathit{pop}^*(\epsilon) = \dagger \qquad \mathit{pop}^*(R \cdot S) = (R, S) \qquad \mathit{pop}^*(\overline{R} \cdot S) = \mathit{pop}^*(S)$$

Nella definizione di $p, s \models^S_R (t, p')$ usiamo la notazione R_k per indicare la ruleset $[r_k, ..., r_n]$ ($k \in [1, n]$) risultante dalla rimozione delle prime $k-1$ regole dalla ruleset $R = [r_1, ..., r_n]$.

In linea con l'idea di rimanere parametrici rispetto allo stato interno del firewall, assumiamo che la funzione $\mathit{establ}(\alpha, X, p)$, data la trasformazione α prescritta dallo stato, un pacchetto p e i campi $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ da modificare, restituisca un pacchetto possibilmente modificato p', ad esempio nel caso questo appartenga a una connessione stabilita.

Assumiamo infine di avere una funzione $\mathit{nat}(p, d_n, s_n)$ che restituisce il pacchetto p modificato secondo l'operazione NAT. L'argomento d_n specifica la modifica alla destinazione di p, cioè il destination NAT (DNAT), mentre s_n specifica la modifica all'origine di p, cioè il source NAT (SNAT).

La tabella 2.2 mostra le regole che definiscono $p, s \models^S_R (t, p')$. La prima regola gestisce il caso in cui il pacchetto p si abbini a una regola con target ACCEPT o DROP; in questo caso la valutazione della ruleset termina e vengono restituiti il pacchetto non modificato e il target. Quando un pacchetto p si abbina a una regola con target CHECK-STATE, controlliamo lo stato s: se p appartiene alle connessioni stabilite, restituiamo ACCEPT e un pacchetto p' ottenuto dalla riscrittura di p. Se p non appartiene a nessuna connessione stabilita allora proseguiamo con la valutazione della ruleset. Quando un pacchetto p si abbina a una regola con target NAT, restituiamo ACCEPT e il pacchetto risultante ottenuto invocando la funzione nat. Ci sono due casi corrispondenti alla situazione in cui p è associato a una regola con target GOTO(_). Se la ruleset R' non è ancora nello stack inseriamo la ruleset corrente R nello stack,

sovralineandola in modo da ignorarla per il ritorno. Altrimenti, se R' è già nello stack, abbiamo trovato un loop e quindi scartiamo p assegnandogli il target DROP. Il caso di un pacchetto p associato a una regola con target CALL(_) è simile, a parte per il fatto che la ruleset inserita nello stack non viene sovralineata. Quando un pacchetto p è associato a una regola con target RETURN, chiamiamo la funzione pop^* sullo stack e confrontiamo il pacchetto con la ruleset ottenuta. Infine, quando nessuna regola si associa al pacchetto, un implicito RETURN occorre: continuiamo dal top dello stack, se esiste. Una regola con target MARK semplicemente modifica il tag del pacchetto a cui è associata. Se nessuna delle regole è applicabile allora restituiamo l'azione di default t_d della ruleset corrente.

Possiamo ora definire la funzione di transizione slave come segue:

$$\frac{c(q) = R \quad p, s \models^{\epsilon}_{R} (\text{ACCEPT}, p') \quad \delta(q, p') = q'}{(q, s, p) \to (q', s, p')}$$

La regola descrive come viene trattato il pacchetto p quando il firewall è nella fase di elaborazione rappresentata dal nodo q e lo stato è s. Confrontiamo p con la ruleset R associata a q e se p è accettato come p', continuiamo considerando la prossima fase di elaborazione rappresentata dalla fase q'.

Il sistema di transizione master è basato sulla chiusura transitiva della relazione $\to$ del sistema slave. Tuttavia è necessario verificare che la sequenza dei nodi attraversati non contenga dei nodi ripetuti, cioè dei cicli, in quanto in questo caso il sistema run time del firewall scarterebbe immediatamente il pacchetto, in modo coerente col comportamento dei firewall supportati. Per questo definiamo $\to^{\oplus}$, una versione priva di cicli della chiusura transitiva della relazione $\to$.

$$\frac{(q_i, s, p) \to^{+}_{\{q_i\}} (q, s, p')}{(q_i, s, p) \to^{\oplus} (q, s, p')}$$

Dove la relazione $\to^{+}_{I}$ è una versione della chiusura transitiva che tiene conto dell'insieme dei nodi visitati I al fine di prevenire i cicli.

$$\frac{(q, s, p) \to (q', s, p') \quad q' \notin I}{(q, s, p) \to^{+}_{I} (q', s, p')} \qquad \frac{(q, s, p) \to (q'', s, p'') \quad q'' \notin I \quad (q'', s, p'') \to^{+}_{I \cup \{q''\}} (q', s, p')}{(q, s, p) \to^{+}_{I} (q', s, p')}$$

Infine definiamo il sistema di transizione master, che trasforma stati e pacchetti come segue:

$$\frac{(q_i, s, p) \to^{\oplus} (q_f, s, p')}{s \xrightarrow{p, p'} s \uplus (p, p')}$$

Questa regola dice che quando un firewall è nello stato s e riceve un pacchetto p, elabora p a partire dal nodo iniziale q_i del suo diagramma di controllo. Se l'elaborazione ha successo, cioè p raggiunge il nodo q_f da cui viene accettato come p', allora il sistema di transizione master accetta p come p' e aggiorna lo stato s inserendo informazioni riguardanti p, la sua trasformazione in p' e l'eventuale connessione a cui appartiene, attraverso la funzione $\uplus$, lasciata non specificata per generalità.

2.4 Normalizzazione

Il linguaggio di configurazione intermedio IFCL permette di gestire il flusso di controllo attraverso i target GOTO(_), CALL(_) e RETURN. Questa proprietà del linguaggio permette di modellare fedelmente i sistemi reali come `iptables` e `ipfw`, tuttavia rende difficile analizzare formalmente le configurazioni per derivare una rappresentazione sintetica della loro semantica. Mostriamo come questi target possono essere eliminati automaticamente dalla configurazione di un firewall, mantenendo inalterata la semantica.

Chiamiamo normalizzata una ruleset in cui non siano presenti target per modificano il flusso di controllo. Presentiamo un'operazione di normalizzazione $(\!|\ \_\ |\!)$ che data una ruleset ne produce una normalizzata equivalente.

Nel seguito denotiamo con $r;R$ una ruleset non vuota composta dalla regola r seguita dalla ruleset (potenzialmente vuota) R; e indichiamo con $R_1@R_2$ la concatenazione di R_1 e R_2.

La normalizzazione di una ruleset R è definita come segue:

$$
\begin{aligned}
(\!|\ R\ |\!) &= (\!|\ R\ |\!)^{true}_{\{R\}} \\
(\!|\ \epsilon\ |\!)^f_I &= \epsilon \\
(\!|\ (\phi,t);R\ |\!)^f_I &= (f\wedge\phi,t);(\!|\ R\ |\!)^f_I \quad \text{se } t \notin \{\texttt{GOTO(R')}, \texttt{CALL(R')}, \texttt{RETURN}\} \\
(\!|\ (\phi,\texttt{RETURN});R\ |\!)^f_I &= (\!|\ R\ |\!)^{f\wedge\neg\phi}_I \\
(\!|\ (\phi,\texttt{CALL(R')});R\ |\!)^f_I &= \begin{cases} (\!|\ R'\ |\!)^{f\wedge\phi}_{I\cup\{R'\}} @ (\!|\ R\ |\!)^f_I & \text{if } R' \notin I \\ (f\wedge\phi,\texttt{DROP});(\!|\ R\ |\!)^f_I & \text{altrimenti} \end{cases} \\
(\!|\ (\phi,\texttt{GOTO(R')});R\ |\!)^f_I &= \begin{cases} (\!|\ R'\ |\!)^{f\wedge\phi}_{I\cup\{R'\}} @ (\!|\ R\ |\!)^{f\wedge\neg\phi}_I & \text{se } R' \notin I \\ (f\wedge\phi,\texttt{DROP});(\!|\ R\ |\!)^{f\wedge\neg\phi}_I & \text{altrimenti} \end{cases}
\end{aligned}
$$

La procedura ausiliaria $(\!|\ R\ |\!)^f_I$ ispeziona ricorsivamente la ruleset R. La formula f accumula congiunzioni di predicati ϕ e rappresenta la condizione sufficiente e necessaria perché la valutazione di un pacchetto da parte della ruleset raggiunga la posizione attualmente in esame della ruleset; l'insieme I tiene traccia delle ruleset attraversate dalla procedura e serve a riconoscere i loop. Se una regola non modifica il flusso di controllo, allora modifichiamo solo la condizione, sostituendo ϕ con $f\wedge\phi$, e continuiamo ad analizzare il resto della ruleset. La sostituzione della condizione corrisponde intuitivamente a specificare che nella ruleset normalizzata una regola deve essere applicata al pacchetto p se e solo se (i) nella ruleset originaria il pacchetto p avrebbe visitato la regola e (ii) il pacchetto p verifica la condizione originaria della regola.

Nel caso di una regola di return $(\phi,\texttt{RETURN})$ non generiamo nessuna nuova regola, ma continuiamo ad analizzare il resto della ruleset aggiornando f con la negazione di ϕ.

Per la regola $(\phi,\texttt{CALL(R')})$ abbiamo due casi, distinti in base all'esito del sanity check che verifica se siamo in un loop o meno. Se la ruleset chiamata R' non è in I, e quindi il sanity check è verificato, allora sostituiamo la regola con la normalizzazione di R', con $f\wedge\phi$ come predicato e aggiungendo R' all'insieme delle ruleset visitate. Se R' invece è già in I, e quindi il sanity check non è verificato, allora abbiamo un loop e quindi rimpiazziamo la regola con una avente DROP come target e $f\wedge\phi$ come condizione. In entrambi i casi continuiamo la normalizzazione del resto della ruleset.

La regola $(\phi,\texttt{GOTO(R')})$ è trattata in maniera simile alla precedente, tranne che nella normalizzazione del resto della ruleset abbiamo $f\wedge\neg\phi$ come predicato, in quanto non si torna indietro dalle ruleset chiamate attraverso i target GOTO(_), condizione necessaria perché un pacchetto visiti le regole che seguono il GOTO(_) è che questo non verifichi la condizione ϕ.

Chiamiamo firewall normalizzato un firewall in cui tutte le ruleset in ρ sono normalizzate. Un firewall normalizzato è ottenuto dalla riscrittura attraverso l'operazione di unfolding delle ruleset associate ai nodi del diagramma di controllo. Si noti che non essendo possibile invocare o chiamare ruleset le uniche ruleset in ρ, per un firewall normalizzato, sono quelle associate ai nodi.

Formalmente:

Definizione 8 (Normalizzazione di un firewall)**.** *Dato un firewall $\mathcal{F}=(\mathcal{C},\Sigma)$, con $\Sigma=(\rho,c)$, la sua versione normalizzata $(\!|\ \mathcal{F}\ |\!)$ è $(\mathcal{C},\Sigma')$ con $\Sigma'=(\rho',c')$ dove $\forall q\in\mathcal{C}.\, c'(q)=(\!|\ c(q)\ |\!)$ e $\rho'=\{(\!|\ c(q)\ |\!) \mid q\in\mathcal{C}\}$.*

$$
\begin{aligned}
P_\epsilon(p,\tilde{p},\mathtt{s}) &= dp(R) \wedge p = \tilde{p} && \\
P_{r;R}(p,\tilde{p},\mathtt{s}) &= (\phi(p,\mathtt{s}) \wedge p = \tilde{p}) \vee (\neg\phi(p,\mathtt{s}) \wedge P_R(p,\tilde{p},\mathtt{s})) && \text{se } r = (\phi, \texttt{ACCEPT}) \\
P_{r;R}(p,\tilde{p},\mathtt{s}) &= \neg\phi(p,\mathtt{s}) \wedge P_R(p,\tilde{p},\mathtt{s}) && \text{se } r = (\phi, \texttt{DROP}) \\
P_{r;R}(p,\tilde{p},\mathtt{s}) &= (\phi(p,\mathtt{s}) \wedge \tilde{p} \in tr(p,d_n,s_n,\leftrightarrow)) \vee (\neg\phi(p,\mathtt{s}) \wedge P_R(p,\tilde{p},\mathtt{s})) && \text{se } r = (\phi, \texttt{NAT}(d_n,s_n)) \\
P_{r;R}(p,\tilde{p},\mathtt{s}) &= (\phi(p,\mathtt{s}) \wedge \tilde{p} \in tr(p,*{:}*,*{:}*,X)) \vee (\neg\phi(p,\mathtt{s}) \wedge P_R(p,\tilde{p},\mathtt{s})) && \text{se } r = (\phi, \texttt{CHECK-STATE}(X)) \\
P_{r;R}(p,\tilde{p},\mathtt{s}) &= (\phi(p,\mathtt{s}) \wedge P_R(p[tag \mapsto m],\tilde{p},\mathtt{s})) \vee (\neg\phi(p,\mathtt{s}) \wedge P_R(p,\tilde{p},\mathtt{s})) && \text{se } r = (\phi, \texttt{MARK}(m))
\end{aligned}
$$

Tabella 2.3: Traduzione delle ruleset in predicati logici.

Vale il seguente teorema che garantisce che ogni firewall è semanticamente equivalente alla sua versione normalizzata:

Teorema 1 (Correttezza della normalizzazione)**.** *Sia $\mathcal{F}$ un firewall e $(\!(\ \mathcal{F}\ )\!)$ la sua versione normalizzata. Chiamiamo $s \xrightarrow{p,p'}_X s'$ un passo del sistema di transizione master del firewall $X \in \{\mathcal{F}, (\!(\ \mathcal{F}\ )\!)\}$. Vale che*

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{(\!(\ \mathcal{F}\ )\!)} s'.$$

2.5 Caratterizzazione dichiarativa

Mostriamo come costruire un predicato logico che caratterizza il comportamento di un firewall normalizzato: quali pacchetti accetta e con quali trasformazioni in quale stato.

Per gestire il NAT, definiamo una funzione ausiliaria $tr(p,d_n,s_n,X)$ che calcola il pacchetto risultante dall'applicazione della trasformazione definita dagli indirizzi d_n ed s_n sul pacchetto in input p. Il parametro $X \in \{\leftarrow,\rightarrow,\leftrightarrow\}$ specifica se la trasformazione si applica all'origine, al destinatario o a entrambi gli indirizzi, similmente a quanto avviene per $\texttt{CHECK-STATE}(X)$.

$$
\begin{aligned}
tr(p,d_n,s_n,\leftrightarrow) &\triangleq \{p[da \mapsto a_d, sa \mapsto a_s] \mid a_d \in d_n, a_s \in s_n\} \\
tr(p,d_n,s_n,\rightarrow) &\triangleq \{p[da \mapsto a_d] \mid a_d \in d_n\} \\
tr(p,d_n,s_n,\leftarrow) &\triangleq \{p[sa \mapsto a_s] \mid a_s \in s_n\}
\end{aligned}
$$

Modelliamo la policy di default di una ruleset R con un predicato dp, vero se la policy è ACCEPT, falso altrimenti.

Data una ruleset normalizzata R, costruiamo il predicato $P_R(p,\tilde{p},\mathtt{s})$ che vale se e solo se il pacchetto p è accettato come $\tilde{p}$ da R nello stato s. La sua definizione Tabella 2.3 è data induttivamente sulle regole della ruleset R.

Facciamo una piccola digressione riguardo a come vogliamo studiare il comportamento del firewall in funzione dello stato s. In teoria il corretto modo di esprimere attraverso un predicato il comportamento di una regola con target $\texttt{CHECK-STATE}(X)$ sarebbe il seguente:

$$P_{(\phi,\texttt{CHECK-STATE}(X));R}(p,\tilde{p},s) = (\phi(p,s) \wedge p \vdash_s \alpha \wedge \tilde{p} = establ(\alpha,X,p)) \vee ((\neg\phi(p,s) \vee p \nvdash_s) \wedge P_R(p,\tilde{p},s))$$

Questo corrisponde esattamente al comportamento dinamico del firewall in cui dato un pacchetto p e uno stato s, un solo pacchetto risultante è prodotto nel caso in cui p appartenga a una connessione stabilita. Innanzitutto possiamo omettere $p \nvdash_s$ in quanto abbiamo assunto che $\phi(p,s) \Rightarrow \exists\alpha\ .\ p \vdash_s \alpha$. Successivamente, dato che l'obiettivo è quello di rappresentare il comportamento di un firewall in

modo sintetico, osserviamo che non possiamo analizzare uno per uno tutti i possibili stati del firewall. Per questo astraiamo dal comportamento esatto dello stato, cioè del predicato $p \vdash_s \alpha$ e dalla funzione $establ(\alpha, X, p)$, e consideriamo solo due possibilità per un pacchetto in uno stato s: il pacchetto non appartiene a nessuna connessione stabilita e quindi il predicato ϕ non viene verificato e l'azione CHECK-STATE(X) non viene mai applicata, oppure il pacchetto appartiene a una connessione stabilita e in questo caso approssimiamo tutti i possibili esiti delle trasformazioni prescritte dallo stato in un unico caso in cui la trasformazione produce non deterministicamente ogni possibile pacchetto ottenibile dalla modifica dei campi specificati da X, esprimiamo questo per mezzo della funzione $tr(p, *{:}*, *{:}*, X)$.

Dato che scegliamo di approssimare in questo modo il comportamento del firewall rispetto allo stato, possiamo considerare una versione approssimata dello stato che si limita ad assegnare a ogni possibile pacchetto un'etichetta che indica se questo appartiene o meno a una connessione stabilita $\mathtt{s} : \mathbb{P} \to \{\mathtt{ENSTABLISHED}, \mathtt{NEW}\}$. Notiamo che il fatto che un pacchetto p appartenga o meno a una connessione stabilita non influenza in alcun modo il comportamento del firewall rispetto a un secondo pacchetto $p' \neq p$, pertanto non è necessario studiare il comportamento del firewall per ogni possibile stato approssimato, ma è sufficiente ridursi alle due funzioni costanti $\mathtt{s}_{\mathtt{ENSTABLISHED}}$, stato nel quale tutti i pacchetti appartengono a una connessione stabilita, e $\mathtt{s}_{\mathtt{NEW}}$, stato nel quale nessun pacchetto appartiene a una connessione stabilita.

Definiamo formalmente il concetto di stato approssimato del firewall come segue

Definizione 9 (Stato approssimato di un firewall)**.** *Dato uno stato di firewall $s \in S$ definiamo la sua versione approssimata $\hat{s} : \mathbb{P} \to \{\mathit{ENSTABLISHED}, \mathit{NEW}\}$ come:*

$$\hat{s}(p) = \begin{cases} \mathit{ENSTABLISHED} & \text{se } \exists \alpha \ . \ p \vdash_s \alpha \\ \mathit{NEW} & \text{altrimenti} \end{cases}$$

Non abbiamo descritto formalmente come vengono valutate le condizioni ϕ delle regole, ma abbiamo assunto che una condizione del tipo $state = \mathtt{ENSTABLISHED}$ venga valutata correttamente su un pacchetto p dato lo stato attuale del firewall s; estendiamo la stessa ipotesi per quanto riguarda lo stato approssimato $\mathtt{s}$ nel predicato Tabella 2.3.

Commentiamo brevemente la costruzione del predicato $P_R(p, \tilde{p}, \mathtt{s})$, definita ricorsivamente su R, per casi sulla prima regola. Intuitivamente, la ruleset vuota non trasforma il pacchetto e lo accetta solo se lo prevede la policy di default, quindi solo coppie in cui vale $p = \tilde{p}$ possono verificare il predicato e solo se vale $dp(R)$. Se la ruleset inizia con la regola (ϕ, ACCEPT), seguita da R, consideriamo due casi: quando $\phi(p, \mathtt{s})$ vale e quindi il pacchetto viene accettato com'è; e quando invece vale $\neg\phi(p, \mathtt{s})$, e p è accettato come $\tilde{p}$ solo se il resto della ruleset R lo accetta. La ruleset che inizia con la regola (ϕ, DROP) accetta p solo se è la continuazione a farlo e se non vale $\phi(p, \mathtt{s})$. La ruleset che inizia con la regola $(\phi, \text{NAT}(d_n, s_n))$ è trattata come quella che inizia con (ϕ, ACCEPT): la differenza è che quando $\phi(p, \mathtt{s})$ vale, il pacchetto è accettato come $\tilde{p} = tr(p, d_n, s_n, \leftrightarrow)$, che è il risultato dell'applicazione delle trasformazioni NAT al pacchetto p. La ruleset che inizia con la regola $(\phi, \text{CHECK-STATE}(X))$ si comporta in modo simile a NAT applicando la trasformazione al pacchetto, nei campi X (scritto come $tr(p, *{:}*, *{:}*, X)$). Questa trasformazione dovrebbe avvenire solo se il pacchetto appartiene a una connessione stabilita. Per la verifica di questa condizione ci affidiamo al predicato ϕ che, come abbiamo assunto, nel caso di regola con target CHECK-STATE(X) deve comportare che il pacchetto appartenga a una connessione stabilita $\phi(p, s) \Rightarrow \exists \alpha \ . \ p \vdash_s \alpha$. Infine, la regola $(\phi, \text{MARK}(m))$, se $\phi(p, \mathtt{s})$ vale, trasforma il pacchetto in un nuovo $p' = p[tag \mapsto m]$, come in una NAT, ma il pacchetto non viene accettato una volta modificato, quindi p, p' non è una automaticamente una soluzione del predicato, ma la continuazione verrà valutata usando p' come pacchetto in arrivo.

Il predicato Tabella 2.3 è semanticamente corretto: se un pacchetto p è accettato da una ruleset R, nello stato s come p', allora $P_R^{\hat{s}}(p, p')$ vale, e vice versa.

Lemma 1. *Data una ruleset R abbiamo che*

1. $\forall p, s.\ p, s \models_R^{\epsilon} (\textit{ACCEPT}, p') \implies P_R(p, p', \hat{s})$;
2. $\forall p, p', \mathtt{s}\ .\ P_R(p, p', \mathtt{s}) \implies \exists s \in S\ .\ \hat{s} = \mathtt{s}\ \wedge\ p, s \models_R^{\epsilon} (\textit{ACCEPT}, p')$

Definiamo il predicato associato a un intero firewall come segue.

Definizione 10. *Sia $\mathcal{F} = (\mathcal{C}, \Sigma)$ un firewall con diagramma di controllo $\mathcal{C} = (Q, A, q_i, q_f)$ e configurazione $\Sigma = (\rho, c)$. Il predicato associato a $\mathcal{F}$ è definito come*

$$\mathcal{P}_{\mathcal{F}}(p, \tilde{p}, \mathtt{s}) \triangleq \mathcal{P}_{q_i}^{\emptyset}(p, \tilde{p}, \mathtt{s}) \quad \textit{where}$$

$$\mathcal{P}_{q_f}^{I}(p, \tilde{p}, \mathtt{s}) \triangleq p = \tilde{p} \qquad \mathcal{P}_{q}^{I}(p, \tilde{p}, \mathtt{s}) \triangleq \exists p'.P_{c(q)}(p, p', \mathtt{s}) \wedge \left(\bigvee_{\substack{(q,\psi,q') \in A \\ q' \notin I}} \psi(p') \wedge \mathcal{P}_{q'}^{I \cup \{q\}}(p', \tilde{p}, \mathtt{s}) \right)$$

per ogni $q \in Q$ tale che $q \neq q_f$, a dove $P_{c(q)}$ è il predicato costruito a partire dalla ruleset $c(q)$, cioè quella associata al nodo q del diagramma di controllo.

Intuitivamente, del nodo finale q_f accettiamo p così com'è. In tutti gli altri nodi q, p è accettato come $\tilde{p}$ se e solo se viene accettato come un qualche pacchetto intermedio p' dalla ruleset associata al nodo q e se questo pacchetto intermedio viene accettato dal nodo successivo nel percorso di p, tutto per un firewall nello stato s. Più precisamente, cerchiamo un pacchetto intermedio p', per il quale valga che (i) p è accettato come p' dalla ruleset associata la nodo q; (ii) p' verifica uno dei predicati ψ degli archi del diagramma di controllo uscenti da q; e (iii) p' è accettato come p' dal nodo raggiunto q'. Vogliamo chiaramente ignorare i percorsi contenenti loop, sia per essere aderenti alla semantica operazionale data, sia perché questo è il comportamento reale dei firewall. Perciò il predicato usa un insieme I nel quale tenere traccia dei nodi già attraversati ed esiste una quarta condizione che deve essere verificata da p': (iv) il nodo successivo che valuterà il pacchetto non deve appartenere già I, cioè $q' \notin I$.

Concludiamo questa sezione stabilendo una connessione fra la caratterizzazione logica di un firewall e la sua semantica operazionale.

Teorema 2 (Correttezza della caratterizzazione logica)**.** *Dato un firewall $\mathcal{F}$ ed il suo predicato corrispondente $\mathcal{P}_{\mathcal{F}}$ abbiamo che*

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p', \hat{s})$
2. $\forall p, p', \mathtt{s}.\ \mathcal{P}_{\mathcal{F}}(p, p', \mathtt{s}) \implies \exists s \in S\ .\ \hat{s} = \mathtt{s}\ \wedge\ s \xrightarrow{p,p'} s \uplus (p, p')$

Capitolo 3

Caratterizzazione funzionale

In questo capitolo presentiamo una caratterizzazione funzionale della semantica di IFCL. Come per la caratterizzazione dichiarativa dell'articolo [4], presentata precedentemente, ci interessa trattare solo i firewall normalizzati, senza istruzioni `CALL(_)`, `RETURN` o `GOTO(_)`. Definiamo per prima cosa la semantica di una ruleset e poi per composizione quella di un firewall. Introduciamo il concetto di trasformazione su un pacchetto e presentiamo una seconda caratterizzazione funzionale basata su questo concetto. Questa seconda versione della semantica funzionale è equivalente alla prima, ma risulta più comoda per l'implementazione della fase di sintesi della pipeline di transcompilazione.

A differenza della caratterizzazione logica presentata precedentemente, quella funzionale non si basa su un'approssimazione dello stato, alla fine del capitolo discuteremo le conseguenze di questa scelta.

3.1 Semantica denotazionale

Definiamo la semantica di una ruleset R e di un firewall IFCL $\mathcal{F}$ come una funzione che dato uno stato interno restituisce una funzione che associa ogni pacchetto al pacchetto nel quale viene trasformato quando viene accettato o $\perp$ se il pacchetto viene scartato.

La semantica di una ruleset R è definita da:

$$[\![R]\!] : S \to \mathbb{P} \to \mathbb{P} \cup \{\perp\}$$
$$[\![R]\!](s) = [\![R]\!]_s : \mathbb{P} \to \mathbb{P} \cup \{\perp\}$$

$$[\![\epsilon]\!]_s\ (p) = \begin{cases} p & \text{se } dp = \texttt{ACCEPT} \\ \perp & \text{altrimenti} \end{cases}$$

$$[\![(\phi, \texttt{ACCEPT}); R]\!]_s\ (p) = \begin{cases} p & \text{se } \phi(p, s) \\ [\![R]\!]_s\ (p) & \text{altrimenti} \end{cases}$$

$$[\![(\phi, \texttt{DROP}); R]\!]_s\ (p) = \begin{cases} \perp & \text{se } \phi(p, s) \\ [\![R]\!]_s\ (p) & \text{altrimenti} \end{cases}$$

$$[\![(\phi, \texttt{NAT}(d_n, s_n)); R]\!]_s\ (p) = \begin{cases} nat(p, d_n, s_n) & \text{se } \phi(p, s) \\ [\![R]\!]_s\ (p) & \text{altrimenti} \end{cases}$$

$$[\![(\phi, \texttt{CHECK-STATE}(X)); R]\!]_s\ (p) = \begin{cases} estabil(\alpha, X, p) & \text{se } \phi(p, s) \wedge p \vdash_s \alpha \\ [\![R]\!]_s\ (p) & \text{altrimenti} \end{cases}$$

$$[\![(\phi, \texttt{MARK}(m)); R]\!]_s\ (p) = \begin{cases} [\![R]\!]_s\ (p[tag \mapsto m]) & \text{se } \phi(p,s) \\ [\![R]\!]_s\ (p) & \text{altrimenti} \end{cases}$$

Dove ricordiamo che il predicato $p \vdash_s \alpha$ è vero se il pacchetto p appartiene ad una connessione stabilita secondo lo stato s del firewall che prescrive per p l'azione α, e la funzione $establ(\alpha, X, p)$ effettua la trasformazione α sul pacchetto p limitatamente ai campi specificati da $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$.

La funzione $[\![R]\!]$, dato uno stato s, è definita attraverso la funzione ricorsiva $[\![R]\!]_s$, che a sua volta è definita per casi sulla forma della prima regola di R. Se R è vuota allora ogni pacchetto viene mappato in se stesso se la policy di default della ruleset dp è ACCEPT, in $\perp$ se la policy di default è DROP. Se R comincia con una regola con target ACCEPT e condizione ϕ allora ogni pacchetto che verifica la condizione ϕ viene mappato in se stesso, ogni altro pacchetto viene trattato secondo la funzione associata al resto della ruleset. Se la prima regola della ruleset R ha target DROP o $\texttt{NAT}(d_n, s_n)$ allora ci si comporta come nel caso precedente, ma i pacchetti che verificano la condizione sono mappati rispettivamente in $\perp$ o in un p' ottenuto a partire da p modificano i campi secondo le specifiche di d_n e s_n attraverso la funzione *nat* introdotta nel capitolo 2. Il caso in cui R cominci con una regola con target $\texttt{CHECK-STATE}(X)$ è piuttosto simile ai precedenti, dove però non è sufficiente che la condizione ϕ sia verificata, serve anche che il pacchetto appartenga ad una connessione stabilita, cioè $p \vdash_s \alpha$, e la trasformazione applicata è α, quella prescritta dallo stato, e solo nei campi specificati da X. L'ultimo caso, quello in cui la prima regola della ruleset ha target $\texttt{MARK}(m)$ è particolare in quanto la valutazione viene comunque demandata al resto della ruleset, con un pacchetto eventualmente modificato nel campo *tag* se la condizione della regola è verificata.

La semantica appena definita per le ruleset è corretta rispetto alla semantica operazionale definita attraverso la relazione $\models$ nel capitolo 2.

Lemma 2. *Sia R una ruleset normalizzata IFCL, abbiamo che*

1. $\forall p, p', s.\ (\ p, s \models^{\epsilon}_R (\textit{ACCEPT}, p') \iff [\![R]\!](s)(p) = p'\ )$
2. $\forall p, s.\ (\ [\![R]\!](s)(p) = \perp \iff \exists p''.\ p, s \models^{\epsilon}_R (\textit{DROP}, p')\ )$

La semantica di un firewall $\mathcal{F}$ in uno stato s è definita da

$$[\![\mathcal{F}]\!] : S \to \mathbb{P} \to \mathbb{P} \cup \{\perp\}$$
$$[\![\mathcal{F}]\!](s) = [\![\mathcal{F}]\!]_s : \mathbb{P} \to \mathbb{P} \cup \{\perp\}$$

$$[\![\mathcal{F}]\!]_s = [\![q_i]\!]^{\mathcal{F},\emptyset}_s$$

Dove la funzione $[\![q]\!]^{\mathcal{F},I}_s : \mathbb{P} \to \mathbb{P} \cup \{\perp\}$, per un firewall $\mathcal{F}$, uno stato s e un insieme di ruleset I è definita per ogni $q \in Q,\ q \neq q_f$ come:

$$[\![q]\!]^{\mathcal{F},\mathcal{I}}_s\ (p) = \begin{cases} [\![q']\!]^{\mathcal{F},I\cup\{q\}}_s\ (p') & \text{se } p' \neq \perp \wedge q' \notin I \\ \perp & \text{altrimenti} \end{cases}$$
$$\text{dove } p' = [\![c(q)]\!]_s\ (p) \text{ e se } p' \neq \perp \text{ allora } q' = \delta(q, p')$$

e per il nodo finale come:

$$[\![q_f]\!]^{\mathcal{F},\mathcal{I}}_s\ (p) = p$$

La semantica di un firewall $[\![\mathcal{F}]\!]$, dato uno stato s è definita dalla funzione $[\![q]\!]_s^{\mathcal{F},I}$ in cui q è il nodo del diagramma di controllo da cui parte la valutazione del pacchetto, I è l'insieme dei nodi visitati e viene usato per individuare i loop. La funzione $[\![q]\!]_s^{\mathcal{F},I}$ è definita ricorsivamente sul diagramma di controllo, dato il nodo di partenza q ed il pacchetto da valutare p, la funzione

- calcola qual è il risultato della valutazione del pacchetto p da parte della ruleset associata al nodo q;
- se il risultato è un pacchetto p' allora calcola il prossimo nodo del grafo che sarà visitato $q' = \delta(q, p')$;
- se il risultato è $\perp$ oppure se q' è già stato visitato, cioè è in I, restituisce $\perp$;
- se q' non è in I allora $[\![q]\!]_s^{\mathcal{F},I}$ associa a p il risultato dell'applicazione di $[\![q']\!]_s^{\mathcal{F},I\cup\{q\}}$ al pacchetto p'.

Il nodo finale accetta ogni pacchetto senza modificarlo.

La caratterizzazione funzionale è corretta rispetto alla semantica operazionale.

Teorema 3 (Correttezza della caratterizzazione funzionale)**.** *Dato un firewall normalizzato $\mathcal{F}$ abbiamo che*

1. $\forall p, p', s.\ (\ s \xrightarrow{p,p'} s \uplus (p, p')\ \iff\ [\![\mathcal{F}]\!](s)(p) = p'\ )$
2. $\forall p, s.\ (\ [\![\mathcal{F}]\!](s)(p) = \perp\ \iff\ \neg\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')\ )$

Si noti che il teorema 3 ha un corollario interessante, ovvero che il comportamento di un firewall è deterministico. Questo è dovuto alla scelta dell'insieme di target che abbiamo deciso di introdurre nel linguaggio, e in particolare al fatto che non abbiamo permesso di effettuare NAT verso intervalli di indirizzi. Questa scelta potrebbe non rispettare il reale comportamento di alcuni firewall (esclusi da questa trattazione), in cui il destino di un pacchetto è deciso non deterministicamente, ad esempio per effettuare bilanciamento del carico. Meccanismi più evoluti di bilanciamento del carico possono tuttavia essere modellati sfruttando lo stato interno del firewall.

Corollario 1 (Determinismo dei firewall)**.** *Dato un firewall IFCL $\mathcal{F}$, il destino associato ad un pacchetto p è unico, ovvero*

$$\forall p, s.\ (!\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')) \vee (\neg\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')\ )$$

3.2 Semantica a trasformazioni

Abbiamo espresso la semantica denotazionale del firewall nella maniera più naturale ovvero specificando per ogni pacchetto se questo può attraversare il firewall e quale è il suo stato dopo averlo attraversato. Tuttavia, per gli algoritmi che implementano la pipeline di transcompilazione, lavorare con le coppie (pacchetto ricevuto, pacchetto trasformato) di questa semantica sarebbe costoso e scomodo. Supponiamo ad esempio di voler rappresentare con una tabella tutte le coppie $(p, [\![\mathcal{F}]\!](p))$, avremmo bisogno di una tabella con $\#\mathbb{P}$ righe. A poco servirebbe unire fra loro le righe con la stessa immagine, in quanto ad esempio tutti i pacchetti che sono accettati senza alcuna modifica, avendo immagine diversa, necessiterebbero di una riga dedicata.

Vogliamo associare ad ogni pacchetto accettato la trasformazione che gli viene applicata, anziché il valore finale della trasformazione stessa. In questo modo ogni pacchetto che viene accettato senza

modifica ha la stessa immagine, così come anche ogni pacchetto che viene accettato modificando solo l'indirizzo IP di destinazione in una data maniera e così via. Abbiamo quindi la possibilità di raccogliere i pacchetti che sono trattati alla stessa maniera dal firewall.

Vogliamo definire una nuova semantica, che opera su domini diversi e che associa ad ogni pacchetto la trasformazione che il firewall applica ai suoi campi. In pratica per ogni campo del pacchetto abbiamo due possibilità: il firewall può modificare il campo (attraverso un NAT ad esempio) o lasciarlo com'è. Il concetto rimane lo stesso della semantica precedente, ma in questo modo tutti i pacchetti che vengono "trattati alla stessa maniera" dal firewall hanno la stessa immagine nella funzione che definisce la semantica e questo ci permetterà di trattarli come un unico insieme semplificando gli algoritmi della pipeline.

Definizione 11 (Trasformazione su un campo del pacchetto)**.** *Dato A l'insieme di valori possibili per il campo di un pacchetto, l'insieme delle trasformazioni possibili sull'insieme A è:*

$$\mathcal{T}(A) = \{id\} \cup \{cost(a) \mid a \in A\}$$

Una trasformazione può essere quindi una funzione costante o una funzione identità. Abbiamo dunque $\mathcal{T}(\mathbf{IP})$, $\mathcal{T}(\mathbf{Port})$ e $\mathcal{T}(\mathbf{Tag})$. $\mathcal{T}(\mathbb{P})$, l'insieme di tutte le possibili trasformazioni su pacchetti, è definito come il prodotto cartesiano delle trasformazioni sui campi del pacchetto[1].

Definizione 12 (Trasformazione su un pacchetto)**.** *L'insieme delle trasformazioni possibili sui pacchetti $\mathcal{T}(\mathbb{P})$ è:*

$$\mathcal{T}(\mathbb{P}) := \mathcal{T}(\mathbf{IP}) \times \mathcal{T}(\mathbf{Port}) \times \mathcal{T}(\mathbf{IP}) \times \mathcal{T}(\mathbf{Port}) \times \mathcal{T}(\mathbf{Tag})$$

Come per i pacchetti, anche per le trasformazioni di pacchetti assumiamo di poter accedere ai campi del prodotto cartesiano con una notazione simile a quella per l'accesso ai campi di un oggetto, cioè scrivendo $t.sIP$, $t.dIP$, $t.sPort$, $t.dPort$ e $t.tag$. Inoltre rappresentiamo le trasformazioni di pacchetti $t \in \mathcal{T}(\mathbb{P})$ in modo simile ai pacchetti, scrivendo per leggibilità $(t.sIP : t.sPort,\ t.dIP : t.dPort, t.tag)$ anziché $(t.sIP, t.sPort, t.dIP, t.dPort, t.tag)$

La scelta del dominio per esprimere le trasformazioni dei campi di un pacchetto è chiaramente arbitraria, ogni alternativa che permetta di derivare la semantica originaria è accettabile. Questa scelta è semplicemente un buon compromesso fra semplicità ed espressività che tiene conto sia del fatto che un firewall spesso lascia passare i pacchetti senza modificarli, sia del fatto che spesso la modifica è ristretta solo ad alcuni campi del pacchetto e imponendo un valore costante piuttosto che una funzione arbitrariamente complicata. Di fatto il linguaggio nel quale esprimiamo le trasformazioni è quello usato dal tool *FireWall Synthesizer*[5] per mostrare la sintesi della semantica di un firewall, dove nella tabella mostrata dalla sintesi un valore esplicito nelle colonne relative al NAT rappresenta la funzione costante e il carattere _ rappresenta la funzione identità.

Le trasformazioni possono chiaramente essere applicate ai pacchetti, in modo simile alle funzioni. Useremo in effetti la stessa notazione usata per le funzioni. Data una trasformazione t ed un pacchetto p, l'applicazione di t a p è definita come il risultato dell'applicazione delle trasformazioni dei campi di t ai campi di p. Formalmente

$$t(p) = (t.sIP(p.sIP) :\ t.sPort(p.sPort),\ t.dIP(p.dIP) :\ t.dPort(p.dPort),\ t.tag(p.tag))$$

dove per il valore di un campo di un pacchetto val e per una trasformazione su quel campo t vale

$$t(val) = \begin{cases} val & \text{se } t = id \\ c & \text{se } t = cost(c) \end{cases}$$

[1] Useremo $id \in \mathcal{T}(\mathbb{P})$ per indicare $id \times id \times id \times id \times id$.

Possiamo vedere l'insieme delle trasformazioni sui pacchetti $\mathcal{T}(\mathbb{P})$ come un preordine in cui $t \lesssim t'$ se e solo se t' è una trasformazione che modifica ogni pacchetto almeno quanto t. Formalmente la relazione d'ordine $\lesssim$ è definita sulle trasformazioni del campo di tipo $A \in \{\mathbf{IP}, \mathbf{Port}, \mathbf{Tag}\}$ del pacchetto come la chiusura riflessiva e transitiva di $\lesssim_0$, dove

$$\forall a \in A.\ id \lesssim_0 cost(a)$$
$$\forall a, a' \in A.\ cost(a) \lesssim_0 cost(a')$$

Per l'insieme delle trasformazioni sui pacchetti $\mathcal{T}(\mathbb{P})$ il preordine è definito nella seguente maniera:

$$t \lesssim t' \iff t.sIP \lesssim t'.sIP \wedge t.sPort \lesssim t'.sPort \wedge t.dIP \lesssim t'.dIP \wedge t.dPort \lesssim t'.dSort \wedge t.tag \lesssim t'.tag$$

Notiamo che $(\mathcal{T}(A), \lesssim)$ è un insieme diretto, per ogni $A \in \{\mathbf{IP}, \mathbf{Port}, \mathbf{Tag}\}$, cioè per ogni coppia di elementi $t_1, t_2 \in \mathcal{T}(A)$, esiste un $t' \in \mathcal{T}(A)$ tale che $t_1 \lesssim t'$ e $t_2 \lesssim t'$. Lo stesso vale conseguentemente per $(\mathcal{T}(\mathbb{P}), \lesssim)$. La relazione $\lesssim$ è totale su $\mathcal{T}(A)$, per $A \in \{\mathbf{IP}, \mathbf{Port}, \mathbf{Tag}\}$. In questo insieme, il least upper bound di un insieme di elementi esiste sempre, dato che per ogni coppia di trasformazioni (t, t') o vale $t \lesssim t'$ o vale $t' \lesssim t$. Tuttavia il least upper bound non è sempre unico. Infatti data una qualunque trasformazione $t \in \mathcal{T}(A)$, per $A \in \{\mathbf{IP}, \mathbf{Port}, \mathbf{Tag}\}$ l'insieme dei maggioranti di t in $\mathcal{T}(A)$ comprende sempre almeno $cost(a)$ per ogni $a \in A$, nei quali ogni elemento è "minore" di tutti gli altri; se anche id appartiene ai maggioranti allora il minore dei maggioranti è unico e coincide con id, altrimenti non abbiamo un unico least upper bound. La relazione $\lesssim$ non è invece totale su $\mathcal{T}(\mathbb{P})$, infatti le due trasformazioni $t_1 = cost(192.168.0.8) \times id \times id \times id \times id$ e $t_2 = id \times id \times cost(192.168.0.6) \times id \times id$ non sono in relazione. Anche su $\mathcal{T}(A)$ esiste sempre un insieme di least upper bound di un insieme di elementi, e anche qui spesso non si tratta di un unico valore. Ad esempio l'insieme dei least upper bound dell'insieme $\{t_1, t_2\}$ è $\{cost(ip) \times id \times cost(ip') \times id \times id \mid ip, ip' \in \mathbf{IP}\}$.

Definiamo l'operazione di aggiornamento $\_ \ltimes \_$ tale che $t' \ltimes t$ è una aggiornamento della trasformazione t con una nuova trasformazione t' ed è uguale a t' nei campi del pacchetto in cui questa è diversa da id, nei campi rimanenti la trasformazione risultante ha lo stesso valore di t (sostanzialmente prendiamo fra i least upper bound dell'insieme $\{t, t'\}$, quello in cui i valori delle trasformazioni $cost(\_)$ sono presi da t' e da t, dando precedenza ai valori di t' su quelli di t). Formalmente

$$t' \ltimes t = (t'.sIP \dot{\ltimes} t.sIP,\ t'.sPort \dot{\ltimes} t.sPort,\ t'.dIP \dot{\ltimes} t.dIP,\ t'.dPort \dot{\ltimes} t.dPort,\ t'.tag \dot{\ltimes} t.tag)$$

dove

$$val_1 \dot{\ltimes} val_2 \begin{cases} val_2 & \text{se } val_2 \neq id \\ val_1 & \text{altrimenti} \end{cases}$$

L'idea è la stessa della composizione di funzioni, cioè applicare una trasformazione t, aggiornata con t' ad un pacchetto p ha lo stesso effetto di applicargli la trasformazione t e poi al pacchetto risultante $p' = t(p)$ applicare t', ovvero $(t \ltimes t')(p) = t(t'(p))$.

Possiamo ora definire una nuova semantica che sfrutti le trasformazioni. Per prima cosa presentiamo la semantica di una ruleset R:

$$(\!|R|\!) : S \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$$
$$(\!|R|\!)(s) = (\!|R|\!)_s : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$$

$$(\!|R|\!)_s \ = (\!|R|\!)_s^{id}$$

Dove la funzione $(\!|R|\!)_s^t : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, per un firewall $\mathcal{F}$, uno stato s e una trasformazione su pacchetti $t \in \mathcal{T}(\mathbb{P})$ è definita come:

$$(\!|\epsilon|\!)_s^t\ (p) = \begin{cases} t & \text{se } dp = \texttt{ACCEPT} \\ \bot & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{ACCEPT}); R|\!)_s^t\ (p) = \begin{cases} t & \text{se } \phi(p,s) \\ (\!|R|\!)_s^t\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{DROP}); R|\!)_s^t\ (p) = \begin{cases} \bot & \text{se } \phi(p,s) \\ (\!|R|\!)_s^t\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{NAT}(d_n, s_n)); R|\!)_s^t\ (p) = \begin{cases} tr_{nat}(d_n, s_n) \ltimes t & \text{se } \phi(p,s) \\ (\!|R|\!)_s^t\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{CHECK-STATE}(X)); R|\!)_s^t\ (p) = \begin{cases} tr_{stato}(\alpha, X) \ltimes t & \text{se } \phi(p,s) \wedge p \vdash_s \alpha \\ (\!|R|\!)_s^t\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{MARK}(m)); R|\!)_s^t\ (p) = \begin{cases} (\!|R|\!)_s^{(id:id,\ id:id,\ cost(m)) \ltimes t}\ (p[tag \mapsto m]) & \text{se } \phi(p,s) \\ (\!|R|\!)_s^t\ (p) & \text{altrimenti} \end{cases}$$

La funzione $tr_{nat}(d_n, s_n)$ restituisce la trasformazione applicata ad un pacchetto secondo i valori indirizzo IP e porta specificati da d_n ed s_n, compresa l'identità se uno dei due valori contiene $\star$. Più precisamente

$$tr_{nat}(dIP : dPort,\ sIP : sPort) = (tr'(dIP) : tr'(dPort),\ tr'(sIP) : tr'(sPort),\ id)$$

dove

$$tr'(add) = \begin{cases} id & \text{se } add = \star \\ cost(add) & \text{altrimenti} \end{cases}$$

La funzione $tr_{stato}(\alpha, X)$ restituisce la trasformazione rappresentata dall'azione α, prescritta dallo stato per il pacchetto, ristretta ad i campi specificati da X. Lasciamo astratta questa funzione dato che per generalità non abbiamo descritto in dettaglio quale sia la forma delle azioni α.

La semantica è espressa in modo simile a quella precedente, attraverso una funzione ricorsiva $(\!|R|\!)_s^t$ definita per casi sulla prima regola di R. L'apice t serve a collezionare la trasformazione attuale applicata al pacchetto, cioè l'eventuale modifica al campo tag, di modo che quando troviamo l'azione che determina l'accettazione del pacchetto possiamo ricavare la trasformazione complessiva avendo sia t, sia la trasformazione finale data dall'ultima regola applicata al pacchetto. Sfruttiamo l'operazione di aggiornamento della trasformazione per unire la trasformazione data dall'ultima regola a quella accumulata nell'apice.

Se la ruleset è vuota allora la trasformazione applicata al pacchetto p è quella accumulata t, tranne nel caso in cui la policy di default della ruleset sia DROP, in quel caso restituiamo $\bot$. Le ruleset che cominciano con una regola con target ACCEPT o DROP restituiscono rispettivamente t o $\bot$ se la condizione è verificata, altrimenti restituiscono il risultato dell'applicazione della semantica del resto della ruleset al pacchetto. Se la prima regola della ruleset ha target NAT(d_n, s_n) o CHECK-STATE(X) restituiscono la

trasformazione t aggiornata con la loro trasformazione, se la condizione ϕ è verificata e, nel caso di CHECK-STATE(X) se il pacchetto appartiene ad una connessione stabilita; associano al pacchetto la trasformazione associatagli dalla semantica del resto della ruleset altrimenti. Se la ruleset comincia con una regola con target MARK(m) allora la semantica restituisce il risultato dell'applicazione della semantica della continuazione: se la condizione della regola è verificata allora la semantica della continuazione è applicata al pacchetto modificato nel campo *tag* e con trasformazione accumulata t aggiornata da una trasformazione identità su tutti i campi tranne sul campo *tag*, il cui valore è $cost(m)$; se la condizione della regola non è verificata allora la semantica della continuazione è applicata al pacchetto originale e la trasformazione accumulata rimate t.

La semantica a trasformazioni definita per una ruleset R è equivalente a quella denotazionale definita precedentemente.

Lemma 3. *Per ogni ruleset normalizzata R, stato s e pacchetto p valgono*

1. $[\![R]\!]_s\ (p) = \bot \iff (\!|R|\!)_s\ (p) = \bot$
2. $(\!|R|\!)_s\ (p) \neq \bot \Rightarrow [\![R]\!]_s\ (p) = (\!|R|\!)_s\ (p)\ (p)$

La semantica di un firewall $\mathcal{F}$ è definita allora da

$$(\!|\mathcal{F}|\!) : S \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$$
$$(\!|\mathcal{F}|\!)(s) = (\!|\mathcal{F}|\!)_s : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$$

$$(\!|\mathcal{F}|\!)_s = (\!|q_i|\!)_s^{\mathcal{F},\emptyset}$$

Dove la funzione $(\!|q|\!)_s^{\mathcal{F},I} : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, per un firewall $\mathcal{F}$, uno stato s e un insieme di ruleset I è definita per ogni $q \in Q,\ q \neq q_f$ come:

$$(\!|q|\!)_s^{\mathcal{F},I}\ (p) = \begin{cases} (\!|q'|\!)_s^{\mathcal{F},I\cup\{q\}}\ (p') \ltimes t & \text{se } t \neq \bot \wedge q' \notin I \\ \bot & \text{altrimenti} \end{cases}$$
$$\text{dove } t = (\!|c(q)|\!)_s\ (p) \text{ e se } t \neq \bot \text{ allora } p' = t(p) \text{ e } q' = \delta(q, p')$$

e per il nodo finale come:

$$(\!|q_f|\!)_s^{\mathcal{F},I}\ (p) = id$$

Questa parte della semantica rimane quasi invariata rispetto alla versione presentata precedentemente. La semantica della ruleset di un nodo q associa una trasformazione t al pacchetto p, quindi per ottenere il pacchetto p' prodotto dal nodo q devo applicare la trasformazione t al pacchetto p, ottenendo così $p' = t(p)$. Nella versione precedente della semantica, una volta stabilito che il pacchetto p fosse trasformato in p' dal nodo q, e che il nodo successivo ad essere visitato fosse q', la semantica associava al pacchetto p il risultato dell'applicazione della funzione associata a q' al pacchetto p'. Nella versione con trasformazioni, quando ho stabilito che il nodo successivo q' associa la trasformazione t' al pacchetto $p' = t(p)$ non mi basta restituire t', devo tenere traccia anche della trasformazione t applicata al pacchetto da q e restituire quindi t aggiornata con t', cioè $t' \ltimes t$.

La semantica a trasformazioni è corretta rispetto alla semantica denotazionale, e quindi rispetto alla semantica dichiarativa di IFCL. Cioè, dato un firewall $\mathcal{F}$, per ogni pacchetto $p \in \mathbb{P}$, se calcolo

la trasformazione t associata a p dalla semantica a trasformazioni di $\mathcal{F}$ e la applico al pacchetto p, ottengo un nuovo pacchetto $p' = t(p)$ che è uguale al pacchetto associato a p dalla semantica $[\![\mathcal{F}]\!]$. Inoltre le due semantiche concordano sullo scarto dei pacchetti.

Teorema 4 (Correttezza della semantica a trasformazioni)**.** *Per ogni firewall $\mathcal{F}$, stato s e pacchetto p valgono*

1. $[\![\mathcal{F}]\!]_s\ (p) = \bot \iff (\!|\mathcal{F}|\!)_s\ (p) = \bot$
2. $(\!|\mathcal{F}|\!)_s\ (p) \neq \bot \Rightarrow [\![\mathcal{F}]\!]_s\ (p) = (\!|\mathcal{F}|\!)_s\ (p)\ (p)$

Dove con la notazione $(\!|\mathcal{F}|\!)_s\ (p)\ (p)$ rappresentiamo il risultato dell'applicazione della trasformazione associata al pacchetto p da $\mathcal{F}$ (cioè $(\!|\mathcal{F}|\!)_s\ (p)$), al pacchetto p stesso.

Si noti che l'uso delle trasformazioni per esprimere il comportamento del firewall introduce un certo grado di arbitrarietà: ad esempio un pacchetto p che viene accettato senza modifiche dal firewall, ovvero tale che $[\![\mathcal{F}]\!]_s\ (p) = p$, può essere associato a $id \in \mathcal{T}(\mathbb{P})$ dalla semantica, ma anche a $(cost(p.sIP), cost(p.sPort), cost(p.dIP), cost(p.dPort), cost(p.tag))$ e a molte altre trasformazioni. Dato un firewall $\mathcal{F}$, esistono infatti più funzioni $f : S \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ tali che $\forall s, p.\ [\![\mathcal{F}]\!]_s\ (p) = \bot \iff f(s)(p) = \bot$ e $\forall s, p.\ [\![\mathcal{F}]\!]_s\ (p) \neq \bot \Rightarrow [\![\mathcal{F}]\!]_s\ (p) = f(s)(p)(p)$. Non consideriamo questo un problema in quanto la caratterizzazione del comportamento del firewall è comunque corretta e la trasformazione associata ad un pacchetto dalla semantica può essere standardizzata, dato che questa ambiguità si presenta unicamente nel caso in cui un pacchetto sia trasformato da un NAT in se stesso. È quindi possibile riconosce questi casi e forzare una preferenza, ad esempio nei confronti della trasformazione id.

Definiamo una relazione di equivalenza $\equiv$ per le funzioni che assegnano trasformazioni ai pacchetti $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ che lega coppie di funzioni che hanno lo stesso risultato finale sull'insieme dei pacchetti $\mathbb{P}$. Formalmente $f \equiv f'$ se e solo se $\forall p \in \mathbb{P}.\ f(p) = \bot \iff f'(p) = \bot$ e $\forall p \in \mathbb{P}.\ f(p) \neq \bot \Rightarrow f(p)(p) = f'(p)(p)$.

Si noti che la caratterizzazione funzionale della semantica non comprende un'approssimazione sul comportamento dello stato del firewall, come invece accade per la caratterizzazione logica presentata nella sezione 2.5. Questo è un problema, perché come abbiamo detto, nella sintesi non possiamo aspettarci di studiare il comportamento del firewall per tutti i possibili stati analizzandoli uno ad uno. Di fatto è possibile modificare la semantica appena espressa per gestire il target CHECK-STATE(X) con la stessa approssimazione della caratterizzazione dichiarativa. La versione approssimata della semantica, tuttavia, complica grandemente lo sviluppo degli algoritmi per l'implementazione della pipeline, ed è troppo grossolana per la rappresentazione del comportamento reale di un firewall. Per questo abbiamo deciso di basarci sulla versione esatta e deterministica della semantica nel resto della tesi, e di limitare i requisiti sulla configurazione generata dalla transcompilazione alla gestione di pacchetti appartenenti a nuove connessioni, assumendo che il comportamento di default del sistema target sia soddisfacente per quanto riguarda i pacchetti appartenenti a connessioni stabilite (vedi capitolo 4). Presentiamo comunque per completezza la versione approssimata della semantica nel capitolo 8.

Capitolo 4

Pipeline di transcompilazione

Il problema che vogliamo affrontare è quello della transcompilazione fra linguaggi di configurazione di firewall. L'idea è di avere una configurazione per un sistema e di volerne creare una equivalente, cioè avente la stessa semantica, per un secondo sistema scelto.

Dal punto di vista dei sistemi firewall questo procedimento deve tenere conto sia della diversa struttura dei diagrammi di controllo dei due sistemi, cioè degli algoritmi di controllo che applicano le policy, sia del linguaggio di configurazione in sé. Sfruttiamo il linguaggio di modellazione comune IFCL per esprimere delle condizioni sulla traduzione che garantiscano la conservazione della semantica del firewall. Riformulando in questo modo, vogliamo, dato un sistema e una configurazione source e dato un sistema target, calcolare una configurazione target tale che la formalizzazione IFCL del firewall source e quella del firewall target abbiano la stessa semantica.

Basiamo la transcompilazione su una versione lievemente modificata della pipeline proposta in [4]. Come detto, l'approccio originale si basa su una pipeline composta da tre stadi: (*i*) formalizzazione del firewall source in IFCL, (*ii*) derivazione della semantica del firewall in una forma sintetica ed esplicita, (*iii*) generazione del firewall target avente la stessa semantica. Nella versione originale della pipeline la semantica del firewall veniva rappresentata come un predicato $P_{\mathcal{F}}(p, \tilde{p}, \mathtt{s})$ e la rappresentazione sintetica era basata sui modelli del predicato: il secondo stadio della pipeline restituiva l'insieme delle triplette $(p, \tilde{p}, \mathtt{s})$ che verificano il predicato $P_{\mathcal{F}}$, espresso in maniera sintetica grazie ai multicubi.

Per convenienza nel trattare la parte finale del processo di transcompilazione, speziamo l'ultima fase della pipeline originale in due parti: la prima, a partire dalla descrizione sintetica della semantica, produce un firewall IFCL; la seconda lo traduce nel linguaggio di configurazione target (si può notare una certa simmetria con le prime due fasi della pipeline originale). Questo ci permette di analizzare con chiarezza il passaggio di traduzione da IFCL al linguaggio target, trascurato da [4].

Inoltre, anziché basarci sulla caratterizzazione logica della semantica del firewall usiamo la descrizione denotazionale presentata: la semantica astratta di un firewall non è rappresentata da un predicato ma da una funzione definita sul dominio dei pacchetti. È diversa pertanto anche la rappresentazione sintetica della semantica del firewall.

Presentiamo la pipeline specificando una serie di passi intermedi per ciascuna fase e presentiamo per ogni passaggio la specifica del comportamento: quali sono i dati in input e quali sono i dati calcolati e restituiti alla fase successiva. Inoltre per ogni stadio della pipeline presentiamo i domini degli elementi trattati e studiamo la loro rappresentazione sintetica, necessaria per questioni di efficienza. Dimostriamo che la transcompilazione è corretta, cioè che la semantica del firewall è conservata, se l'implementazione della pipeline rispetta le specifiche.

La ridefinizione della pipeline nella forma che presenteremo permette di

- parametrizzare il processo di transcompilazione rispetto alla specifica dei sistemi firewall in modo da facilitare l'estensione della teoria
- garantire la conservazione delle trasformazioni effettuate sui pacchetti
- fornire un algoritmo efficiente per il calcolo della rappresentazione sintetica del firewall
- evidenziare le scelte arbitrarie nella fase di generazione del firewall finale sulle quali è possibile intervenire per ottimizzare il firewall prodotto

4.1 Transcompilazione di configurazioni firewall

Assumiamo, per ogni sistema firewall supportato, di avere una funzione che dato un file di configurazione source ne deriva la formalizzazione in IFCL, $for_k(\texttt{file.conf}) = \Sigma$ per $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$, dove `file.conf` è il file di configurazione per il sistema source e Σ è una configurazione IFCL. Chiamiamo $\mathcal{C}_k$ il diagramma di controllo del sistema firewall k. Le funzioni di formalizzazione e i diagrammi di controllo per i sistemi supportati sono presentati informalmente nella sezione 2.2.

Formalmente il problema che vogliamo affrontare è il seguente: dato un firewall source definito come la coppia ($k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$, `file.conf`) e un sistema target desiderato k', produrre un file di configurazione `file.conf`$'$ per il sistema target tale che la semantica dei due firewall sia equivalente, ovvero, chiamando $s \xrightarrow{p,p'}_X s'$ un passo del sistema di transizione master del firewall $X \in \{\mathcal{F}, \mathcal{F}'\}$, dove $\mathcal{F} = (\mathcal{C}_k, for_k(\texttt{file.conf}))$ e $\mathcal{F}' = (\mathcal{C}_{k'}, for_{k'}(\texttt{file.conf}'))$, deve valere:

$$\forall s, s' \in S,\ p, p' \in \mathbb{P}.\ s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{\mathcal{F}'} s'$$

In realtà, come abbiamo dichiarato precedentemente, tralasciamo dalla trattazione i pacchetti appartenenti a connessioni stabilite. Consideriamo per questi pacchetti che il comportamento di default dei sistemi sia quello corretto. Dato che lo stato associato ad un pacchetto non influenza il destino di altri pacchetti, per studiare il comportamento di tutti i pacchetti quando non appartengono a connessioni stabilite è sufficiente studiare il firewall in un solo stato: $s_{\texttt{NEW}}$, lo stato in cui non esiste nessuna connessione stabilita. Formalmente possiamo limitarci a questo stato in quanto per ogni firewall $\mathcal{F}$ vale:

$$\forall s \in S, p \in \mathbb{P}.\ s \not\vdash_p \implies (\forall p'.\ s \xrightarrow{p,p'}_{\mathcal{F}} s \uplus (p, p') \iff s_{\texttt{NEW}} \xrightarrow{p,p'}_{\mathcal{F}} s_{\texttt{NEW}} \uplus (p, p'))$$

Inoltre per comodità preferiamo basarci sulla semantica denotazionale anziché sul sistema di transizioni etichettate. Dati i teoremi 2.4, 3 e 4 vale il seguente lemma.

Lemma 4. *Due firewall IFCL $\mathcal{F}$ e $\mathcal{F}'$ sono equivalenti secondo la semantica operazionale se e solo se la loro normalizzazione è equivalente secondo la semantica denotazionale, ovvero*

$$\forall s \in S, p \in \mathbb{P}.\ \forall p', s'.\ s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{\mathcal{F}'} s'$$

se e solo se

$$(\!|\mathbb{C}\ \mathcal{F}'\ \mathbb{D}|\!) \equiv (\!|\mathbb{C}\ \mathcal{F}'\ \mathbb{D}|\!)$$

Riassumendo, dato un firewall source definito come la coppia composta dal sistema $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$ e dal file di configurazione `file.conf`, e dato un sistema target desiderato k', vogliamo produrre un file di configurazione `file.conf`$'$ per il sistema target tale che:

$$(\!|\mathbb{C}\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ \mathbb{D}|\!)(s_{\texttt{NEW}}) \equiv (\!|\mathbb{C}\ (\mathcal{C}_{k'}, for_{k'}(\texttt{file.conf}'))\ \mathbb{D}|\!)(s_{\texttt{NEW}})$$

4.2 Presentazione della pipeline

La pipeline di transcompilazione è composta da quattro stadi: il primo riguarda la traduzione del firewall source in IFCL, il secondo consiste nell'estrazione della semantica del firewall come funzione da pacchetti a trasformazioni, il terzo genera un nuovo firewall IFCL del tipo scelto e infine il quarto traduce il firewall IFCL in un file di configurazione per il sistema target.

Come abbiamo detto per la parte centrale della pipeline, quelle relativa a firewall IFCL, ci basiamo sulla semantica denotazionale presentata nel capitolo 3. La versione precedente della pipeline, presentata in [4] si basava sulla caratterizzazione logica presentata nella sezione 2.5.

Gli stadi della pipeline sono i seguenti, per alcuni di loro abbiamo evidenziato una serie di passi intermedi:

1. traduzione del firewall iniziale in IFCL
2. estrazione della semantica astratta del firewall
   (a) normalizzazione del firewall IFCL
   (b) astrazione delle ruleset associate ai nodi del diagramma di controllo del firewall IFCL
   (c) astrazione del firewall come funzione sui pacchetti
3. generazione del firewall IFCL finale
   (a) calcolo delle funzioni associate ai nodi del diagramma di controllo del firewall IFCL
   (b) traduzione delle funzioni associate ai nodi in ruleset IFCL
4. traduzione in configurazione nel linguaggio target

4.2.1 Esempio di transcompilazione

Presentiamo un esempio di transcompilazione completo nel quale mostriamo una ad una le fasi della pipeline. Consideriamo il caso di un firewall `ipfw` del quale vogliamo portare la configurazione in `iptables`. Supponiamo che il file di configurazione `ipfw` sia il seguente:

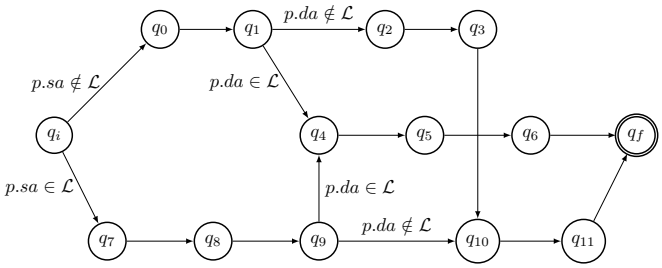
```
ipfw -q nat 1 config redirect_port tcp 192.168.0.8:22 22
ipfw -q nat 2 config ip 151.15.185.183

ipfw -q add 0010 deny all from any to 8.8.8.8
ipfw -q add 0020 nat 1 all from not 192.168.0.0/24 to 151.15.185.183 22
ipfw -q add 0030 nat 2 all from 192.168.0.0/24 to not 192.168.0.0/24 80
ipfw -q add 0040 allow all from 127.0.0.1 to 192.168.0.8 22
ipfw -q add 0050 allow all from 151.15.185.183 to 192.168.0.8 22
ipfw -q add 0060 deny all from any to any
```

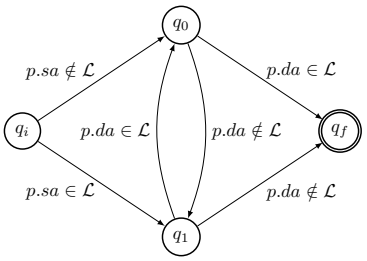
Seguendo la pipeline, effettuiamo la formalizzazione del firewall `ipfw` in IFCL (stadio 1.). Prendiamo quindi il diagramma di controllo $\mathcal{C}_{\texttt{ipfw}}$ di `ipfw`, in figura 4.1b. Sfruttando la procedura descritta in 2.2, costruiamo un insieme di ruleset ρ e un assegnamento c ai nodi del diagramma $\mathcal{C}_{\texttt{ipfw}}$. Il risultato di questa prima fase della pipeline è la configurazione IFCL $\Sigma = (\rho, c)$ dove:

$\rho = \{R^I_{10}, R^I_{20}, R^I_{30}, R^I_{40}, R^I_{10}, R^O_{20}, R^O_{30}, R^O_{40}\}$
$c(q_i) = c(q_f) = R_\epsilon,\ c(q_0) = R^I_{10},\ c(q_1) = R^O_{10}$

$R^{I/O}_{10} =$



(a) Diagramma di controllo di `iptables`



(b) Diagramma di controllo di `ipfw`

Figura 4.1: Diagrammi di controllo dei sistemi supportati.

$$(\texttt{p.dIP} = 8.8.8.8, \textsc{drop});$$
$$(true, \textsc{goto} R_{20}^{I/O})$$

$$R_{20}^{I/O} =$$
$$(\texttt{p.sIP} \notin 192.168.0.0/24 \wedge \texttt{p.dIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 22,\ \textsc{nat}(\star : \star, 192.168.0.8 : \star));$$
$$(true, \textsc{goto} R_{30}^{I/O})$$

$$R_{30}^{I/O} =$$
$$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \notin 192.168.0.0/24 \wedge \texttt{p.dPort} = 80,\ \textsc{nat}(151.15.185.183 : \star, \star : \star));$$
$$(true, \textsc{goto} R_{40}^{I/O})$$

$$R_{40}^{I/O} =$$
$$(\texttt{p.sIP} = 127.0.0.1 \wedge \texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22,\ \textsc{accept});$$
$$(true, \textsc{goto} R_{50}^{I/O})$$

$$R_{50}^{I/O} =$$
$$(\texttt{p.sIP} = 151.15.185.183 \wedge \texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22,\ \textsc{accept});$$
$$(true, \textsc{goto} R_{60}^{I/O})$$

$$R_{60}^{I/O} =$$
$$(true, \textsc{drop})$$

Lo stadio 2. della pipeline è composto da tre passi: per prima cosa normalizziamo il firewall attraverso la procedura $(\!|\ \_\ |\!)$. Otteniamo:

$$\rho = \{R^I, R^O\}$$
$$c(q_i) = c(q_f) = R_\epsilon,\ c(q_0) = R^I,\ c(q_1) = R^O$$

$$R^I = R^O =$$
$$(\texttt{p.dIP} = 8.8.8.8, \textsc{drop});$$
$$(\texttt{p.sIP} \notin 192.168.0.0/24 \wedge \texttt{p.dIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 22,\ \textsc{nat}(\star : \star, 192.168.0.8 : \star));$$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \notin 192.168.0.0/24 \wedge \texttt{p.dPort} = 80,\ \textsc{nat}(151.15.185.183 : \star, \star : \star))$;

$(\texttt{p.sIP} = 127.0.0.1 \wedge \texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22,\ \textsc{accept})$;

$(\texttt{p.sIP} = 151.15.185.183 \wedge \texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22,\ \textsc{accept})$;

$(true, \textsc{drop})$

Dove per questioni di leggibilità le regole con condizione equivalente a $false$ sono state omesse, e le condizioni della forma $true \wedge \phi$ sono riscritte come ϕ. Entrambe le semplificazioni non modificano la semantica delle ruleset, si limitano a renderle più leggibili. A questo punto, a partire dalle ruleset, usando $(\!|\_|\!)(s_{\texttt{NEW}})$, calcoliamo per ogni nodo q_i la funzione $\lambda_i : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ (fase 2.b).

$$\lambda_0(p) = \lambda_1(p) = \begin{cases} id \times id \times cost(192.168.0.8) \times id \times id & \text{se } \texttt{p.sIP} \notin 192.168.0.0/24 \wedge \\ & \quad \texttt{p.dIP} = 151.15.185.183 \wedge \\ & \quad \texttt{p.dPort} = 22 \\ cost(151.15.185.183) \times id \times id \times id \times id & \text{se } \texttt{p.sIP} \in 192.168.0.0/24 \wedge \\ & \quad \texttt{p.dIP} \notin 192.168.0.0/24 \cup \{8.8.8.8\} \wedge \\ & \quad \texttt{p.dPort} = 80 \\ id \times id \times id \times id \times id & \text{se } \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dIP} = 192.168.0.8 \wedge \\ & \quad \texttt{p.dPort} = 22 \\ \bot & \text{altrimenti} \end{cases}$$

Nella fase 2 calcoliamo la funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ ottenuta combinando le funzioni λ_0 e λ_1 sulla base del diagramma di controllo. Assumendo che l'insieme degli indirizzi locali sia $\mathcal{L} = \{\ 127.0.0.1,\ 192.168.0.1,\ 151.15.185.183\ \}$ la funzione λ è:

$$\lambda(p) = \begin{cases} cost(151.15.185.183) \times id \times id \times id \times id & \text{se } (\ \texttt{p.sIP} \in 192.168.0.0/24 \wedge \\ & \quad \texttt{p.dIP} \in \{151.15.185.183,\ 127.0.0.1\} \wedge \\ & \quad \texttt{p.dPort} = 80\ ) \\ & \vee\ (\ \texttt{p.sIP} = 192.168.0.1 \wedge \\ & \quad \texttt{p.dIP} \notin 192.168.0.0/24 \cup \{8.8.8.8\} \wedge \\ & \quad \texttt{p.dPort} = 80\ ) \\ id \times id \times cost(192.168.0.8) \times id \times id & \text{se } \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dIP} = 151.15.185.183 \wedge \\ & \quad \texttt{p.dPort} = 22 \\ id \times id \times id \times id \times id & \text{se } \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dIP} = 192.168.0.8 \wedge \\ & \quad \texttt{p.dPort} = 22 \\ \bot & \text{altrimenti} \end{cases}$$

A questo punto conosciamo la semantica del firewall e vogliamo generarne uno con semantica identica per `iptables`. Per prima cosa prendiamo il diagramma di controllo di iptables (figura 4.1a). Vogliamo assegnare ad ogni nodo q_i del diagramma una funzione $\lambda_i : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ in modo tale che il risultato complessivo sia coerente con la semantica attesa (fase 3.a). La descrizione di un metodo efficace per derivare le funzioni λ_i a partire dalla funzione λ e dalla forma del diagramma di controllo sarà oggetto del capitolo 7. Per il momento ci limitiamo ad osservare che non tutti i nodi possono effettuare trasformazioni arbitrarie, questo è immediatamente evidente in `iptables` in quanto i nodi corrispondono a coppie (ruleset, tabella) dove solo nella tabella `NAT` è possibile modificare gli indirizzi dei pacchetti. Diamo senza ulteriori chiarimenti un'assegnazione di funzioni ai nodi:

$$\lambda_0(p) = \lambda_1(p) = \lambda_2(p) = \lambda_4(p) = \lambda_7(p) = \lambda_{10}(p) = id$$

$$\lambda_3(p) = \bot$$

$$\lambda_5(p) = \begin{cases} cost(151.15.185.183) \times id \times id \times id \times id & \text{se } \texttt{p.sIP} \in 192.168.0.0/24 \wedge \\ & \quad \texttt{p.dIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dPort} = 80 \\ id & \text{altrimenti} \end{cases}$$

$$\lambda_6(p) = \begin{cases} id & \text{se } \texttt{p.sIP} = 151.15.185.183 \wedge \\ & \quad \texttt{p.dIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dPort} = 80 \\ \bot & \text{altrimenti} \end{cases}$$

$$\lambda_8(p) = \begin{cases} id \times id \times cost(192.168.0.8) \times id \times id & \text{se } \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dIP} = 151.15.185.183 \wedge \\ & \quad \texttt{p.dPort} = 22 \\ id & \text{altrimenti} \end{cases}$$

$$\lambda_9(p) = \begin{cases} id & \text{se } (\ \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \\ & \quad \texttt{p.dIP} = 192.168.0.8 \wedge \\ & \quad \texttt{p.dPort} = 22\ ) \\ & \vee\ (\ \texttt{p.sIP} = 192.168.0.1 \wedge \\ & \quad \texttt{p.dIP} \notin 192.168.0.0/24 \cup \{8.8.8.8\} \wedge \\ & \quad \texttt{p.dPort} = 80\ ) \\ \bot & \text{altrimenti} \end{cases}$$

$$\lambda_{11}(p) = \begin{cases} cost(151.15.185.183) \times id \times id \times id \times id & \text{se } \texttt{p.sIP} = 192.168.0.1 \wedge \\ & \quad \texttt{p.dIP} \notin 192.168.0.0/24 \cup \\ & \qquad \{127.0.0.1,\ 151.15.185.183,\ 8.8.8.8\} \wedge \\ & \quad \texttt{p.dPort} = 80 \\ id & \text{altrimenti} \end{cases}$$

Dove la ruleset λ_i è assegnata al nodo q_i. Il passaggio successivo, 3.b, prevede di generare una ruleset IFCL R_i per ogni nodo q_i del diagramma di controllo che abbia semantica corrispondente alla funzione

λ_i.

$R_0 = R_1 = R_2 = R_4 = R_7 = R_{10} = R_i = R_f = R_\epsilon$

$R_3 = (true, \texttt{DROP})$

$R_5 =$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \texttt{p.dPort} = 80\ ,\ \texttt{NAT}(151.15.185.183 : \star,\ \star : \star))$

$R_6 =$

$(\texttt{p.sIP} = 151.15.185.183 \wedge \texttt{p.dIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \texttt{p.dPort} = 80\ ,\ \texttt{ACCEPT});$

$(true\ ,\ \texttt{DROP})$

$R_8 =$

$(\texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \texttt{p.dIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 22\ ,\ \texttt{NAT}(\star : \star,\ 192.168.0.8 : \star))$

$R_9 =$

$(\ (\ \texttt{p.sIP} \in \{127.0.0.1,\ 151.15.185.183\} \wedge \texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22) \vee$

$(\ \texttt{p.sIP} = 192.168.0.1 \wedge \texttt{p.dIP} \notin 192.168.0.0/24 \cup \{8.8.8.8\} \wedge \texttt{p.dPort} = 80\ )\ ,\ \texttt{ACCEPT}\ );$

$(true\ ,\ \texttt{DROP})$

$R_{11} =$

$(\ \texttt{p.sIP} = 192.168.0.1 \wedge \texttt{p.dIP} \notin 192.168.0.0/24 \cup \{127.0.0.1,\ 151.15.185.183,\ 8.8.8.8\} \wedge \texttt{p.dPort} = 80\ ,$

$\texttt{NAT}(151.15.185.183 : \star,\ \star : \star))$

Infine non ci resta che tradurre la configurazione ottenuta nel linguaggio target (stadio 4. della pipeline):

```
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

-A OUTPUT -s 127.0.0.1,151.15.185.183 -d 151.15.185.183 --dport 22 -j DNAT --to 192.168.0.8
-A INPUT -s 192.168.0.0/24 -d 127.0.0.1,151.15.185.183 --dport 80 -j SNAT --to 151.15.185.183
-A POSTROUTING -s 192.168.0.1 --dport 80 -j SNAT --to 151.15.185.183

COMMIT

*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]

-A OUTPUT -s 127.0.0.1,151.15.185.183 -d 192.168.0.8 --dport 22 -j ACCEPT
-A OUTPUT -s 192.168.0.1 -d 8.8.8.8,192.168.0.0/24 --dport 80 -j DROP
-A OUTPUT -s 192.168.0.1 --dport 80 -j ACCEPT
-A INPUT -s 151.15.185.183 -d 127.0.0.1,151.15.185.183 --dport 80 -j ACCEPT

COMMIT
```

4.3 Domini della pipeline

Possiamo considerare ogni fase della pipeline come una funzione che data una rappresentazione del firewall ne produce una equivalente in un diverso dominio. In questo modo la transcompilazione può essere vista come una composizione di più funzioni. Individuiamo cinque domini per la rappresentazione dei firewall, ognuno corrispondente ad un diverso livello di astrazione. Ad ogni livello distinguiamo quali attributi di un firewall sono dovuti al sistema scelto e quali alla sua configurazione. Per semplificare la notazione, dato che abbiamo deciso di concentrarci unicamente sullo stato $s_{\texttt{NEW}}$, nelle definizioni omettiamo l'eventuale parametro relativo allo stato.

Livello 0 (firewall concreti): A questo livello di astrazione un sistema firewall è rappresentato da un processo di sistema e una configurazione da un file di configurazione.

Livello 1 (firewall IFCL): Un sistema firewall è un diagramma di controllo $\mathcal{C}$, una configurazione Σ è una coppia composta da un insieme di ruleset IFCL ρ e da un'assegnazione delle ruleset ai nodi del diagramma $c : Q \to \rho$.

Livello 2 (firewall IFCL normalizzati): Un sistema firewall è un diagramma di controllo C, una configurazione Σ è una coppia composta da un insieme di ruleset IFCL normalizzate ρ e da un'assegnazione delle ruleset ai nodi del diagramma $c : Q \to \rho$.

Livello 3 (firewall semiastratti): Un sistema firewall è un diagramma di controllo C, una configurazione è una funzione $f : Q \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ che assegna ad ogni nodo una funzione su pacchetti $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$.

Livello 4 (firewall astratti): Un firewall è la sua configurazione astratta, cioè una funzione che associa ogni pacchetto al modo in cui viene trattato $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$.

Indichiamo per ogni firewall, il livello di astrazione a cui ci stiamo riferendo attraverso un pedice: $\mathcal{F}_0$ è un firewall concreto, $\mathcal{F}_1$ è un firewall IFCL, $\mathcal{F}_2$ è un firewall IFCL normalizzato, $\mathcal{F}_3$ è un firewall semiastratto e $\mathcal{F}_4$ è un firewall astratto.

Abbiamo presentato i domini in ordine dal più concreto al più astratto. Le prime due fasi della pipeline, corrispondenti al frontend, operano traduzioni dal dominio dei firewall concreti a quello dei firewall astratti; le ultime due, corrispondenti al backend, operano nel verso opposto, dal dominio dei firewall astratti ritornano al dominio dei firewall concreti.

La funzione di traduzione dal primo al secondo livello è la *formalizzazione* del linguaggio di configurazione. Quella che traduce dal secondo al terzo è la *normalizzazione* (rimozione di `CALL` e `GOTO` e `RETURN`). Chiamiamo la traduzione dal terzo al quarto livello *semiastrazione* ed infine la traduzione dal quarto al quinto livello *composizione*.

Per realizzare le ultime due fasi della pipeline occorre definire delle funzioni di traduzione nel verso opposto: dall'astratto al concreto. La funzione di traduzione dal quinto al quarto livello è la *decomposizione*. Quella che traduce dal quarto al terzo è la *codifica*. Chiamiamo la traduzione dal terzo al secondo livello *rifattorizzazione* ed infine la traduzione dal secondo al primo livello *concretizzazione*.

Tutte le funzioni di traduzione, ad eccezione di quelle fra il primo ed il secondo livello, sono definite genericamente su ogni diagramma di controllo. Con questo approccio per estendere la teoria includendo il supporto a nuovi linguaggi è sufficiente definirne la formalizzazione IFCL.

La figura 4.2 illustra le varie fasi della pipeline in relazione con i domini di riferimento. Assumiamo di avere un file di configurazione `file.conf` per il sistema source $k \in \{\texttt{pf}, \texttt{iptables}, \texttt{ipfw}\}$ e di avere un sistema target k', allora gli stadi della pipeline di configurazione possono essere formalizzati come segue:

1. **formalizzazione** del *firewall concreto* come *firewall IFCL*
   - riceve $\mathcal{F}_0 = (k, \texttt{file.conf})$
   - prende $\mathcal{C}_k$ definito per il sistema se questo è supportato
   - calcola $\Sigma = for_k(\texttt{file.conf})$
   - restituisce $\mathcal{F}_1 = (\mathcal{C}_k, \Sigma)$
2. estrazione della semantica astratta del firewall
   (a) **normalizzazione** del *firewall IFCL*
      - riceve $\mathcal{F}_1 = (\mathcal{C}_k, \Sigma)$
      - calcola e restituisce $\mathcal{F}_2 = (\!|\ \mathcal{F}_1\ |\!)$
   (b) **semiastrazione** del *firewall normalizzato*
      - riceve $\mathcal{F}_2 = (\mathcal{C}_k, \Sigma_n)$, dove $\Sigma_n = (\rho, c)$
      - per ogni ruleset $R \in \rho$ calcola $(\!|R|\!)$
      - calcola f tale che per ogni $q \in Q_k$ vale $f(q) = (\!|c(q)|\!)(s_{\texttt{NEW}})$
      - restituisce $\mathcal{F}_3 = (\mathcal{C}_k, f)$
   (c) **composizione** del *firewall astratto*
      - riceve $\mathcal{F}_3 = (\mathcal{C}_k, f)$
      - calcola e restituisce $\mathcal{F}_4 = \odot\mathcal{F}_3$
3. generazione del *firewall IFCL* finale
   (a) **decomposizione** del *firewall astratto* in un *firewall semiastratto* del tipo target
      - riceve $\mathcal{F}'_4 = \mathcal{F}_4 : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ e k' il sistema target
      - prende $\mathcal{C}_{k'}$ definito per il sistema se questo è supportato
      - calcola $f' : Q_{k'} \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ tale che $\odot(\mathcal{C}_{k'}, f') = \mathcal{F}'_4$
      - restituisce $\mathcal{F}'_3 = (\mathcal{C}_{k'}, f')$
   (b) **codifica** del *firewall IFCL normalizzato*
      - riceve $\mathcal{F}'_3 = (\mathcal{C}_{k'}, f')$
      - per ogni funzione $\lambda \in \Lambda$ assegnato ai nodi da f (la definizione esatta di Λ è $\{f'(q) \mid q \in Q_{k'}\}$) genera una ruleset R tale che $(\!|R|\!)(s_{\texttt{NEW}}) = \lambda$ e colleziona queste ruleset nell'insieme ρ'
      - calcola $c' : Q_{k'} \to \rho'$ tale che per ogni $q \in Q_{k'}$ vale $(\!|c'(q)|\!)(s_{\texttt{NEW}}) = f'(q)$
      - restituisce $\mathcal{F}'_2 = (\mathcal{C}_{k'}, \Sigma')$, dove $\Sigma' = (\rho', c')$
   (c) **rifattorizzazione** del *firewall IFCL normalizzato* in un generico *firewall IFCL*
      - riceve $\mathcal{F}'_2 = (\mathcal{C}_{k'}, \Sigma')$
      - calcola una nuova configurazione Σ'_r tale che $(\!|\ (\mathcal{C}_{k'}, \Sigma'_r)\ |\!) = (\mathcal{C}_{k'}, \Sigma')$
      - restituisce $\mathcal{F}'_1 = (\mathcal{C}_{k'}, \Sigma'_r)$
4. **concretizzazione** del *firewall IFCL* in un *firewall concreto*
   - riceve $\mathcal{F}'_1 = (\mathcal{C}_{k'}, \Sigma'_r)$
   - calcola $\texttt{file.conf}' = con_{k'}(\mathcal{C}_{k'}, \Sigma'_r)$
   - restituisce $\mathcal{F}_0 = (k', \texttt{file.conf}')$

Dove nei passi 2.c e 3.a abbiamo usato la funzione di composizione $\odot$, che dato un firewall semiastratto restituisce il firewall astratto ottenuto componendo le funzioni associate ai nodi secondo la semantica denotazionale. Vale che $\odot(\mathcal{C}_k, f) = (\!|(\mathcal{C}_k, \Sigma)|\!)(s_{\texttt{NEW}})$, con $\Sigma = (\rho, c)$ se per ogni $q \in Q_k$ vale $f(q) = (\!|c(q)|\!)(s_{\texttt{NEW}})$[1] .

Formalmente la funzione $\odot$, per un firewall semiastratto $\mathcal{F} = (\mathcal{C}, f)$, è definita come $\odot^{\mathcal{F}}_{\emptyset}(q_i)$. Dove la funzione $\odot^{\mathcal{F}}_{I}(q)$ per un dato insieme di nodi I, è definita per ogni $q \in Q,\ q \neq q_f$ come:

$$\odot^{\mathcal{F}}_{I}\ (q)\ (p) = \begin{cases} \odot^{\mathcal{F}}_{I\cup\{q\}}(q')\ (p') \ltimes t & \text{se } t \neq \bot \wedge q' \notin I \\ \bot & \text{altrimenti} \end{cases}$$

$$\text{dove } t = f(q)(p) \text{ e se } t \neq \bot \text{ allora } p' = t(p) \text{ e } q' = \delta(q, p')$$

e per il nodo finale come:

$$\odot^{\mathcal{F}}_{I}\ (q_f)\ (p) = id$$

La funzione con_k è la funzione di concretizzazione del sistema k che data una configurazione IFCL Σ restituisce un file di configurazione `file.conf` nel linguaggio di configurazione di k. Questa è concettualmente la funzione inversa della funzione di formalizzazione for_k, tuttavia è bene osservare che non è necessariamente l'inversa dal punto di vista funzionale. Infatti la formalizzazione delle configurazioni prodotta dalla funzione for_k ha spesso una forma molto particolare (si prenda ad esempio quella di `ipfw` presentata nella sezione 2.2.3); non vogliamo limitarci alla possibilità di concretizzare solo configurazioni IFCL aventi una struttura così particolare. L'idea è quella di definire una funzione che per ogni configurazione IFCL per il diagramma di controllo del sistema k restituisca una configurazione nel linguaggio target "il più simile possibile"; esprimiamo questo concetto dicendo che la normalizzazione del firewall IFCL e di quello ottenuto applicando con_k e poi formalizzando di nuovo, devono essere uguali: $(\!|\ (\mathcal{C}_k, \Sigma)\ |\!) = (\!|\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ |\!)$. La forma di queste funzioni sarà descritta in maggior dettaglio nel capitolo 7.

In questa trattazione trascuriamo la fase di *rifattorizzazione*, assumendo che il firewall restituito da questa fase sia sempre quello ottenuto in input dello stadio di *codifica* ($\Sigma'_r = \Sigma'$). Si ricordi che è banalmente vero che ogni *firewall normalizzato* è anche un *firewall IFCL*. Inoltre, come già annunciato, abbiamo trascurato il comportamento dei pacchetti appartenenti a connessioni stabilite, concentrandoci unicamente sullo stato $s_{\texttt{NEW}}$.

Teorema 5 (Correttezza della pipeline)**.** *Sia* `file.conf` *un firewall concreto in uno qualunque dei sistemi* $k \in \{$`iptables`, `ipfw`, `pf`$\}$*. Il firewall target* `file.conf`$'$ *prodotto dalla pipeline di transcompilazione, per il sistema target* k'*, ha la stessa semantica del firewall source per quanto riguarda pacchetti non appartenenti a connessioni stabilite. Formalmente vale:*

$$(\!|\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ |\!)(s_{NEW}) \equiv (\!|\ (\mathcal{C}_{k'}, for_{k'}(\texttt{file.conf}'))\ |\!)(s_{NEW})$$

4.4 Domini sintetici

Quella che è stata presentata è la versione concettuale della pipeline, con le specifiche per ogni fase dei valori in input, di quelli calcolati e di quelli restituiti. Per alcune fasi abbiamo una descrizione accurata

[1] Si noti che abbiamo definito la semantica denotazionale per firewall IFCL normalizzati, non semiastratti, il passaggio intermedio dei firewall semiastratti ci è utile dal punto di vista pratico per l'implementazione degli algoritmi della pipeline e per la ricompilazione.

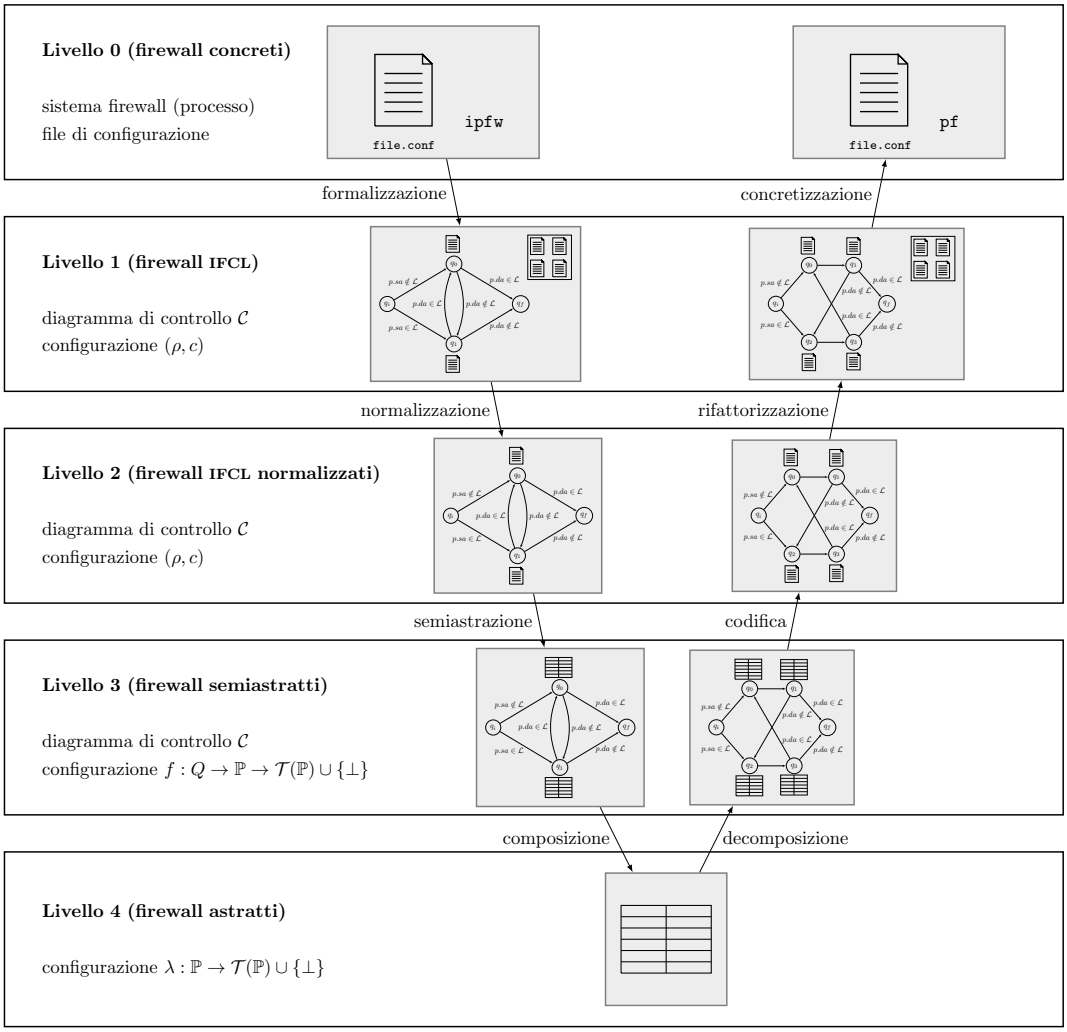


Figura 4.2: schema della pipeline di transcompilazione

dell'algoritmo da applicare, come per la fase di normalizzazione che sfrutta la procedura $(\!|\ \_\ |\!)$. Per dare un'implementazione della pipeline occorre definire le funzioni di traduzione del backend, delle quali abbiamo solo una descrizione non operativa.

La parte complicata da implementare è quella relativa alle fasi *2.b*, *2.c*, *3.a* e *3.b*. Questo perché abbiamo a che fare con firewall astratti e semiastratti, che sono basati su funzioni da pacchetti a trasformazioni $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$. Queste funzioni devono essere rappresentate in un modo che consenta di effettuare le operazioni della fase *3.a* e *3.b*. Una rappresentazione sintattica come quella prodotta dalla semantica denotazionale sarebbe molto difficile da maneggiare, quindi abbiamo optato per una rappresentazione esplicita, basata su una visione delle funzioni come insiemi di coppie (pacchetto, trasformazione). Ovviamente dato che abbiamo un enorme numero di input possibili per le funzioni, $\#\mathbb{P} = 2^{96}$, non è possibile calcolare e memorizzare le coppie una ad una. Ci affidiamo ad una rappresentazione sintetica dell'insieme delle coppie, basata su multicubi. Implementiamo la parte centrale della pipeline di transcompilazione attraverso un *algoritmo di sintesi* e un *algoritmo di generazione*, corrispondenti rispettivamente alle fasi *2.b* e *2.c*, e alle fasi *3.a* e *3.b*.

Definiamo una rappresentazione sintetica ed esplicita dei domini dei *firewall semiastratti* e *astratti*, che sarà la forma dei dati realmente gestiti dagli algoritmi. Stabiliamo ovviamente una corrispondenza fra domini di specifica e sintetici, definendo l'interpretazione teorica di ogni elemento dei domini sintetici e basandoci su di questa per dimostrare la correttezza degli algoritmi di sintesi e generazione.

Come abbiamo detto la rappresentazione sintetica si basa su multicubi. Il multicubo è una generalizzazione del concetto di cubo: dove un cubo di dimensione n può essere visto come il prodotto cartesiano di n intervalli, un multicubo è il prodotto cartesiano di n unioni di intervalli. Ad esempio, in R^3, il cubo di lato 1 con uno spigolo sull'origine degli assi e resto dei punti nel sottospazio di coordinate positive, può essere rappresentato come $[0,1] \times [0,1] \times [0,1]$, mentre è un multicubo l'insieme $([0,1] \cup [2,3]) \times [0,1] \times ([0,1] \cup [2,3] \cup [6,7])$.

Definizione 13 (Multicubo)**.** *Dato un insieme A, prodotto cartesiano di n insiemi $A = A_1 \times \cdots \times A_n$, un suo sottoinsieme $M \subseteq A$ è un multicubo se e solo se esistono n insiemi $M_1 \subseteq A_1, \cdots, M_n \subseteq A_n$ tali che $M = M_1 \times \cdots \times M_n$. Solitamente gli insiemi $M_1, \cdots, M_n$ sono rappresentati come unioni di intervalli disgiunti.*

In particolare un multicubo nel nostro sistema sarà un insieme di pacchetti $P \subseteq \mathbb{P}$ per il quale esistono $IP_s, IP_d \subseteq \mathbf{IP}$, $Port_s, Port_d \subseteq \mathbf{Port}$ e $Tag \subseteq \mathbf{Tag}$ tali che $P = IP_s \times Port_s \times IP_d \times Port_d \times Tag$. Denotiamo l'insieme dei multicubi in $\mathbb{P}$ con $\mathcal{M}(\mathbb{P})$, formalmente

$$\mathcal{M}(\mathbb{P}) = 2^{\mathbf{IP}} \times 2^{\mathbf{Port}} \times 2^{\mathbf{IP}} \times 2^{\mathbf{Port}} \times 2^{\mathbf{Tag}}$$

Un *firewall sintetizzato* di un qualunque livello è una rappresentazione sintetica del firewall, in cui ogni funzione $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ è espressa come insieme di coppie (P, t) dove $t \in \mathcal{T}(\mathbb{P})$ è una trasformazione e P è un *multicubo* di pacchetti che hanno t come immagine. Rappresentiamo a volte queste funzioni come una tabella con una riga per ogni coppia (P, t) in cui $t \neq \bot$ (lasciando implicito che il destino di tutti i pacchetti non rappresentati nella tabella è quello di essere scartati).

Per prima cosa definiamo formalmente cosa intendiamo per rappresentazione sintetizzata di una funzione su pacchetti, successivamente definiamo di conseguenza firewall semiastratti sintetizzati e firewall astratti sintetizzati (non essendo presenti funzioni da pacchetti a trasformazioni nei firewall ad altri livelli di astrazione, non ha senso parlare di loro versioni sintetizzate).

Definizione 14 (Funzione sintetizzata su pacchetti)**.** *Chiamiamo funzione sintetizzata su pacchetti un insieme di coppie $\widetilde{\lambda} \in 2^{\mathcal{M}(\mathbb{P}) \times \mathcal{T}(\mathbb{P}) \cup \{\bot\}}$ tale che l'insieme delle parti sinistre delle coppie di $\widetilde{\lambda}$ è una*

partizione di $\mathbb{P}$. *Formalmente:*

1. $\forall (P,t) \in \widetilde{\lambda}.\ P \neq \emptyset$
2. $\forall (P_1,t_1),(P_2,t_2) \in \widetilde{\lambda}.\ P_1 \cap P_2 = \emptyset$
3. $\bigcup_{(P,t)\in\widetilde{\lambda}} P = \mathbb{P}$

Chiamiamo *firewall semiastratto sintetizzato* una coppia $\widetilde{\mathcal{F}_3} = (\mathcal{C}, \widetilde{f})$ dove $\mathcal{C}$ è un diagramma di controllo e $\widetilde{f} : Q \to 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\bot\}}$ assegna ad ogni nodo q una funzione sintetizzata su pacchetti $\widetilde{\lambda}$. Allo stesso modo chiamiamo *firewall astratto sintetizzato* una funzione sintetizzata su pacchetti. Definiamo l'interpretazione di una funzione sintetizzata su pacchetti $i(\widetilde{\lambda}) : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ come segue:

$$i(\widetilde{\lambda})(p) = t \iff \exists (P,t) \in \widetilde{\lambda}.\ p \in P$$

Dato che l'insieme delle parti sinistre di $\widetilde{\lambda}$ è una partizione di $\mathbb{P}$, è immediato verificare che la definizione appena data non è ambigua e definisce davvero una funzione del tipo dichiarato. L'interpretazione di un firewall semiastratto sintetizzato $\widetilde{\mathcal{F}_3} = (\mathcal{C}, \widetilde{f})$ è un firewall semiastratto $\mathcal{F}_3 = (\mathcal{C}, f)$ dove per ogni nodo q, $f(q) = i(\tilde{f}(q))$. L'interpretazione di un firewall astratto sintetizzato $\widetilde{\mathcal{F}_4} = \widetilde{\lambda}$ è un firewall astratto $\mathcal{F}_4 = i(\widetilde{\lambda})$.

Gli algoritmi di sintesi e generazione del firewall, che realizzano le fasi di *semiastrazione* (*2.b*), *composizione* (*2.c*), *decomposizione* (*3.a*) e *codifica* (*3.b*), operano dunque sulla versione sintetizzata dei domini. Perché siano corretti è sufficiente che l'interpretazione dei firewall prodotti corrisponda sempre alla specifica della pipeline nello stadio di riferimento. Più precisamente: l'algoritmo di sintesi prende come input un firewall IFCL normalizzato e restituisce un firewall astratto sintetizzato $\tilde{\mathcal{F}}_4 \in 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\bot\}}$, mentre l'algoritmo di generazione prende come input un firewall astratto sintetizzato $\tilde{\mathcal{F}'}_4$ e restituisce un firewall IFCL normalizzato.

Per applicare il teorema 5 è sufficiente che, nell'implementazione della pipeline in cui i domini sintetizzati sostituiscono quelli normali, siano garantite le seguenti condizioni: (*i*) l'interpretazione del firewall astratto sintetizzato restituito dall'algoritmo di sintesi verifica le condizioni dell'output della fase *2.c* della pipeline, (*ii*) se l'interpretazione del firewall astratto sintetizzato in input all'algoritmo di generazione verifica le condizioni dell'input della fase *3.a* della pipeline, allora il firewall IFCL normalizzato prodotto dall'algoritmo di sintesi verifica le condizioni dell'output della fase *3.b* della pipeline. Se queste condizioni valgono allora vale anche il risultato del teorema 5. Formalmente, perché valga il risultato del teorema 5, che garantisce la conservazione della semantica del firewall, è sufficiente che siano verificate dagli algoritmi di sintesi e generazione le due condizioni seguenti:

1. Sia $\mathcal{F}_2$ un firewall normalizzato, sia $\tilde{\lambda}$ il risultato dell'algoritmo di sintesi con input $\mathcal{F}_2$, allora $(\!|\mathcal{F}_2|\!)(s_{\texttt{NEW}}) = i(\tilde{\lambda})$
2. Sia $\tilde{\lambda}$ un firewall astratto sintetizzato, sia $\mathcal{F}'_2$ il risultato dell'algoritmo di generazione con input $\tilde{\lambda}$ e sia $\lambda = i(\tilde{\lambda})$, allora che $(\!|\mathcal{F}'_2|\!)(s_{\texttt{NEW}}) = \lambda$

Capitolo 5

Algoritmo di sintesi

L'algoritmo di sintesi realizza le ultime due fasi dello stadio 2 della pipeline di transcompilazione: prende in input un firewall IFCL normalizzato e restituisce un firewall astratto, cioè una funzione $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, rappresentato come un *firewall astratto sintetizzato*. Dividiamo l'algoritmo in due parti, la prima realizza la *semiastrazione*, fase 2.b della pipeline, ovvero a partire dal firewall IFCL genera un firewall semiastratto in cui ad ogni nodo del diagramma di controllo è associata una funzione $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$. La seconda parta realizza la fase 2.c, componendo le funzioni associate ai nodi sulla base del diagramma di controllo, calcola una rappresentazione sintetica della funzione corrispondente alla semantica del firewall.

L'esatta implementazione degli algoritmi, e la scelta delle strutture dati verranno discussi nel capitolo 8.

5.1 Semiastrazione

La prima parte della sintesi, corrispondente alla semiastrazione, è realizzata nell'algoritmo 1 dalla funzione FIREWALL_SEMI_ABSTRACTION (riga 1), che attraverso la funzione RULESET_SYNTHESIS, dato un firewall normalizzato $\mathcal{F}_2$ in input, calcola la funzione su pacchetti corrispondente alla semantica della ruleset associata ad ognuno dei nodi del diagramma di controllo. L'output è un firewall semiastratto sintetizzato $\tilde{\mathcal{F}}_3$.

Nell'implementazione abbiamo usato la funzione SPLIT(P: insieme di pacchetti, t: trasformazione, ϕ: predicato), che divide l'insieme di pacchetti P in due parti: la prima, P_s, è composta dai pacchetti che dopo aver subito la trasformazione t verificano il predicato ϕ; la seconda, $\mathbf{P}_n$, da quelli che non lo verificano. Il primo insieme viene restituito com'è, il secondo insieme invece viene restituito come insieme di multicubi disgiunti non vuoti.

$$
\begin{aligned}
&\text{SPLIT}(\text{P, t, } \phi) = (P_s, \mathbf{P}_n) \\
&P_s = \{\ p \in P \mid \phi(t(p))\ \} \ \in \mathcal{M}(\mathbb{P}) \\
&\mathbf{P}_n = \{{P_n}^1, {P_n}^2, \dots {P_n}^m\} \in 2^{\mathcal{M}(\mathbb{P})} \quad \text{tale che} \quad 1.\ \forall P_n^i \in \mathbf{P}_n.\ P_n^i \neq \emptyset \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 2.\ \forall P_n^i, P_n^j \in \mathbf{P}_n.\ P_n^i \cap P_n^j = \emptyset \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 3.\ \bigcup_{P_n^i \in \mathbf{P}_n} P_n^i = \{\ p \in P \mid \neg\phi(t(p))\ \}
\end{aligned}
$$

Algorithm 1

1: **function** FIREWALL_SEMI_ABSTRACTION($\mathcal{F}_2$: firewall normalizzato)
2: $(\mathcal{C}, \Sigma) \leftarrow \mathcal{F}_2$
3: $(\rho, c) \leftarrow \Sigma$
4: $(Q, A, q_i, q_f) \leftarrow \mathcal{C}$
5: $\tilde{f} \leftarrow [\,]$
6: **for all** $q \in Q$ **do**
7: $\tilde{f}[q] \leftarrow$ RULESET_SYNTHESIS($c(q)$)
8: $\tilde{\mathcal{F}}_3 \leftarrow (\mathcal{C}, \tilde{f})$
9: **return** $\tilde{\mathcal{F}}_3$
10:
11: **function** RULESET_SYNTHESIS(R: ruleset)
12: **return** RULESET_SYNTHESIS_REC($\mathbb{P}$, R, id)
13:
14: **function** RULESET_SYNTHESIS_REC(P: insieme di pacchetti, R: ruleset, t: trasformazione)
15: **if** $R = [\,]$ **then return** $\{(P, t)\}$
16: $(\phi, Action) \cdot R' \leftarrow R$
17: $(P_s, \mathbf{P}_n) \leftarrow$SPLIT($P$, t, ϕ)
18: $\tilde{\lambda}' \leftarrow \bigcup_{P'_n \in \mathbf{P}_n}($ RULESET_SYNTHESIS_REC(P'_n, R', t) $)$
19: **if** $Action =$ `ACCEPT` **then**
20: **return** $\{(P_s, t)\} \cup \tilde{\lambda}'$
21: **if** $Action =$ `DROP` **then**
22: **return** $\{(P_s, \bot)\} \cup \tilde{\lambda}'$
23: **if** $Action =$ `NAT`(sn, dn) **then**
24: **return** $\{(P_s, tr_{nat}(dn, sn) \ltimes t)\} \cup \tilde{\lambda}'$
25: **if** $Action =$ `CHECK-STATE`(X) **then**
26: **return** RULESET_SYNTHESIS_REC(P, R', t)
27: **if** $Action =$ `MARK`(m) **then**
28: **return** RULESET_SYNTHESIS_REC(P_s, R', $(id : id,\ id : id,\ cost(m)) \ltimes t) \cup \tilde{\lambda}'$

Il motivo dietro a questa scelta è che, assumendo che P sia un multicubo, P_s è sicuramente un multicubo, mentre $\mathbf{P}_n$ può non esserlo. Daremo dettagli maggiori sulla forma di P_s e $\mathbf{P}_n$ alla fine di questa sezione.

La funzione FIREWALL_SEMI_ABSTRACTION itera sui nodi del diagramma di controllo $\mathcal{C}$ (riga 6), per ogni nodo q calcola la funzione sintetizzata $\tilde{\lambda}$ applicando la funzione RULESET_SYNTHESIS alla ruleset associata al nodo q da c, e aggiorna l'array f in modo tale che $f[q]$ restituisca $\tilde{\lambda}$ (riga 7). La funzione RULESET_SYNTHESIS, applicata ad una ruleset R, restituisce una funzione su pacchetti sintetizzata $\tilde{\lambda}$ tale che la sua interpretazione è uguale alla semantica della ruleset nello stato $s_{\texttt{NEW}}$, cioè tale che $i(\tilde{\lambda}) = (\!|R|\!)(s_{\texttt{NEW}})$. La sua definizione è simile a quella della semantica denotazionale di una ruleset, e sfrutta una funzione ricorsiva ausiliaria RULESET_SYNTHESIS_REC che calcola la sintesi di una parte di ruleset, dato l'insieme dei pacchetti che non sono stati gestiti dalle regole precedenti P e la trasformazione t già subita per via delle regole precedenti. Inizialmente la funzione viene chiamata con parametri corrispondenti all'insieme di tutti i pacchetti $\mathbb{P}$, l'intera ruleset R e la trasformazione identità id (riga 12).

La ricorsione è sulla porzione di ruleset ancora da analizzare R. Se è vuota allora tutti i pacchetti che ancora non sono stati gestiti saranno accettati con trasformazione t, restituiamo dunque un singoletto contenente (P, t) (riga 15). Abbiamo infatti assunto una politica di default ACCEPT per semplicità, l'estensione al caso generale è banale. Altrimenti si estrae la prima regola della ruleset $(\phi, Action)$ e si chiama R' il resto della ruleset (riga 16). Dividiamo P nei due insiemi di pacchetti P_s e $\mathbf{P}_n$ per mezzo della funzione SPLIT(P, t, ϕ) (riga 17). P_s è un multicubo e contiene tutti e soli i pacchetti in P che dopo aver subito la trasformazione t verificano la condizione ϕ della regola, $\mathbf{P}_n$ è un insieme di multicubi che contengono tutti gli altri pacchetti di P. Si calcola la funzione su pacchetti sintetizzata $\tilde{\lambda}'$ corrispondente alla semantica del resto della ruleset R' riguardo l'insieme di pacchetti $\mathbf{P}_n$, facendo una chiamata ricorsiva per ognuno dei multicubi nell'insieme (con trasformazione invariata) e unendo i risultati fra loro (riga 18). La prosecuzione dell'algoritmo dipende dal target della regola. Se il target è ACCEPT allora tutti i pacchetti che verificano la condizione sono accettati con trasformazione t e di tutti gli altri sono trattati secondo $\tilde{\lambda}'$ (riga 20). Il caso in cui *Action* è uguale a DROP è simile, i pacchetti in P_s sono associati a $\perp$ e $\mathbf{P}_n$ viene trattato alla stessa maniera con $\tilde{\lambda}'$ (riga 22). Se la ruleset comincia con una regola con target NAT allora l'insieme P_s viene associato a t aggiornata con il risultato della funzione tr_{nat} (riga 24), che restituisce la trasformazione specificata dai parametri del target e che abbiamo definito formalmente in 3.2. Il target CHECK-STATE è trattato in maniera molto particolare in quanto ci stiamo concentrando sullo stato $s_{\texttt{NEW}}$ e quindi nessun pacchetto appartiene a connessioni stabilite: nessuna pacchetto viene trasformato per effetto dello stato, P_s e $\mathbf{P}_n$ non servono, passiamo alla chiamata ricorsiva l'insieme P, tutti i suoi pacchetti sono gestiti sulla base del resto della ruleset R' (riga 26). Se il target è MARK(m) allora la valutazione di P_s procede con una chiamata ricorsiva in cui la trasformazione t viene aggiornata con la trasformazione relativa alla scrittura del campo *tag* con il valore m; come negli altri casi $\mathbf{P}_n$ viene gestito con $\tilde{\lambda}'$ (riga 28).

Osserviamo che per poter chiamare il risultato prodotto dall'algoritmo un firewall semiastratto sintetizzato occorre che l'array f rappresenti una funzione $Q \to 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\perp\}}$, perciò gli insiemi di pacchetti nella parte sinistra delle coppie associate ai nodi devono essere dei multicubi. L'insieme di tutti i pacchetti $\mathbb{P}$ è banalmente un multicubo, tutti gli altri insiemi di pacchetti coi quali lavoriamo sono prodotti dalla funzione SPLIT. Abbiamo detto che questa funzione restituisce una coppia $(P_s, \mathbf{P}_n)$, dove P_s è un multicubo e $\mathbf{P}_n$ è insieme di multicubi, se l'insieme P in input è un multicubo. Ogni insieme finito può essere espresso come unione di finiti multicubi, pertanto $\mathbf{P}_n$ non dà problemi da questo punto di vista. In generale non è vero, per una generica funzione ϕ, che l'insieme $\{p \in P \mid \phi(p, s_{\texttt{NEW}})\}$ è un multicubo se P lo è. Tuttavia dato che il predicato ϕ è la condizione di una regola IFCL, deve essere possibile scomporlo nella seguente maniera:

$$\phi(p,s) = \phi_{sIP}(p.sIP) \wedge \phi_{sPort}(p.sPort) \wedge \phi_{dIP}(p.dIP) \wedge \phi_{dPort}(p.dPort) \wedge \phi_{tag}(p.tag) \wedge \phi_s(p,s)$$

per qualche predicato ϕ_{sIP}, ϕ_{sPort}, ϕ_{dIP}, ϕ_{dPort} e ϕ_{tag}, e per un predicato $\phi_s(p,s)$ che può essere solo $\exists\alpha.\ p \vdash_s \alpha$ oppure $\neg\exists\alpha.\ p \vdash_s \alpha$, corrispondenti rispettivamente a $false$ e $true$ nei casi in questione, dato che abbiamo assunto che $s = s_{\texttt{NEW}}$. La scomposizione garantisce che S_n sia un multicubo, in quanto $\{p \in \mathbb{P} \mid \phi(p, s_{\texttt{NEW}})\}$ è un multicubo e l'intersezione di due multicubi è sempre un multicubo.

Questa scomposizione del predicato $\phi(p,s)$ suggerisce anche un metodo per calcolare $\mathbf{P}_n$ [1]: è sufficiente prendere i predicati $\phi_1(p) = \phi_{sIP}(p.sIP)$, $\phi_2(p) = \phi_{sPort}(p.sPort)$, $\phi_3(p) = \phi_{dIP}(p.dIP)$, $\phi_4(p) = \phi_{dPort}(p.dPort)$ e $\phi_5(p) = \phi_{tag}(p.tag)$ e combinarli per calcolare degli insiemi P_n^1, P_n^2, ..., $P_n^m \in \mathcal{M}(\mathbb{P})$ tali che:

1. $\forall P_n^i \in \mathbf{P}_n.\ P_n^i \neq \emptyset$

[1] Assumiamo qui che il predicato $\phi_s(p,s)$ possa essere ignorato in quanto valuta a $true$ (e quindi elemento neutro della congiunzione); nel caso contrario il congiunto stesso vale $false$, quindi $\mathbf{P}_n = \{P\}$ e $P_s = \emptyset$.

2. $\forall P_n^i, P_n^j \in \mathbf{P}_n.\ P_n^i \cap P_n^j = \emptyset$

3. $\bigcup_{P_n^i \in \mathbf{P}_n} P_n^i = \{\ p \in P \mid \neg\phi(t(p))\ \}$

Possiamo comporre i predicati in modo diverso per ottenere un risultato che soddisfi i requisiti, il più semplice probabilmente è quello di prendere fra gli elementi dell'insieme $\{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$ tutti i predicati diversi da *true*, chiamiamo questi predicati φ_1, φ_2, ..., φ_m. A seconda della condizione originale della regola, m può essere un qualunque numero naturale minore o uguale di 5. A questo punto possiamo costruire gli m multicubi nella seguente maniera:

$$\forall i \in \{1, \dots m\}.\ P_n^i = \{\ p \in P \mid \neg\varphi_i(p) \wedge \forall j < i.\ \varphi_j(p)\ \}$$

Si noti che per leggibilità non abbiamo trattato esplicitamente il caso in cui l'insieme P_s sia l'insieme vuoto, assumiamo che il controllo sia effettuato al momento dell'aggiunta della coppia (P_s, t) all'insieme $\tilde{\lambda}'$; la versione completa dell'algoritmo è immediatamente derivabile da quella presentata.

Il risultato dell'algoritmo è un firewall semiastratto sintetizzato la cui interpretazione corrisponde alle specifiche della pipeline di transcompilazione.

Teorema 6. *Sia $\mathcal{F}_2 = (\mathcal{C}, \Sigma)$ con $\Sigma = (\rho, c)$ il firewall normalizzato in input all'algoritmo di semiastrazione. L'algoritmo produce un firewall $\tilde{\mathcal{F}}_3 = (\mathcal{C}, \tilde{f})$ tale che*

- *$\tilde{\mathcal{F}}_3$ è un firewall semiastratto sintetizzato*
- *la funzione sintetizzata $\tilde{f}$ è tale che per ogni $q \in Q_k$ vale $i(\tilde{f})(q) = (\!|c(q)|\!)(s_{\texttt{NEW}})$*

5.2 Composizione

La fase di composizione (2.c) a partire dal firewall semiastratto sintetizzato $\tilde{\mathcal{F}}_3$ calcola il firewall astratto sintetizzato relativo, $\tilde{\mathcal{F}}_4$. Per semplificare questa fase ci basiamo su una versione del firewall avente diagramma di controllo aciclico. Sebbene sia possibile in teoria comporre le funzioni sintetizzate associate ai nodi su un grafo aciclico, questo richiederebbe di tener traccia per ogni multicubo che valutiamo, del percorso fatto all'interno del diagramma di controllo, in modo da scoprire quando siamo in presenza di un ciclo e da assegnare correttamente il valore $\perp$. In un firewall con diagramma di controllo aciclico invece possiamo effettuare la composizioni in ordine, dal nodo finale al primo, propagando all'indietro il valore ottenuto per un nodo e combinandolo con la semantica della ruleset associata al nodo precedente. A livello algoritmico questo si traduce nella possibilità di usare, per ogni nodo, una ricorsione sui nodi successori (cioè raggiunti da un arco uscente), e di avere automaticamente garanzia di terminazione.

Presentiamo dunque un algoritmo per la trasformazione di un firewall IFCL in una sua versione aciclica, garantendo la conservazione della semantica nel processo. Successivamente forniamo l'algoritmo di composizione in sé, che a partire da un firewall semiastratto sintetizzato con diagramma di controllo aciclico calcola la rappresentazione sintetizzata del firewall astratto.

5.2.1 Firwall aciclici

L'obiettivo è quello di calcolare la versione aciclica di un firewall, trasformando il diagramma di controllo $\mathcal{C}$ in una versione equivalente aciclica e aggiornando la funzione c di conseguenza. Un diagramma di controllo aciclico è un digramma di controllo in cui non esiste un percorso $\pi = q_i, \cdots, q_f$ tale che un nodo q compare più di una volta in π, dove in un percorso possiamo passare dal nodo q al

nodo, q' se e solo se esiste un arco (q, ψ, q') per qualche $\psi \neq false$. La versione aciclica di un firewall $\mathcal{F} = (\mathcal{C}, \Sigma)$ è un firewall $\mathcal{F}_u = (\mathcal{C}_u, \Sigma_u)$ tale che: il diagramma di controllo $\mathcal{C}_u$ è aciclico e i due firewall hanno la stessa semantica, ovvero $(\!|\mathcal{F}|\!) = (\!|\mathcal{F}_u|\!)$.

Consideriamo l'uguaglianza della semantica dei due firewall, $(\!|\mathcal{F}|\!) = (\!|\mathcal{F}_u|\!)$, come la congiunzione di due condizioni, la prima relativa alla simulazione del firewall $\mathcal{F}$ da parte di $\mathcal{F}'$, la seconda a quei pacchetti che in $\mathcal{F}$ percorrono un ciclo nel diagramma di controllo.

Il firewall prodotto dall'algoritmo *simula* il firewall in input, ovvero per ogni percorso legale, cioè aciclico, nel firewall originale esiste un percorso legale nel firewall prodotto tale che i predicati sugli archi e le ruleset associate ai nodi sono le stesse.

Definizione 15 (Simulazione di un firewall)**.** *Il firewall $\mathcal{F}'$ simula il firewall $\mathcal{F}$, scritto $\mathcal{F} \trianglerighteq \mathcal{F}'$, se e solo se $(\mathcal{F}, q_i, \{q_i\}) \trianglerighteq (\mathcal{F}', q_i', \{q_i'\})$ dove $\trianglerighteq$ è una relazione fra triple (firewall, nodo, insieme di nodi) definita come:*

$$
\begin{aligned}
(\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}', q', I') \iff\ & c(q) = c'(q') \wedge \\
& (q = q_f \iff q' = q_f') \wedge \\
& \forall q_1 \in\ Q \setminus I\ .\ \exists (q, \psi, q_1) \in A\ \Rightarrow \\
& \qquad (\exists q_1' \in Q' \setminus I', (q', \psi, q_1') \in A' \wedge (\mathcal{F}, q_1, I \cup \{q_1\}) \trianglerighteq (\mathcal{F}', q_1', I' \cup \{q_1'\}))
\end{aligned}
$$

dove $\mathcal{F} = (\mathcal{C}, \Sigma)$ e $\mathcal{F}' = (\mathcal{C}', \Sigma')$, con $\mathcal{C} = (Q, A, q_i, q_f)$, $\Sigma = (\rho, c)$ e $\mathcal{C}' = (Q', A', q_i', q_f')$, $\Sigma' = (\rho', c')$.

Chiamiamo *pacchetto ciclante* per un firewall un pacchetto la cui valutazione comporta la visita di uno stesso nodo più volte per un qualche stato.

Definizione 16 (Pacchetti ciclanti)**.** *L'insieme dei pacchetti ciclanti in un dato firewall $\mathcal{F}$, $pc(\mathcal{F})$, è l'insieme $\{p \in \mathbb{P} \mid \exists q \in Q, p', p'' \in \mathbb{P}, s \in S.\ (q_i, s, p) \rightarrow^* (q, s, p') \wedge (q, s, p') \rightarrow^+ (q, s, p'')\}$.*

Non è una sorpresa che due firewall simili abbiano la stessa semantica per quanto riguarda i pacchetti non ciclanti.

Teorema 7. *Se due firewall sono simili allora hanno semantica equivalente per quanto riguarda i pacchetti non ciclanti di $\mathcal{F}$, ovvero: $\mathcal{F} \trianglerighteq \mathcal{F}' \Longrightarrow (\ \forall p \notin pc(\mathcal{F}), s \in S.\ (\!|\mathcal{F}|\!)(s)(p) = (\!|\mathcal{F}'|\!)(s)(p)\ )$.*

L'idea è dunque quella di generare un firewall che simuli quello originale, replicando alcuni nodi in modo tale da "srotolare" il diagramma di controllo ed evitare che ci siano cicli. La procedura si basa sul fatto che ogni pacchetto in un diagramma di controllo può passare per ogni nodo al massimo una volta prima di essere scartato automaticamente. Occorre un trattamento ad hoc per i pacchetti ciclanti, che nel firewall originale verrebbero scartati. Perché il grafo risultante sia un diagramma di controllo occorre che da ogni nodo diverso da quello finale si possa sempre passare ad un nodo successivo, per ogni pacchetto. Pertanto i pacchetti ciclanti del firewall originale non possono bloccarsi nell'ultimo nodo prima della fine del ciclo, devono essere mandati verso un nodo che li gestisca in modo opportuno, cioè scartandoli. Creiamo per questo un nodo ad hoc, $q_\perp$, nel quale ridirigiamo tutti i pacchetti ciclanti del firewall originale e che è configurato in modo tale da scartare ogni pacchetto. Chiamiamo la ruleset che gli è assegnata $R_\perp$, questa è composta da un unica istruzione $(true, \texttt{DROP})$.

La funzione UNLOOP($\mathcal{F}$) dell'algoritmo 2 (riga 1) calcola la versione aciclica del firewall $\mathcal{F}$ attraverso la funzione ricorsiva UNLOOP_REC($\mathcal{F}$, q, q_u, I) che visita il diagramma nodo per nodo, seguendo gli archi e costruendo man mano il nuovo firewall. Il parametro $\mathcal{F}$ è il firewall originale, q è il nodo di $\mathcal{F}$ che stiamo considerando, q_u è il nodo del firewall prodotto che stiamo considerando e I è l'insieme dei nodi visitati per raggiungere q nel firewall originale, e ci serve a riconoscere i cicli. La funzione

UNLOOP_REC($\mathcal{F}$, q, q_u, I) restituisce una tripla (Q_u, A_u, c_u) dove Q_u sono i nodi del diagramma del firewall prodotto dall'algoritmo, A_u sono gli archi e c_u è un insieme di coppie che rappresenta la funzione di assegnamento di ruleset ai nodi (riga 5). Nella definizione della funzione UNLOOP_REC (riga 10), per prima cosa si inizializzano Q_u e A_u come insiemi vuoti e c_u come la funzione che associa a q_u la stessa ruleset associata a q nel firewall originale: q_u è una "copia" del nodo q del firewall originale (righe 13, 14 e 15). Si noti che nel firewall aciclico prodotto dall'algoritmo possono esistere più "copie" diverse del nodo q del firewall originale. Assumiamo di avere una funzione SUCCESSORI (q: nodo, A: archi) che dato un nodo e l'insieme degli archi restituisce la lista di coppie (q', ψ) tale che (q, ψ, q') appartiene ad A. Attraverso la funzione SUCCESSORI, itero sui nodi q' raggiungibili da q con predicato ψ, nel firewall $\mathcal{F}$ (riga 16): se il nodo causa un ciclo nel firewall (riga 17) allora aggiungo nel firewall prodotto il nodo $q_\perp$ e un arco con predicato ψ fra q_u e $q_\perp$, e associo al nodo $q_\perp$ la ruleset $R_\perp$ in c_u; per la definizione che abbiamo dato di diagramma di controllo non è possibile che il nodo $q_\perp$ non abbia archi uscenti, pertanto aggiungiamo anche un arco uscente con predicato *true* verso il nodo q_f (righe 18, 19 e 20).

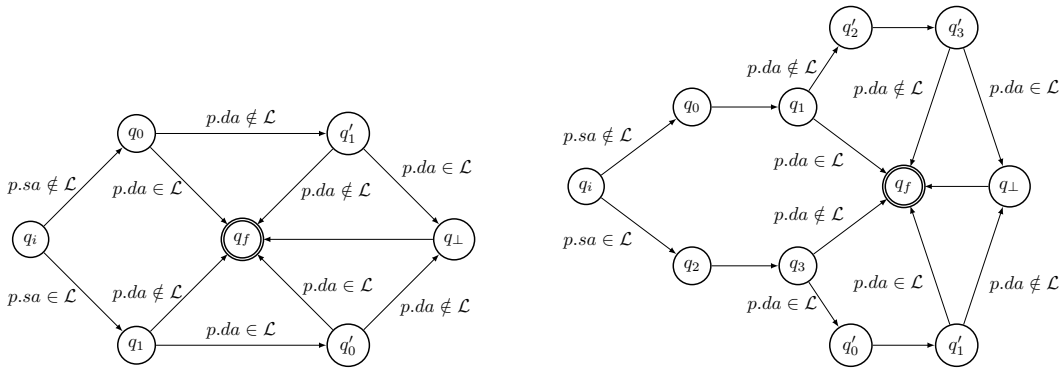
Se il nodo q' non causa un ciclo (riga 21) allora devo aggiungerlo al firewall prodotto, se $q' = q_f$ allora lo aggiungo com'è (riga 23), altrimenti genero e aggiungo una sua "copia" (riga 25), cioè un nuovo nodo che avrà nel diagramma finale lo stesso ruolo che q' ha nel diagramma originale per quanto riguarda i percorsi di cui ci stiamo occupando, ma che sarà separato dagli altri nodi del diagramma finale generati a partire dallo stesso nodo q'. La generazione di un nuovo nodo viene effettuata dalla funzione GENERA_NODO() che assumiamo generi sempre nodi differenti. Aggiungo il nuovo nodo q'_u a Q e copio in A_u l'arco (q, ψ, q'), che diventa (q_u, ψ, q'_u) (righe 26 e 27). Effettuo una chiamata ricorsiva sul nodo q' in $\mathcal{F}$ e su q'_u sul firewall che sto costruendo, aggiungendo il nodo q' in I (riga 28). Unisco il risultato della chiamata ricorsiva alla tripla (Q_u, A_u, c_u) ottenuta dalle iterazioni precedenti e proseguo con il successivo fra i nodi raggiungibili da q nel firewall iniziale (righe 29, 30 e 31). Alla fine restituisco la tripla (Q_u, A_u, c_u).

La semantica del firewall aciclico prodotto è equivalente a quella del firewall di partenza.

Teorema 8. *Sia $\mathcal{F}$ un firewall IFCL, sia $\mathcal{F}_u$ il risultato dell'applicazione della funzione UNLOOP al firewall $\mathcal{F}$:*

1. *$\mathcal{F}_u$ è un firewall IFCL aciclico*
2. *$\mathcal{F} \trianglerighteq \mathcal{F}_u$*
3. *$\forall p \in pc(\mathcal{F}), s \in S.\ (\!|\mathcal{F}_u|\!)(s)(p) = \perp$*

Il diagramma di controllo di `iptables` non contiene cicli, perciò non c'è necessità di applicare l'algoritmo ai firewall `iptables`. I diagrammi di controllo di `pf` e `ipfw`, invece, contengono cicli: la loro versione aciclica, restituita dall'algoritmo, è mostrata in figura 5.1b e 5.1a.



(a) `ipfw` (b) `pf`

Figura 5.1: Versione aciclica dei diagrammi di controllo

Algorithm 2

```
1:  function UNLOOP(F: firewall)
2:      (C, Σ) ← F
3:      (ρ, c) ← Σ
4:      (Q, A, q_i, q_f) ← C
5:      (Q_u, A_u, c_u) ← UNLOOP_REC(F, q_i, q_i, {q_i})
6:      C_u ← (Q_u, A_u, q_i, q_f)
7:      Σ_u ← (ρ ∪ {R_⊥}, c_u)
8:      return (C_u, Σ_u)
9:
10: function UNLOOP_REC(F: firewall, q: nodo, q_u: nodo , I: insieme di nodi)
11:     (C, ρ, c) ← F
12:     (Q, A, q_i, q_f) ← C
13:     Q_u ← ∅
14:     A_u ← ∅
15:     c_u ← {(q_u, c(q))}
16:     for all (q', ψ) ∈ SUCCESSIORI(q, A) do
17:         if q' ∈ I then
18:             Q_u ← Q_u ∪ {q_⊥}
19:             A_u ← A_u ∪ {(q_u, ψ, q_⊥), (q_⊥, true, q_f)}
20:             c_u ← c_u ∪ {(q_⊥, R_⊥)}
21:         else
22:             if q' = q_f then
23:                 q'_u ← q_f
24:             else
25:                 q'_u ← GENERA_NODO( )
26:             Q_u ← Q_u ∪ {q'_u}
27:             A_u ← A_u ∪ {(q_u, ψ, q'_u)}
28:             (Q', A', c' ← UNLOOP_REC(F, q', q'_u, I ∪ {q'})
29:             A_u ← A_u ∪ A'
30:             Q_u ← Q_u ∪ Q'
31:             c_u ← c_u ∪ c'
32:     return (Q_u, A_u, c_u)
```

5.2.2 Algoritmo di composizione

La seconda parte dell'algoritmo di sintesi realizza la fase di composizione della pipeline. In questa fase le funzioni sintetizzate associate ai vari nodi del diagramma di controllo sono composte fra loro in modo da ottenere la semantica del firewall source. L'algoritmo 3 realizza questa fase attraverso la funzione COMPOSITION (riga 1) che dato un firewall semiastratto sintetizzato $\tilde{\mathcal{F}}_3$ restituisce un firewall astratto sintetizzato equivalente $\tilde{\mathcal{F}}_4$. L'algoritmo termina solo se il firewall in input è aciclico, condizione garantita dall'applicazione dell'algoritmo 2.

La funzione COMPOSITION è definita attraverso la funzione ricorsiva COMPOSITION_REC che ha come parametri il firewall semiastratto sintetizzato $\tilde{\mathcal{F}}_3$ e il nodo q a partire dal quale calcolare la semantica, inizialmente q_i. La funzione COMPOSITION_REC($\tilde{\mathcal{F}}_3$, q) (riga 6) calcola la semantica del firewall considerando il diagramma di controllo solo dal nodo q in avanti. La semantica di un firewall composto da un unico nodo, il nodo finale q_f, è la semantica della ruleset associata al nodo stesso, ovvero avendo assunto che in ogni firewall IFCL debba valere $c(q_f) = R_\epsilon$, la funzione sintetizzata $(\mathbb{P}, id)$ (riga 9). Se invece non siamo nel nodo q_f, prediamo la funzione sintetizzata su pacchetti associata a q, $\tilde{\lambda} = f(q)$ (riga 10). Inizializziamo $\tilde{\lambda}_q$ con le coppie $(P, \perp)$ di $\tilde{\lambda}$ (riga 11). Per ogni nodo q' raggiungibile da q con arco etichettato dal predicato ψ:

- usando la funzione FILTER prendiamo $\tilde{\lambda}_{(q,q')}$, la parte di $\tilde{\lambda}$ relativa ai pacchetti che attraversano l'arco, cioè che verificano ψ (riga 13);
- calcoliamo $\tilde{\lambda}_{q'}$, la semantica del firewall dal nodo q' in poi, attraverso una chiamata ricorsiva COMPOSITION_REC($\tilde{\mathcal{F}}_3$, q') (riga 14);
- concateniamo in maniera opportuna le due funzioni sintetizzate $\tilde{\lambda}_{(q,q')}$ e $\tilde{\lambda}_{q'}$ usando la funzione CONCAT (riga 15).

Collezioniamo gli insiemi di coppie ottenuti per ognuno dei nodi raggiungibili q' e creiamo e restituiamo la funzione sintetizzata finale.

Nell'implementazione abbiamo assunto di avere a disposizione le funzioni DROPPER, FILTER e CONCAT. La funzione DROPPER($\tilde{\lambda}$: funzione su pacchetti sintetizzata) dato un insieme di coppie (multicubo di pacchetti, trasformazione) restituisce l'insieme ottenuto togliendo tutte le coppie in cui la trasformazione è diversa da DROP.

$$\text{DROPPER}(\tilde{\lambda}) = \{(P, \perp) \mid (P, \perp) \in \tilde{\lambda}\}$$

La funzione FILTER($\tilde{\lambda}$: funzione su pacchetti sintetizzata, ψ: predicato) dato un insieme di coppie (multicubo di pacchetti, trasformazione) e un predicato, filtra ogni termine sinistro delle coppie del primo parametro $\tilde{\lambda}$ togliendo tutti i pacchetti che non verificano ψ dopo aver subito la trasformazione specificata dalla parte destra della coppia.

$$\text{FILTER}(\tilde{\lambda}, \psi) = \{\ (P', t) \mid (P, t) \in \tilde{\lambda}\ \wedge\ P' = t^{-1}(\psi(t(P)))\ \wedge\ P' \neq \emptyset\ \}$$

Dove ci siamo permessi un piccolo abuso di notazione e abbiamo scritto $t(P)$ per $\{t(p) \mid p \in P\}$ e $\psi(P)$ per $\{p \in P \mid \psi(p)\}$; e con $t^{-1}(P'')$ intendiamo la preimmagine di P'' nell'insieme P tramite t. In ordine quindi per calcolare P': applichiamo la trasformazione t, filtriamo i pacchetti secondo il predicato ψ e infine torniamo indietro dalla trasformazione. Formalmente $t^{-1}(\psi(t(P))) = \{p \in P \mid \psi(t(p))\}$.

Per poter chiamare il risultato della funzione FILTER occorre che le parti sinistre delle coppie P' dell'insieme prodotto siano multicubi di pacchetti. Assumendo che $\tilde{\lambda}$ sia una funzione sintetizzata, se il predicato ψ è scomponibile in una congiunzione di predicati secondo la formula

$$\psi(p) = \psi_{sIP}(p.sIP) \wedge \psi_{sPort}(p.sPort) \wedge \psi_{dIP}(p.dIP) \wedge \psi_{dPort}(p.dPort) \wedge \psi_{tag}(p.tag)$$

Algorithm 3

function COMPOSITION($\tilde{\mathcal{F}}_3$: firewall semiastratto sintetizzato aciclico)
    $(\mathcal{C}, \tilde{f}) \leftarrow \tilde{\mathcal{F}}_3$
    $(Q, A, q_i, q_f) \leftarrow \mathcal{C}$
    return COMPOSITION_REC($\tilde{\mathcal{F}}_3$, q_i)

function COMPOSITION_REC($\tilde{\mathcal{F}}_3$: firewall semiastratto sintetizzato aciclico, q: nodo)
    $(\mathcal{C}, \tilde{f}) \leftarrow \tilde{\mathcal{F}}_3$
    $(Q, A, q_i, q_f) \leftarrow \mathcal{C}$
    if $q = q_f$ **then return** $\{(\mathbb{P}, id)\}$
    $\tilde{\lambda} \leftarrow \tilde{f}(q)$
    $\tilde{\lambda}_q \leftarrow$ DROPPER($\tilde{\lambda}$)
    for all $(q', \psi) \in$ SUCCESSORI(q, A) **do**
        $\tilde{\lambda}_{(q,q')} \leftarrow$ FILTER($\tilde{\lambda}$, ψ)
        $\tilde{\lambda}_{q'} \leftarrow$ COMPOSITION_REC($\tilde{\mathcal{F}}_3$, q')
        $\tilde{\lambda}_q \leftarrow \tilde{\lambda}_q \cup$ CONCAT($\tilde{\lambda}_{(q,q')}$, $\tilde{\lambda}_{q'}$)
    return $\tilde{\lambda}_q$

allora il risultato della funzione è una funzione sintetizzata su pacchetti, ovvero se $P \in \mathcal{M}(\mathbb{P})$ allora $t^{-1}(\psi(t(P))) \in \mathcal{M}(\mathbb{P})$.

La funzione CONCAT($\tilde{\lambda}_1$: funzione su pacchetti sintetizzata, $\tilde{\lambda}_2$: funzione su pacchetti sintetizzata) concatena due funzioni sintetizzate espresse come coppie (multicubo di pacchetti, trasformazione). L'idea è quella di prendere il primo insieme e calcolare per ogni coppia (P_1, t_1) quali coppie (P_2, t_2) sono associate a $t(P_1)$ nel secondo insieme e restituire delle coppie in cui il primo termine è composto dagli elementi di P_1 che trasformati da t_1 appartengono a P_2 (preimmagine di P_2 in P_1 tramite t_1) e il secondo termine è la trasformazione t_1 aggiornata con t_2.

$$\text{CONCAT}(\tilde{\lambda}_1, \tilde{\lambda}_2) = \{\ (P', t_2 \ltimes t_1) \mid (P_1, t_1) \in \tilde{\lambda}_1 \wedge (P_2, t_2) \in \tilde{\lambda}_2 \wedge P' = t_1^{-1}(P_2 \cap t_1(P_1)) \wedge P' \neq \emptyset\ \}$$

Anche qui occorre controllare che il risultato della funzione sia effettivamente una funzione su pacchetti sintetizzata, cioè che le parti sinistre delle coppie che contiene siano in effetti multicubi di pacchetti. Non occorre nessuna assunzione in quanto se $P1$ e $P2$ sono multicubi di pacchetti e t è una trasformazione su pacchetti allora anche $t_1^{-1}(P_2 \cap t_1(P_1))$ è un multicubo.

L'algoritmo di sintesi è corretto: dato un firewall IFCL normalizzato aciclico, calcola un firewall astratto sintetizzato la cui interpretazione corrisponde alla semantica del firewall nello stato $s_{\texttt{NEW}}$.

Teorema 9 (Correttezza dell'algoritmo di sintesi)**.** *thm:sintesi Sia $\mathcal{F}_2$ un firewall IFCL normalizzato aciclico, sia $\tilde{\mathcal{F}}_4$ il firewall astratto sintetizzato restituito dall'algoritmo di sintesi, allora vale:*

$$i(\tilde{\mathcal{F}}_4) = (\!|\mathcal{F}_2|\!)(s_{NEW})$$

5.3 Esempio di sintesi in pf

Presentiamo un esempio di sintesi di un firewall `pf`. Si consideri la seguente configurazione:

```
rdr from any to 151.15.185.183 port 22 -> 192.168.0.8
nat from 192.168.0.0/24 to ! 192.168.0.0/24 -> 151.15.185.183
```

$p \in \mathbb{P}$					$t \in \mathcal{T}(\mathbb{P})$				
sIP	$sPort$	dIP	$dPort$	tag	sIP	$sPort$	dIP	$dPort$	tag
$*$	$*$	$*$	$*$	$*$	id	id	id	id	id

(a) $\tilde{\lambda}_{\epsilon}$

$p \in \mathbb{P}$					$t \in \mathcal{T}(\mathbb{P})$				
sIP	$sPort$	dIP	$dPort$	tag	sIP	$sPort$	dIP	$dPort$	tag
192.168.0.8/24	$*$	¬192.168.0.8/24	$*$	$*$	$cost$(151.15.185.183)	id	id	id	id
¬192.168.0.8/24	$*$	$*$	$*$	$*$	id	id	id	id	id
$*$	$*$	192.168.0.8/24	$*$	$*$	id	id	id	id	id

(b) $\tilde{\lambda}_{snat}$

$p \in \mathbb{P}$					$t \in \mathcal{T}(\mathbb{P})$				
sIP	$sPort$	dIP	$dPort$	tag	sIP	$sPort$	dIP	$dPort$	tag
$*$	$*$	151.15.185.183	22	$*$	id	id	$cost$(192.168.0.8)	id	id
$*$	$*$	¬151.15.185.183	$*$	$*$	id	id	id	id	id
$*$	$*$	$*$	¬22	$*$	id	id	id	id	id

(c) $\tilde{\lambda}_{dnat}$

$p \in \mathbb{P}$					$t \in \mathcal{T}(\mathbb{P})$				
sIP	$sPort$	dIP	$dPort$	tag	sIP	$sPort$	dIP	$dPort$	tag
151.15.185.183	$*$	$*$	80	$*$	id	id	id	id	id
192.168.0.0/24	$*$	192.168.0.0/24	$*$	$*$	id	id	id	id	id
¬192.168.0.0/24	$*$	192.168.0.8	22	$*$	id	id	id	id	id

(d) $\tilde{\lambda}_{filt}$

Tabella 5.1: Rappresentazione tabellare delle funzioni su pacchetti sintetizzate.

```
block all
pass from any to 192.168.0.8 port 22
pass from 192.168.0.0/24 to 192.168.0.0/24
pass from 151.15.185.183 to any port 80
```

La traduzione in IFCL della configurazione è la seguente:

$$\rho = \{R_{snat}, R_{dnat}, R_{finp}, R_{finpr}, R_{fout}, R_{foutr}, R_{\epsilon}\}$$

$$c(q_i) = c(q_f) = R_{\epsilon},\ c(q_0) = R_{snat},\ c(q_1) = R_{finp}\ c(q_2) = R_{dnat}, c(q_3) = R_{fout}$$

$R_{snat} =$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \notin 192.168.0.0/24\ ,\ \texttt{NAT}(151.15.185.183 : \star, \star : \star))$

$R_{dnat} =$

$(\texttt{p.dIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 22\ ,\ \texttt{NAT}(\star : \star, 192.168.0.8 : \star))$

$R_{finp} =$

$(true\ ,\ \texttt{GOTO}(R_{finpr}))$

$R_{fout} =$

$(true\ ,\ \texttt{GOTO}(R_{foutr}))$

$R_{finpr} = R_{foutr}$

$(\texttt{p.sIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 80\ ,\ \texttt{ACCEPT}))$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \in 192.168.0.0/24\ ,\ \texttt{ACCEPT})$

$p \in \mathbb{P}$					$t \in \mathcal{T}(\mathbb{P})$				
sIP	$sPort$	dIP	$dPort$	tag	sIP	$sPort$	dIP	$dPort$	tag
$*$	$*$	192.168.0.8	22	$*$	id	id	id	id	id
192.168.0.0/24	$*$	192.168.0.0/24	$*$	$*$	id	id	id	id	id
151.15.185.183	$*$	$*$	80	$*$	id	id	id	id	id
192.168.0.1	$*$	$\neg$192.168.0.0/24	80	$*$	$cost$(151.15.185.183)	id	id	id	id
$*$	$*$	151.15.185.183	22	$*$	id	id	$cost$(192.168.0.8)	id	id

Tabella 5.2: Rappresentazione tabellare del firewall astratto sintetizzato.

$(\texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22\ ,\ \texttt{ACCEPT})$

$(true, \texttt{DROP})$

Il grafico è quello di `pf`, in figura 2.1c. La normalizzazione consiste semplicemente nella rimozione delle GOTO.

$$\rho = \{R_{snat}, R_{dnat}, R_{finp}, R_{fout}, R_\epsilon\}$$
$$c(q_i) = c(q_f) = R_\epsilon,\ c(q_0) = R_{snat},\ c(q_1) = R_{finp}\ c(q_2) = R_{dnat}, c(q_3) = R_{fout}$$

$R_{snat} =$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \notin 192.168.0.0/24\ ,\ \texttt{NAT}(151.15.185.183 : \star, \star : \star))$

$R_{dnat} =$

$(\texttt{p.dIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 22\ ,\ \texttt{NAT}(\star : \star, 192.168.0.8 : \star))$

$R_{finp} = R_{fout} =$

$(\texttt{p.sIP} = 151.15.185.183 \wedge \texttt{p.dPort} = 80\ ,\ \texttt{ACCEPT}))$

$(\texttt{p.sIP} \in 192.168.0.0/24 \wedge \texttt{p.dIP} \in 192.168.0.0/24\ ,\ \texttt{ACCEPT})$

$(\texttt{p.dIP} = 192.168.0.8 \wedge \texttt{p.dPort} = 22\ ,\ \texttt{ACCEPT})$

$(true, \texttt{DROP})$

Il passaggio alla versione aciclica del firewall può essere fatto in ogni momento, in questo caso per semplicità lo rimandiamo a dopo la fase di semiastrazione. Applichiamo l'algoritmo 1 e calcoliamo la versione semiastratta sintetizzata del firewall IFCL, con configurazione f tale che:

$$f(q_i) = f(q_f) = \tilde{\lambda}_\epsilon,\ c(q_0) = \tilde{\lambda}_{snat}, c(q_2) = \tilde{\lambda}_{dnat},\ c(q_1) = c(q_3) = \tilde{\lambda}_{filt}$$

Dove le funzioni su pacchetti sintetizzate $\tilde{\lambda}_\epsilon$, $\tilde{\lambda}_{snat}$, $\tilde{\lambda}_{dnat}$ e $\tilde{\lambda}_{filt}$ sono presentate nella tabella 5.1 come tabelle in cui ogni riga corrisponde ad una coppia (multicubo, trasformazione) e dove le coppie in cui la parte destra è $\perp$ sono lasciate implicite.

A questo punto, grazie all'algoritmo 2, passiamo alla versione aciclica: il diagramma di controllo diventa come quello in figura 5.1b, e la configurazione f diventa:

$$f(q_i) = f(q_f) = \tilde{\lambda}_\epsilon,\ c(q_0) = c(q_0') = \tilde{\lambda}_{snat}, c(q_2) = c(q_2') = \tilde{\lambda}_{dnat},\ c(q_1) = c(q_1') = c(q_3) = c(q_3') = \tilde{\lambda}_{filt}$$

Possiamo infine comporre le tabelle grazie all'algoritmo 3: assumendo che l'insieme $\mathcal{L}$ degli indirizzi locali sia $\{127.0.0.1, 192.168.0.1, 151.15.185.183\}$, la funzione su pacchetti sintetizzata risultante $\tilde{\lambda}$ è espressa dalla tabella 5.2.

Capitolo 6

Espressività dei sistemi firewall

Come abbiamo anticipato, i diversi sistemi firewall non sono ugualmente espressivi. La forma dei diagrammi di controllo e i vincoli legati al linguaggio di configurazione source inducono delle condizioni sulle configurazioni esprimibili, rendendo alcune funzioni su pacchetti impossibili da codificare.

Ci interessa caratterizzare dunque per ogni sistema quali sono le configurazioni esprimibili e studiarle nel dominio dei firewall astratti, per poter meglio confrontare l'espressività di diversi sistemi fra loro. Ci basiamo in effetti sulla parte della pipeline di transcompilazione relativa alla sintesi: per prima cosa caratterizziamo i diversi sistemi supportati a livello di firewall IFCL, e quindi di firewall semastratti. Successivamente a partire da questi cerchiamo di derivare quali firewall astratti possono essere prodotti dalla fase di composizione.

Formalmente, dato un sistema firewall $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$, chiamiamo $Conf_k$ l'insieme dei file di configurazione legali per il sistema k. L'insieme dei firewall astratti esprimibili nel sistema k è dunque:

$$\Lambda_k = \{(\!|\mathbb{C}\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ \mathbb{D}|\!) \mid \texttt{file.conf} \in Conf_k\}$$

Sostanzialmente chiamiamo *esprimibili* da un sistema i firewall le configurazioni ottenibili a partire da file di configurazione legali per quel sistema, tramite le funzioni di formalizzazione e trasformazione della parte di sintesi della pipeline.

Per prima cosa introduciamo l'assegnamento di etichette, un meccanismo per rappresentare le configurazioni IFCL esprimibili di un sistema e ne diamo una descrizione per i sistemi supportati. Successivamente descriviamo l'insieme dei firewall astratti esprimibili da un sistema, sulla base dei firewall IFCL esprimibili dallo stesso.

In questo capitolo e nel successivo trascuriamo completamente il campo *tag* dei pacchetti, questo perché il target MARK, nei diversi sistemi firewall supportati, è soggetto a vincoli diversi e difficili da modellare usando IFCL. Solo per questo capitolo e per il successivo ridefiniamo dunque $\mathbb{P}$ come $\mathbf{IP} \times \mathbf{Port} \times \mathbf{IP} \times \mathbf{Port}$ e $\mathcal{T}(\mathbb{P})$ come $\mathcal{T}(\mathbf{IP}) \times \mathcal{T}(\mathbf{Port}) \times \mathcal{T}(\mathbf{IP}) \times \mathcal{T}(\mathbf{Port})$.

6.1 Configurazioni IFCL esprimibili

A livello di IFCL ogni firewall è una coppia composta da un diagramma di controllo e da una configurazione. Per ogni sistema k è definito il relativo diagramma di controllo $\mathcal{C}_k$ e la funzione for_k che traduce un file di configurazione source nella sua formalizzazione IFCL. Non tutte le possibili configurazioni IFCL Σ sono esprimibili dal linguaggio di configurazione di k, ovvero la funzione for_k non è una funzione surgettiva. Chiamiamo Γ_k l'insieme delle configurazioni IFCL ottenibili dalla formalizzazione

di file di configurazione legali del sistema k, formalmente:

$$\Gamma_k = \{for_k(\texttt{file.conf}) \mid \texttt{file.conf} \in Conf_k\}$$

L'insieme Γ_k potrebbe essere studiato partendo dai file di configurazione legali per i sistemi `iptables`, `pf` e `ipfw`, e delle relative funzioni di formalizzazione for_k che abbiamo definito nel capitolo 2. Tuttavia in mancanza di una formalizzazione ufficiale di questi linguaggi di configurazione, preferiamo caratterizzare direttamente l'insieme Γ_k all'interno del dominio di IFCL.

Inoltre, dato l'obiettivo finale è quello di caratterizzare Λ_k, e dato che la forma dei firewall prodotti da for_k è piuttosto complessa (si pensi ad esempio al modo con il quale abbiamo rappresentato i firewall `ipfw`, presentato nella sezione 2.2.3) e anche decisamente arbitraria in quanto avremmo potuto definire delle funzioni di formalizzazione diverse ma equivalenti (ad esempio decidendo di modellare in modo diverso i salti di `ipfw`), evitiamo di caratterizzare esattamente Γ_k, concentrandoci immediatamente sull'insieme dei firewall normalizzati ottenibili da esso. Nel seguito chiamiamo esprimibili da un sistema k le configurazioni IFCL ottenibili dalla normalizzazione di una configurazione in Γ_k.

Dobbiamo quindi tener conto del fatto che non tutti gli assegnamenti di ruleset ai nodi del diagramma di controllo di un sistema sono esprimibili. In particolare risulta, nei sistemi analizzati, che non sia possibile associare ruleset contenenti determinate operazioni ad alcuni dei nodi. In alcuni nodi è possibile scartare pacchetti, in altri no; in alcuni i pacchetti possono solo essere scartati o accettati senza modifiche mentre altri possono trasformare i pacchetti modificando alcuni campi e così via.

Assumendo che ogni nodo possa accettare i pacchetti senza modificarli diciamo che l'insieme delle operazioni interessanti sono $\{SNAT, DNAT, DROP\}$. Dove $SNAT$ e $DNAT$ indicano che un nodo può modificare rispettivamente i campi sIP e $sPort$, e dIP e $dPort$ di un pacchetto; e $DROP$ indica che nel nodo i pacchetti possono essere scartati.

Forniamo uno strumento per caratterizzare l'insieme delle configurazioni IFCL esprimibili da un sistema k sulla base delle operazioni che si possono associare ai nodi del suo diagramma di controllo. Un assegnamento di etichette per un diagramma di controllo è una funzione che assegna a ogni nodo del diagramma un sottoinsieme delle etichette $\{SNAT, DNAT, DROP\}$; le etichette associate a un nodo specificano quali operazioni possono essere effettuate sui pacchetti che passano da quel nodo.

Definizione 17 (Assegnamento di etichette a un diagramma di controllo)**.** *Un assegnamento di etichette a un diagramma di controllo $\mathcal{C} = (Q, A, q_i, q_f)$ è una funzione $v : Q \to 2^{\{SNAT,DNAT,DROP\}}$ che associa un insieme di etichette $L \in 2^{\{SNAT,DNAT,DROP\}}$ a ogni nodo del diagramma di controllo.*

Una configurazione normalizzata $\Sigma = (\rho, c)$ di un diagramma di controllo $\mathcal{C}$ è legale secondo un assegnamento di etichette v a $\mathcal{C}$ se e solo se per ogni nodo $q \in Q$ la ruleset associatagli $c(q)$ contiene solo target permessi dalle etichette $v(q)$. Scriviamo $\Sigma \models v$ per dire che la configurazione Σ è legale per l'assegnamento di etichette v.

Definizione 18 (Configurazione IFCL normalizzata legale)**.** *La configurazione normalizzata Σ di un firewall IFCL con diagramma di controllo $\mathcal{C} = (Q, A, q_i, q_f)$ è legale secondo un assegnamento di etichette v, scritto $\Sigma \models v$, se e solo se $\forall q \in Q.\ c(q) \models v(q)$, dove $R_\epsilon \models L$ per ogni insieme di etichette L, altrimenti sia $R = r \cdot R'$:*

- *se $r = (\phi, \textsc{accept})$, $r = (\phi, \textsc{check-state}(\_))$ o $r = (\phi, \textsc{mark}(\_))$ allora $R \models L$ se e solo se $R' \models L$*
- *se $r = (\phi, \textsc{drop})$ allora $R \models L$ se e solo se $DROP \in L$ e $R' \models L$*
- *se $r = (\phi, \textsc{nat}(ip : port, \star : \star))$ allora $R \models L$ se e solo se $SNAT \in L$ e $R' \models L$*

- *se* $r = (\phi, \textsc{nat}(\star : \star, ip : port))$ *allora* $R \models L$ *se e solo se* $DNAT \in L$ *e* $R' \models L$
- *se* $r = (\phi, \textsc{nat}(ip_1 : port_1, ip_2 : port_2))$ *allora* $R \models L$ *se e solo se* $SNAT \in L$ *e* $DNAT \in L$ *e* $R' \models L$

Chiamiamo $\mathbb{M}_2(\mathcal{C}, v)$ l'insieme delle configurazioni normalizzate Σ per il diagramma di controllo $\mathcal{C}$, legali secondo v, dove l'assegnamento di etichette v è definito su $\mathcal{C}$ [1]. Formalmente $\mathbb{M}_2(\mathcal{C}, v) = \{\Sigma \mid \Sigma \models v\}$

Assumiamo il seguente risultato (senza poterlo dimostrare dato che non abbiamo formalmente introdotto sintassi e semantica dei linguaggi di configurazione dei firewall concreti).

Ipotesi 1. *Per ogni sistema firewall* $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$*, esiste un assegnamento di etichette* v_k *per il diagramma di controllo* $\mathcal{C}_k$ *tale che una configurazione* IFCL Σ *è esprimibile in* k *se e solo se* Σ_n *è legale secondo* v_k*, dove* $\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D} = (\mathcal{C}_k, \Sigma_n)$*. Formalmente:*

$$\mathbb{M}_2(\mathcal{C}_k, v_k) = \{\Sigma_n \mid \Sigma \in \Gamma_k \wedge \mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D} = (\mathcal{C}_k, \Sigma_n)\}$$

Assegnamento di etichette per i sistemi supportati

Presentiamo gli assegnamenti di etichette che caratterizzano le configurazioni IFCL esprimibili nei sistemi supportati: `iptables`, `pf` e `ipfw`. Nella figura 6.1 mostriamo i diagrammi dei sistemi firewall, con le etichette assegnate a ognuno dei nodi, dove l'assenza di etichette indica che l'insieme vuoto è assegnato al nodo.

In `iptables` l'assegnamento è rappresentato in figura 6.1a.

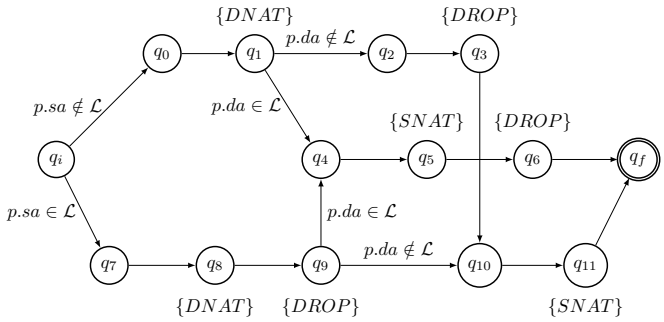
$$\begin{aligned}
&v_{\texttt{iptables}}(q_i) = v_{\texttt{iptables}}(q_f) = \emptyset \\
&v_{\texttt{iptables}}(q_0) = v_{\texttt{iptables}}(q_2) = v_{\texttt{iptables}}(q_4) = v_{\texttt{iptables}}(q_7) = v_{\texttt{iptables}}(q_{10}) = \emptyset \\
&v_{\texttt{iptables}}(q_1) = v_{\texttt{iptables}}(q_8) = \{DNAT\} \\
&v_{\texttt{iptables}}(q_3) = v_{\texttt{iptables}}(q_6) = v_{\texttt{iptables}}(q_9) = \{DROP\} \\
&v_{\texttt{iptables}}(q_5) = v_{\texttt{iptables}}(q_{11}) = \{SNAT\}
\end{aligned}$$

Ogni nodo nel diagramma di controllo di `iptables` corrisponde a una coppia (tabella, chain); le etichette associate a un nodo dipendono dalla tabella in quanto solo nella tabella `NAT` è possibile effettuare traduzioni di indirizzi, e solo nella tabella `FILTER` è possibile scartare dei pacchetti. Inoltre nel caso della tabella `NAT`, a seconda della chain associata al nodo possiamo fare solo NAT sugli indirizzi di origine o su quelli di destinazione.

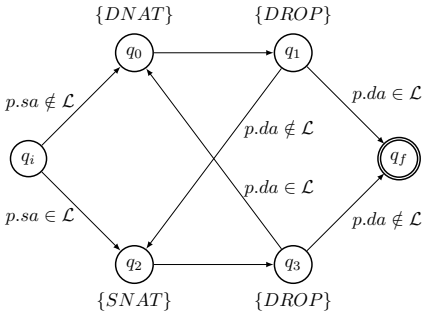
Nel linguaggio di configurazione di `pf` non è presente una nozione simile a quella di ruleset, tutte le regole fanno parte di un unico insieme che viene letto più volte. È la funzione $for_{\texttt{pf}}$ che realizza la divisione delle regole fra i nodi del diagramma di controllo $\mathcal{C}_{\texttt{pf}}$. Il modo in cui le regole vengono divise fa sí che solo nei nodi q_1 e q_3 si possa scartare un pacchetto, solo nel nodo q_0 si possa modificare l'indirizzo di destinazione e in q_2 quello di origine. L'assegnamento di etichette, rappresentato in 6.1b è dunque:

$$\begin{aligned}
&v_{\texttt{pf}}(q_i) = v_{\texttt{pf}}(q_f) = \emptyset \\
&v_{\texttt{pf}}(q_1) = v_{\texttt{pf}}(q_3) = \{DROP\} \\
&v_{\texttt{pf}}(q_0) = \{DNAT\} \\
&v_{\texttt{pf}}(q_2) = \{SNAT\}
\end{aligned}$$

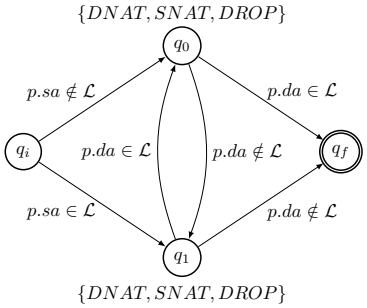
[1] Il pedice in $\mathbb{M}_2(\mathcal{C}, v)$ dipende dal dominio di riferimento, che è quello dei firewall IFCL normalizzati.



(a) `iptables`



(b) `pf`



(c) `ipfw`

Figura 6.1: Diagrammi di controllo dei sistemi supportati con etichette associate ai nodi.

Il diagramma di controllo di `ipfw` è minimale: come in `pf`, le regole di filtro e di traduzione sono mischiate, ma a differenza di `pf` l'ordine di applicazione non dipende dal tipo della regola. Le regole sono divise solo sulla base dell'eventuale presenza delle keyword `in` e `out`. Ogni tipo di regola può essere etichettata con entrambe le keyword, pertanto nei due nodi q_0 e q_1 possiamo trovare ogni genere di regola, le etichette assegnate sono le seguenti:

$$v_{\texttt{ipfw}}(q_i) = v_{\texttt{ipfw}}(q_f) = \emptyset$$
$$v_{\texttt{ipfw}}(q_0) = v_{\texttt{ipfw}}(q_1) = \{SNAT, DNAT, DROP\}$$

La figura 6.1c mostra le etichette a fianco dei relativi nodi del diagramma di controllo.

6.2 Configurazioni astratte esprimibili

Dati i vincoli sulle configurazioni IFCL esprimibili da un sistema k, rappresentati dall'assegnamento di etichette sul diagramma di controllo, vogliamo studiare quali sono i firewall astratti esprimibili dallo stesso sistema.

Definizione 19 (Firewall astratto esprimibile)**.** *L'insieme dei firewall astratti esprimibili nel sistema k è:*

$$\Lambda_k = \{(\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!) \mid \Sigma \in \Gamma_k\}$$

Dato che abbiamo assunto nell'ipotesi 1 che la normalizzazione dei firewall con configurazioni in Γ_k sia esattamente l'insieme di firewall con configurazioni legali secondo v_k per il sistema k, $\mathbb{M}_2(\mathcal{C}_k, v_k)$,

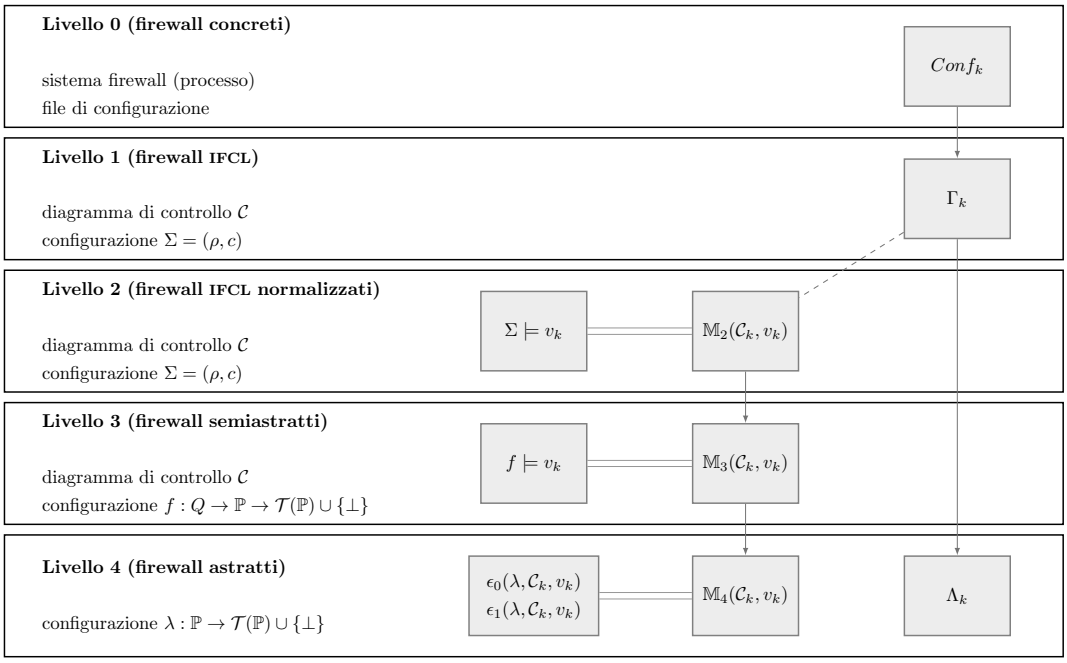


Figura 6.2: Schema delle configurazioni esprimibili e legali nei vari livelli di astrazione: le relazioni che abbiamo assunto per ipotesi sono rappresentate tratteggiate, quelle che valgono per definizione sono delle frecce e quelle che abbiamo dimostrato sono delle linee doppie.

possiamo basarci su questi per calcolare Λ_k. Chiamiamo $\mathbb{M}_3(\mathcal{C}_k, v_k)$ le configurazioni dei firewall risultanti dalla semiastrazione dei firewall normalizzati con diagramma di controllo $\mathcal{C}_k$ e configurazione $\Sigma \in \mathbb{M}_2(\mathcal{C}_k, v_k)$. I firewall astratti ottenuti combinando i firewall semiastratti con diagramma di controllo $\mathcal{C}_k$ e configurazione $f \in \mathbb{M}_3(\mathcal{C}_k, v_k)$ saranno infine chiamati $\mathbb{M}_4(\mathcal{C}_k, v_k)$. Dall'ipotesi 1 segue che $\mathbb{M}_4(\mathcal{C}_k, v_k) = \Lambda_k$.

L'obiettivo è quello di caratterizzare gli insiemi $\mathbb{M}_3(\mathcal{C}_k, v_k)$ ed $\mathbb{M}_4(\mathcal{C}_k, v_k)$ sulla base dell'assegnamento di etichette v_k, in modo da avere una descrizione degli insiemi Λ_k per $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$ tramite la quale sia agevole:

- confrontare l'espressività di due sistemi firewall
- verificare se una funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ appartiene o meno a Λ_k

In realtà per il confronto ci accontentiamo al momento di definire una condizione necessaria per l'uguaglianza di espressività di due sistemi, facile da verificare e sufficientemente espressiva, che ci permette di dimostrare che `pf` è meno espressivo di `iptables` e `ipfw`.

Per descrivere l'insieme $\mathbb{M}_3(\mathcal{C}_k, v_k)$ definiamo, per ogni possibile insieme di etichette assegnato a un nodo, l'insieme delle possibili funzioni $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ che possono essere assegnate a quel nodo. Usiamo la stessa notazione $f \models v$ per dire che l'assegnamento f di funzioni $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ ai nodi del diagramma di controllo del firewall semiastratto è legale secondo l'assegnamento di etichette v. Scriviamo sempre $\lambda \models L$ per dire che la funzione λ può essere assegnata a un nodo a cui sia assegnato l'insieme di etichette L.

Per prima cosa definiamo per ogni etichetta $l \in \{SNAT, DNAT, DROP\}$ l'insieme $\Lambda_{\Box l}$ delle funzioni $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ che corrispondono alla semantica delle ruleset che non possono essere assegnate a un nodo che non abbia l'etichetta l. Formalmente $\Lambda_{\Box l} = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \lambda \models L \Rightarrow$

$l \in L\}$.

$$\Lambda_{\Box SNAT} = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \exists p \in \mathbb{P}.\ \lambda(p) = t \wedge t \neq \bot \wedge (t.sIP \neq id \vee t.sPort \neq id)\}$$
$$\Lambda_{\Box DNAT} = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \exists p \in \mathbb{P}.\ \lambda(p) = t \wedge t \neq \bot \wedge (t.dIP \neq id \vee t.dPort \neq id)\}$$
$$\Lambda_{\Box DROP} = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \exists p \in \mathbb{P}.\ \lambda(p) = \bot\}$$

Definizione 20 (Configurazione semiastratta legale)**.** *La configurazione f di un firewall semiastratto con diagramma di controllo $\mathcal{C} = (Q, A, q_i, q_f)$ è legale secondo un assegnamento di etichette v, scritto $f \models v$, se e solo se $\forall q \in Q.\ c(q) \models v(q)$ dove per una funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ e un insieme di etichette L vale $\lambda \models L$ se e solo se λ non appartiene a nessuno degli insiemi $\Lambda_{\Box l}$ per una etichetta l che non appartiene a L. Formalmente:*

$$\lambda \models L \iff \lambda \notin \bigcup_{l \notin L} \Lambda_{\Box l}$$

Teorema 10. *L'insieme delle configurazioni semiastratte di un diagramma di controllo $\mathcal{C} = (Q, A, q_i, q_f)$, legali secondo un assegnamento di etichette v, è l'insieme delle configurazioni di firewall ottenute dalla semiastrazione di firewall IFCL normalizzati legali secondo v.*

$$\mathbb{M}_3(\mathcal{C}, v) = \{f : Q \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid f \models v\}$$

Per caratterizzare $\mathbb{M}_4(\mathcal{C}, v)$ ci basiamo sulla coppia di predicati $\epsilon_0(\lambda, \mathcal{C}, v)$ e $\epsilon_1(\lambda, \mathcal{C}, v)$. I due predicati sono i seguenti:

- $\epsilon_0(\lambda, \mathcal{C}, v)$, che chiamiamo *fattibilità locale*, è vero se ogni pacchetto p viene trattato dal firewall secondo una trasformazione t tale che l'assegnamento di t a p è singolarmente esprimibile dal sistema, ovvero

$$\forall p \in \mathbb{P}.\ \exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p)$$

- $\epsilon_1(\lambda, C, v)$, che chiamiamo *compatibilità*, è vero se, assumendo che gli assegnamenti di trasformazioni ai pacchetti siano singolarmente esprimibili dal sistema, esiste una configurazione che li verifica tutti contemporaneamente, ovvero

$$(\forall p \in \mathbb{P}.\ \exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p)) \Rightarrow$$
$$(\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ \forall p \in \mathbb{P}.\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p))$$

La prima proprietà riguarda l'espressività del diagramma di controllo e dell'assegnamento di etichette in senso stretto. Ad esempio un pacchetto con origine locale e destinazione non locale in `pf` non può subire una trasformazione che modifichi i campi destinazione in quanto non attraversa mai, nel diagramma di controllo $\mathcal{C}_{\texttt{pf}}$, un nodo etichettato con $DNAT$. Per questo motivo un firewall astratto che preveda una trasformazione di questo tipo per uno di questi pacchetti non si può esprimere in `pf`.

La seconda proprietà riguarda la coerenza fra i diversi assegnamenti (pacchetto, trasformazione) del firewall astratto da esprimere ed è legata al fatto che il comportamento dei nodi del diagramma di controllo nei confronti di un dato pacchetto possa dipendere unicamente dalla forma attuale del pacchetto stesso. Se nel sistema in esame per accettare il pacchetto p_1 con trasformazione t_1 è necessario che la funzione λ_n associata al nodo q_n accetti il pacchetto p' con trasformazione t'_1 e per accettare il pacchetto p_2 con trasformazione t_2 è necessario che la stessa funzione λ_n accetti il pacchetto p' con trasformazione $t'_2 \neq t'_1$, allora un firewall astratto $\mathcal{F}_4$ tale che $\mathcal{F}_4(p_1) = t_1$ e $\mathcal{F}_4(p_2) = t_2$ non è esprimibile dal sistema.

Teorema 11. *Un firewall astratto λ è legale secondo un diagramma di controllo $\mathcal{C}$ e un assegnamento di etichette v se e solo se valgono $\epsilon_0(\lambda, C, v)$ e $\epsilon_1(\lambda, C, v)$.*

$$\mathbb{M}_4(\mathcal{C}, v) = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \epsilon_0(\lambda, C, v) \wedge \epsilon_1(\lambda, C, v)\}$$

La figura 6.2 sintetizza le relazioni fra i vari insiemi e predicati che abbiamo definito, evidenziando sia il livello di astrazione di ciascun elemento, sia la natura della relazione: se vale per definizione allora è rappresentata come una freccia, se dipende da un'assunzione che abbiamo fatto è una linea tratteggiata, infine se è dimostrata la rappresentiamo come una linea doppia. Gli elementi sono distribuiti su cinque righe, a seconda del livello di astrazione, e su tre colonne: la colonna più a destra contiene le configurazioni esprimibili dal sistema; la colonna centrale contiene le configurazioni legali secondo l'assegnamento di etichette v_k; la colonna a sinistra contiene i predicati che usiamo per caratterizzare le configurazioni legali a un dato livello di astrazione.

In pratica ϵ_0 ed ϵ_1 sono i vincoli che abbiamo sul firewall astratto dati i vincoli imposti sul diagramma di controllo del firewall IFCL. Per controllare se una funzione λ sia esprimibile da un sistema, e per poter confrontare l'espressività di diversi sistemi, vogliamo esprimere i due predicati senza fare riferimento a $\mathbb{M}_3(\mathcal{C}, v)$.

Per fare questo studiamo i percorsi che attraversano il diagramma di controllo $\mathcal{C}$. Chiamiamo $\ddot{\Pi}(\mathcal{C}) = \{\ddot{\pi}_1, \ddot{\pi}_2, \ddot{\pi}_3 \ldots, \ddot{\pi}_m\}$ i percorsi che partono da q_i e terminano in q_f, questi sono i percorsi che un pacchetto può fare prima di essere accettato. Chiamiamo $\bar{\Pi}(\mathcal{C}) = \{\bar{\pi}_1, \bar{\pi}_2, \bar{\pi}_3 \ldots, \bar{\pi}_n\}$ i percorsi che partono da q_i e terminano in un q_n dove q_n o è etichettato con $DROP$ oppure compare più di una volta nel percorso, questi sono i percorsi che un pacchetto può fare prima di essere scartato. Chiamiamo $\Pi(\mathcal{C}) = \{\pi_1, \pi_2, \pi_3, \ldots, \pi_{n+m}\}$ i percorsi che attraversano il diagramma di controllo (di qualunque tipo siano, ovvero: $\Pi(\mathcal{C}) = \ddot{\Pi}(\mathcal{C}) \cup \bar{\Pi}(\mathcal{C})$).

Ridefiniamo le proprietà sulla base dei percorsi:

- la *fattibilità locale* è verificata se e solo se per ogni pacchetto p l'insieme dei percorsi che p può seguire all'interno del diagramma di controllo permette una trasformazione $t = \lambda(p)$.
- la *compatibilità* è verificata se e solo se per ogni coppia (p, t) tale che $p \in \mathbb{P}$ e $t = \lambda(p) \in \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, le condizioni che dobbiamo imporre sulle funzioni associate ai nodi del diagramma affinché uno dei percorsi del diagramma di controllo trasformi p secondo t non sono in contraddizione con il resto della funzione λ.

Se esprimiamo il firewall astratto come una tabella allora possiamo pensare a $\epsilon_0(\lambda, C, v)$ come a una condizione di validità delle singole righe, mentre a $\epsilon_1(\lambda, C, v)$ come a una condizione riguardo il rapporto fra più righe.

6.2.1 Fattibilità locale

La fattibilità locale di un firewall astratto λ rispetto a un diagramma di controllo $\mathcal{C}$ e un assegnamento di etichette v può essere espressa considerando i percorsi del diagramma come segue:

$$\epsilon_0(\lambda, \mathcal{C}, v) = \forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ (p, \lambda(p)) \in E(\pi, \mathcal{C}, v)$$

dove la funzione E, dato un percorso π, un diagramma di controllo $\mathcal{C}$ e un assegnamento di vincoli v, restituisce un insieme di coppie (p, t) tale che esiste una configurazione coerente con il diagramma di controllo $\mathcal{C}$ e con le etichette v per la quale il pacchetto p percorre il percorso π nel diagramma e viene trattato secondo t. Si tratta sostanzialmente di una funzione che associa a ogni percorso in

un diagramma di controllo la sua espressività, in relazione ai vincoli espressi dall'assegnamento di etichette. Formalmente:

$$E(\pi, \mathcal{C}, v) = \{\ (p,t) \mid \exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f))\ (p) = t \wedge \Delta((\mathcal{C}, f), p) = \pi\ \}$$

Nella definizione abbiamo usato la funzione Δ che dato un firewall associa a ogni pacchetto $p \in \mathbb{P}$ il percorso π che questi compie all'interno del diagramma di controllo. Poiché i diagrammi di controllo sono deterministici la funzione Δ è ben definita; e può essere espressa in modo standard sulla base della funzione δ che dato un nodo del diagramma di controllo e un pacchetto restituisce il prossimo nodo a essere visitato.

Caratterizziamo la funzione E in termini delle etichette del diagramma di controllo. Per prima cosa definiamo una funzione $\ell(\pi, \mathcal{C}, v)$ che dato un percorso π all'interno di un diagramma di controllo $\mathcal{C}$ etichettato secondo v, restituisce l'insieme delle etichette presenti sui nodi del percorso.

$$\ell(\pi, \mathcal{C}, v) = \bigcup_{q \in Q} v(q) \qquad \text{dove } \ \mathcal{C} = (Q, A, q_i, q_f)$$

Definiamo una funzione μ che dato un insieme di etichette restituisce l'insieme dei campi di un pacchetto che possono essere modificati da un nodo da esso etichettato.

$$\mu(L) = \mu_{SNAT}(L) \cup \mu_{DNAT}(L)$$

$$\mu_{SNAT}(L) \begin{cases} \{sIP, sPort\} & \text{se } SNAT \in L \\ \emptyset & \text{altrimenti} \end{cases} \qquad \mu_{DNAT}(L) \begin{cases} \{dIP, dPort\} & \text{se } DNAT \in L \\ \emptyset & \text{altrimenti} \end{cases}$$

A questo punto, per verificare che una coppia (p,t) sia esprimibile da un percorso, per ogni campo del pacchetto controlliamo che il valore del campo verifichi i predicati sugli archi attraversati dal pacchetto. Dal nodo iniziale, fino al primo nodo in cui avviene una modifica al valore di quel campo, è il valore dal campo del pacchetto p a dover verificare le condizioni; dall'ultimo nodo in cui avviene una modifica al valore del campo fino al nodo finale, è invece il valore del campo ottenuto applicando t al pacchetto p a dover verificare i predicati sugli archi. Nel caso di un pacchetto scartato occorre verificare solo la prima parte, cioè occorre verificare per ogni campo del pacchetto che questo verifichi i predicati sugli archi, dal nodo iniziale fino al primo nodo capace di modificare quel campo (dopo la trasformazione, che è arbitraria, ogni predicato diverso da $false$ può essere verificato da ogni pacchetto iniziale dopo una trasformazione ad hoc).

Per prima cosa ci serve una notazione per esprimere una condizione sull'arco fra due nodi q, q', che predichi unicamente sul valore associato a un campo specifico $x \in \{sIP, sPort, dIP, dPort\}$. Se ψ è il predicato sull'arco fra q e q', cioè $(q, \psi, q') \in A$, allora scriviamo $\psi^x_{q,q'}(a)$ per la versione di ψ che predica unicamente sul campo x del pacchetto, formalmente:

$$\psi^x_{q,q'}(a) = \exists p \in \mathbb{P}.\ p.x = a \wedge \psi(p)$$

Possiamo ora definire la funzione $E(\pi, \mathcal{C}, v)$.

$$E(\pi, \mathcal{C}, v) = \begin{cases} \{\ (p, \bot) \mid \forall x \in \{sIP, sPort, dIP, dPort\}.\ \overrightarrow{V}_x(p.x, \pi, \mathcal{C}, v)\ \} & \text{se } \pi \in \bar{\Pi}(\mathcal{C}) \\ \{\ (p,\ t) \mid t \neq \bot\ \wedge \forall x \in \{sIP, sPort, dIP, dPort\}.\ \overleftrightarrow{V}_x(p.x, t.x, \pi, \mathcal{C}, v)\ \} & \text{se } \pi \in \ddot{\Pi}(\mathcal{C}) \end{cases}$$

dove

$$\overrightarrow{V}_x(a, \pi, \mathcal{C}) = \begin{cases} \psi^x_{q,q'}(a) \wedge \overrightarrow{V}_x(a, q' \cdot \pi', \mathcal{C}) & se\ \pi = q \cdot q' \cdot \pi' \\ true & altrimenti \end{cases}$$

$$\overrightarrow{V}_x(a,\pi,\mathcal{C},v) = \begin{cases} \psi^x_{q,q'}(a) \wedge \overrightarrow{V}_x(a,q'\cdot\pi',\mathcal{C},v) & se\ \pi = q\cdot q'\cdot\pi' \wedge x \notin \mu(v(q)) \\ true & altrimenti \end{cases}$$

$$\overleftarrow{V}_x(a,\pi,\mathcal{C},v) = \begin{cases} \psi^x_{q,q'}(a) \wedge \overleftarrow{V}_x(a,\pi'\cdot q',\mathcal{C},v) & se\ \pi = \pi'\cdot q'\cdot q \wedge x \notin \mu(v(q)) \\ true & altrimenti \end{cases}$$

$$\overleftrightarrow{V}_x(a,ta,\pi,\mathcal{C},v) = \begin{cases} \overrightarrow{\overrightarrow{V}}_x(a,\pi,\mathcal{C}) & \text{se } ta = id \\ x \in \mu(\ell(\pi)) \wedge \overrightarrow{V}_x(a,\pi,\mathcal{C},v) \wedge \overleftarrow{V}_x(a',\pi,\mathcal{C},v) & \text{se } ta = cost(a') \end{cases}$$

Data una coppia (p,t), per verificare se questa appartiene a $E(\pi,\mathcal{C},v)$, come prima cosa verifichiamo se il percorso è uno che porta a scartare il pacchetto o ad accettarlo: nel primo caso, dato che $\pi \in \bar{\Pi}$, t deve essere $\perp$ e il pacchetto p deve poter percorrere il percorso π. Esprimiamo la seconda condizione attraverso il predicato $\overrightarrow{V}_x(a,\pi,\mathcal{C},v)$, che è verificato da un valore a per il campo x, da un percorso π e da un diagramma di controllo $\mathcal{C}$ se e solo se a verifica tutte le condizioni presenti sugli archi del percorso π a partire dal nodo iniziale q_i, fino al primo nodo capace di modificare il campo x, cioè tale che $x \in \mu(v(q))$. Questo perché, dal primo nodo capace di modificare il campo in poi, ogni condizione sugli archi può essere verificata da un assegnamento ad hoc: la forma che assume il pacchetto non è importante infatti, dato che alla fine verrà scartato.

Nel caso invece in cui il percorso sia in $\ddot{\Pi}$, il pacchetto p può essere trattato secondo t se e solo se $t \neq \perp$ e per ogni campo x il valore del pacchetto per quel campo, subite le trasformazioni apportate dai vari nodi, verifica tutte le condizioni sugli archi del percorso, cioè deve valere $\overleftrightarrow{V}_x(a,ta,\pi,\mathcal{C},v)$. Il predicato $\overleftrightarrow{V}_x(a,ta,\pi,\mathcal{C},v)$ è vero se il valore a del campo x può attraversare il percorso π del diagramma di controllo $\mathcal{C}$ etichettato secondo v e subire una trasformazione complessiva ta. La definizione del predicato dipende dalla trasformazione ta: se questa è id allora è necessario che il valore iniziale del campo del pacchetto verifichi tutte le condizioni sugli archi, cioè $\overrightarrow{\overrightarrow{V}}_x(a,\pi,\mathcal{C})$; se invece viene applicata una trasformazione è necessario che questa sia consentita dalle etichette associate al percorso, cioè $x \in \mu(\ell(\pi))$ e che il valore a del campo x consenta al pacchetto di attraversare tutti gli archi fino al primo capace di trasformare il campo x, cioè $\overrightarrow{V}_x(a,\pi,\mathcal{C},v)$, e che il valore finale del campo, a', ottenuto applicando la trasformazione ad a, consenta di attraversare gli archi a partire dall'ultimo nodo capace di trasformare il campo x in poi, cioè $\overleftarrow{V}_x(a',\pi,\mathcal{C},v)$. Si noti che fra il primo e l'ultimo nodo capace di trasformare il campo x non imponiamo alcuna condizione, questo perché per ogni coppia $(a, cost(a'))$, qualunque sia il predicato da verificare, possiamo assegnare al campo un valore a'' che lo verifichi e sovrascrivere la trasformazione con $cost(a')$ nell'ultimo nodo disponibile.

Come detto, il predicato $\overrightarrow{\overrightarrow{V}}_x(a,\pi,\mathcal{C})$ è verificato se e solo se a verifica tutte le condizioni presenti sugli archi del percorso π. Il predicato $\overrightarrow{V}_x(a,\pi,\mathcal{C},v)$ invece è verificato da un valore a per il campo x, da un percorso π e da un diagramma di controllo $\mathcal{C}$ se e solo se a verifica tutte le condizioni presenti sugli archi del percorso π, ma solo a partire dal nodi iniziale fino al primo nodo capace di modificare il campo x, cioè tale che $x \in \mu(v(q))$. Entrambi i predicati sono definiti iterativamente attraverso una condizione sui primi due nodi del percorso e lo stesso predicato applicato al percorso ottenuto rimuovendo il primo nodo; se i nodi del percorso non sono almeno due non c'è nessun arco da verificare e quindi il predicato è vero automaticamente. La definizione di $\overleftarrow{V}_x(a,\pi,\mathcal{C},v)$ ricorre in modo diverso sul percorso, si controlla la condizione sull'arco che collega gli ultimi due nodi e la ricorsione è sul percorso a partire dal nodo iniziale fino al penultimo. Il predicato è vero se e solo se il valore a, assegnato al campo x verifica le condizioni sugli archi del percorso π dall'ultimo nodo capace di modificare x in poi. L'iterazione avviene nel verso opposto rispetto a $\overrightarrow{V}_x(a,\pi,\mathcal{C},v)$ proprio per riconoscere facilmente l'ultimo nodo tale che $x \in \mu(v(q))$.

Per semplificare ulteriormente la verifica di ϵ_0, anziché basarci sull'insieme di coppie (p, t) esprimibili da un particolare percorso, calcolato dalla funzione $E(\pi, \mathcal{C}, v)$, riformuliamo l'espressività calcolando per ogni coppia (p, t) quale sia l'insieme dei percorsi capaci di esprimerla.

$$\mathcal{P}(\mathcal{C}, v, p, t) = \{\pi \in \Pi(\mathcal{C}) \mid (p, t) \in E(\pi, \mathcal{C}, v)\}$$

Quindi ridefiniamo ϵ_0 nella seguente maniera:

$$\epsilon_0(\lambda, \mathcal{C}, v) = \forall p \in \mathbb{P}.\ \mathcal{P}(\mathcal{C}, v, p, \lambda(p)) \neq \emptyset$$

La rappresentazione di $\mathcal{P}(\mathcal{C}, v, p, t)$ è adeguata a verificare l'appartenenza o meno di una coppia (p, t) all'insieme $E(\pi, \mathcal{C}, v)$. Per quanto riguarda un firewall astratto $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, la verifica di $\epsilon_0(\lambda, \mathcal{C}, v)$ dovrebbe procedere verificando una a una tutte le coppie $(p, \lambda(p))$. Chiaramente possiamo supporre di avere a che fare con la versione sintetizzata di λ, pertanto il numero di controlli non è intrattabile essendo limitato dal numero di multicubi e quindi legato al numero di regole della configurazione IFCL.

Il confronto dell'espressività di due sistemi diversi, k e k', quindi dell'insieme delle λ che verificano $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$ rispetto a quelle che verificano $\epsilon_0(\lambda, \mathcal{C}_{k'}, v_{k'})$, non è invece immediatamente trattabile. Chiaramente, data la formulazione, è evidente che la capacità di un sistema di esprimere una funzione che associa un dato pacchetto a una trasformazione è indipendente dall'eventuale capacità di esprimere altre associazioni. Pertanto anziché verificare una a una quali delle possibili funzioni $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ verificano il predicato $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$ possiamo considerare le coppie (p, t) separatamente e considerare quali verifichino $\mathcal{P}(\mathcal{C}_k, v_k, p, t) \neq \emptyset$ per un dato sistema k. Dal punto di vista generale possiamo solo affidarci a un risolutore automatico, non potendo fare assunzioni sulle condizioni che etichettano gli archi.

Dato però che le condizioni ψ nei sistemi analizzati sono tutte molto simili e riguardano unicamente l'appartenenza o meno dell'indirizzo di origine o destinazione all'insieme degli indirizzi locali $\mathcal{L}$, forniamo una procedura semplificata che funziona per i sistemi supportati. Vogliamo caratterizzare l'insieme delle coppie (p, t) che verificano $\mathcal{P}(\mathcal{C}_k, v_k, p, t) \neq \emptyset$ per un dato sistema k in maniera abbastanza sintetica da permettere il confronto con un secondo sistema k'. Non è necessario calcolare il risultato della funzione coppia per coppia, è sufficiente calcolare il risultato per tutte le possibili combinazioni di alcuni valori canonici per i campi del pacchetto e della trasformazione.

L'insieme $\mathbb{P}$ viene suddiviso in quattro classi di equivalenza in base alla località o meno dell'indirizzo IP di origine e destinazione, l'insieme delle trasformazioni viene suddiviso in $3^2 \cdot 2^2 = 36$ sulla base delle trasformazioni sui campi: per gli indirizzi IP consideriamo tre tipi di trasformazioni id, $cost(\mathcal{L})$ e $cost(\neg\mathcal{L})$ mentre per le porte consideriamo solo id e $cost$. In aggiunta dobbiamo considerare anche $\bot$ come possibile destino di un pacchetto. Definiamo dunque degli insiemi di valori canonici:

- per i campi sIP e dIP del pacchetto definiamo la coppia di valori $p_{IP} = \{\mathcal{L},\ \neg\mathcal{L}\}$, corrispondenti a un generico indirizzo appartenente a $\mathcal{L}$ e a un generico indirizzo non appartenente a $\mathcal{L}$ (la porta non è importante ai fini della verifica dei predicati ψ);
- per i campi sIP e dIP della trasformazione abbiamo invece tre valori $t_{IP} = \{id,\ cost(\mathcal{L}),\ cost(\neg\mathcal{L})\}$ corrispondenti rispettivamente a una trasformazione id, a una trasformazione in un indirizzo appartenente a $\mathcal{L}$ e a una trasformazione non appartenente a $\mathcal{L}$;
- per i campi $sPort$ e $dPort$ della trasformazione i valori possibili sono $t_{Port} = \{id,\ cost\}$ a seconda se la trasformazione sia id o meno.

Modelliamo quindi l'insieme $\mathbb{P}$ come il prodotto cartesiano $\tilde{\mathbb{P}} = (p_{IP} \times p_{IP})$ e l'insieme $\mathcal{T}(\mathbb{P})$ come $\widetilde{\mathcal{T}(\mathbb{P})} = t_{IP} \times t_{Port} \times t_{IP} \times t_{Port}$. La definizione della funzione $\alpha_{\mathbb{P}}(p)$ che dato un pacchetto $p \in \mathbb{P}$ restituisce un pacchetto con i valori canonici corrispondenti, $\tilde{p} \in \tilde{\mathbb{P}}$, e quella della funzione $\alpha_{\mathcal{T}(\mathbb{P})\cup\{\bot\}}(t)$ che

data una trasformazione $\mathcal{T}(\mathbb{P})\cup\{\bot\}$ restituisce una trasformazione con i valori canonici corrispondenti, $\widetilde{t} \in \widetilde{\mathcal{T}(\mathbb{P})} \cup \{\bot\}$, sono banali.

$$\alpha_{\mathbb{P}}(p) = (\alpha_{pIP}(p.sIP), \alpha_{pIP}(p.dIP)) \qquad \alpha_{pIP}(a) = \begin{cases} \mathcal{L} & \text{se } a \in \mathcal{L} \\ \neg\mathcal{L} & \text{altrimenti} \end{cases}$$

$$\alpha_{\mathcal{T}(\mathbb{P})\cup\{\bot\}}(t) = \begin{cases} \bot & \text{se } t = \bot \\ (\alpha_{tIP}(t.sIP), \alpha_{tPort}(t.sPort), \alpha_{tIP}(t.dIP), \alpha_{tPort}(t.dPort)) & \text{altrimenti} \end{cases}$$

$$\alpha_{tIP}(ta) = \begin{cases} id & \text{se } ta = id \\ cost(\mathcal{L}) & \text{se } ta = cost(a) \wedge a \in \mathcal{L} \\ cost(\neg\mathcal{L}) & \text{altrimenti} \end{cases} \qquad \alpha_{tPort}(ta) = \begin{cases} id & \text{se } ta = id \\ cost & \text{altrimenti} \end{cases}$$

Definiamo la funzione $\tilde{\mathcal{P}}_{\mathcal{C},v}(\tilde{p},\tilde{t})$ con $\tilde{p} \in \tilde{\mathbb{P}}$ e $\tilde{t} \in \widetilde{\mathcal{T}(\mathbb{P})} \cup \{\bot\}$ come la versione definita sui valori canonici di $\mathcal{P}$, in particolare vale:

$$\forall p \in \mathbb{P}, t \in \mathcal{T}(\mathbb{P}) \cup \{\bot\}.\ \mathcal{P}(\mathcal{C}, v, p, t) = \tilde{\mathcal{P}}_{\mathcal{C},v}(\alpha(p), \alpha(t))$$

Dell'insieme delle coppie $(\tilde{p},\tilde{t})$ che verificano il predicato $\tilde{\mathcal{P}}_{\mathcal{C}_k,v_k}(\tilde{p},\tilde{t})$, per un dato sistema k, possiamo ora dare una descrizione completa in tempo ragionevole in quanto il numero di combinazioni da provare non è troppo alto ($2^2 \cdot (2^2 \cdot 3^2 + 1) = 148$ per la precisione). Per i sistemi supportati definiremo la funzione $\tilde{\mathcal{P}}$ in forma tabellare, per leggibilità e facilità di confronto dei sistemi.

Fattibilità locale nei sistemi supportati

Per ognuno dei sistemi supportati $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$ calcoliamo come prima cosa l'insieme dei percorsi nel diagramma di controllo. Successivamente per ogni percorso π calcoliamo la funzione $E(\pi, \mathcal{C}_k, v_k)$. Infine, basandoci sulla funzione E ottenuta, forniamo in formato tabellare il risultato della funzione $\tilde{\mathcal{P}}_{\mathcal{C}_k,v_k}(\tilde{p},\tilde{t})$ per ogni possibile coppia di input $(\tilde{p},\tilde{t})$. Per risparmiare spazio, laddove il valore di un campo non sia rilevante per il risultato della funzione usiamo il carattere _ ed evitiamo di ripetere righe identiche. Sempre per risparmiare spazio usiamo una tabella a parte per le coppie in cui $t = \bot$.

pf

I percorsi $\pi \in \Pi(\mathcal{C}_{\texttt{pf}})$ sono i seguenti:

$$\ddot{\pi}_1 = q_i; q_0; q_1; q_f \qquad \ddot{\pi}_2 = q_i; q_2; q_3; q_f$$
$$\ddot{\pi}_3 = q_i; q_0; q_1; q_2; q_3; q_f \qquad \ddot{\pi}_4 = q_i; q_2; q_3; q_0; q_1; q_f$$

$$\bar{\pi}_1 = q_i; q_0; q_1 \qquad \bar{\pi}_2 = q_i; q_2; q_3$$
$$\bar{\pi}_3 = q_i; q_0; q_1; q_2; q_3 \qquad \bar{\pi}_4 = q_i; q_2; q_3; q_0; q_1$$
$$\bar{\pi}_5 = q_i; q_0; q_1; q_2; q_3; q_0 \qquad \bar{\pi}_6 = q_i; q_2; q_3; q_0; q_1; q_2$$

Abbiamo quindi che la funzione E per i percorsi vale:

$$E(\ddot{\pi}_1, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t.sIP = t.sPort = id \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$

$$E(\ddot{\pi}_2, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t.dIP = t.dPort = id \wedge t \neq \bot\}$$

$p.sIP$	$p.dIP$	$t.sIP$	:	$t.sPort$	$t.dIP$	:	$t.dPort$	π
$\mathcal{L}$	$\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	$\mathcal{L}$	_	:	_	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	$\mathcal{L}$	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\emptyset$
$\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	id	$\{\ddot{\pi}_2\}$
$\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	$cost(\_)$	:	_	$\emptyset$
$\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	_	:	$cost$	$\emptyset$
$\neg\mathcal{L}$	$\mathcal{L}$	id	:	id	id	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	$\mathcal{L}$	id	:	id	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	$\mathcal{L}$	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_3\}$
$\neg\mathcal{L}$	$\mathcal{L}$	id	:	$cost$	id	:	_	$\emptyset$
$\neg\mathcal{L}$	$\mathcal{L}$	id	:	$cost$	$cost(\mathcal{L})$	:	_	$\emptyset$
$\neg\mathcal{L}$	$\mathcal{L}$	$cost(\_)$	:	_	id	:	_	$\emptyset$
$\neg\mathcal{L}$	$\mathcal{L}$	$cost(\_)$	:	_	$cost(\mathcal{L})$	:	_	$\emptyset$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	id	:	id	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_3\}$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_3\}$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	$cost(\_)$	:	_	$cost(\mathcal{L})$	:	_	$\emptyset$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	id	:	$cost$	$cost(\mathcal{L})$	:	_	$\emptyset$

(a) $\tilde{t} \neq \bot$

$p.sIP$	$p.dIP$	t	π
$\mathcal{L}$	$\mathcal{L}$	$\bot$	$\{\bar{\pi}_2, \bar{\pi}_4, \bar{\pi}_6\}$
$\mathcal{L}$	$\neg\mathcal{L}$	$\bot$	$\{\bar{\pi}_2\}$
$\neg\mathcal{L}$	_	$\bot$	$\{\bar{\pi}_1, \bar{\pi}_3, \bar{\pi}_5\}$

(b) $\tilde{t} = \bot$

Tabella 6.1: Rappresentazione tabellare della funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}}(\tilde{p}, \tilde{t})$.

$$E(\ddot{\pi}_3, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t \neq \bot\}$$

$$E(\ddot{\pi}_4, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge p.dIP \in \mathcal{L} \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$

$$E(\bar{\pi}_1, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_2, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$

$$E(\bar{\pi}_3, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_4, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L} \wedge p.dIP \in \mathcal{L}\}$$

$$E(\bar{\pi}_5, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_6, \mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L} \wedge p.dIP \in \mathcal{L}\}$$

La funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\mathtt{pf}}, v_{\mathtt{pf}}}(\tilde{p}, \tilde{t})$ è definita per casi su ogni possibile input dalla tabella 6.1. Si noti che alcune coppie $(\tilde{p}, \tilde{t})$ non sono esprimibili dal sistema, pertanto `pf` non è del tutto generale.

$p.sIP$	$p.dIP$	$t.sIP$	:	$t.sPort$	$t.dIP$	:	$t.dPort$	π
$\mathcal{L}$	_	_	:	_	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	_	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_2\}$
$\mathcal{L}$	$\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_2\}$
$\neg\mathcal{L}$	_	_	:	_	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	_	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_3\}$
$\neg\mathcal{L}$	$\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_3\}$

(a) $\tilde{t} \neq \bot$

$p.sIP$	$p.dIP$	t	π
$\mathcal{L}$	_	$\bot$	$\{\bar{\pi}_2, \bar{\pi}_4\}$
$\neg\mathcal{L}$	_	$\bot$	$\{\bar{\pi}_1, \bar{\pi}_3\}$

(b) $\tilde{t} = \bot$

Tabella 6.2: Rappresentazione tabellare della funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}}(\tilde{p}, \tilde{t})$.

iptables

Definiamo i percorsi $\pi \in \Pi(\mathcal{C}_{\texttt{iptables}})$ nella seguente maniera:

$$\ddot{\pi}_1 = q_i; q_0; q_1; q_2; q_3; q_{10}; q_{11}; q_f \qquad \ddot{\pi}_2 = q_i; q_7; q_8; q_9; q_{10}; q_{11}; q_f$$
$$\ddot{\pi}_3 = q_i; q_0; q_1; q_4; q_5; q_6; q_f \qquad \ddot{\pi}_4 = q_i; q_7; q_8; q_9; q_4; q_5; q_6; q_f$$

$$\bar{\pi}_1 = q_i; q_0; q_1; q_2; q_3 \qquad \bar{\pi}_2 = q_i; q_7; q_8; q_9$$
$$\bar{\pi}_3 = q_i; q_0; q_1; q_4; q_5; q_6 \qquad \bar{\pi}_4 = q_i; q_7; q_8; q_9; q_4; q_5; q_6$$

Abbiamo quindi che la funzione E per i percorsi vale:

$$E(\ddot{\pi}_1, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$
$$E(\ddot{\pi}_2, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t \neq \bot\}$$
$$E(\ddot{\pi}_3, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t \neq \bot\}$$
$$E(\ddot{\pi}_4, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$
$$E(\bar{\pi}_1, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$
$$E(\bar{\pi}_2, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$
$$E(\bar{\pi}_3, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$
$$E(\bar{\pi}_4, \mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$

La funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\texttt{iptables}}, v_{\texttt{iptables}}}(\tilde{p}, \tilde{t})$ è definita per casi su ogni possibile input dalla tabella 6.2. Si noti che per ogni coppia $(\tilde{p}, \tilde{t})$ la funzione restituisce un insieme non vuoto, pertanto `iptables`, a differenza di `pf`, è generale, almeno per quanto riguarda la condizione di fattibilità locale.

ipfw

Definiamo i percorsi di `ipfw` nella seguente maniera:

$$\ddot{\pi}_1 = q_i; q_0; q_f \qquad \ddot{\pi}_2 = q_i; q_1; q_f \qquad \ddot{\pi}_3 = q_i; q_0; q_1; q_f \qquad \ddot{\pi}_4 = q_i; q_1; q_0; q_f$$

$$\bar{\pi}_1 = q_i; q_0 \qquad \bar{\pi}_2 = q_i; q_1 \qquad \bar{\pi}_3 = q_i; q_0; q_1 \qquad \bar{\pi}_4 = q_i; q_1; q_0$$
$$\bar{\pi}_5 = q_i; q_0; q_1; q_0 \qquad \bar{\pi}_6 = q_i; q_1; q_0; q_1$$

Abbiamo

$$E(\ddot{\pi}_1, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$

$$E(\ddot{\pi}_2, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t \neq \bot\}$$

$$E(\ddot{\pi}_3, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,t) \mid p.sIP \notin \mathcal{L} \wedge t(p).dIP \notin \mathcal{L} \wedge t \neq \bot\}$$

$$E(\ddot{\pi}_4, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,t) \mid p.sIP \in \mathcal{L} \wedge t(p).dIP \in \mathcal{L} \wedge t \neq \bot\}$$

$$E(\bar{\pi}_1, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_2, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$

$$E(\bar{\pi}_3, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_4, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$

$$E(\bar{\pi}_5, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \notin \mathcal{L}\}$$

$$E(\bar{\pi}_6, \mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}) = \{(p,\bot) \mid p.sIP \in \mathcal{L}\}$$

La funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}}(\tilde{p}, \tilde{t})$ è definita per casi su ogni possibile input dalla tabella 6.3. Si noti che per ogni coppia $(\tilde{p}, \tilde{t})$ la funzione restituisce un insieme non vuoto, pertanto `ipfw`, come `iptables`, è generale, almeno per quanto riguarda la condizione di fattibilità locale.

Confronto

Come abbiamo notato `iptables` e `ipfw` sono generali, cioè è possibile configurarli in modo tale da creare qualunque assegnamento possibile fra uno specifico pacchetto e una specifica trasformazione. Infatti la tabella non ha "buchi", tutte le coppie sono esprimibili. Chiaramente questo non basta a garantire che tutti i possibili firewall astratti siano esprimibili, dato che non abbiamo ancora trattato la *coerenza*. Tuttavia quanto descritto è sufficiente ad affermare che alcuni firewall `iptables` e `ipfw` hanno una semantica impossibile da replicare esattamente in `pf`. In particolare, non sono esprimibili in `pf` quelle funzioni $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ per le quali esiste un $p \in \mathbb{P}$ tale che la coppia $(p, \lambda(p))$ è relative a una delle righe della tabella 6.1 in cui il campo π è $\emptyset$.

Portiamo un esempio di firewall astratto $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ che non è esprimibile da `pf`, non verificando $\epsilon_0(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$. Si supponga che l'insieme degli indirizzi locali sia $\mathcal{L} = \{192.168.0.0, 127.0.0.0\}$. Il firewall astratto non esprimibile è il seguente:

$$\lambda(p) = \begin{cases} (id : id,\ cost(4.3.2.1) : id) & \text{se } p = (192.168.0.0 : 80,\ 1.2.3.4 : 80) \\ \bot & \text{altrimenti} \end{cases}$$

Notiamo che $\mathcal{P}(\mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}, (192.168.0.0 : 80,\ 1.2.3.4 : 80), (id : id,\ 4.3.2.1 : id)) = \emptyset$. Questo segue dal fatto che $\alpha(192.168.0.0 : 80,\ 1.2.3.4 : 80) = (\mathcal{L}, \neg\mathcal{L})$, $\alpha(id : id,\ cost(4.3.2.1) : id) = (id : id,\ cost(\neg\mathcal{L}) : id)$ e dal risultato della funzione in tabella 6.1: $\bar{\mathcal{P}}_{\mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}}((\mathcal{L}, \neg\mathcal{L}), (id : id,\ cost(\neg\mathcal{L}) : id)) = \emptyset$.

$p.sIP$	$p.dIP$	$t.sIP$	:	$t.sPort$	$t.dIP$	:	$t.dPort$	π
$\mathcal{L}$	_	_	:	_	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	_	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_2\}$
$\mathcal{L}$	$\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_4\}$
$\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_2\}$
$\neg\mathcal{L}$	_	_	:	_	$cost(\mathcal{L})$	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	_	_	:	_	$cost(\neg\mathcal{L})$	:	_	$\{\ddot{\pi}_3\}$
$\neg\mathcal{L}$	$\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_1\}$
$\neg\mathcal{L}$	$\neg\mathcal{L}$	_	:	_	id	:	_	$\{\ddot{\pi}_3\}$

(a) $\tilde{t} \neq \bot$

$p.sIP$	$p.dIP$	t	π
$\mathcal{L}$	_	$\bot$	$\{\bar{\pi}_2, \bar{\pi}_4, \bar{\pi}_6\}$
$\neg\mathcal{L}$	_	$\bot$	$\{\bar{\pi}_1, \bar{\pi}_3, \bar{\pi}_5\}$

(b) $\tilde{t} = \bot$

Tabella 6.3: Rappresentazione tabellare della funzione $\tilde{\mathcal{P}}_{\mathcal{C}_{\texttt{ipfw}}, v_{\texttt{ipfw}}}(\tilde{p}, \tilde{t})$.

6.2.2 Coerenza

La coerenza è la proprietà di un firewall astratto λ, rispetto a un diagramma di controllo $\mathcal{C}$ e a un assegnamento di etichette v, secondo la quale per nessuna coppia (p, t) tale che $t = \lambda(p)$, imporre che la configurazione del firewall f sia tale che $\odot(\mathcal{C}, f)(p) = t$ causa delle restrizioni sulla configurazione stessa che siano contraddittorie con altre coppie (p', t') tali che $t' = \lambda(p')$. Formalmente $\epsilon_1(\lambda, C, v)$ può essere espressa considerando i percorsi del diagramma nella seguente maniera:

$$\begin{aligned}\epsilon_1(\lambda, \mathcal{C}, v) = &(\forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ (p, \lambda(p)) \in E(\pi, \mathcal{C}, v)) \\ &\Rightarrow (\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ \forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p))\end{aligned}$$

dove ci siamo concessi un piccolo abuso di notazione e abbiamo scritto $\odot(\pi, f)$ per intendere la concatenazione delle funzioni $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ associate ai nodi del percorso π da f. Formalmente:

$$\odot(\pi, f)(p) = \begin{cases} \odot(\pi', f)(p') \ltimes t & \text{se } \pi = q \cdot \pi' \wedge t \neq \bot \\ \bot & \text{se } \pi = q \cdot \pi' \wedge t = \bot \\ id & \text{altrimenti} \end{cases} \qquad \text{dove } t = f(q)(p) \text{ e } p' = t(p)$$

In effetti, dato che abbiamo a disposizione la funzione $\mathcal{P}(\mathcal{C}, v, p, t)$, possiamo sfruttarla per limitare la scelta del percorso $\pi \in \Pi(\mathcal{C})$ scrivendo:

$$\begin{aligned}\epsilon_1(\lambda, \mathcal{C}, v) = &(\forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ (p, \lambda(p)) \in E(\pi, \mathcal{C}, v)) \\ &\Rightarrow (\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ \forall p \in \mathbb{P}.\ \exists \pi \in \mathcal{P}(\mathcal{C}, v, p, \lambda(p)).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p))\end{aligned}$$

Inoltre, se la funzione $\mathcal{P}(\mathcal{C}, v, p, \lambda(p))$ restituisce un singoletto, allora la condizione $\Delta((\mathcal{C}, f), p)$ è automaticamente soddisfatta e possiamo riscrivere la coerenza come:

$$\begin{aligned}\epsilon_1(\lambda, \mathcal{C}, v) = &(\forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ (p, \lambda(p)) \in E(\pi, \mathcal{C}, v)) \\ &\Rightarrow (\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ \forall p \in \mathbb{P}.\ \exists \pi \in \mathcal{P}(\mathcal{C}, v, p, \lambda(p)).\ \odot(\pi, f)(p) = \lambda(p))\end{aligned}$$

L'algoritmo di generazione, che realizza la fase 3. della pipeline di transcompilazione, si occupa sostanzialmente di trovare una f che verifichi la seconda parte del predicato, possibilmente generandola ad hoc perché verifichi $\odot(\pi, f)(p) = \lambda(p)$, tenendo $f \in \mathbb{M}_3(\mathcal{C}, v)$ come vincolo.

L'idea è quella di calcolare per ogni coppia (p, t), le restrizioni cui un firewall del sistema studiato è soggetto accettando il pacchetto p con trasformazione t. Una policy è esprimibile solo se le restrizioni causate da ciascuna delle coppie non impedisce il firewall dal realizzare il resto della funzione λ.

La formula proposta non è la più vantaggiosa per il confronto dell'espressività di diversi sistemi, ma fornisce un predicato che data una funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ consente in teoria di verificare se questa può o meno essere la semantica di un firewall del sistema in esame.

Forniamo un esempio di firewall astratto λ che nel sistema `pf` verifica la condizione di fattibilità locale $\epsilon_0(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$ ma che non è esprimibile in quanto non verifica la condizione di coerenza $\epsilon_1(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$. Supponiamo $\mathcal{L} = \{192.168.0.0, 127.0.0.0\}$, si consideri la funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ definita come:

$$\lambda(p) = \begin{cases} (cost(1.2.3.4) : id,\ id : id) & \text{se } p = (192.168.0.0 : 22,\ 192.168.0.0 : 23) \quad (1) \\ (id : id,\ cost(4.3.2.1) : id) & \text{se } p = (1.2.3.4 : 22,\ 192.168.0.0 : 23) \quad (2) \\ \bot & \text{altrimenti} \end{cases}$$

Riguardo al caso (1), $\ddot{\pi}_4$ è l'unico percorso che il pacchetto $(192.168.0.0 : 22,\ 192.168.0.0 : 23)$ può percorrere se gli deve essere associata la trasformazione $(cost(1.2.3.4) : id,\ id : id)$; formalmente:

$$\mathcal{P}(\mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}, (192.168.0.0 : 22, 192.168.0.0 : 23), (cost(1.2.3.4) : id, id : id)) = \{\pi_4\}$$

Quindi è necessario, perché $\epsilon_1(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$ sia verificato, che esista una configurazione f tale che $\odot(\pi_4, f)(192.168.0.0 : 22,\ 192.168.0.0 : 23) = (cost(1.2.3.4) : id,\ id : id)$. Nel percorso π_4 esiste un solo nodo con etichetta $SNAT$, adatto ad applicare la trasformazione $cost(1.2.3.4)$ al campo sIP del pacchetto, questo nodo è q_2. Dunque il nodo q_2 applicherà questa trasformazione e il resto dei nodi del percorso assocerà la trasformazione identità al risultato:

$$\begin{aligned} f(q_2)(192.168.0.0 : 22,\ 192.168.0.0 : 23) &= (cost(1.2.3.4) : id,\ id : id) \\ f(q_3)(1.2.3.4 : 22,\ 192.168.0.0 : 23) &= (id : id,\ id : id) \\ f(q_0)(1.2.3.4 : 22,\ 192.168.0.0 : 23) &= (id : id,\ id : id) \\ f(q_1)(1.2.3.4 : 22,\ 192.168.0.0 : 23) &= (id : id,\ id : id) \end{aligned}$$

Nel caso (2) invece, $\ddot{\pi}_1$ è l'unico percorso che il pacchetto $(1.2.3.4 : 22,\ 192.168.0.0 : 23)$ può percorrere se deve essergli associata la trasformazione $(id : id,\ cost(4.3.2.1) : id)$; formalmente:

$$\mathcal{P}(\mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}}, (1.2.3.4 : 22,\ 192.168.0.0 : 23), (id : id,\ cost(4.3.2.1) : id)) = \{\pi_1\}$$

È quindi necessario, perché $\epsilon_1(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$ sia verificato, che la stessa configurazione f del caso precedente sia tale che $\odot(\pi_3, f)(1.2.3.4 : 22,\ 192.168.0.0 : 23) = (id : id,\ cost(4.3.2.1) : id)$. Nel percorso π_3 esiste un solo nodo con etichetta $DNAT$, adatto ad applicare la trasformazione $cost(4.3.2.1)$ al campo dIP del pacchetto, questo nodo è q_0. Dunque il nodo q_0 applicherà questa trasformazione e il resto dei nodi del percorso assocerà la trasformazione identità al risultato:

$$\begin{aligned} f(q_0)(1.2.3.4 : 22,\ 192.168.0.0 : 23) &= (id : id,\ cost(4.3.2.1) : id) \\ f(q_1)(1.2.3.4 : 22,\ 4.3.2.1 : 23) &= (id : id,\ id : id) \end{aligned}$$

Il firewall astratto λ non è esprimibile dunque, dato che non è possibile che la funzione di trasformazione associata al nodo q_0 sia tale che $f(q_0)(1.2.3.4 : 22,\ 192.168.0.0 : 23) = (id : id,\ cost(4.3.2.1) : id)$ e contemporaneamente $f(q_0)(1.2.3.4 : 22,\ 192.168.0.0 : 23) = (id : id,\ id : id)$.

In particolare il problema di questo firewall astratto in `pf` è legato al fatto che se un pacchetto p viene accettato con trasformazione t dal percorso π_4, allora il pacchetto p' tale che $p'.sIP = t(p).sIP$, $p'.sPort = t(p).sPort$, $p'.dIP = p.dIP$ e $p'.dPort = p.dPort$ deve essere accettato con trasformazione $t' = (id : id,\ t.dIP : t.dPort)$. Altri vincoli simili possono essere individuati dall'analisi del predicato $\epsilon_1(\lambda, \mathcal{C}_{\texttt{pf}}, v_{\texttt{pf}})$, sulla base del percorso associato a una coppia (p, t).

Capitolo 7

Generazione di un firewall

La generazione è la terza fase della pipeline di transcompilazione, relativa alla creazione di un firewall IFCL, del tipo designato, che abbia una semantica equivalente a quella del firewall astratto di partenza. Dato un firewall astratto sintetizzato $\tilde{\lambda}$, l'obiettivo è dunque quello di produrre una configurazione IFCL Σ per il sistema k tale che:

$$(\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!)(s_{\texttt{NEW}}) = i(\tilde{\lambda})$$

Nonostante avvenga in un'altra fase della pipeline, dobbiamo occuparci anche della concretizzazione: cioè la procedura che, a partire dalla configurazione per il firewall IFCL, restituisce il file di configurazione del sistema target. La quarta ed ultima fase della pipeline è realizzata per mezzo di una funzione con_k; questa funzione non può essere definita su tutte le possibili configurazione IFCL assegnabili al diagramma di controllo del sistema target, $\mathcal{C}_k$. In effetti per poter implementare la fase finale della pipeline è necessario che la configurazione Σ prodotta dall'algoritmo di sintesi appartenga al dominio della funzione con_k. Alcune configurazione potrebbero non essere compilabili perché la loro semantica non è esprimibile dal sistema target, come abbiamo dimostrato nel capitolo precedente infatti non tutti i sistemi possono esprimere tutte le funzioni $\mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$. In questo caso non possiamo fare niente, il fallimento non dipende dall'algoritmo che scegliamo per la fase 3. della pipeline o dalla funzione di concretizzazione. Tuttavia, se la funzione di concretizzazione con_k non è definita su tutte le configurazioni Σ tali che $(\!|(\mathcal{C}_k, \Sigma)|\!)$ è una funzione esprimibile dal sistema k, è possibile produrre dei firewall IFCL che non possono essere concretizzati anche quando la funzione di partenza è esprimibile dal sistema target.

Per prima cosa forniremo qualche dettaglio sulla funzione di concretizzazione, specificando quali tipi di configurazioni IFCL sono concretizzabili nei vari sistemi; successivamente forniremo due approcci alternativi per la fase di generazione: uno che segue passo passo le fasi intermedie della pipeline proposta nel capitolo 4; l'altro proposto inizialmente in [4] basato sulla generazione in un passo solo della configurazione IFCL per il sistema target, basato sull'uso intensivo del campo *tag* dei pacchetti.

Dato un firewall astratto sintetizzato $\tilde{\lambda}$ e un sistema k, vorremmo verificare in anticipo se sia possibile ottenere un firewall con semantica equivalente a $\lambda = i(\tilde{\lambda})$ per il sistema k, cioè $\lambda \in \Lambda_k$, ma come abbiamo visto la procedura per un controllo è tanto complessa quanto tentare la generazione in sé. Verifichiamo comunque la condizione necessaria $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$, se questa non è verificata allora sicuramente non potremo generare il firewall target, in caso contrario c'è comunque la possibilità che la generazione non vada a buon fine, nel qual caso ce ne accorgeremo durante l'applicazione dell'algoritmo e la generazione terminerà segnalando errore.

Come detto decidiamo di analizzare i sistemi solo per quanto riguarda il modo di trattare pacchetti appartenenti a nuove connessioni, pertanto assumeremo sempre che il firewall sia nello stato $s_{\texttt{NEW}}$.

7.1 Concretizzazione

L'ultima traduzione effettuata dalla pipeline è la concretizzazione del firewall IFCL come file di configurazione del sistema target. Concettualmente la funzione con_k è l'inversa della funzione di formalizzazione for_k, ma questo non è strettamente vero a livello funzionale. Non vale infatti in genere che $for_k(con_k(\Sigma)) = \Sigma$; quello che abbiamo è che data un configurazione IFCL Σ, per il sistema k, la sua concretizzazione restituisce un file di configurazione `file.conf` la cui semantica sia equivalente a quella di Σ. Formalmente:

$$(\!|\mathbb{C}\ (\mathcal{C}_k, for_k(con_k(\Sigma)))\ \mathbb{D}|\!) \equiv (\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!)$$

Per una funzione di concretizzazione chiamiamo correttezza la proprietà di produrre sempre un file di configurazione legale e la cui semantica sia equivalente a quella della configurazione di partenza.

La funzione con_k non può essere definita su ogni possibile configurazione, si tratta di una funzione parziale. Ad esempio tutte le configurazioni per le quali la semantica del firewall non è esprimibile dal sistema k non saranno concretizzabili in file di configurazione per k. Chiamiamo Γ'_k l'insieme delle configurazioni Σ sulle quali è definita la funzione con_k. Si noti che Γ_k non è affatto equivalente a Γ'_k. Idealmente vorremmo poter concretizzare ogni configurazione con semantica esprimibile. Chiamiamo con_k *completa* se e solo se:

$$\forall \lambda \in \Lambda_k.\ \forall \Sigma.\ (\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!) = \lambda \Rightarrow \Sigma \in \Gamma'_k$$

In realtà ci accontentiamo di una proprietà più debole:

$$\forall \lambda \in \Lambda_k.\ \exists \Sigma.\ (\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!) = \lambda \wedge \Sigma \in \Gamma'_k$$

E ci sincereremo che la configurazione prodotta dall'algoritmo di generazione sia proprio una dei Σ che appartengono a Γ'_k. In particolare la funzione con_k per $k \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$ è definita sull'insieme $\Gamma'_k = \mathbb{M}_2(\mathcal{C}_k, v_k)$.

La concretizzazione per i sistemi supportati è relativamente banale se ci limitiamo a supportare le configurazioni in $\mathbb{M}_2(\mathcal{C}_k, v_k)$. Trattandosi di configurazioni normalizzate infatti, e non dovendo quindi trattare salti, tag e chiamate, è sufficiente applicare una trasformazione sintattica immediatamente derivabile dalla definizione delle funzioni di formalizzazione for_k.

7.2 Generazione per livelli

Presentiamo l'algoritmo di generazione per livelli, che funziona attraverso una serie di trasformazioni, dal dominio dei firewall astratti fino a quello dei firewall concreti, passando per livelli intermedi e mantenendo inalterata la semantica. Per realizzare la traduzione fra due sistemi firewall differenti è necessario definire delle funzioni di traduzione che vadano nel verso opposto rispetto a quelle dell'algoritmo di sintesi, dal livello più astratto a quello più concreto. Lo studio dell'espressività dei sistemi di firewall ci serve anche a capire il dominio delle funzioni e a guidare l'implementazione. L'algoritmo di generazione implementa le fasi 3.a e 3.b della pipeline di transcompilazione.

Come per la sintersi anche nella generazione è importante mantenere una rappresentazione sintetica delle funzioni per questioni di trattabilità. Prima di passare alla fase di generazione, viene controllato che la funzione su pacchetti sintetizzata $\tilde{\lambda}$ da compilare verifichi $\epsilon_0(i(\tilde{\lambda}), \mathcal{C}_k, v_k)$.

Notiamo che la fase complicata è la *decomposizione* (fase 3.a), la quale prevede di generare una configurazione semiastratta sintetizzata $\tilde{f}$, legale secondo l'assegnamento di etichette del sistema, tale che la composizione della sua interpretazione, secondo il diagramma di controllo $\mathcal{C}_k$, corrisponda

all'interpretazione del firewall astratto sintetizzato di partenza.

$$i(\tilde{f}) \in \mathbb{M}_3(\mathcal{C}_k, v_k) \wedge \forall p \in \mathbb{P}.\ \odot(\mathcal{C}_k, i(\tilde{f}))(p) = i(\tilde{\lambda})(p)$$

Infatti, una volta calcolata una $\tilde{f}$ adeguata, la compilazione delle funzioni sintetizzate assegnate ai nodi in ruleset IFCL si riduce semplicemente ad una procedura sintattica banale, il cui risultato è un firewall IFCL normalizzato. Dato che il risultato è un firewall IFCL normalizzato, se il firewall astratto è esprimibile allora la configurazione prodotta appartiene sicuramente all'insieme Γ'_k su cui la funzione di concretizzazione è definita.

Per prima cosa analizziamo la generazione di $\tilde{f}$ dal punto di vista teorico, concentrandoci quindi su λ e f, successivamente passiamo alla versione reale con funzioni sintetizzate.

7.3 Generazione del firewall semiastratto

Formalmente il problema che vogliamo affrontare è quello, dato un sistema k e un firewall astratto λ tale che $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$, di trovare una configurazione semiastratta f tale che:

1. f sia legale secondo l'assegnamento di etichette che caratterizza il sistema k: $f \in \mathbb{M}_3(\mathcal{C}, v)$
2. f imponga una semantica coerente con la funzione λ: $\forall p \in \mathbb{P}.\ \odot(\mathcal{C}_k, f)(p) = \lambda(p)$

Ovvero, facendo riferimento ai percorsi all'interno del diagramma di controllo:

$$f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p) \tag{i}$$

Per prima cosa definiamo una proprietà dei diagramma di controllo e degli assegnamenti di etichette che garantisce la possibilità di individuare, per ogni coppia (p, t), un preciso percorso all'interno del diagramma di controllo. Chiamiamo *sistemi uniterali* i sistemi con diagrammi di controllo tali per cui, in ogni possibile configurazione legale secondo l'assegnamento di etichette, il percorso che un dato pacchetto p può percorrere, dato che la trasformazione finale risultante deve essere t, è unico.

Definizione 21 (Sistema uniterale)**.** *Un sistema uniterale è un sistema firewall k tale che per ogni coppia (p, t), con $p \in \mathbb{P}$ e $t \in \mathcal{T}(\mathbb{P})$, esiste un percorso $\pi \in \Pi(\mathcal{C}_k)$ tale che per ogni possibile configurazione semiastratta f legale secondo v_k par la quale vale $\odot(\mathcal{C}_k, f)(p) = t$, il percorso di p nel firewall è π, cioè $\Delta((\mathcal{C}_k, f), p) = \pi$.*

Si noti che la definizione non comprende vincoli riguardo i pacchetti che vengono scartati, questi infatti verranno trattati in maniera particolare dall'algoritmo di generazione. Si noti inoltre che i sistemi `iptables`, `pf` e `ipfw` sono tutti uniterali, come testimonia la funzione $\mathcal{P}$ del capitolo precedente, la quale restituisce sempre singoletti o insiemi vuoti per coppie in cui $t \neq \bot$.

Se il sistema in questione è uniterale e la funzione λ da compilare verifica la fattibilità locale, allora per ogni coppia (p, t) tale che $t = \lambda(p) \neq \bot$, vale che $\mathcal{P}(\mathcal{C}_k, v_k, p, t)$ è un singoletto. In questo caso possiamo fare a meno della quantificazione esistenziale sui percorsi, quando parliamo di pacchetti accettati; possiamo scrivere quindi:

$$\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \Delta((\mathcal{C}_k, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p) \quad \text{dove } \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p))$$

Occorre ancora la condizione $\Delta((\mathcal{C}_k, f), p) = \pi$ in quanto, sebbene il percorso adeguato sia uno solo, è comunque necessario verificare non solo che la composizione delle funzioni di trasformazione associate ai pacchetti mappino p in $\lambda(p)$, ma anche che la configurazione scelta imponga al pacchetto di percorrere il percorso corretto.

Diverso è il caso in cui la sequenza di trasformazioni che permettono ai nodi del percorso di mappare p in $\lambda(p)$ sia unica. Chiamiamo compatto un sistema k per il quale se due pacchetti subiscono la stessa trasformazione $t \neq \bot$ per una data configurazione, percorrendo lo stesso percorso nel diagramma di controllo, allora i due pacchetti subiscono esattamente le stesse trasformazioni negli stessi nodi.

Definizione 22 (Sistema compatto)**.** *Un sistema k con diagramma di controllo $\mathcal{C}_k$, etichettato secondo v_k, è detto compatto se*

$$\forall f \in \mathbb{M}_3(\mathcal{C}_k, v_k).\ \forall p, p' \in \mathbb{P}.$$
$$(\Delta((\mathcal{C}_k, f), p) = \Delta((\mathcal{C}_k, f), p') = \pi \wedge \odot(\pi, f)(p) = \odot(\pi, f)(p') \neq \bot) \Longrightarrow p \underset{(\pi,f)}{\cong} p'$$

Dove la relazione $\cong$ è definita come:

$$p \underset{(\pi,f)}{\cong} p' = \begin{cases} t = t' \wedge t(p) \underset{(\pi',f)}{\cong} t'(p') & \text{se } \pi = q \cdot \pi' \\ true & \text{altrimenti} \end{cases} \qquad \text{Dove } t = f(q)(p) \text{ e } t' = f(q)(p')$$

Definiamo dunque la versione semplificata del predicato sui pacchetti accettati:

$$\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \odot(\pi, f)(p) = \lambda(p) \qquad \text{dove } \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p)) \qquad \text{(ii)}$$

Definiamo inoltre un predicato vero se e solo se i pacchetti per i quali $\lambda(p) = \bot$ siano gestiti correttamente, cioè:

$$\forall p \in \mathbb{P}.\ \lambda(p) = \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \bot \qquad \text{(iii)}$$

Vale allora il seguente teorema.

Teorema 12. *Se il sistema k è uniterale e compatto, e se la funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ verifica la fattibilità locale, $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$, allora $(i) \iff (ii) \wedge (iii)$*

L'obiettivo del resto della sezione è definire una forma equivalente per il predicato (ii) che dia una traccia per l'implementazione dell'algoritmo in sé. Definiamo un nuovo predicato $\chi(\mathcal{C}_k, v_k, \pi, p, t, f)$ equivalente a $f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \odot(\pi, f)(p) = \lambda(p)$ se $t \neq \bot$, dove $\pi = \mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p))$, del quale diamo una caratterizzazione operativa che sarà la base della parte dell'algoritmo di generazione che si occupa dei pacchetto accettati.

$$\chi(\mathcal{C}_k, v_k, \pi, p, t, f) = \begin{cases} \exists\, t', t''.\ t' \ltimes t'' = t \wedge t' \in \nu(v(q)) \wedge t'' \in \nu(\ell(\pi, \mathcal{C}_k, v_k)) \wedge \\ \qquad f(q)(p) = t' \wedge t'(p) = p' \wedge \\ \qquad \chi(\mathcal{C}_k, v_k, \pi', p', t'', f) & \text{se } \pi = q \cdot \pi' \\ \\ t = id & \text{altrimenti} \end{cases}$$

Dove la funzione ν dato un inseme di etichette restituisce l'insieme delle trasformazioni consentite su un generico pacchetto: $\nu : 2^{\{SNAT, DNAT, DROP\}} \to 2^{\mathcal{T}(\mathbb{P})}$.

$$\nu(L) = \prod_{x \in \{sIP, sPort, dIP, dPort\}} \nu_x(L)$$

$$\nu(L) = \begin{cases} \{cost(a) \mid a \in Dom_x\} \cup \{id\} & \text{se } x \in \mu(L) \\ \{id\} & \text{altrimenti} \end{cases} \quad \text{dove } Dom_x = \begin{cases} \textbf{IP} & \text{se } x \in \{sIP, dIP\} \\ \textbf{Port} & \text{se } x \in \{sPort, dPort\} \end{cases}$$

Ridefiniamo dunque il predicato (ii) come $\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow \chi(\mathcal{C}_k, v_k, \pi, p, t, f)$.

Anziché considerare il predicato $\chi(\mathcal{C}_k, v_k, \pi, p, t, f)$ per ogni coppia (p, t) tali che $t = \lambda(p) = \bot$ come una condizione da verificare per una data configurazione f, la consideriamo una descrizione operativa dei vincoli che abbiamo sulla selezione della configurazione f. In particolare questi vincoli sono della forma $f(q)(p) = t$ per qualche nodo q, pacchetto p e trasformazione t, e sono generati a partire dal percorso π, dal pacchetto p e dalla trasformazione t. Idealmente un algoritmo per la generazione di f potrebbe cercare di verificare il predicato con un assegnamento di valore per la variabile libera f, aggiornandola di volta in volta quando trova una condizione esplicita del tipo $f(q)(p) = t$, scegliendo arbitrariamente i valori in caso di quantificazione esistenziale e facendo backtrack quando ci si trova di fronte ad una contraddizione. La realizzabilità dell'algoritmo cambia drasticamente sulla base delle soluzioni di $\exists\ t', t''.\ t' \ltimes t'' = t \wedge t' \in \nu(v(q)) \wedge t'' \in \nu(\ell(\pi, \mathcal{C}_k, v_k))$. A seconda di quanti modi legali ho per scomporre la trasformazione t in due parti: t' e t'', posso avere più o meno casi da verificare. Il caso ideale è quello in cui t' e t'' sono uniche data t, in questo modo la scomposizione è unica e la verifica del predicato può essere fatta senza backtrack; in questo caso, se si trova un'inconsistenza allora la compilazione fallisce in quanto non può esistere una configurazione che verifichi il predicato χ.

Consideriamo la trasformazione t campo per campo. L'idea è che, per ogni campo che non rimale invariato, vorremmo che solo uno dei nodi del percorso fosse capace di modificarlo, in questo modo la trasformazione t può essere scomposta in una trasformazione costante su quel nodo e id in tutti gli altri; quando invece la trasformazione può essere applicata in più nodi abbiamo libertà di scelta, possiamo applicare la modifica prima o dopo, e possiamo anche modificare il campo in più nodi sovrascrivendo la prima modifica con la seconda.

Formalmente, dati una trasformazione t, un percorso $\pi = q \cdot \pi'$ in un diagramma di controllo $\mathcal{C}_k$ e un assegnamento di etichette v_k; per ogni campo $x \in \{sIP, sPort, dIP, dPort\}$, le scomposizioni che devo provare per la verifica di $\chi(\mathcal{C}_k, v_k, \pi, p, t, f)$ sono:

- se la trasformazione $t.x$ è id allora la scomposizione è unica: $t' = t'' = id$;
- se si tratta di una trasformazione costante, $t.x = cost(a)$ per un qualche a, e $x \in \mu(v(q))$ e $x \notin \mu(\ell(\pi', \mathcal{C}_k, v_k))$, allora la scomposizione è sempre unica: $t'.x = cost(a)$ e $t''.x = id$;
- se si tratta di una trasformazione costante, $t.x = cost(a)$ per un qualche a, e $x \notin \mu(v(q))$ e $x \in \mu(\ell(\pi', \mathcal{C}_k, v_k))$, allora la scomposizione è unica e deve essere: $t'.x = id$ e $t''.x = cost(a)$;
- se si tratta di una trasformazione costante, $t.x = cost(a)$ per un qualche a, e $x \in \mu(v(q))$ e $x \in \mu(\ell(\pi', \mathcal{C}_k, v_k))$, allora la scomposizione non è unica, sono possibili in totale:
  - $t'.x = cost(a)$ e $t''.x = id$;
  - $t'.x = id$ e $t''.x = cost(a)$;
  - $t'.x = cost(b)$ e $t''.x = cost(a)$ per ogni possibile valore b.

Qualche chiarimento: per prima cosa il caso in cui $x \notin \mu(v(q))$ e $x \notin \mu(\ell(\pi', \mathcal{C}_k, v_k))$ non è possibile in quanto la fattibilità locale è garantita per ipotesi e abbiamo scelto il percorso restituito dalla funzione $\mathcal{P}$; secondariamente è bene notare che la libertà di scelta in questo contesto non è una cosa positiva, infatti vuol dire che abbiamo più alternative da provare. Non possiamo limitarci a prendere solo alcune delle alternative possibili perché, a causa dell'interferenza fra la verifica di diverse coppie (p, t), legata al problema della *coerenza*, ϵ_1, rischieremmo di escludere valori che si combinano bene fra loro e quindi la soluzione.

Nel caso quindi in cui, in nessun percorso $\pi \in \Pi(\mathcal{C}_k)$, vi siano più di un etichetta NAT dello stesso tipo (due $SNAT$ o due $DNAT$), la scomposizione di t in t' e t'' è unica. Chiaramente infatti, per ogni

campo x della trasformazione, per ogni nodo q del percorso π, non possono valere entrambi $x \in \mu(v(q))$ e $x \in \mu(\ell(\pi', \mathcal{C}_k, v_k))$, dove π' è la parte di π che segue q.

Definizione 23 (Sistema senza NAT ripetuti)**.** *Chiamiamo sistema senza NAT ripetuti un sistema k con diagramma di controllo $\mathcal{C}_k$ e assegnamento di etichette v_k tale che per ogni percorso $\pi \in \Pi(\mathcal{C}_k)$, il numero di occorrenze dell'etichetta $SNAT$ in π è al più uno e lo stesso vale per $DNAT$.*

È immediato verificare che `iptables` e `pf` sono sistemi senza NAT ripetuti; `ipfw` invece no.

Notiamo inoltre che, in un sistema senza NAT ripetuti, non c'è alternativa sul nodo nel quale effettuare una traduzione per due pacchetti che debbano subire la stessa trasformazione seguendo lo stesso percorso, infatti solo un nodo è disponibile. Vale infatti il seguente teorema:

Teorema 13. *Se un sistema è senza NAT ripetuti, allora è compatto.*

7.4 Decomposizione sintetizzata

Passiamo quindi a considerare la vera forma dei dati che sarà trattata dall'algoritmo, ovvero le funzioni su pacchetti sintetizzate. In effetti, il firewall astratto da compilare sarà fornito come funzione sintetizzata $\tilde{\lambda} \in 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\bot\}}$ e il firewall semiastratto prodotto sarà esso stesso sintetizzato, pertanto la sua configurazione sarà $\tilde{f} : Q \to 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\bot\}}$.

I pacchetti scartati dal firewall saranno trattati separatamente, consideriamo quindi le coppie (p, t) tali che $t = i(\tilde{\lambda})(p) \neq \bot$. Definiamo l'equivalente dell'espressione (ii), verificata se e solo se la configurazione sintetizzata $\tilde{f}$ è legale e rispetta la semantica attesa per quanto riguarda i pacchetti accettati.

$$\forall (P,t) \in \tilde{\lambda}.\ t \neq \bot \Rightarrow \forall p \in P.\ \chi(\mathcal{C}_k, v_k, \pi, p, t, i(\tilde{f})) \qquad \text{dove } \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, t) \qquad \text{(iv)}$$

Dove abbiamo assunto che il sistema target sia uniterale e compatto.

Vogliamo definire un predicato $\tilde{\chi}$, equivalente del predicato χ per i firewall sintetizzati, che caratterizzi operativamente le configurazioni $\tilde{f}$ desiderate, attraverso una serie di vincoli sulle coppie (P, t) appartenenti alle funzioni sintetizzate $\tilde{f}(q)$. Ogni coppia $(P, t) \in \tilde{\lambda}$ comporta la generazione di un certo numero di questi vincoli, per i nodi appartenenti al percorso π seguito dai pacchetti $p \in P$. Per prima cosa vogliamo la garanzia che tutti i pacchetti in un insieme P tale che $(P, t) \in \tilde{\lambda}$ percorrano lo stesso percorso $\pi \in \Pi(\mathcal{C}_k)$. Non si tratta di una proprietà strettamente legata al diagramma di controllo, quanto piuttosto alla forma stessa della funzione sintetizzata, in particolare alla divisione dei pacchetti fra i vari multicubi.

Definizione 24 (Funzione sintetizzata disgiunta)**.** *Una funzione sintetizzata su pacchetti $\tilde{\lambda} \in 2^{\mathcal{M}(\mathbb{P})\times\mathcal{T}(\mathbb{P})\cup\{\bot\}}$ è detta disgiunta rispetto ad un sistema uniterale k, con diagramma di controllo $\mathcal{C}_k$ e assegnamento di etichette v_k, se e solo se il percorso associato ai pacchetti p nella parte sinistra di ogni coppia $(P, t) \in \tilde{\lambda}$, con $t \neq \bot$, è sempre lo stesso. Formalmente:*

$$\forall (P,t) \in \tilde{\lambda}.\ t \neq \bot \Rightarrow \forall p, p' \in P.\ \mathcal{P}(\mathcal{C}_k, v_k, p, t) = \mathcal{P}(\mathcal{C}_k, v_k, p', t)$$

Nella pratica è possibile ottenere una funzione sintetizzata disgiunta a partire da una funzione sintetizzata qualunque, per i sistemi `iptables`, `pf` e `ipfw`, spezzando ogni coppia $(P, t) \in \tilde{\lambda}$ in quattro parti con trasformazione t e multicubo di pacchetti uguale al sottoinsieme di P in cui gli indirizzi di origine e destinazione sono rispettivamente tutti locali, il primo locale e il secondo non locale, il primo non locale e il secondo locale, e infine tutti non locali. La divisione presentata funziona in quanto gli insiemi prodotti sono tutti dei multicubi (l'intersezione di multicubi è un multicubo) e in quanto la funzione $\tilde{\mathcal{P}}$ restituisce solo insiemi vuoti o singoletti per i sistemi supportati (vedi capitolo 6).

Se la funzione sintetizzata da compilare è disgiunta rispetto al sistema k allora possiamo definire un nuovo predicato $\tilde{\chi}$ tale che la formula seguente sia equivalente all'espressione (iv):

$$\forall (P,t) \in \tilde{\lambda}.\ t \neq \bot \Rightarrow \tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) \qquad \text{dove } \forall p \in P.\ \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, t)$$

Definiamo formalmente il predicato $\tilde{\chi}$ come segue:

$$\tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) = \forall p \in P.\ \chi(\mathcal{C}_k, v_k, \pi, p, t, i(\tilde{f}))$$

Vogliamo una caratterizzazione operativa del predicato $\tilde{\chi}$, tuttavia stavolta non vale direttamente un'equivalente fra il predicato e la caratterizzazione operativa, che chiamiamo quindi $\tilde{\tilde{\chi}}$.

$$\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) = \begin{cases} \exists\, t', t''.\ t' \ltimes t'' = t \wedge t' \in \nu(v(q)) \wedge t'' \in \nu(\ell(\pi, \mathcal{C}_k, v_k)) \wedge \\ \qquad (P, t') \,\tilde{\in}\, \tilde{f}(q) \wedge t'(P) = P' \wedge \\ \qquad \tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi', P', t'', \tilde{f}) & \text{se } \pi = q \cdot \pi' \\ \\ t = id & \text{altrimenti} \end{cases}$$

Dove $(P,t) \,\tilde{\in}\, \tilde{\lambda}$ è vero se e solo se $\forall p \in P.\ i(\tilde{\lambda})(p) = t$; ovvero secondo una qualche divisione degli elementi di P in multicubi vale che a tutti gli elementi di P è associata la trasformazione t da $\tilde{\lambda}$. La notazione evidenzia il fatto che l'operazione più semplice per garantire che $(P,t') \,\tilde{\in}\, \tilde{f}(q)$ sia verificato, consiste nell'aggiungere la coppia (P',t) all'insieme $\tilde{f}(q)$; in effetti questo è più o meno quello che fa l'algoritmo di generazione che presenteremo.

È immediato verificare che $\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) \Rightarrow \tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$; tuttavia se k è un sistema uniterale compatto allora vale anche l'opposto.

Teorema 14. *Per ogni sistema uniterale e compatto k con diagramma di controllo $\mathcal{C}_k$, etichettato secondo v_k, percorso $\pi \in \Pi(\mathcal{C}_k)$, multicubo P e trasformazione t vale che*

$$\forall \tilde{f}.\tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) \iff \tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$$

L'algoritmo di generazione che segue è sostanzialmente una riscrittura in pseudocodice di una valutazione del predicato $\forall (P,t) \in \tilde{\lambda}.\ t \neq \bot \Rightarrow \tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$, in cui la verifica delle condizioni del tipo $(P,t) \in \tilde{f}(q)$ viene usata costruttivamente per determinare il valore di $\tilde{f}$.

Ma prima di presentare l'algoritmo dobbiamo risolvere il problema dei pacchetti scartati, i quali non verificano nessuna delle proprietà necessarie ad essere trattati attraverso il predicato $\tilde{\tilde{\chi}}$. Quello che proponiamo è di costruire $\tilde{f}$ in quattro passaggi: (i) inizializzazione di $\tilde{f}$ come funzione che associa ad ogni nodo un insieme vuoto di coppie (multicubo, trasformazione); (ii) per ogni coppia $(P,t) \in \tilde{\lambda}$ in cui $t \neq \bot$ inserimento in $\tilde{f}$ delle coppie (multicubo trasformazione) necessarie affinché $\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$ sia verificato; (iii) per ogni nodo q, completamento di $\tilde{f}(q)$ affinché definisca una trasformazione per ogni pacchetto in $\mathbb{P}$, dove la trasformazione è scelta in modo tale da scartare il maggior numero possibile di pacchetti; (iv) verifica che tutti i pacchetti che devono essere scartati siano effettivamente scartati dalla configurazione ottenuta, terminazione con errore in caso contrario.

Nella fase (ii), dichiariamo un fallimento quando occorrono assegnamenti contrastanti per un qualche nodo q, come $(P,t) \in \tilde{f}(q)$ e $(P',t') \in \tilde{f}(q)$ con $t \neq t'$ e $P \cap P' \neq \emptyset$. Assumiamo inoltre che ogni aggiornamento della funzione sintetizzata associata ad un nodo q sia effettuato in modo tale da non inserire coppie (P,t) tali che $P \cap P' \neq \emptyset$ per un qualche $(P',t) \in \tilde{f}(q)$. Per questo definiamo una funzione di aggiornamento che nell'aggiungere una coppia (P,t) ad una funzione sintetizzata controlla le coppie già presenti, terminando con fallimento se verifica delle incompatibilità e "ritagliando" P se

si verificano delle sovrapposizioni non contraddittorie. Dato che la fase in cui configuriamo il firewall per gestire i pacchetti da accettare viene prima di quella in cui completiamo la configurazione per scartare più pacchetti possibile, e dato che nella fase (*iii*) non tocchiamo nessuna coppia già inserita in $\tilde{f}$, non è possibile che vengano scartati per errore dei pacchetti da accettare.

Alla fine della fase (*ii*), la configurazione $\tilde{f}$ è tale che assegna ad ogni nodo $q \in Q$ un insieme di coppie (P, t); dato che sono inserite solo le coppie necessarie a realizzare il corretto comportamento riguardo ai nodi accettati, è possibile che questi insiemi non contengano, per ogni p, una coppia (P, t) tale che $p \in P$: cioè ci possono essere dei pacchetti per i quali non abbiamo deciso come il nodo q li debba gestire. Chiamiamo $\mathbf{P}_{\#}$ l'insieme dei multicubi contenenti i pacchetti a cui $\tilde{f}(q)$ non associa trasformazioni. Associamo dunque ai pacchetti in $\mathbf{P}_{\#}$ delle trasformazioni in modo tale da scartare più pacchetti possibili: chiamiamo $\mathbf{P}_{\perp}$ l'insieme dei multicubi contenenti i pacchetti che vengono scartati in q. I pacchetti in $\mathbf{P}_{\perp}$ possono essere scartati direttamente o passati ad un nodo che li scarta a sua volta.

Quello che possiamo fare per scartare i pacchetti in q dipende da $v_k(q)$. Se $DROP \in v_k(q)$ allora aggiungiamo a $\tilde{f}$ una coppia $(P, \perp)$ per ogni P in $\mathbf{P}_{\#}$; cioè scartiamo direttamente tutto il possibile, abbiamo dunque che $\mathbf{P}_{\perp} = \mathbf{P}_{\#}$. Altrimenti, se $DROP \notin v_k(q)$, l'unico modo che abbiamo per scartare un pacchetto p di $\mathbf{P}_{\#}$ è quello di trasformarlo in un pacchetto che venga scartato nel prossimo nodo q' (direttamente o a sua volta attraverso una trasformazione ed un passaggio ad un nodo successivo). Il problema è che il pacchetto p potrebbe visitare nodi diversi in base alla trasformazione associatagli da q, ed è possibile che solo in alcuni dei cammini possa finire per essere scartato, quindi dovremmo considerare tutte le alternative possibili.

Anziché seguire questo approccio, sfruttiamo il fatto che tutti i pacchetti, per essere scartati devono visitare prima o poi un nodo etichettato con $DROP$. Quindi completiamo le funzioni $\tilde{f}(q)$ partendo da questi nodi e seguendo gli archi in A a ritroso, occupandoci per primi dei nodi etichettati con $DROP$, poi dei loro predecessori, dei predecessori dei predecessori e così via. Dato un nodo q per cui abbiamo completato $\tilde{f}(q)$, con $\mathbf{P}_{\perp}$ non vuoto, per ognuno dei predecessori q' di q, tale che $(q', \psi, q) \in A$, filtriamo i multicubi di $\mathbf{P}_{\perp}$ in modo da eliminare i pacchetti che non verificano ψ, chiamiamo l'insieme ottenuto $\mathbf{P}'_{\triangleright\perp}$. Nel nodo q' vogliamo applicare ad ogni pacchetto p una trasformazione t tale che $t(p)$ appartiene ad un multicubo in $\mathbf{P}'_{\triangleright\perp}$.

Se $v(q')$ non contiene né $SNAT$, né $DNAT$, allora l'unica possibilità è quella di associare id ai multicubi in $\mathbf{P}'_{\#}$; occorre comunque calcolare $\mathbf{P}'_{\perp}$, l'insieme dei multicubi che, se lasciati identici, sono passati al nodo q e poi scartati. Questi possono essere calcolati facendo l'intersezione fra i multicubi di $\mathbf{P}'_{\#}$ e quelli di $\mathbf{P}'_{\triangleright\perp}$.

Se $v(q')$ contiene sia $SNAT$ che $DNAT$ allora è possibile prendere un pacchetto qualsiasi $p_{\perp}$ di un multicubo qualsiasi di $\mathbf{P}'_{\triangleright\perp}$ e trasformare ogni pacchetto di $\mathbf{P}'_{\#}$ in $p_{\perp}$ attraverso la trasformazione $cost(p_{\perp}) = (cost(p_{\perp}.sIP) : cost(p_{\perp}.sPort),\ cost(p_{\perp}.dIP) : cost(p_{\perp}.dPort))$. In questo caso dunque, per ogni $P \in \mathbf{P}'_{\#}$, aggiungiamo $(P, cost(p_{\perp}))$ a $\tilde{f}(q')$ e abbiamo che $\mathbf{P}'_{\perp} = \mathbf{P}'_{\#}$.

Se $v(q')$ contiene solo $SNAT$ allora per ogni multicubo di pacchetti $P_1 \in \mathbf{P}'_{\#}$, e per ogni multicubo $P_2 \in \mathbf{P}_{\triangleright\perp}$ prendiamo un pacchetto qualsiasi $p_{\perp} \in P_2$, definiamo una trasformazione $t = (cost(p_{\perp}.sIP) : cost(p_{\perp}.sPort), id : id)$ e calcoliamo P'_1 l'insieme dei pacchetti di P_1 che una volta trasformati secondo t appartengono a P_2, formalmente $P'_1 = \{p \in P_1 \mid t(p) \in P_2\}$; se $P'_1 \neq \emptyset$ allora aggiungiamo la coppia (P'_1, t) in $\tilde{f}(q')$, e P'_1 all'insieme $\mathbf{P}'_{\perp}$.

Il caso in cui $v(q')$ contenga solo $DNAT$ è identico al precedente, dove però la trasformazione t è $(id : id, cost(p_{\perp}.sIP) : cost(p_{\perp}.sPort))$. Una volta completata la configurazione $\tilde{f}$ per quanto riguarda q', proseguiamo passando ai nodi predecessori di q', sfruttando l'insieme $\mathbf{P}'_{\perp}$ che abbiamo calcolato.

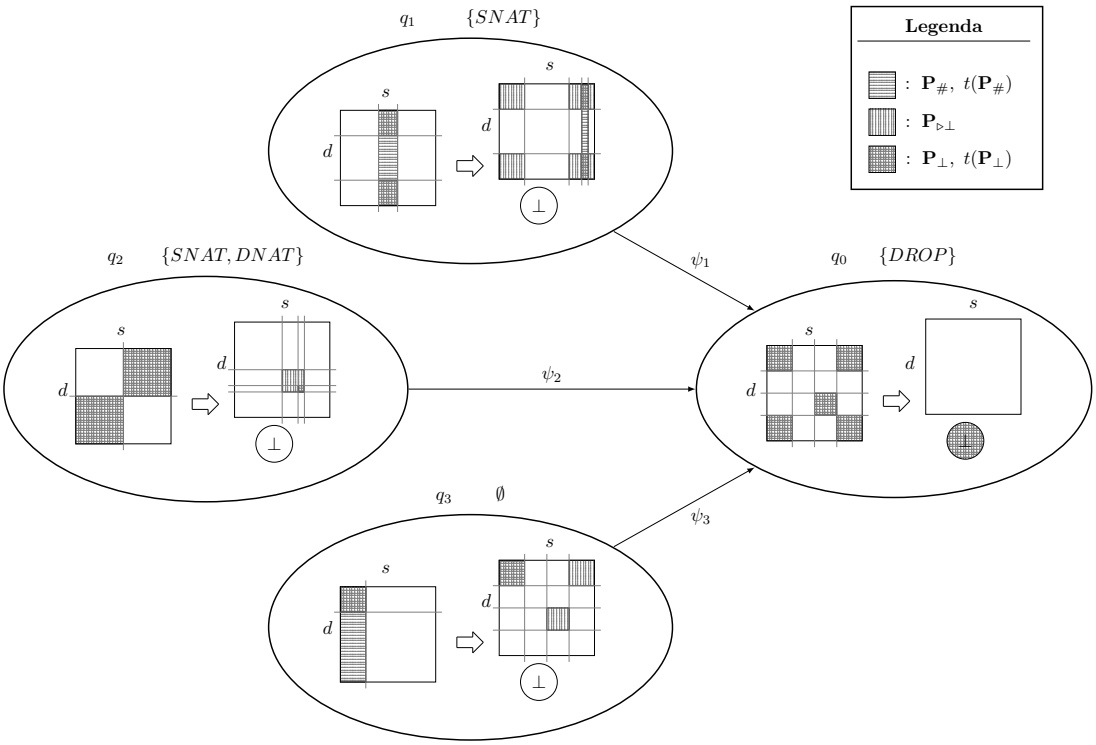


Figura 7.1: Esempio di come vengono completati gli insiemi di coppie (multicubo, trasformazione) assegnate ai nodi del diagramma di controllo: in ogni nodi rappresentiamo l'insieme dei pacchetti $\mathbb{P}$ prima e dopo l'applicazione delle trasformazioni.

Si veda ad esempio la figura 7.1, contenente quattro nodi di un diagramma di controllo, con i relativi nomi e l'insieme delle etichette assegnate loro da v_k. In ogni nodo abbiamo rappresentato la funzione di trasformazione dei pacchetti, assegnatagli dalla configurazione astratta alla fine della fase (*ii*): a sinistra abbiamo il dominio della funzione, l'insieme $\mathbb{P}$ rappresentato su due dimensioni per comodità (indirizzo di origine e di destinazione), a destra abbiamo il codominio costituito dall'insieme $\mathbb{P}$ stesso più $\perp$. Nel dominio di ogni nodo, abbiamo rappresentato con delle linee orizzontali l'insieme $\mathbf{P}_{\#}$ dei pacchetti per i quali non c'è trasformazione assegnata, e con una quadrettatura l'insieme $\mathbf{P}_{\perp}$ dei pacchetti che riusciamo a scartare. Nei nodi che non sono etichettati con $DROP$, l'insieme di pacchetti $\mathbf{P}_{\triangleright\perp}$ nei quali vogliamo trasformare i pacchetti da scartare, affinché siano scartati dal nodo successivo, è rappresentato da linee verticali. Nel codominio rappresentiamo anche le immagini di $\mathbf{P}_{\#}$ e $\mathbf{P}_{\perp}$, secondo la solita notazione.

Il nodo q_0 è etichettato con $DROP$, pertanto possiamo scartare direttamente tutti i pacchetti in $\mathbf{P}_{\#}$ assegnando loro la trasformazione $\perp$; poiché tutti i pacchetti in $\mathbf{P}_{\#}$ sono scartati, essi sono anche in $\mathbf{P}_{\perp}$; abbiamo usato quindi la quadrettatura. Nel codominio del nodo q_1, abbiamo rappresentato con le righe verticali $\mathbf{P}_{\triangleright\perp}$, cioè i pacchetti dell'insieme $\mathbf{P}_{\perp}$ di q_0 che verificano ψ_1. Il nodo è etichettato soltanto con $SNAT$, pertanto riusciamo a scartare solo i pacchetti in $\mathbf{P}_{\#}$ che hanno indirizzo di destinazione presente anche in $\mathbf{P}_{\triangleright\perp}$, mentre ogni indirizzo di origine va bene in quanto è possibile trasformarlo in quello di un pacchetto qualsiasi appartenente a $\mathbf{P}_{\triangleright\perp}$. Il nodo q_2 è etichettato sia con $SNAT$, sia con $DNAT$, pertanto riusciamo a scartare tutti i pacchetti in $\mathbf{P}_{\#}$ trasformandoli in un pacchetto qualsiasi appartenente a $\mathbf{P}_{\triangleright\perp}$. Nel nodo q_3 non abbiamo etichette, quindi possiamo solo assegnare l'identità come trasformazione ai pacchetti in $\mathbf{P}_{\#}$; verranno scartati solo quelli che hanno indirizzo di origine e destinazione all'interno di $\mathbf{P}_{\triangleright\perp}$, ovvero l'intersezione fra i due insiemi.

La configurazione ottenuta scarta ogni pacchetto possibile; tuttavia non abbiamo alcuna garanzia

che vengano scartati tutti i pacchetti che devono esserlo. Per questo alla fine della generazione di $\tilde{f}$, nella fase (*iv*), verifichiamo che ogni coppia $(P, \bot) \in \tilde{\lambda}$ sia rispettata dal firewall prodotto, in caso contrario terminiamo segnalando errore.

Si noti che nella fase (*iii*) ignoriamo la possibilità di scartare i pacchetti dirottandoli verso un percorso che contenga un loop. È possibile per questo che l'algoritmo fallisca anche in casi in cui esiste una configurazione corretta. Il comportamento dei firewall rispetto ai pacchetti ciclanti serve a gestire una situazione di errore (il loop), pertanto preferiamo non sfruttarlo deliberatamente in fase di configurazione.

La funzione FIREWALL_GENERATION dell'algoritmo 4 realizza l'algoritmo di generazione (riga 1). Gli input sono il firewall astratto sintetizzato $\tilde{\lambda}$, il diagramma di controllo $\mathcal{C}_k$ e l'assegnamento di etichette v_k. L'obiettivo della funzione è quello di restituire una configurazione semiastratta sintetizzata $\tilde{f}$. Come prima cosa si inizializza questa funzione, assegnando ad ogni nodo del diagramma l'insieme vuoto di coppie (multicubo di pacchetti, trasformazione), righe 3 e 4. Usiamo una notazione ad array per la funzione $\tilde{f}$ per comodità. Le righe 5, 6, 7 e 8 corrispondono alla generazione della parte di $\tilde{f}$ che verifica le coppie di $\tilde{\lambda}$ in cui $t \neq \bot$. La riga 9 realizza invece il riempimento delle funzioni sintetizzate associate ai nodi, attraverso la funzione FILL che scarta tutto il possibile e assegna *id* come trasformazione di default ai pacchetti che non riesce a scartare. Questa funzione è definita nell'algoritmo 5. Infine le righe 10 e 11 verificano che la configurazione prodotta rispetti le coppie $(P, \bot) \in \tilde{\lambda}$.

Alla riga 7 assumiamo di aver accesso alla funzione $\mathcal{P}$; in realtà per i sistemi supportati consulteremo la tabella relativa a $\tilde{\mathcal{P}}$ presentata nel capitolo 6. La costruzione di $\tilde{f}$ è effettuata grazie alla funzione ricorsiva CHI, che implementa la verifica del predicato $\tilde{\tilde{\chi}}$. I parametri sono: una configurazione semiastratta sintetizzata $\tilde{f}$, un assegnamento di etichette v_k, un percorso π, un multicubo di pacchetti P e una trasformazione t.

Alla riga 11 usiamo una funzione PASS che dato un diagramma di controllo $\mathcal{C}_k$, una configurazione semiastratta sintetizzata $\tilde{f}$ e un multicubo di pacchetti P, restituisce *true* se e solo se almeno uno dei pacchetti $p \in P$ passa attraverso il firewall $(\mathcal{C}_k, i(\tilde{f}))$ (cioè non viene scartato). Usiamo questa funzione per verificare che i pacchetti da scartare siano gestiti correttamente.

La funzione CHI copia la funzione sintetizzata ricevuta in input (riga 14), successivamente, se il percorso non è vuoto, divide la trasformazione t in due parti t' e t'' tali che $t'' \ltimes t' = t$ (riga 18), aggiorna $\tilde{f}'$ con la coppia (P, t'), controllando se la coppia contraddice altre coppie inserite precedentemente attraverso la funzione update (riga 19), calcola l'insieme di pacchetti che deve gestire il prossimo nodo del cammino[1] ed effettua una chiamata ricorsiva aggiornando la configurazione, l'insieme di pacchetti e la trasformazione (riga 21).

La funzione DIVIDE prende come input una trasformazione da scomporre t e un insieme di etichette L (riga 24), e restituisce una coppia di trasformazioni, di cui la prima sarà applicata dal nodo a cui fa riferimento l'insieme di etichette e la seconda sarà applicata dal resto del percorso. Le trasformazioni sono ricavate campo per campo (riga 25): per ognuno di questi, se la trasformazione t è *id* allora entrambe t' e t'' sono *id* (righe 26, 27 e 28); altrimenti, se la trasformazione è $cost(a)$ per qualche a, se le etichette del nodo permettono la trasformazione del campo allora t' trasforma il campo e t'' è *id* (righe 32 e 33), altrimenti vale il contrario (righe 35 e 36).

La funzione update prende come input una funzione sintetizzata $\tilde{\lambda}$, un multicubo di pacchetti P e una trasformazione t (riga 39), e restituisce la funzione sintetizzata aggiornata con la nuova coppia (P, t), controllando però che P non intersechi nessun insieme di pacchetti già trattato da $\tilde{\lambda}$. La procedura scorre tutto l'insieme di coppie $\tilde{\lambda}$ in modo ricorsivo, se trova una coppia (P', t') tale che P e P' hanno degli elementi in comune e $t \neq t'$ allora c'è una contraddizione e l'algoritmo

[1] Con un abuso di notazione abbiamo scritto $t(P)$ per intendere $\{t(p) \mid p \in P\}$.

Algorithm 4

function FIREWALL_GENERATION($\tilde{\lambda}$: funzione su pacchetti sintetizzata, $\mathcal{C}_k$: diagramma di controllo, v_k: assegnamento di etichette)
    $(Q, A, q_i, q_f) = \mathcal{C}_k$
    for all $q \in Q$ **do**
        $\tilde{f}[q] \leftarrow \emptyset$
    for all $(P, t) \in \tilde{\lambda}$ **do**
        if $t \neq \perp$ **then**
            $\{\pi\} \leftarrow \mathcal{P}(\mathcal{C}_k, v_k, P, t)$
            $\tilde{f} \leftarrow$ CHI($\tilde{f}$, v_k, π, P, t)
    $\tilde{f} \leftarrow$ FILL($\tilde{f}$, $\mathcal{C}_k$, v_k)
    for all $(P, t) \in \tilde{\lambda}$ **do**
        if $t = \perp \wedge$ PASS($\tilde{f}$, $\mathcal{C}_k$, P) **then** FAIL
    return $\tilde{f}$

function CHI($\tilde{f}$: configurazione, v_k: assegnamento di etichette, π: percorso, P: multicubo di pacchetti, t: trasformazione)
    $\tilde{f}' \leftarrow \tilde{f}$
    if LENGTH(π) > 0 **then**
        $q \cdot \pi' \leftarrow \pi$
        $(t', t'') \leftarrow$DIVIDE(t, $v_k(q)$)
        $\tilde{f}'[q] \leftarrow$UPDATE($\tilde{f}[q]$, P, t')
        $P' \leftarrow t'(P)$
        $\tilde{f}' \leftarrow$ CHI($\tilde{f}'$, v_k, π', P', t'')
    return $\tilde{f}'$

function DIVIDE(t: trasformazione, L: insieme di etichette)
    for all $x \in \{sIP, sPort, dIP, dPort\}$ **do**
        if $t.x = id$ **then**
            $t'.x \leftarrow id$
            $t''.x \leftarrow id$
        else
            $cost(a) \leftarrow t.x$
            if $x \in \mu(L)$ **then**
                $t'.x \leftarrow cost(a)$
                $t''.x \leftarrow id$
            else
                $t'.x \leftarrow id$
                $t''.x \leftarrow cost(a)$
    return (t', t'')

function UPDATE($\tilde{\lambda}$: funzione sintetizzata, P: multicubo, t: trasformazione)
    if $\tilde{\lambda} = \emptyset$ **then return** $\{(P, t)\}$
    else
        $(P', t') \cup \tilde{\lambda}' \leftarrow \tilde{\lambda}$
        $(P_s, \mathbf{P}_n) \leftarrow$SPLIT($P$, id, $\in P'$)
        if $P_s \neq \emptyset \wedge t \neq t'$ **then** FAIL
        return $\{(P', t')\} \cup \bigcup_{P'' \in \mathbf{P}_n}$ UPDATE($\tilde{\lambda}'$, P'', t)

Algorithm 5

function FILL($\tilde{f}$: configurazione semiastratta sintetizzata, $\mathcal{C}_k$: diagramma di controllo, v_k: assegnamento di etichette)
  $(Q, A, q_i, q_f) = \mathcal{C}_k$
  $\tilde{f}' \leftarrow \tilde{f}$
  for all $q \in Q$ **do**
    if $DROP \in v(q)$ **then**
      $\mathbf{P}_\perp \leftarrow$REST($\tilde{f}'(q)$)
      if $\mathbf{P}_\perp \neq \emptyset$ **then**
        for all $P \in \mathbf{P}_\perp$ **do**
          $\tilde{f}' \leftarrow \tilde{f}' \cup \{(P, \perp)\}$
        $\tilde{f}' \leftarrow$BACK_FILL($\tilde{f}$, q, $\mathcal{C}_k$, v_k, $\mathbf{P}_\perp$)
  $\tilde{f}' \leftarrow$ FILL_ALL($\tilde{f}'$, id)
  return $\tilde{f}'$

function BACK_FILL($\tilde{f}$: configurazione semiastratta sintetizzata, q: nodo, $\mathcal{C}_k$: diagramma di controllo, v_k: assegnamento di etichette, $\mathbf{P}_\perp$: insieme di multicubi)
  $(Q, A, q_i, q_f) = \mathcal{C}_k$
  $\tilde{f}' \leftarrow \tilde{f}$
  for all $(q', \psi) \in$ PREDECESSORI(q, A) **do**
    $\mathbf{P}'_\# \leftarrow$REST($\tilde{f}'(q')$)
    $\mathbf{P}'_{\triangleright\perp} \leftarrow$FILTER_BACK($\mathbf{P}_\perp$, ψ)
    $\mathbf{P}'_\perp \leftarrow \emptyset$
    if $DROP \in v(q') \wedge \mathbf{P}'_{\triangleright\perp} \neq \emptyset \wedge \mathbf{P}'_\# \neq \emptyset$ **then**
      if $SNAT \in v(q') \wedge DNAT \in v(q')$ **then**
        $p_\perp \leftarrow$ TAKE_ONE(TAKE_ONE($\mathbf{P}''_\perp$))
        $\tilde{f}' \leftarrow \tilde{f}' \cup (\bigcup_{p \in \mathbf{P}'_\#} \{(P, cost(p_\perp))\})$
        $\mathbf{P}'_\perp \leftarrow \mathbf{P}'_\#$
      else
        while $\mathbf{P}'_\# \neq \emptyset$ **do**
          $P' \leftarrow$ HEAD($\mathbf{P}'_\#$)
          $\mathbf{P}'_\# \leftarrow$ TAIL($\mathbf{P}'_\#$)
          for all $P'' \in \mathbf{P}'_{\triangleright\perp}$ **do**
            $p_\perp \leftarrow$ TAKE_ONE(P'')
            if $SNAT \in v(q') \wedge DNAT \notin v(q')$ **then**
              $t \leftarrow (cost(p_\perp.sIP) : cost(p_\perp.sPort),\ id : id)$
            else if $SNAT \notin v(q') \wedge DNAT \in v(q')$ **then**
              $t \leftarrow (id : id,\ cost(p_\perp.dIP) : cost(p_\perp.dPort))$
            else
              $t \leftarrow id$
            $(P_s, \mathbf{P}_N) \leftarrow$ SPLIT(P', t, $p \in P''$)
            if $P_s \neq \emptyset$ **then**
              $\tilde{f}' \leftarrow \tilde{f}' \cup \{(P_S,\ t)\}$
              $\mathbf{P}'_\# \leftarrow \mathbf{P}'_\# \cup \mathbf{P}_N$
              $\mathbf{P}'_\perp \leftarrow \mathbf{P}'_\perp \cup \{P_S\}$
              break
    if $\mathbf{P}'_\perp \neq \emptyset$ **then**
      $f' \leftarrow$BACK_FILL($\tilde{f}'$, q', $\mathcal{C}_k$, v_k, $\mathbf{P}'_\perp$)
  return $\tilde{f}'$

segnala fallimento; altrimenti l'insieme P viene filtrato per togliere i pacchetti a cui $\tilde{\lambda}$ assegna già la trasformazione t, e a quello che rimane viene assegnata la trasformazione t aggiungendo nuove coppie alla funzione $\tilde{\lambda}$. Per prima cosa si controlla se siamo nel caso base, ovvero se non esistono più coppie all'interno dell'insieme $\tilde{\lambda}$, in questo caso (riga 40) si inserisce semplicemente la nuova coppia (P, t). Altrimenti si prende una coppia (P', t') da $\tilde{\lambda}$ e si divide P nel multicubo P_s, corrispondente all'intersezione con P', e nell'insieme di multicubi $\mathbf{P}_n$, che non intersecano P'. Per fare questo si usa la funzione SPLIT introdotta nel capitolo 5 (riga 43), controllando che la parte a comune fra P e P' associ la stessa trasformazione e in caso contrario terminando con fallimento (riga 44). Alla fine viene restituita l'unione della coppia (P', t') con i risultati delle chiamate ricorsive, una per ogni multicubo in $\mathbf{P}_n$, sul resto delle coppie di $\tilde{\lambda}$ (riga 45). Si noti che la coppia (P', t') viene appunto reinserita così com'è nel risultato, infatti la funzione update non modifica nessuna coppia già presente nel parametro $\tilde{\lambda}$, adegua le coppie nuove che inserisce a quelle già presenti.

La funzione FILL, dell'algoritmo 5 (riga 1), ha lo scopo di aggiungere coppie a $\tilde{f}$ per scartare quanti più pacchetti possibile, e completare la configurazione assegnando ai pacchetti rimanenti la trasformazione *id*.

Partiamo sequenzialmente da ogni nodo etichettato con *DROP* (riga 5), assumiamo alla riga 6 di avere una funzione REST($\tilde{\lambda}$: funzione sintetizzata su pacchetti) che dato un insieme di coppie (multicubo di pacchetti, trasformazione) $\tilde{\lambda}$, restituisce un insieme di multicubi contenenti tutti e soli i pacchetti di $\mathbb{P}$ che non compaiono in nessuna parte sinistra di $\tilde{\lambda}$. Se l'insieme dei multicubi contenenti pacchetti non trattati da $\tilde{f}'(q)$ non è vuoto allora aggiungiamo a $\tilde{f}'(q)$ una coppia $(P, \perp)$ per ogni multicubo P di pacchetti non trattati (riga 9), e propaghiamo l'aggiornamento di $\tilde{f}'$ all'indietro nel diagramma di controllo, attraverso la funzione ricorsiva BACK_FILL (riga 10). Dopo aver aggiornato la configurazione per scartare quanti più pacchetti possibile, potrebbero ancora esistere dei pacchetti non trattati, pertanto invochiamo la funzione FILL_ALL($\tilde{f}'$, *id*) (riga 11). Assumiamo che la funzione FILL_ALL($\tilde{f}$, t) assegni la trasformazione t ad ogni pacchetto libero nella configurazione $\tilde{f}$.

La funzione BACK_FILL($\tilde{f}$: configurazione semiastratta sintetizzata, q: nodo, C_k : diagramma di controllo, v_k : assegnamento di etichette, $\mathbf{P}_\perp$: insieme di multicubi), definita alla riga 14, dato un nodo q e l'insieme dei multicubi $\mathbf{P}_\perp$ contenenti i pacchetti che vengono scartati da q, restituisce una versione della configurazione $\tilde{f}$ aggiornata in modo tale che i predecessori di q assegnino ai propri pacchetti delle trasformazioni tali da far sì che più pacchetti possibili siano ridiretti verso $\mathbf{P}_\perp$ in q. Assumiamo (riga 17) di avere a disposizione una funzione PREDECESSORI(q: nodo, A: insieme di archi) che restituisce l'insieme delle coppie (q', ψ) tale che $(q', \psi, q) \in A$. Inoltre, assumiamo (riga 17) di avere a disposizione una funzione FILTER_BACK($\mathbf{P}$: insieme di multicubi di pacchetti, ψ: condizione sui pacchetti) che restituisce una copia di $\mathbf{P}$ nella quale in ogni multicubo sono rimossi i pacchetti che non verificano ψ.

$$\text{FILTER\_BACK}(\mathbf{P}, \psi) = \{P' \mid P \in \mathbf{P} \wedge P' = \psi(P) \wedge P' \neq \emptyset\}$$

Come nel capitolo 5 assumiamo che ψ possa essere scomposta in una congiunzione di predicati, uno per ogni campo del pacchetto; pertanto l'insieme dei pacchetti che verificano ψ è un multicubo e quindi l'operazione alla base della funzione FILTER_BACK è l'intersezione fra multicubi.

La funzione TAKE_ONE(X: insieme) è una funzione di utilità che dato un insieme restituisce uno qualunque fra i suoi elementi. Per quanto riguarda $\mathbf{P}'_\#$ usiamo le funzioni HEAD e TAIL per specificare che seguiamo un ordine nella valutazione dei multicubi, e l'unione alla riga 41 inserisce infondo all'insieme. Il risultato è che i multicubi in $\mathbf{P}_N$ sono valutati all'interno del ciclo **while**.

Teorema 15 (Correttezza del firewall generato)**.** *Se il sistema k è senza NAT ripetuti e la funzione sintetizzata $\tilde{\lambda}$ è disgiunta, se esiste una configurazione $\Sigma \in \Gamma_k$ tale che $i(\tilde{\lambda}) = (\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!)$ e*

tale che non esistono pacchetti ciclanti in $(\mathcal{C}_k, \Sigma)$*, allora l'algoritmo 4 restituisce una configurazione semiastratta sintetizzata* $\tilde{f}$ *tale che*

$$i(\tilde{f}) \in \mathbb{M}_3(\mathcal{C}_k, v_k) \wedge \forall p \in \mathbb{P}.\ \odot(\mathcal{C}_k, i(\tilde{f}))(p) = i(\tilde{\lambda})(p)$$

altrimenti l'algoritmo termina segnalando errore.

7.5 Generazione diretta della configurazione IFCL usando i tag

Il metodo seguente prevede una generazione diretta della configurazione IFCL, senza passare da firewall semiastratti, e si basa sul campo *tag* dei pacchetti. Più nel dettaglio, la generazione segue tre fasi:

- a partire dalla configurazione astratta $\tilde{\lambda}$, generiamo una ruleset IFCL R_λ che ne è la traduzione coppia per coppia;
- creiamo quattro ruleset IFCL derivate da R_λ: R_{fil}, R_{snat}, R_{dnat} e R_{nat};
- assegniamo ad ogni nodo del diagramma di controllo del sistema target una ruleset composta a partire da quelle prodotte dal passo precedente o R_ϵ, la rulset vuota.

Parleremo della correttezza della configurazione prodotta in termini di semantica della configurazione IFCL e della sua concretizzazione come file di configurazione per il sistema target.

7.5.1 Generazione delle ruleset

Per prima cosa a partire dalla funzione astratta sintetizzata $\tilde{\lambda}$ generiamo la ruleset R_λ in modo tale che la valutazione semantica della ruleset corrisponda all'interpretazione della funzione astratta:

$$(\!|R_\lambda|\!)(s_{\texttt{NEW}}) = i(\tilde{\lambda})$$

Questo viene ottenuto concatenando, per ogni coppia $(P, t) \in \tilde{\lambda}$, una regola $(p \in P, target(t))$, dove $target(t)$ è un target la cui applicazione realizza la trasformazione t.

$$target(t) = \begin{cases} \texttt{ACCEPT} & \text{se } t = id \\ \texttt{DROP} & \text{se } t = \bot \\ \texttt{NAT}(d_n, s_n) & \text{altrimenti} \end{cases} \qquad \text{dove } d_n \text{ e } s_n \text{ tali che } tr_{nat}(d_n, s_n) = t$$

Per definizione di funzione astratta sintetizzata, nessuna delle regole si sovrappone con le altre, e quindi l'ordine è completamente irrilevante.

A questo punto a partire dalla ruleset R_λ, attraverso la funzione RULESET_GENERATION dell'algoritmo 6 (riga 1), creiamo quattro ruleset: R_{fil}, R_{snat}, R_{dnat} e R_{nat}. L'obiettivo è quello di produrre una ruleset R_{mark} che assegni ad ogni multicubo P presente in $\tilde{\lambda}$ un'etichetta diversa per il campo *tag*; di fare in modo che questa ruleset sia sempre la prima ad essere valutata per ogni pacchetto; e di far dipendere le modifiche effettuate al pacchetto unicamente sulla base del campo *tag*. Per ogni riga della ruleset R_λ, se il target è ACCEPT allora la regola viene inserita nella ruleset R_{fil} (riga 5). Altrimenti, se siamo di fronte ad una regola di NAT, viene generato un nuovo tag m, attraverso la procedura NEW_TAG che assumiamo restituire sempre nomi freschi (riga 7); si aggiunge a R_{mark} una regola che associa il tag generato ai pacchetti che verificano la condizione e che non sono ancora stati

Algorithm 6

1: **function** RULESET_GENERATION(R_λ)
2: $R_{nat} = R_{dnat} = R_{fil} = R_{snat} = R_{mark} = \epsilon$
3: **for** r in R_λ **do**
4: **if** $r = (\phi, \texttt{ACCEPT})$ **then**
5: $R_{fil} \leftarrow R_{fil} \cdot r$
6: **else if** $r = (\phi, \texttt{NAT}(d_n, s_n))$ **then**
7: $m \leftarrow$ NEW_TAG( )
8: $R_{mark} \leftarrow R_{mark} \cdot (\phi \wedge tag(p) = \bullet, \texttt{MARK}(m))$
9: $R_{dnat} \leftarrow R_{dnat} \cdot (tag(p) = m, \texttt{NAT}(d_n, \star))$
10: $R_{snat} \leftarrow R_{snat} \cdot (tag(p) = m, \texttt{NAT}(\star, s_n))$
11: $R_{nat} \leftarrow R_{nat} \cdot (tag(p) = m, \texttt{NAT}(d_n, s_n))$
12: $R_{fil} \leftarrow R_{fil} \cdot (tag(p) \neq \bullet, \texttt{ACCEPT}) \cdot (true, \texttt{DROP})$
13: $R_{dnat} \leftarrow R_{mark} \cdot R_{dnat}$
14: $R_{fil} \leftarrow R_{mark} \cdot R_{fil}$
15: $R_{snat} \leftarrow R_{mark} \cdot R_{snat}$
16: $R_{nat} \leftarrow R_{mark} \cdot R_{nat}$
17: **return** $(R_{fil}, R_{snat}, R_{dnat}, R_{nat})$

etichettati (riga 8), e si aggiunge una regola che applica la trasformazione ai pacchetti etichettati con m alle rimanenti ruleset, applicando per ogni ruleset solo la parte di trasformazione corrispondente (righe 9, 10 e 11). Nella condizione $tag(p) = \bullet$, assumiamo che $\bullet$ sia il valore di default per il campo tag, verificano la condizione tutti e soli i pacchetti che non sono ancora stati etichettati.

La condizione $tag(p) = \bullet$ serve perché altri pacchetti, non in P, dopo aver subito qualche trasformazione in altri nodi, potrebbero verificare la condizione ϕ, ma su loro non voglio applicare la trasformazione legata all'etichetta m. Per concludere, se ad un pacchetto è associato un tag allora non lo dobbiamo scartare, altrimenti, se nessuna regola con target ACCEPT si applica al pacchetto e questi non ha alcun tag assegnato, lo scartiamo con un'ultima regola $(true, \texttt{DROP})$ (riga 12). La ruleset R_{mark} viene preposta a tutte le altre ruleset (righe 13, 14, 15 e 16), come abbiamo già detto infatti, l'algoritmo funziona se l'assegnamento di etichette è la prima cosa che facciamo su ogni pacchetto in transito sul firewall.

7.5.2 Assegnamento delle ruleset ai nodi

In [4] si propone un assegnamento delle ruleset ai nodi di `iptables` e `pf`. Daremo un metodo per la generazione di un assegnamento di ruleset sulla base dell'assegnamento di etichette v_k coerente con i sistemi già trattati ed applicabile anche a `ipfw`.

Assumiamo di avere un assegnamento di etichette per il sistema target v_k, la funzione $c : Q \to \rho$ vale dunque:

$$c(q) = c_{nat}(q) \cdot c_{fil}(q)$$

$$c_{nat}(q) = \begin{cases} R_{snat} & \text{se } SNAT \in v_k(q) \wedge DNAT \notin v_k(q) \\ R_{dnat} & \text{se } SNAT \notin v_k(q) \wedge DNAT \in v_k(q) \\ R_{nat} & \text{se } SNAT \in v_k(q) \wedge DNAT \in v_k(q) \\ R_\epsilon & \text{altrimenti} \end{cases} \qquad c_{fil}(q) = \begin{cases} R_{fil} & \text{se } DROP \in v_k(q) \\ R_\epsilon & \text{altrimenti} \end{cases}$$

Da notare che per quanto riguarda *pf* ed *iptables* l'assegnamento di ruleset ai nodi del diagramma è lo stesso descritto nell'articolo.

7.5.3 Correttezza della configurazione generata

Nell'articolo [4] si dimostra che se ogni percorso $\pi \in \Pi(\mathcal{C}_k)$ passa da almeno un nodo a cui è assegnato R_{fil} (quindi nel nostro caso un nodo con etichetta $DROP$), allora il firewall generato accetta tutti e soli i pacchetti accettati dal firewall originale. Notiamo che, oltre a quelli presentati precedentemente, anche l'assegnamento di ruleset che abbiamo dato per `ipfw` rispetta la condizione.

Dimostriamo qualcosa di più forte, se la funzione su pacchetti da compilare è localmente fattibile e non ci sono NAT ripetuti sul diagramma di controllo allora la semantica del firewall prodotto è esattamente la stessa del firewall di partenza.

Teorema 16. *Sia* $\rho = \{R_{snat}, R_{dnat}, R_{nat}, R_{fil}, R_{snat} \cdot R_{fil}, R_{dnat} \cdot R_{fil}, R_{nat} \cdot R_{fil}, R_\epsilon\}$*, dove* R_{fil}*,* R_{snat}*,* R_{dnat} *e* R_{nat} *sono prodotto dall'algoritmo 6 con input* R_λ*. Sia* $c : Q \to \rho$ *l'assegnamento di ruleset ai nodi del diagramma di controllo del sistema target* $\mathcal{C}_k$*, generato secondo* v_k*. Se il sistema target* k *è senza* NAT *ripetuti, se ogni percorso da* q_i *a* q_f *comprende almeno un nodo etichettato con* $DROP$*, se l'interpretazione di* $\tilde{\lambda}$ *è localmente fattibile dal sistema target,* $\epsilon_0(i(\tilde{\lambda}), \mathcal{C}_k, v_k)$*, e se le etichette sugli archi non predicano sul campo tag, allora la semantica del firewall* $(\mathcal{C}_k, \Sigma)$ *con* $\Sigma = (\rho, c)$ *per lo stato* $s_{\texttt{NEW}}$ *è identica all'interpretazione di* $\tilde{\lambda}$*.*

$$(\!|(\mathcal{C}_k, \Sigma)|\!)(s_{\texttt{NEW}}) = i(\tilde{\lambda})$$

Notiamo che questo non vale per `ipfw` che però ha un forma abbastanza peculiare in quanto ogni percorso $\pi \in \Pi(\mathcal{C}_{\texttt{ipfw}})$ contiene solo nodi con assegnate tutte le etichette possibili (a eccezione del nodo iniziale e di quello finale).

Questo secondo teorema garantisce la correttezza della generazione della configurazione per `ipfw`.

Teorema 17. *Sia* $\rho = \{R_{snat}, R_{dnat}, R_{nat}, R_{fil}, R_{snat} \cdot R_{fil}, R_{dnat} \cdot R_{fil}, R_{nat} \cdot R_{fil}, R_\epsilon\}$*, dove* R_{fil}*,* R_{snat}*,* R_{dnat} *e* R_{nat} *sono prodotto dall'algoritmo 6 con input* R_λ*. Sia* $c : Q \to \rho$ *l'assegnamento di ruleset ai nodi del diagramma di controllo del sistema target* $\mathcal{C}_k$*, generato secondo* v_k*. Se tutti i percorsi del sistema target* k*,* $\Pi(\mathcal{C}_k)$*, sono tali che* $\ell(\ddot{\pi}) = \{SNAT, DNAT, DROP\}$*, e se nessun pacchetto a cui siano applicate trasformazioni* SNAT *e* DNAT *al massimo una volta percorre dei loop nel diagramma di controllo, allora la semantica del firewall* $(\mathcal{C}_k, \Sigma)$ *con* $\Sigma = (\rho, c)$ *per lo stato* $s_{\texttt{NEW}}$ *è identica all'interpretazione di* $\tilde{\lambda}$*.*

$$(\!|(\mathcal{C}_k, \Sigma)|\!)(s_{\texttt{NEW}}) = i(\tilde{\lambda})$$

7.5.4 Problemi di concretizzazione

Come mai abbiamo deciso di progettare un algoritmo che non facesse uso di tag se quello proposto nell'articolo [4] funziona correttamente? Il motivo principale è legato al fatto che il campo *tag* stesso, e l'operazione MARK nei vari linguaggi di configurazione, sono soggetti a vincoli differenti, rendendo difficile la concretizzazione della configurazione IFCL in un file di configurazione per il sistema target. Per i sistemi attualmente supportati abbiamo individuato delle tecniche di compilazione ad hoc, questo però è un problema da risolvere singolarmente per ogni nuovo sistema da supportare, e spesso per permettere le operazioni di cui abbiamo bisogno è necessario produrre delle configurazioni bizantine.

Le parti delle ruleset prodotte dall'algoritmo 6 immediatamente individuabili come potenzialmente problematiche da implementare nei linguaggi di configurazione target sono:

1. La condizione $tag(p) = \bullet$, in quanto non è definito il tag nullo in nessuno dei linguaggi di configurazione supportato. Questa condizione può essere sostituita chiaramente con un controllo del tipo $tag(p) \in M$ dove M è l'insieme dei tag creati dall'algoritmo, ma solo se il linguaggio permette di esprimere condizioni come l'appartenenza del *tag* ad un insieme.

2. La condizione $tag(p) \neq \bullet$, che soffre dello stesso problema che può però essere risolto in questo caso creando più regole, una per ogni $m \in M$ con condizione $tag(p) = m$ e aventi tutte l'azione associata alla regola iniziale (ovvero ACCEPT).

3. Il fatto di dover modificare il campo *tag* dei pacchetti, attraverso il target MARK in ogni nodo del diagramma di controllo, o comunque di dover eseguire la ruleset R_{mark} come prima cosa per ogni pacchetto.

4. Il fatto che in IFCL, dopo aver compiuto l'operazione MARK, la valutazione della ruleset prosegue, mentre in alcuni linguaggi l'operazione di modifica del *tag* corrisponde necessariamente all'accettazione immediata. La traduzione deve tener conto di questo usando un'istruzione del linguaggio target che scrive il campo *tag* ma che lascia che il pacchetto continui la valutazione della ruleset, oppure deve garantire la preservazione della semantica associando, già al momento dell'applicazione del *tag*, l'azione corretta.

Valutiamo questi potenziali problemi nei sistemi attualmente supportati e proponiamo soluzioni adeguate. Per risolvere questi problemi è necessario in alcuni casi modificare le ruleset prodotte. La mancanza di un approccio unificato, estendibile a nuovi sistemi è il motivo principale che ci ha portati a sviluppare un nuovo algorimo per la generazione di configurazioni IFCL che, anche se per il momento non supporta tutti i sistemi, crediamo vada nella direzione più corretta per individuare una soluzione generale al problema della transcompilazione fra linguaggi di configurazione.

ipfw

In `ipfw` i *tag* sono numeri interi ed è supportato il filtro su intervalli di *tag*, quindi se generiamo i *tag* in ordine e teniamo traccia del massimo possiamo risolvere i punti 1 e 2 usando `not tagged 0-max` per esprimere $tag(p) = \bullet$ e `tagged 0-max` per esprimere $tag(p) \neq \bullet$. L'ordine delle regole in `ipfw` non è soggetto a restrizioni quindi non abbiamo nessun problema per quanto riguarda il punto 3. L'applicazione di un *tag* al pacchetto non è un'azione di per sé in `ipfw` ma un'opzione associata ad un'azione. Pertanto in `ipfw` non è direttamente esprimibile un'istruzione del tipo $(\phi, \text{MARK}(m))$. Questo ci crea un problema con il punto 4, che possiamo risolvere in due modi:

- possiamo sfruttare l'azione `count`, che non ha effetti sulla valutazione del pacchetto da parte della ruleset, ed esprimere $(\phi, \text{MARK}(m))$ come `ipfw -q add count tag m` ϕ. Questa è sicuramente la soluzione più semplice ma usa l'azione `count` in modo sicuramente diverso da quello atteso e rende la configurazione prodotta poco leggibile e potrebbe interferire con gli strumenti che usano `count`, per esempio per monitorare o debuggare la configurazione.

- possiamo, attraverso una fase di preprocessing, verificare quale sarà la regola successiva a $(\phi, \text{MARK}(m))$ che stabilirà il destino del pacchetto e riscrivere la regola combinando i due target. Osserviamo che, dato che non vale $tag(p) = \bullet$, nessuna altra regola di R_{mark} sarà applicata al pacchetto. La regola che sarà successivamente applicata al pacchetto nella ruleset sarà necessariamente una della forma $(tag(p) = m, \text{NAT}(d_n, s_n))$ dalla ruleset R_{nat}. Quindi l'azione corrispondente, insieme alla MARK, può essere scritta come target della regola.

Algorithm 7

1: **function** RULESET_GENERATION(R_λ)
2: $R_{nat} = R_{dnat} = R_{fil} = R_{snat} = R_{mark} = \epsilon$
3: **for** r in R_λ **do**
4: **if** $r = (\phi, \texttt{ACCEPT})$ **then**
5: $R_{fil} \leftarrow R_{fil} \cdot r$
6: **else if** $r = (\phi, \texttt{NAT}(d_n, s_n))$ **then**
7: $m \leftarrow$ NEW_TAG( )
8: $R_{mark} \leftarrow R_{mark} \cdot (\phi, \texttt{MARK}(m))$
9: $R_{dnat} \leftarrow R_{dnat} \cdot (tag(p) = m, \texttt{NAT}(d_n, \star))$
10: $R_{snat} \leftarrow R_{snat} \cdot (tag(p) = m, \texttt{NAT}(\star, s_n))$
11: $R_{nat} \leftarrow R_{nat} \cdot (tag(p) = m, \texttt{NAT}(d_n, s_n))$
12: $R_{fil} \leftarrow R_{fil} \cdot R_{mark} \cdot R_{fil} \cdot (tag(p) \neq \bullet, \texttt{ACCEPT}) \cdot (true, \texttt{DROP})$
13: $R_{dnat} \leftarrow R_{dnat} \cdot R_{mark} \cdot R_{dnat}$
14: $R_{snat} \leftarrow R_{snat} \cdot R_{mark} \cdot R_{snat}$
15: $R_{nat} \leftarrow R_{nat} \cdot R_{mark} \cdot R_{nat}$
16: **return** $(R_{fil}, R_{snat}, R_{dnat}, R_{nat})$

iptables

In `iptables` i *tag* sono numeri interi e le condizioni sui *tag* possono comprendere la specifica di una maschera, in questo caso ogni numero che è identico a quello specificato dalla condizione, modulo la maschera, verifica la condizione. Possiamo quindi risolvere i problemi dei punti 1 e 2 decidendo di usare solo *tag* dispari e usando una maschera `/1`. Esprimiamo quindi $tag(p) = \bullet$ come `! --mark 1/1` e $tag(p) \neq \bullet$ come `--mark 1/1`. In `iptables` `MARK` è un taget a sé e una volta applicato la valutazione della ruleset prosegue dalla regola successiva, esattamente come in IFCL, quindi non abbiamo problemi per quanto riguarda il punto 4. Per quanto riguarda il punto 3 sembrano esserci indicazioni contrastanti su dove possono essere inserite regole con target MARK: in molte fonti si raccomanda di usarle solo nella tabella `mangle`, talvolta dicendo che altrimenti il *tag* non viene associato al pacchetto [18]; nel manuale di `iptables` invece non c'è traccia di vincoli simili [21]. Che si tratti di una questione di stile o di un requisito necessario per il corretto funzionamento della configurazione, una semplice fase di preprocessing è sufficiente a spostare il contenuto della ruleset R_{mark} nei nodi del diagramma di controllo relativi alla tabella `MANGLE` (i nodi relativi alle ruleset `PREROUTING` e `OUTOUT` della tabella `MANGLE` dovrebbero essere sufficienti).

pf

In `pf` i tag sono stringhe arbitrarie, confrontate unicamente per identità: non è supportata nessuna forma di controllo su insieme, lista o intervallo. Questo rende molto difficile realizzare il controllo $tag(p) = \bullet$ del punto 1. A complicare ulteriormente le cose concorre il fatto che le regole di trasformazione siano separate da quelle di filtro, pertanto la traduzione di R_{mark} per le ruleset R_{snat} e R_{dnat} deve produrre una lista di regole di trasformazione, che hanno una sintassi più limitata di quelle di filtro [10]. Un limite importante per quanto riguarda i *tag* nelle regole di traduzione, è che la condizione `tagged` non può essere negata usando `!` come succede invece nelle regole di filtro. Proponiamo un algoritmo alternativo per la generazione delle configurazioni, molto simile a quello originale: l'algoritmo 7.

La ruleset R_λ è tale che le condizioni delle regole sono tutte mutualmente esclusive. Il controllo $tag(p) = \bullet$ nell'algoritmo 6 quindi non serve a impedire che un *tag* sovrascriva un altro precedentemente assegnato dalla stessa ruleset, ma ad impedire che siano assegnati nuovi *tag* a pacchetti già trasformati da una ruleset precedente. Poiché per essere modificato un pacchetto deve prima essere taggato, il controllo $tag(p) = \bullet$ serve allo scopo. Possiamo quindi rimuovere il controllo $tag(p) = \bullet$ dalle regole se prima di controllare se il pacchetto verifica le condizioni ci sinceriamo che il pacchetto non sia già stato taggato. Aggiungiamo quindi una parte iniziale alle ruleset R_{snat} , R_{dnat} e R_{fil} che controlla se il pacchetto è già stato taggato e in quel caso gli associa il destino previsto dall'algoritmo; questo corrisponde esattamente a replicare le regole del tipo $(tag(p) = m, \texttt{NAT}(d_n, d_s))$ all'inizio della ruleset.

Come detto, il problema del punto 2 è facilmente risolvibile producendo una regola della forma $(tag(p) = m, \texttt{ACCEPT})$ per ogni tag m creato dal programma. Il punto 3 non rappresenta alcun problema: i tag possono essere scritti sia dalle regole di filtro che da quelle di traduzione. Anche in `pf`, come in `ipfw`, l'operazione di associare un tag a un pacchetto non è un'azione, ma un'opzione associata ad un'altra azione. Per il punto 4 occorre fare un distinguo: nelle regole di filtro l'azione a cui associamo l'opzione di tag può essere solo `block` o `pass`, quindi non è possibile tradurre direttamente il target MARK; per le regole di traduzione la questione non è altrettanto chiara. Dalla grammatica presente nel manuale notiamo che la parte delle istruzioni di trasformazione che specifica la traduzione da applicare al pacchetto (`["->" ( redirhost | "" redirhost-list "") [ portspec ] [ pooltype ] [ "static-port" ] ]`) è opzionale. Tuttavia non è espressamente definito il comportamento di una regola di traduzione che non preveda alcuna traduzione. Ad ogni modo sia per le regole di traduzione, sia per quelle di filtro è possibile applicare la stessa fase di postprocessing proposta per `ipfw`: combinare la regola da tradurre (con target MARK) con la successiva la cui condizione viene verificata dal pacchetto. Inoltre dal diagramma di controllo (o dal fatto che le regole di traduzione siano sempre considerate prima di quelle di filtro) risulta evidente che in realtà sia sufficiente associare i tag ai pacchetti in R_{snat} e R_{dnat}, tralasciando la cosa in R_{fil}.

Capitolo 8

Conclusioni

Abbiamo presentato una pipeline di transcompilazione fra linguaggi di configurazione di sistemi firewall differenti, che al momento supporta i sistemi `iptables`, `pf` e `ipfw`. Abbiamo dimostrato entro quali limiti sia garantita l'equivalenza fra il firewall di partenza e quello prodotto, e attraverso uno studio dell'espressività dei sistemi firewall abbiamo valutato il risultato ottenuto rispetto a quanto teoricamente possibile.

Il cuore del nostro approccio è il linguaggio IFCL, un linguaggio intermedio che permette di rappresentare firewall definiti con i vari linguaggi supportati e del quale è definita una semantica operazionale. IFCL astrae dai dettagli dei sistemi rappresentati, usando un linguaggio standard per la definizione delle regole di filtro e traduzione e modellando il procedimento di applicazione delle regole del sistema attraverso un diagramma di controllo. Il lavoro presentato estende quello esposto in [4], definendo una nuova semantica denotazionale per il linguaggio intermedio IFCL, che caratterizza il comportamento di un firewall rappresentandolo come una funzione dall'insieme dei pacchetti a quello delle trasformazioni possibili. In base a questa caratterizzazione funzionale vengono ridefiniti gli algoritmi che implementano il processo di astrazione del firewall source, e la rappresentazione sintetica della sua semantica; vengono studiati i limiti teorici della transcompilazione e viene presentato un algoritmo per la generazione della configurazione IFCL target.

Il primo stadio della pipeline corrisponde alla formalizzazione del firewall source, usando IFCL; questa fase è semplificata molto dal fatto che IFCL sia definito in modo tale che ogni azione dei linguaggi supportati abbia un corrispettivo diretto in IFCL.

Nello stadio due vogliamo calcolare una rappresentazione sintetica della semantica del firewall IFCL: per prima cosa rimuoviamo i target che modificano il flusso di controllo, come CALL e GOTO; poi astraiamo le ruleset calcolando la funzione da pacchetti a trasformazioni associata ad ogni nodo del diagramma di controllo, in forma sintetica; infine procediamo a comporre fra loro le funzioni per ottenere la semantica stessa del firewall (per semplificare il calcolo trasformiamo prima il diagramma di controllo in una versione equivalente aciclica). La correttezza della semantica denotazionale rispetto a quella operazionale e le condizioni poste sulla rappresentazione sintetica, basata su multicubi, garantiscono la conservazione della semantica del firewall nei vari passaggi. Prima di applicare la terza fase della pipeline, e derivare un firewall IFCL del tipo target, controlliamo che sia verificata la *fattibilità locale*, una condizione necessaria (ma non sufficiente) perché il sistema target sia in grado di replicare il comportamento del firewall di origine.

Se la condizione è verificata, nella fase tre scomponiamo la funzione sintetizzata, ottenuta dall'astrazione del firewall di origine, in una serie di funzioni, una per ogni nodo del diagramma di controllo del sistema target; successivamente traduciamo le funzioni in ruleset IFCL. La scomposizione può fallire qualora il firewall da implementare non sia esprimibile nel sistema target, in caso contrario il firewall

IFCL prodotto e quello di partenza hanno la stessa semantica, sia per quanto riguarda i pacchetti accettati, sia per quanto riguarda i pacchetti scartati.

L'algoritmo che abbiamo definito purtroppo non dà garanzie di trovare una scomposizione se nel diagramma di controllo del sistema target sono presenti più nodi, consecutivi in almeno un percorso, che siano capaci di effettuare le stesse trasformazioni su un campo del pacchetto (due nodi capaci di fare SNAT o due nodi capaci di fare DNAT). Per questi sistemi, come `ipfw`, rimane applicabile l'algoritmo basato sull'uso del campo *tag* dei pacchetti, definito inizialmente in [4] per `iptables` e `pf`, e riproposto qui in versione lievemente modificata per essere applicabile a `ipfw`. Inoltre, abbiamo studiato i problemi legati alla traduzione delle configurazioni IFCL generate da questo algoritmo verso i linguaggi dei sistemi target, mostrando in particolare come questi possano essere risolti almeno per `iptables` e `ipfw`.

Infine, nella fase quattro della pipeline, il firewall IFCL viene tradotto nel linguaggio di configurazione target; la traduzione è banale in quanto il firewall prodotto dalla fase precedente è normalizzato e quindi privo di azioni complesse come CALL e GOTO.

La necessità di trovare strumenti affidabili per la gestione dei firewall, definendo possibilmente soluzioni applicabili a diversi sistemi, è fuor di dubbio. I nostri risultati sono a nostro avviso molto promettenti, in quanto basati su un approccio formale e generale, sebbene risultino incapaci di trattare adeguatamente alcuni casi importanti, come la generazione di firewall qualora il sistema target consenta di applicare lo stesso tipo di trasformazione ad un pacchetto in più momenti diversi nel corso della valutazione.

Presentiamo qualche accenno riguardo all'implementazione effettiva della pipeline, dando qualche dettaglio in più dal punto di vista degli algoritmi e delle strutture dati, con l'obiettivo principale di mostrare che quanto esposto è realizzabile in modo efficiente. Presentiamo infine una serie di lavori futuri che prevedono di estendere la teoria modellando ulteriori aspetti del comportamento di un firewall e rilassando alcune delle assunzioni fatte.

8.1 Implementazione

Per la produzione di uno strumento software, che concretizzi quanto esposto in questa tesi, occorre valutare alcuni aspetti implementativi e algoritmici tralasciati fino a qui. Nel capitolo 5 abbiamo parlato della rappresentazione sintetica delle funzioni $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, definite come insiemi di coppie $(P, t) \in \tilde{\lambda}$ con $P \in \mathcal{M}(\mathbb{P})$ e $t \in \mathcal{T}(\mathbb{P}) \cup \{\bot\}$, quindi abbiamo già un modello di dato abbastanza preciso per le coppie; manca da discutere il modello di dati per l'insieme stesso.

Fra le operazioni che la nostra rappresentazione delle funzioni sintetizzate deve supportare, quelle potenzialmente problematiche di cui ci occuperemo sono:

- la funzione FILTER($\tilde{\lambda}$, ψ) che restituisce una nuova funzione sintetizzata in cui la parte sinistra delle coppie contiene solo elementi che verificano ψ:

$$\{\ (P', t) \mid (P, t) \in \tilde{\lambda}\ \wedge\ P' = t^{-1}(\psi(t(P)))\ \wedge\ P' \neq \emptyset\ \}$$

- la funzione CONCAT($\tilde{\lambda_1}$, $\tilde{\lambda_2}$) che restituisce la funzione sintetizzata corrispondente alla concatenazione delle due funzioni sintetizzate:

$$\{\ (P', t_2 \ltimes t_1) \mid (P_1, t_1) \in \tilde{\lambda}_1 \wedge (P_2, t_2) \in \tilde{\lambda}_2 \wedge P' = t_1^{-1}(P_2 \cap t_1(P_1)) \wedge P' \neq \emptyset\ \}$$

- l'unione di due funzioni sintetizzate $\tilde{\lambda_1} \cup \tilde{\lambda_2}$[1]

[1] Della funzione SPLIT(P, t, $\emptyset$) abbiamo descritto già brevemente nel capitolo 5.

La soluzione banale consiste nel rappresentare $\tilde{\lambda}$ come una lista di coppie (P, t). È possibile comunque usare anche strutture dati differenti, fra le varie possibili mostriamo una proposta che sfrutta i *segment tree* [7].

8.1.1 Implementazione banale

Supponiamo di rappresentare le funzioni sintetizzate $\tilde{\lambda}$ come liste di coppie $(P, t) \in \mathcal{M}(\mathbb{P}) \times \mathcal{T}(\mathbb{P}) \cup \{\bot\}$. L'implementazione della funzione FILTER($\tilde{\lambda}$, ψ) prevede allora di calcolare sequenzialmente, per ogni (P, t) di $\tilde{\lambda}$:

1. il multicubo $t(P)$, che viene calcolato come

   $$(t.sIP(P.sIP) : t.sPort(P.sPort),\ t.dIP(P.dIP) : t.dPort(P.dPort), t.tag(P.tag))$$

   dove $t.x(A)$ è $\{a\}$ se $t.x = cost(a)$, altrimenti, se $t.x = id$, è A.

2. il multicubo $\psi(t(P)) = \{p \in t(P) \mid \psi(p)\}$ dove abbiamo assunto che la funzione ψ possa essere scomposta secondo

   $$\psi(p) = \psi_{sIP}(p.sIP) \wedge \psi_{sPort}(p.sPort) \wedge \psi_{dIP}(p.dIP) \wedge \psi_{dPort}(p.dPort) \wedge \psi_{tag}(p.tag)$$

   quindi possiamo calcolare $\psi(t(P))$ semplicemente come

   $$\psi_{sIP}(t(P).sIP) \times \psi_{sPort}(t(P).sPort) \times \psi_{dIP}(t(P).dIP) \times \psi_{dPort}(t(P).dPort) \times \psi_{tag}(t(P).tag)$$

   dove scriviamo $\psi(P)$ per $\{p \in P \mid \psi(p)\}$, possiamo verificare a questo punto se $\psi(t(P)) = \emptyset$, nel qual caso scartiamo la coppia (P, t) e passiamo alla successiva.

3. il multicubo $t^{-1}(\psi(t(P)))$ viene ottenuto prendendo il risultato dalla fase precedente $\psi(t(P))$ che chiamiamo P', e invertendo la trasformazione t, lavorando su ogni campo separatamente:

   $$(t.sIP^{-1}(P'.sIP) : t.sPort^{-1}(P'.sPort),\ t.dIP^{-1}(P'.dIP) : t.dPort^{-1}(P'.dPort), t.tag^{-1}(P'.tag))$$

   dove $t.x^{-1}(P'.x)$ è uguale a $P'.x$ se $t.x = id$, altrimenti è uguale a $P.x$ se $t.x = cost(a)$ per un qualche a.

In modo simile, la funzione CONCAT($\tilde{\lambda_1}$, $\tilde{\lambda_2}$) prevede, per ogni coppia (P_1, t_1) di $\tilde{\lambda_1}$, per ogni coppia (P_2, t_2) di $\tilde{\lambda_2}$ (quindi in totale $O(n^2)$ volte, se n è il numero di coppie in una funzione sintetizzata), di calcolare in ordine:

1. il multicubo $t_1(P_1)$, seguendo lo stesso procedimento del primo passaggio della funzione FILTER.

2. il multicubo $P_2 \cap t_1(P_1)$ che viene calcolato campo per campo, essendo entrambi gli operandi dei multicubi possiamo infatti calcolare per ogni x:

   $$(P_2 \cap t_1(P_1)).x = P_2.x \cap t_1(P_1).x$$

   anche in questo caso possiamo verificare a questo punto se il risultato è $\emptyset$ oppure no (e nel caso scartare la coppia e passare alla successiva).

3. il multicubo $t_1^{-1}(P_2 \cap t_1(P_1))$ viene calcolato come nell'ultimo passo della funzione FILTER, dove P' è però $P_2 \cap t_1(P_1)$ e $t_1.x^{-1}(P'.x)$ è uguale a $P'.x$ se $t_1.x = id$, altrimenti è uguale a $P_1.x$ se $t_1.x = cost(a)$ per un qualche a.

Infine, l'unione di due funzioni sintetizzate può essere realizzata semplicemente concatenando le due liste.

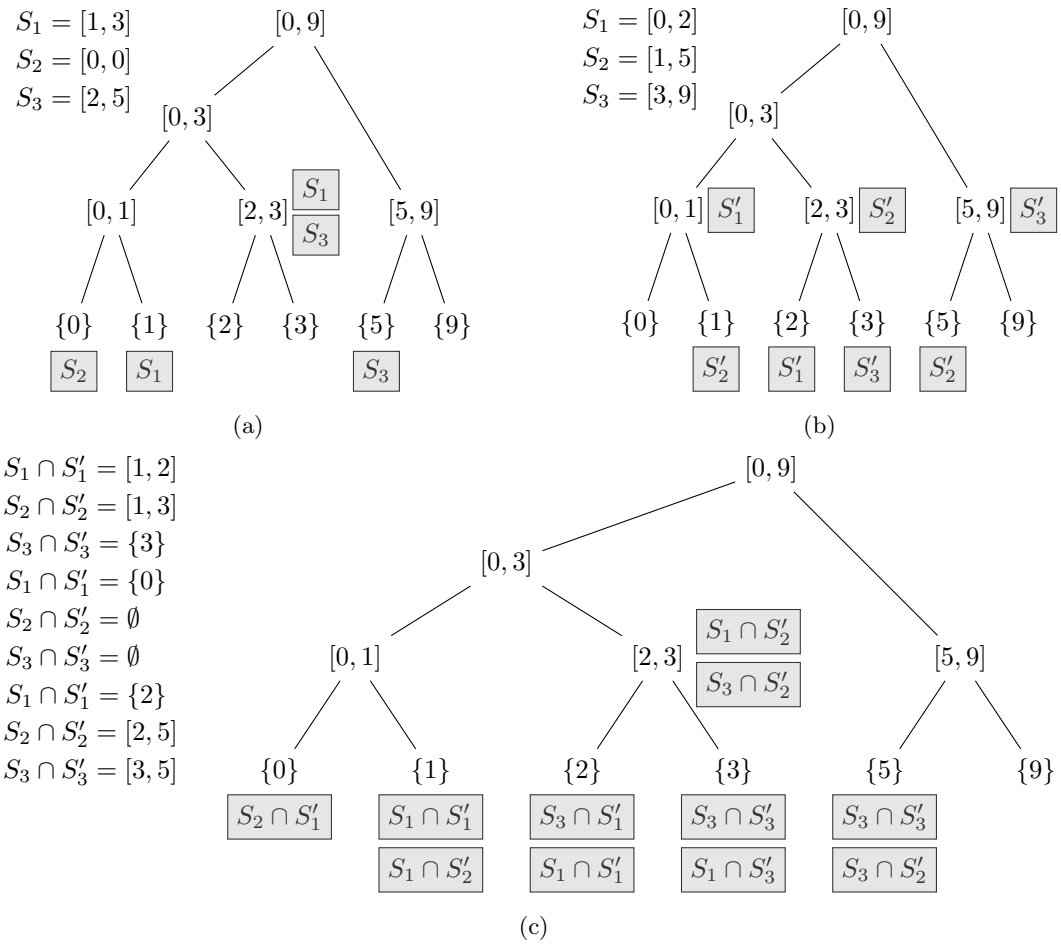


Figura 8.1: due esempi di segment tree 8.1a, 8.1b e i relativi insiemi di segmenti rappresentati, e segment tree relativo all'insieme delle possibili intersezioni fra elementi del primo insieme e del secondo 8.1c.

8.1.2 Implementazione con segment tree

Come abbiamo visto nella sezione precedente, la parte algoritmicamente più complessa è sicuramente l'applicazione della funzione CONCAT, in quanto l'operazione descritta dai tre passi deve essere ripetuta mettendo in relazione ogni coppia del primo insieme con ogni coppia del secondo insieme. Questo è uno spreco in molti casi: data una coppia $(P_1, t_1) \in \tilde{\lambda}_1$, non vorremmo considerare la sua concatenazione con tutte le coppie $(P_2, t_2) \in \tilde{\lambda}_2$, ma solo con quelle per cui $t_1(P_1) \cap P_2 \neq \emptyset$. In altre parole, dato $t_1(P_1) = P'$ vorremmo poter cercare in $\tilde{\lambda}_2$ tutte e solo le coppie (P_2, t_2) in cui P_2 interseca P'. Se rappresentiamo le funzioni sintetizzate $\tilde{\lambda}$ come liste di coppie l'unico modo per determinare questo insieme è quello di valutare uno ad una tutte le coppie, come in effetti abbiamo fatto quando abbiamo discusso l'implementazione banale.

Possiamo tuttavia considerare una rappresentazione di $\tilde{\lambda}$ più adeguata a questo tipo di ricerca. Perché sia vero che $t_1(P_1) \cap P_2 \neq \emptyset$, è necessario che, per ogni campo $x \in \{sIP, sPort, dIP, dPort, tag\}$, $t_1.x(P_1.x) \cap P_2.x \neq \emptyset$. L'idea è quella di usare, per ogni campo dei pacchetti, una struttura dati che permetta una ricerca per intervalli in maniera efficiente, come un albero binario di ricerca, in modo da poter trovare in tempo logaritmico il sottoinsieme di valori per i quali l'intersezione non è vuota. In realtà le cose sono lievemente più complicate in quanto: (i) non abbiamo a che fare con valori unici ma con insiemi arbitrari rappresentati come unioni di segmenti; (ii) non abbiamo in $\mathbb{P}$ un ordinamento totale, in particolare non ci interessa un ordinamento lessicografico in cui l'ordine dipende dal primo

campo e in caso di equivalenza dipende dal secondo e così via: vogliamo un ordinamento separato per ogni campo del pacchetto.

Per quanto riguarda il punto (i), una soluzione possibile, che approfondiamo qui, consiste nell'applicazione di *segment tree*, alberi specializzati per la rappresentazione di intervalli chiusi (segmenti) [7]. Per il punto (ii) invece proponiamo di considerare i campi in cui i pacchetti sono divisi, in modo indipendente, usando un albero per ogni campo.

Dato un insieme di segmenti, un segment tree può essere costruito prendendo la lista ordinata degli estremi dei segmenti e costruendo un albero binario accoppiando i valori a due a due e costruendo all'insù finché possibile, in modo tale che la lista degli estremi da cui si è partiti siano le foglie dell'albero binario. L'altezza dell'albero è $O(\log(n))$ con n numero degli intervalli di partenza, il numero di nodi $O(n \log(n))$.

Ad ogni nodo viene assegnato il segmento che sottende, dove alle foglie è assegnato il singoletto contenente il valore dal quale sono state create e ad ogni nodo interno viene assegnato il segmento $[m, M]$ dove m è l'estremo sinistro del segmento assegnato al figlio sinistro del nodo e M è l'estremo destro del segmento assegnato al figlio destro. In questo modo ad ogni nodo è associato un segmento che contiene i segmenti dei nodi figli. Chiamiamo $Seg(q) = S$ il segmento assegnato al nodo q. L'idea è che un segmento $S = [m, M]$, all'interno del segment tree, è rappresentato da un insieme di nodi Q tali che $m = \min \{m' \mid q \in Q \wedge Seg(q) = [m', M']\}$, $M = \max \{M' \mid q \in Q \wedge Seg(q) = [m', M']\}$ e che per ogni valore $a \in [m, M]$, se a è il valore di una foglia, allora esiste un $q \in Q$ tale che $a \in Seg(q)$. L'insieme Q definito non è unico, fra quelli possibili prendiamo sempre il minore; è possibile dimostrare che per ogni livello dell'albero servono al massimo due nodi.

Più in dettaglio, la procedura per etichettare i nodi del segment tree, per rappresentare il segmento $S = [m, M]$ è la seguente. A partire dalla radice, dato il nodo q:

1. se $Seg(q) \subseteq S$ allora etichettiamo il nodo con S;
2. altrimenti:
   - se $Seg(q') \cap S \neq \emptyset$, dove q' è il figlio sinistro di q, allora proseguiamo l'etichettatura ricorsivamente dal nodo q';
   - se $Seg(q'') \cap S \neq \emptyset$, dove q'' è il figlio destro di q, allora proseguiamo l'etichettatura ricorsivamente dal nodo q''.

Le figure 8.1a e 8.1b mostrano due esempi di segment tree, creati a partire da insiemi di segmenti diversi, rappresentati a loro volta come etichette sugli alberi.

Per il seguito ci sarà utile definire, dati due insiemi di intervalli $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ e $\mathbf{S'} = \{S'_1, S'_2, \ldots, S'_m\}$, come calcolare l'insieme delle possibili intersezioni fra un intervallo del primo insieme e uno del secondo: $\{S_i \cap S'_j \mid S_i \in \mathbf{S} \wedge S'_j \in \mathbf{S'}\}$. Come prima cosa si noti che per ogni possibile scelta di i e j, l'intersezione $S_i \cap S'_j$ restituisce un segmento oppure l'insieme vuoto. Osserviamo anche che gli estremi dei segmenti risultanti possono essere solo fra quelli dei segmenti di $\mathbf{S}$ o $\mathbf{S'}$. La procedura per calcolare tutte le possibili intersezioni prevede di ispezionare i nodi del segment tree uno ad uno, dalla radice alle foglie, seguendo una visita in profondità per mezzo di una procedura ricorsiva. Nella visita ci ricorderemo all'interno di quali segmenti ci troviamo (cioè quali etichette abbiamo incontrato nella discesa). Più precisamente, a partire dalla radice e da due insiemi vuoti di etichette, uno per $\mathbf{S}$, l'altro per $\mathbf{S'}$, applichiamo il seguente algoritmo, dove q è il nodo nel quale ci troviamo, e gli insiemi $\mathbf{I}$ e $\mathbf{I'}$ sono le etichette incontrate:

1. per ogni etichetta S_i di $\mathbf{S}$ assegnata al nodo q:

- aggiungiamo S_i a $\mathbf{I}$ per ricordarci che da questo nodo in poi siamo dentro S_i;
- per ogni S'_j in $\mathbf{I}'$, scriviamo l'etichetta $S_i \cap S'_j$ sul nodo q.

2. per ogni etichetta S'_j di $\mathbf{S}'$ assegnata al nodo q:
   - aggiungiamo S'_j a $\mathbf{I}'$ per ricordarci che da questo nodo in poi siamo dentro S'_j;
   - per ogni S_i in $\mathbf{I}$, scriviamo l'etichetta $S_i \cap S'_j$ sul nodo q.
3. proseguiamo ricorsivamente su entrambi i nodi figli di q con gli insiemi aggiornati.

Il risultato dell'intersezione degli intervalli del segment tree in figura 8.1a con quelli del segment tree in figura 8.1b è mostrato in figura 8.1c.

L'idea dunque è quella di usare una serie di segment tree per ognuna delle funzioni sintetizzate $\tilde{\lambda}$, in particolare uno per ogni dimensione di $\mathbb{P}$. Consideriamo di avere un nome per ogni multicubo P tale che $(P,t) \in \tilde{\lambda}$, la funzione sintetizzata $\tilde{\lambda}$ viene rappresentata come una coppia (D,T) dove D è una rappresentazione efficiente della funzione che associa al nome di P la trasformazione t per ogni coppia $(P,t) \in \tilde{\lambda}$ (ad esempio una tabella hash), e $T = T_{sIP} \times T_{sPort} \times T_{dIP} \times T_{dPort} \times T_{tag}$ è la lista di alberi, uno per ogni campo dei pacchetti, all'interno dei quali sono inserite le etichette che rappresentano i segmenti dei campi nei multicubi. Come nome per il multicubo P useremo $\breve{P}$. Consideriamo anche, per ogni campo x, di ogni multicubo P, di avere un nome (una numerazione) per ogni segmento di cui $P.x$ è composto. Scriveremo $\breve{P}.x.1$, $\breve{P}.x.2$ e così via, assumendo quindi che dal nome del segmento sia sempre immediatamente derivabile il nome del multicubo. Per indicare il segmento in sé scriveremo invece $P.x.1$, $P.x.2$ e così via. Per ogni multicubo P, per ogni campo x, rappresentiamo dunque ogni segmento $P.x.i$ nel segment tree T_x etichettandolo con il nome $\breve{P}.x.i$.

Supponiamo quindi di voler calcolare $\text{CONCAT}(\tilde{\lambda_1}, \tilde{\lambda_2})$, supponiamo inoltre che le funzioni $\tilde{\lambda_1}$ e $\tilde{\lambda_2}$ siano rappresentate come (D_1,T_1) e (D_2,T_2), dove gli alberi che compongono T_1 e quelli che compongono T_2 sono uguali tranne che per i nomi di segmenti associati ai nodi. Il risultato della funzione sarà $\tilde{\lambda_3}$, rappresentata come (D_3,T_3), sempre con T_3 avente la stessa forma degli altri. Per motivi che saranno chiari dopo, supponiamo di avere una funzione $\oplus$ che dato il nome di due multicubi restituisce un nuovo nome per la loro intersezione, assumendo sempre che data l'intersezione posso ottenere il nome dei due multicubi. Ricordiamo che vogliamo calcolare $t_1^{-1}(P_2 \cap t_1(P_1))$, o meglio, per ogni campo x vogliamo $t_1.x^{-1}(P_2.x \cap t_1.x(P_1.x))$. Esprimendo gli insiemi $P_1.x$ e $P_2.x$ come unioni di segmenti, abbiamo

$$t_1.x^{-1}(\bigcup_i (P_2.x.i) \cap t_1.x(\bigcup_j (P_1.x.j))) \;=\; t_1.x^{-1}(\bigcup_{i,j} (P_2.x.i \cap t_1.x(P_1.x.j)))$$

Consideriamo due casi distinti:

- se $t_1 = id$ allora la formula precedente diventa

  $$\bigcup_{i,j} (P_2.x.i \cap P_1.x.j)$$

  quindi per ogni coppia di segmenti, uno relativo a P_1 nell'albero $T_1.x$ e l'altro relativo a P_2 nell'albero $T_2.x$, calcoliamo l'intersezione, che è un segmento che chiamiamo $(\breve{P}_2 \oplus \breve{P}_1).x.(i,j)$, e la inseriamo nell'albero $T_3.x$;

- se $t_1 = cost(a)$ allora dobbiamo verificare se $a \in \bigcup_i (P_2.x.i)$ (possiamo farlo con una visita sull'albero in $O(\log(n))$), in questo caso allora la formula restituisce semplicemente $\bigcup_j (P_1.x.j)$, cioè tutti i segmenti che compongono $P_1.x$ sono da ricopiare in $T_3.x$; altrimenti nessun segmento è da aggiungere a $T_3.x$.

Il grande vantaggio di questa rappresentazione è che ora possiamo calcolare il risultato di CONCAT($\tilde{\lambda_1}$, $\tilde{\lambda_2}$) visitando in parallelo, per ogni dimensione x, i due alberi $T_1.x$ e $T_2.x$ e collezionando il risultato in $T_3.x$ una volta sola, e non una volta per ogni coppia di multicubi (P_1, P_2). Il procedimento è molto simile a quello per il calcolo dell'intersezione fra due insiemi di intervalli presentato precedentemente e mostrato in figura 8.1; qui però è necessaria una trattazione delle etichette un po' particolare in quanto lavoriamo con insiemi scomposti in segmenti anziché direttamente con segmenti, inoltre occorre tenere conto anche della trasformazione associata all'intervallo (se non è id).

L'algoritmo per il calcolo di CONCAT($\tilde{\lambda_1}$, $\tilde{\lambda_2}$) prevede, per ogni dimensione x:

1. per ogni elemento (P_1, t_1) di $\tilde{\lambda_1}$ per cui $t_1.x = cost(a)$, per qualche a, verificare se a appartiene a qualche $P_2.x.j$ in $T_2.x$, se questo è vero allora si inserisce $\breve{P_1}$ nell'insieme, inizialmente vuoto, C_x contenente i multicubi di cui il campo x va copiato invariato in $T_3.x$, e per ogni $P_2.x$ si aggiorna D_3 in modo tale che $D_3(\breve{P_1}).x = D_2(\breve{P_2}).x \ltimes cost(a)$ (questa operazione complessivamente costa $O(\log(n))$ per ogni elemento di $\tilde{\lambda_1}$, quindi un totale di $O(n \log(n))$);

2. scorrere parallelamente gli alberi $T_1.x$ e $T_3.x$, nodo per nodo, copiando in $T_3.x$ le etichette $\breve{P_1}.x.i$ se e solo se $\breve{P_1}$ appartiene a C_x (questa operazione prevede di scorrere l'albero $T_1.x$ una ed una sola volta, quindi costa $O(n \log(n))$);

3. scorre parallelamente gli alberi $T_1.x$, $T_2.x$ e $T_3.x$, leggendo dai primi due e scrivendo nel terzo, attraverso una procedura ricorsiva che implementa una visita in profondità degli alberi; per ogni nodo visitato teniamo conto dei segmenti all'interno dei quali ci troviamo. Più precisamente assumiamo di essere al nodo q degli alberi, di sapere di essere dentro i segmenti I_1 per quanto riguarda $\tilde{\lambda_1}$ e dentro i segmenti I_2 per quanto riguarda $\tilde{\lambda_2}$, allora quello che facciamo è:

   - per ogni $P_1.x.i$ assegnato al nodo q in $T_1.x$:
     - segniamo di essere all'interno del segmento $P_1.x.i$ d'ora in poi, per il passo successivo e le chiamate ricorsive (cioè aggiungiamo $P_1.x.i$ a I_1)
     - per ogni $P_2.x.j$ all'interno del quale ci troviamo in $T_2.x$ (cioè in I_2), scriviamo $(\breve{P_1} \oplus \breve{P_2}).x.(i, j)$ nel nodo q di $T_3.x$, e aggiorniamo D_3 in modo tale che $D_3(\breve{P_1} \oplus \breve{P_2}).x = D_2(\breve{P_2}).x \ltimes id$
   - per ogni $P_2.x.i$ assegnato al nodo q in $T_2.x$:
     - segniamo di essere all'interno del segmento $P_2.x.i$ d'ora in poi, nelle chiamate ricorsive (cioè aggiungiamo $P_2.x.i$ a I_2)
     - per ogni $P_1.x.j$ all'interno del quale ci troviamo in $T_1.x$ (cioè in I_1), scriviamo $(\breve{P_1} \oplus \breve{P_2}).x.(i, j)$ nel nodo q di $T_3.x$, e aggiorniamo D_3 in modo tale che $D_3(\breve{P_1} \oplus \breve{P_2}).x = D_2(\breve{P_2}).x \ltimes id$
   - facciamo una chiamata ricorsiva per ogni nodo figlio di q rispettivamente nei tre alberi, con I_1 e I_2 aggiornati.

   (il calcolo prevede operazioni dal costo costante applicate su ogni tripletta di nodi corrispondenti nei tre alberi, ogni nodo degli alberi è valutato una ed una sola volta, quindi il costo totale è $O(n \log(n))$).

Il risultato è quindi calcolato in un tempo totale $O(n \log(n))$.

L'assunzione secondo cui tutti gli alberi hanno la stessa forma può essere verificata tenendo conto di tutte le funzioni sintetizzate con cui dobbiamo lavorare durante la fase di creazione degli stessi, ad esempio attraverso una fase di preprocessing. Questi ed altri dettagli, relativi all'eventualità in cui il componente di un multicubo risulti essere vuoto, sono rimandati a future trattazioni; l'intento di questa

digressione era infatti unicamente quello di mostrare che strutture dati adeguate possono essere impiegate per eliminare il costo quadratico della funzione CONCAT, che si presenta nell'implementazione banale con liste.

8.2 Sviluppi futuri

La pipeline presentata qui può essere estesa in molti modi: è possibile aumentare il numero di sistemi supportati attraverso compilatori dedicati per e da IFCL; possiamo inoltre approfondire maggiormente la formalizzazione dello stato interno del firewall modellando fedelmente le feature più avanzate disponibili nei firewall, come il bilanciamento del carico e i rate limit. Sebbene infatti, come abbiamo detto, non sia sempre desiderabile per il porting forzare lo stesso comportamento rispetto allo stato, possiamo comunque studiare il comportamento del firewall sulla base dello stato al fine di analizzare le differenze fra la configurazione originale e quella generata.

La versione precedente di IFCL permetteva la modellazione di sistemi non deterministici, in cui ad un dato pacchetto in input corrispondevano diversi esiti possibili. Anche lo stato interno veniva modellato in un modo simile.

Abbiamo inoltre definito senza sfruttarla, la fase di *refactoring* della pipeline, atta a rimodellare la configurazione IFCL dal punto di vista dello stile, mantenendo inalterata la semantica. Questa fase in particolare offre numerose possibilità dal punto di vista della qualità del codice prodotto, ma complica il lavoro dal punto di vista della traduzione delle configurazioni IFCL nel linguaggio target, pertanto meriterebbe un'indagine a parte.

8.2.1 NAT non deterministico

Nelle versioni precedenti di IFCL, il target NAT aveva come argomenti insiemi arbitrari di indirizzi anziché singoli valori. La semantica attesa era quella di una trasformazione non deterministica secondo la quale la riscrittura del pacchetto p potesse avvenire secondo uno qualunque degli indirizzi specificati.

Questo genere di trasformazioni può essere specificata in `iptables`, dove però la semantica prevede che la trasformazione venga selezionata fra quelle possibili attraverso una politica round robin, quindi non proprio in modo non deterministico, ma in funzione dei pacchetti precedentemente trattati. Opzioni avanzate con effetti simili sono disponibili in molti linguaggi, di solito la trasformazione può essere selezionata attraverso una politica round robin o secondo il bilanciamento del carico.

In effetti quindi, secondo questi esempi un modello non deterministico non è necessario in quanto la trasformazione scelta è funzione dei pacchetti precedentemente osservati e quindi lo stato interno può essere applicato per decidere deterministicamente il trattamento da riservare al prossimo pacchetto.

Inoltre consentire di definire NAT non deterministici complicherebbe molto l'analisi delle configurazioni. Si assuma ad esempio di avere in un percorso due nodi successivi: uno nel quale possiamo fare NAT e l'altro nel quale possiamo scartare alcuni pacchetti; se nel primo nodo modifico l'indirizzo di origine di un pacchetto p scrivendo non deterministicamente 0.0.0.0 oppure 1.1.1.1, e nel secondo nodo scarto ogni pacchetto in cui l'IP di origine è 0.0.0.0 e accetto ogni pacchetto in cui l'IP di origine è 1.1.1.1, allora complessivamente il firewall si comporterà rispetto a p scartandolo o accettandolo non deterministicamente. Questo genere di comportamenti sarebbe sicuramente presente solo in configurazioni "sbagliate", in loro presenza sarebbe perfettamente accettabile rifiutare la compilazione; tuttavia altre difficoltà di modellazione sono più difficili da risolvere.

Ad esempio non sempre è facile tenere traccia dei vincoli sugli insiemi di indirizzi. In `iptables` è possibile specificare traduzioni non deterministiche solo verso insiemi di pacchetti che siano equivalenti

a prodotti cartesiani di intervalli di valori, cioè cubi della forma $([sIP, sIP'] : [sPort, sPort'],\ [dIP, dIP'] : [dPort, dPort'], [tag, tag'])$; questo è dovuto alla sintassi attraverso la quale si definiscono le regole. Tuttavia, avendo due nodi consecutivi in un percorso all'interno del diagramma di controllo, il primo capace di fare SNAT e il secondo DNAT, è possibile che la concatenazione delle trasformazioni produca un insieme di alternative non esprimibile come cubo. Si supponga ad esempio che il primo dei nodi, quello capace di fare SNAT, trasformi un pacchetto p non deterministicamente in uno fra un insieme di pacchetti uguali a p tranne per sIP, preso non deterministicamente all'interno di $[192.168.0.0, 192.168.0.10]$. Supponiamo anche che il secondo nodo applichi a tutti i pacchetti $p[sIP \mapsto 192.168.0.0] \ldots p[sIP \mapsto 192.168.0.9]$ la trasformazione $(id : id,\ cost(1.1.1.1) : id,\ id)$, e a $p[192.168.0.10]$ la trasformazione $(id : id,\ cost(2.2.2.2) : id,\ id)$. Allora la composizione delle due funzioni associa non deterministicamente al pacchetto p, una fra le trasformazioni

$$
\begin{aligned}
&([cost(192.168.0.0), cost(192.168.0.9)] \times \{id\} \times \{cost(1.1.1.1)\} \times \{id\} \times \{id\}) \\
&\cup \ (\{cost(192.168.0.10) : id,\ cost(1.1.1.1) : id,\ id\})
\end{aligned}
$$

che, evidentemente, non può essere espresso come cubo (né come multicubo).

8.2.2 Stato interno

Lo stato interno è stato modellato in IFCL in un modo molto generale, tralasciando per il momento ogni dettaglio riguardo alle informazioni effettivamente memorizzate riguardo il traffico osservato e al loro uso per determinare il destino dei pacchetti in arrivo. Si è semplicemente assunto di avere una funzione $p \vdash_s \alpha$ che associ ad ogni pacchetto p e stato s l'azione prescritta α, ed una funzione $s \uplus (p, p')$ che aggiorna lo stato interno s con le informazioni rilevanti riguardo ad un nuovo pacchetto p accettato come p'. Gli sviluppi futuri rispetto allo stato possono andare nella direzione di modellare uno ad uno i diversi strumenti che fanno uso dello stato interno, come il NAT dinamico e i limit rate; oppure possono andare nella direzione di migliorare la rappresentazione sintetica della semantica di un firewall per includere informazioni sul comportamento in funzione dello stato.

Nella caratterizzazione logica della semantica, formulata originariamente in [4] e presentata nella sezione 2.5, il funzionamento dello stato è approssimato assumendo che un pacchetto appartenente ad una qualunque connessione stabilita possa essere trasformato non deterministicamente in ogni pacchetto possibile. Si noti che la caratterizzazione funzionale della semantica non comprende invece un'approssimazione sul comportamento dello stato del firewall. Di fatto è possibile modificare la semantica per gestire il target CHECK-STATE(X) con la stessa approssimazione della caratterizzazione dichiarativa. La versione approssimata della semantica denotazionale, dato uno stato, associa ogni pacchetto ad un insieme di trasformazioni possibili. A causa del non determinismo dato dall'approssimazione che abbiamo fatto sulla semantica dell'operazione CHECK-STATE(_) infatti non possiamo più associare un pacchetto ad un destino deterministicamente. Come già detto riguardo al NAT, dover gestire il non determinismo complica molto la progettazione di algoritmi che implementino la pipeline di compilazione; per questo abbiamo deciso di basarci sulla versione esatta e deterministica della semantica in questa tesi.

In generale scoraggiamo dunque l'uso di approssimazioni come quella della caratterizzazione logica in quanto eccessivamente grossolane e complicate da gestire. Presentiamo comunque per completezza la versione approssimata non deterministica della semantica denotazionale di un firewall. Come per la caratterizzazione logica, ci baseremo su stati approssimati `S`, che assegnano ad ogni pacchetto un'etichetta di stato $\mathtt{s} \in \{\mathtt{NEW}, \mathtt{ENSTABLISHED}\}$. Come al solito definiamo inizialmente la semantica di

una ruleset e successivamente quella di un firewall.

$$(\!|R|\!) : \mathtt{S} \to \mathbb{P} \to 2^{\mathcal{T}(\mathbb{P}) \cup \{\bot\}}$$
$$(\!|R|\!)(\mathtt{s}) = (\!|R|\!)_{\mathtt{s}} : \mathbb{P} \to 2^{\mathcal{T}(\mathbb{P}) \cup \{\bot\}}$$

$$(\!|R|\!)_{\mathtt{s}} = (\!|R|\!)^{id}_{\mathtt{s}}$$

Dove la funzione $(\!|R|\!)^t_s : \mathbb{P} \to 2^{\mathcal{T}(\mathbb{P}) \cup \{\bot\}}$, per un firewall $\mathcal{F}$, uno stato s e una trasformazione su pacchetti $t \in \mathcal{T}(\mathbb{P})$ è definita come:

$$(\!|\epsilon|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} \{t\} & \text{se } dp = \texttt{ACCEPT} \\ \{\bot\} & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{ACCEPT}); R|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} \{t\} & \text{se } \phi(p, s) \\ (\!|R|\!)^t_{\mathtt{s}}\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{DROP}); R|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} \{\bot\} & \text{se } \phi(p, s) \\ (\!|R|\!)^t_{\mathtt{s}}\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{NAT}(d_n, s_n)); R|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} \{tr_{nat}(d_n, s_n) \ltimes t\} & \text{se } \phi(p, s) \\ (\!|R|\!)^t_{\mathtt{s}}\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{CHECK-STATE}(X)); R|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} tr_{nondet}(X) \ltimes t & \text{se } \phi(p, s) \\ (\!|R|\!)^t_{\mathtt{s}}\ (p) & \text{altrimenti} \end{cases}$$

$$(\!|(\phi, \texttt{MARK}(m)); R|\!)^t_{\mathtt{s}}\ (p) = \begin{cases} (\!|R|\!)^{(id,id,id,id,cost(m)) \ltimes t}_{\mathtt{s}}\ (p[tag \mapsto m]) & \text{se } \phi(p, s) \\ (\!|R|\!)^t_{\mathtt{s}}\ (p) & \text{altrimenti} \end{cases}$$

La funzione $tr_{nondet}(X)$ per $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ restituisce tutte le possibili trasformazioni di pacchetti che agiscono sui campi specificati da X. Formalmente:

$$tr_{nondet}(X) = \begin{cases} any(\mathbf{IP}) \times any(\mathbf{Port}) \times \{id\} \times \{id\} \times \{id\} & \text{se } X = \leftarrow \\ \{id\} \times \{id\} \times any(\mathbf{IP}) \times any(\mathbf{Port}) \times \{id\} & \text{se } X = \rightarrow \\ any(\mathbf{IP}) \times any(\mathbf{Port}) \times any(\mathbf{IP}) \times any(\mathbf{Port}) \times \{id\} & \text{se } X = \leftrightarrow \end{cases}$$

Dove $any(A) = \{id\} \cup \{cost(a) \mid a \in A\}$. Abbiamo abusato della notazione per quanto riguarda $\ltimes$, intendendo con $tr_{nondet}(X) \ltimes t$ l'insieme $\{t' \ltimes t \mid t' \in tr_{nondet}(X)\}$. Per il resto non c'è niente di particolare da notare, la semantica è molto simile alla versione deterministica in cui i valori restituiti dalla funzione sono però inseriti in dei singoletti. L'unica eccezione è la sola operazione non deterministica (o meglio, approssimata in modo non deterministico), cioè $\texttt{CHECK-STATE}(X)$.

Definiamo la semantica di un firewall $\mathcal{F}$ come:

$$(\!|\mathcal{F}|\!) : S \to \mathbb{P} \to 2^{\mathcal{T}(\mathbb{P}) \cup \{\bot\}}$$
$$(\!|\mathcal{F}|\!)(s) = (\!|\mathcal{F}|\!)_s : \mathbb{P} \to 2^{\mathcal{T}(\mathbb{P}) \cup \{\bot\}}$$

$$(\!|\mathcal{F}|\!)_s = (\!|q_i|\!)^{\mathcal{F}, \emptyset}_s$$

Dove per ogni $q \in Q$, $q \neq q_f$

$$(\!|q|\!)^{\mathcal{F},I}_{\mathtt{s}}\ (p) = \begin{cases} T \cup \{\bot\} & \text{se } \exists t \in (\!|c(q)|\!)_{\mathtt{s}}(p)\ .\ t = \bot \vee \delta(q, t(p)) \in I \\ T & \text{altrimenti} \end{cases}$$

$$\text{dove } T = \bigcup_{\substack{t \in (\!|c(q)|\!)_{\mathtt{s}}(p) \\ t \neq \bot \\ \delta(q,t(p)) \notin I}} (\!|\delta(q, t(p))|\!)^{\mathcal{F},I \cup \{q\}}_{\mathtt{s}}\ (t(p)) \ltimes t$$

e per il nodo finale

$$(\!|q_f|\!)^{\mathcal{F},I}_{s}\ p = \{id\}$$

Notiamo che, poiché la semantica della ruleset associata ad un nodo q, dato un pacchetto p restituisce un insieme di possibili trasformazioni T_q è necessario considerare separatamente il risultato di ognuna delle trasformazioni $t \in T_q$. Se esiste almeno una trasformazione $t_i \in T_q$ che è uguale a $\bot$ oppure tale che il prossimo nodo da visitare dato q e il pacchetto $p' = t_i(p)$ creerebbe un loop, allora sappiamo che $\bot$ è una delle possibili trasformazioni associate a p dal firewall. Esclusa la possibilità di essere scartato nel nodo q, tutte le possibili trasformazioni associate a p sono elementi dell'insieme T, che viene costruito prendendo ogni trasformazione $t \in T_q$ che permetta al pacchetto p di essere passato ad un nuovo nodo q' (dipendente da t) e collezionando le trasformazioni associate a $p' = t(p)$ dalla semantica del nodo q'.

Bibliografia

[1] P. Adão, C. Bozzato, G. D. Rossi, R. Focardi, and F. L. Luccio. Mignis: A semantic based tool for firewall configuration. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 351–365, 2014.

[2] Babel. The compiler for writing next generation JavaScript. `https://babeljs.io`.

[3] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. *Firmato*: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[4] C. Bodei, P. Degano, R. Focardi, L. Galletta, and M. Tempesta. Transcompiling firewalls. In L. Bauer and R. Küsters, editors, *Principles of Security and Trust*, pages 303–324, Cham, 2018. Springer International Publishing.

[5] C. Bodei, P. Degano, R. Focardi, L. Galletta, M. Tempesta, and L. Veronese. Language-independent synthesis of firewall policies. In *Proc. 3rd IEEE European Symposium on Security and Privacy*, 2018.

[6] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Formal Aspects in Security and Trust (FAST'04)*, pages 203–218, 2004.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[8] C. Diekmann, J. Michaelis, M. P. L. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *Proceedings of the 15th IFIP Networking Conference, Vienna, Austria, May 17-19, 2016*, pages 252–260, 2016.

[9] S. N. Foley and U. Neville. A firewall algebra for openstack. In *2015 IEEE Conference on Communications and Network Security, CNS 2015*, pages 541–549, 2015.

[10] FreeBSD Packet Filter (PF). `https://www.freebsd.org/cgi/man.cgi?query=pf.conf&sektion=5&n=1`.

[11] M. Gouda and A. Liu. Structured firewall design. *Computer Networks*, 51(4):1106–1120, Mar. 2007.

[12] A. J. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 177–187, 2000.

[13] A. J. Mayer, A. Wool, and E. Ziskind. Offline firewall analysis. *Int. J. Inf. Sec.*, 5(3):125–144, 2006.

[14] Mignis Compiler. `https://github.com/secgroup/Mignis`.

[15] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, November 7-12, 2010*, 2010.

[16] Open Networking Foundation. Software-Defined Networking (SDN) Definition. `https://www.opennetworking.org/sdn-resources/sdn-definition`.

[17] OpenBSD Packet Filter (PF). `https://www.openbsd.org/faq/pf/`.

[18] Oskar Andreasson. `https://www.frozentux.net/iptables-tutorial/iptables-tutorial.html`.

[19] Runtime converter. `http://www.runtimeconverter.com`.

[20] The IPFW Firewall. `https://www.freebsd.org/cgi/man.cgi?ipfw(8)`.

[21] The netfilter project. `http://ipset.netfilter.org/iptables.man.html`.

[22] The Netfilter Project. `https://www.netfilter.org/`.

[23] The Netfilter Project. Traversing of tables and chains. `http://www.iptables.info/en/structure-of-iptables.html`.

[24] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), May 2006, Berkeley, California, USA*, pages 199–213, 2006.

Appendice A

Dimostrazioni

Presentiamo la dimostrazione di teoremi, lemmi e corollari presentati nella tesi, divisi seguendo i capitoli di riferimento.

A.1 Correttezza della normalizzazione e della caratterizzazione logica di IFCL

Teorema 1 (Correttezza della normalizzazione)**.** *Sia $\mathcal{F}$ un firewall e $(\!(\ \mathcal{F}\ )\!)$ la sua versione normalizzata. Chiamiamo $s \xrightarrow{p,p'}_X s'$ un passo del sistema di transizione master del firewall $X \in \{\mathcal{F}, (\!(\ \mathcal{F}\ )\!)\}$. Vale che*

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{(\!(\ \mathcal{F}\ )\!)} s'.$$

Dimostrazione. Si veda [4] per la dimostrazione di questo teorema. □

Lemma 1. *Data una ruleset R abbiamo che*

1. $\forall p, s.\ p, s \models^{\epsilon}_R (\textsc{accept}, p') \implies P_R(p, p', \hat{s})$;
2. $\forall p, p', \mathbf{s}\ .\ P_R(p, p', \mathbf{s}) \implies \exists s \in S\ .\ \hat{s} = \mathbf{s}\ \wedge\ p, s \models^{\epsilon}_R (\textsc{accept}, p')$

Dimostrazione. Si veda [4] per la dimostrazione di questo teorema. □

Teorema 2 (Correttezza della caratterizzazione logica)**.** *Dato un firewall $\mathcal{F}$ ed il suo predicato corrispondente $\mathcal{P}_{\mathcal{F}}$ abbiamo che*

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p', \hat{s})$
2. $\forall p, p', \mathbf{s}.\ \mathcal{P}_{\mathcal{F}}(p, p', \mathbf{s}) \implies \exists s \in S\ .\ \hat{s} = \mathbf{s}\ \wedge\ s \xrightarrow{p,p'} s \uplus (p, p')$

Dimostrazione. Si veda [4] per la dimostrazione di questo teorema. □

A.2 Correttezza della caratterizzazione funzionale

Lemma 2. *Sia R una ruleset normalizzata IFCL, abbiamo che*

1. $\forall p, p', s.\ (\ p, s \models^{\epsilon}_R (\textsc{accept}, p') \iff [\![R]\!](s)(p) = p'\ )$
2. $\forall p, s.\ (\ [\![R]\!](s)(p) = \bot \iff \exists p''.\ p, s \models^{\epsilon}_R (\textsc{drop}, p')\ )$

Dimostrazione. La dimostrazione procede per induzione sulla lunghezza della ruleset R:

caso base: $R = \epsilon$, secondo la semantica denotazionale applico la default policy restituendo p o $\bot$; in entrambi i casi viene applicata la regola (12) della semantica operazionale. Quindi l'enunciato vale, dove nel caso di pacchetto scartato, $p'' = p$.

passo induttivo: Assumo vero l'enunciato per ogni ruleset di lunghezza n, sia $R = r \cdot R'$ una ruleset di lunghezza $n+1$. Sia $r = (\phi(p,s), a)$, se $\phi(p,s)$ non è verificato allora $[\![R]\!](s)(p)$ è definita come $[\![R']\!](s)(p)$ e banalmente, per ogni t, $p, s \Vdash_R (t, i)$ per qualche i se e solo se $p, s \Vdash_{R'} (t, j)$ per qualche j. Essendo R' di lunghezza n, l'enunciato da dimostrare è valido per ipotesi induttiva. Altrimenti, se $\phi(p,s)$ è verificato procediamo per casi in base al target a:

- se $a = \texttt{ACCEPT}$ allora
  $[\![R]\!](s)(p) = p$ e $p, s \models^{\epsilon}_R (\texttt{ACCEPT}, p)$, per la regola (1) della semantica operazionale, dato che $p, s \Vdash_R (\texttt{ACCEPT}, i)$;
- se $a = \texttt{DROP}$ allora
  $[\![R]\!](s)(p) = \bot$ e $p, s \models^{\epsilon}_R (\texttt{DROP}, p)$, per la regola (1) della semantica operazionale, dato che $p, s \Vdash_R (\texttt{DROP}, i)$;
- se $a = \texttt{NAT}(d_n, s_n)$ allora
  $[\![R]\!](s)(p) = nat(p, d_n, s_n)$ e $p, s \models^{\epsilon}_R (\texttt{ACCEPT}, nat(p, d_n, s_n))$, per la regola (4) della semantica operazionale, dato che $p, s \Vdash_R (\texttt{NAT}(d_n, s_n), i)$;
- se $a = \texttt{CHECK-STATE}(X)$ e vale $p \vdash_s \alpha$, allora
  $[\![R]\!](s)(p) = enstabl(\alpha, X, p)$ e $p, s \models^{\epsilon}_R (\texttt{ACCEPT}, enstabl(\alpha, X, p))$, per la regola (2) della semantica operazionale, dato che $p, s \Vdash_R (\texttt{CHECK-STATE}(X), i)$;
- se $a = \texttt{CHECK-STATE}(X)$ ma non vale $p \vdash_s \alpha$, allora
  $[\![R]\!](s)(p) = [\![R']\!](s)(p)$ e, per la regola (3) della semantica operazionale, $p, s \models^{\epsilon}_R (t, p'')$ se e solo se $p, s \models^{\epsilon}_{R'} (t, p'')$;
- se $a = \texttt{MARK}(m)$ allora
  $[\![R]\!](s)(p) = [\![R']\!](s)(p) = p[tag \mapsto m]$ e, per la regola (13) della semantica operazionale, $p, s \models^{\epsilon}_R (t, p'')$ se e solo se $p[tag \mapsto m], s \models^{\epsilon}_{R'} (t, p'')$.

□

Teorema 3 (Correttezza della caratterizzazione funzionale)**.** *Dato un firewall normalizzato $\mathcal{F}$ abbiamo che*

1. $\forall p, p', s.\ (\ s \xrightarrow{p,p'} s \uplus (p, p')\ \iff\ [\![\mathcal{F}]\!](s)(p) = p'\ )$
2. $\forall p, s.\ (\ [\![\mathcal{F}]\!](s)(p) = \bot\ \iff\ \neg\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')\ )$

Dimostrazione. 1. segue banalmente da $(q, s, p) \to^+_I (q_f, s, p')\ \iff\ [\![q]\!]^{\mathcal{F},I}(s)(p) = p'$, che dimostriamo per induzione sul numero di passi nel sistema di transizione slave di IFCL.

caso base: $(q, s, p) \to^+_I (q_f, s, p')$ in un passo se e solo se $(q, s, p) \to (q_f, s, p')$ e $q_f \notin I$, dove $(q, s, p) \to (q_f, s, p')$ a sua volta è vero se e solo se $p, s \models^{\epsilon}_{c(q)} (\texttt{ACCEPT}, p')$ e $\delta(q, p') = q_f$; $[\![\mathcal{F}]\!](s)(p) = p'$ in un passo se e solo se $[\![c(q)]\!](s)(p) = p'$, $\delta(q, p') = q_f$, $q_f \notin I$ e $p' = [\![q_f]\!](s)(p')$. La tesi segue dal fatto che $\forall p \in \mathbb{P}.\ [\![q_f]\!](s)(p) = p$ e dal lemma 2.

passo induttivo: $(q, s, p) \to^+_I (q_f, s, p')$ in n passi se e solo se $(q, s, p) \to (q'', s, p'')$, $q'' \notin I$ e $(q'', s, p'') \to^+_{I \cup \{q''\}} (q_f, s, p')$; dove $(q, s, p) \to (q'', s, p'')$ a sua volta è vero se e solo se $p, s \models^{\epsilon}_{c(q)} (\texttt{ACCEPT}, p'')$ e $\delta(q, p'') = q''$; $[\![\mathcal{F}]\!](s)(p) = p'$ in n passi se e solo se $[\![c(q)]\!](s)(p) = p'''$, $\delta(q, p''') = q'$,

$q' \notin I$ e $p' = [\![q']\!]^{\mathcal{F},I\cup\{q'\}}(s)(p''')$. Dal lemma 2 deriva che $p''' = p''$ e di conseguenza $q' = q''$; dall'ipotesi induttiva, essendo il cammino da q'' a q_f lungo $n-1$, deriva che $[\![q']\!]^{\mathcal{F},I\cup\{q''\}}(s)(p'') = p' \iff (q'', s, p'') \rightarrow^{+}_{I\cup\{q''\}} (q_f, s, p')$.

Per punto *2.*, dimostriamo che la parte sinistra implica la destra per contropositiva: assumiamo $\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')$, allora per il punto *1.* vale che $[\![\mathcal{F}]\!](s)(p) = p' \neq \bot$. Dimostriamo anche l'inverso per contropositiva: assumiamo che $[\![\mathcal{F}]\!](s)(p) \neq \bot$, ma allora $\exists p'.\ [\![\mathcal{F}]\!](s)(p) = p'$ e quindi per il punto *1.* vale che $\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')$. □

Corollario 1 (Determinismo dei firewall)**.** *Dato un firewall* IFCL $\mathcal{F}$*, il destino associato ad un pacchetto p è unico, ovvero*

$$\forall p, s.\ (!\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')) \vee (\neg\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')\ )$$

Dimostrazione. Possiamo assumere che $\mathcal{F}$ sia normalizzato, senza perdita di generalità dato il teorema 2.4. Prendiamo il risultato di $[\![\mathcal{F}]\!](s)(p)$ e consideriamo i due casi possibili:

- se il risultato è $\bot$ allora per il teorema 3 abbiamo che $\neg\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')$.
- se il risultato è $p' \neq \bot$ allora per il teorema 3 abbiamo che $s \xrightarrow{p,p'} s \uplus (p, p')$, quindi $\exists p'.\ s \xrightarrow{p,p'} s \uplus (p, p')$; supponiamo che $\exists p', p''.\ s \xrightarrow{p,p'} s \uplus (p, p') \wedge s \xrightarrow{p,p''} s \uplus (p, p'')$, allora, sempre per il teorema 3 $p' = [\![\mathcal{F}]\!](s)(p) = p''$.

□

Lemma 3. *Per ogni ruleset normalizzata R, stato s e pacchetto p valgono*

1. $[\![R]\!]_s\ (p) = \bot \iff (\!|R|\!)_s\ (p) = \bot$

2. $(\!|R|\!)_s\ (p) \neq \bot \Rightarrow [\![R]\!]_s\ (p) = (\!|R|\!)_s\ (p)\ (p)$

Dimostrazione. Dimostriamo *1.* per induzione sulla lunghezza di R:

caso base: R è la ruleset vuota, in questo caso entrambe le semantiche si comportano in accordo a dp: se $dp =$ ACCEPT allora $[\![R]\!]_s(p) = p$ e $(\!|R|\!)_s(p) = id$, con $id(p) = p$; altrimenti $dp =$ DROP e $[\![R]\!]_s(p) = (\!|R|\!)_s(p) = \bot$.

passo induttivo: $R = r \cdot R'$, con $r = (\phi, a)$: $[\![R]\!]_s(p) = \bot$ se $\phi(p, s)$ e $a =$ DROP, oppure se $\neg\phi(p, s)$ e $[\![R']\!]_s(p) = \bot$; $(\!|R|\!)_s(p) = \bot$ se $\phi(p, s)$ e $a =$ DROP, oppure se $\neg\phi(p, s)$ e $(\!|R'|\!)_s(p) = \bot$, l'enunciato segue banalmente dall'ipotesi induttiva.

Per *ii.* dimostriamo per induzione sulla lunghezza di R che vale $[\![R]\!]_s(t(p)) = p' \iff (\!|R|\!)^t_s(t(p))(p) = p'$:

caso base: R è la ruleset vuota, $dp \neq$ DROP per assunzione, quindi $[\![R]\!]_s(t(p)) = t(p)$ e $(\!|R|\!)^t_s(t(p))(p) = id$.

passo induttivo: assumiamo che l'enunciato valga per ogni ruleset di lunghezza $n-1$, sia $R = r \cdot R'$ una ruleset di lunghezza n, dove $r = (\phi, a)$; se $\neg\phi(p, s)$ allora $[\![R]\!]_s(t(p)) = [\![R']\!]_s(t(p))$ e $(\!|R|\!)^t_s(t(p)) = (\!|R'|\!)^t_s(t(p))$, che sono uguali per ipotesi induttiva. Se invece $\phi(p, s)$ vale allora per casi su a

- se $a =$ ACCEPT allora $(\!|R|\!)^t_s(t(p)) = t$ e $[\![R]\!]_s(t(p)) = t(p)$;
- $a =$ DROP non può essere perché abbiamo assunto che il pacchetto non viene scartato;

- se $a = \texttt{NAT}(d_n, s_n)$ allora $(\!|R|\!)^t_s(t(p)) = tr_{nat}(d_n, s_n) \ltimes t$ e $[\![R]\!]_s(t(p)) = nat(t(p), d_n, s_n)$; $(tr_{nat}(d_n, s_n) \ltimes t)(p) = tr_{nat}(d_n, s_n)(t(p)) = nat(t(p), d_n, s_n)$;
- se $a = \texttt{CHECK-STATE}(X)$ e $p \vdash_s \alpha$ allora $(\!|R|\!)^t_s(t(p)) = tr_{stato}(\alpha, X) \ltimes t$ e $[\![R]\!]_s(t(p)) = enstabl(\alpha, X, t(p))$; $(tr_{stato}(\alpha, X) \ltimes t)(p) = tr_{stato}(\alpha, X)(t(p)) = enstabl(\alpha, X, t(p))$;
- se $a = \texttt{CHECK-STATE}(X)$ e $p \nvdash_s$ allora $[\![R]\!]_s(t(p)) = [\![R']\!]_s(t(p))$ e $(\!|R|\!)^t_s(t(p)) = (\!|R'|\!)^t_s(t(p))$, che sono uguali per ipotesi induttiva;
- se $a = \texttt{MARK}(m)$ allora

$$
\begin{aligned}
&[\![R]\!]_s(t(p)) = \\
&[\![R']\!]_s((id : id,\ id : id, cost(m))(t(p))) = \\
&[\![R']\!]_s(((id : id,\ id : id, cost(m)) \ltimes t)(p)))
\end{aligned}
$$

$$
\begin{aligned}
&(\!|R|\!)^t_s(t(p))(p) = \\
&(\!|R'|\!)^{(id:id,\ id:id, cost(m)) \ltimes t}_s((id : id,\ id : id, cost(m))(t(p))) = \\
&(\!|R'|\!)^{(id:id,\ id:id, cost(m)) \ltimes t}_s((id : id,\ id : id, cost(m)) \ltimes t)(p)
\end{aligned}
$$

  che sono uguali per ipotesi induttiva.

□

Teorema 4 (Correttezza della semantica a trasformazioni)**.** *Per ogni firewall $\mathcal{F}$, stato s e pacchetto p valgono*

1. $[\![\mathcal{F}]\!]_s\ (p) = \bot \iff (\!|\mathcal{F}|\!)_s\ (p) = \bot$
2. $(\!|\mathcal{F}|\!)_s\ (p) \neq \bot \Rightarrow [\![\mathcal{F}]\!]_s\ (p) = (\!|\mathcal{F}|\!)_s\ (p)\ (p)$

Dimostrazione. Dimostriamo per induzione sulla lunghezza del cammino che $[\![q]\!]^{\mathcal{F},I}_s(p) = p' \neq \bot$ se e solo se $(\!|q|\!)^{\mathcal{F},I}_s(p) = t \neq \bot \wedge t(p) = p'$.

caso base: consideriamo un cammino composto da un unico nodo q, se il pacchetto non viene scartato allora $q = q_f$ e quindi $[\![q_f]\!]^{\mathcal{F},I}_s(p) = p \neq \bot$, $(\!|q_f|\!)^{\mathcal{F},I}_s(p) = id \neq \bot \wedge id(p) = p$; altrimenti $[\![q]\!]^{\mathcal{F},I}_s(p) = \bot$ vale se e solo se $[\![c(q)]\!](s)(p) = \bot$ oppure $[\![c(q)]\!](s)(p) = p'$ e $\delta(q, p') \in I$, il primo caso, per il lemma 3, è vero se e solo se $(\!|c(q)|\!)(s)(p) = \bot$, il secondo caso invece, sempre per il lemma 3, è vero se e solo se $[\![c(q)]\!](s)(p) = t \neq \bot$ e $\delta(q, t(p)) \in I$, dove uno dei due casi è vero se e solo se $(\!|q|\!)^{\mathcal{F},I}_s(p) = \bot$.

passo induttivo: assumiamo l'enunciato per cammini lunghi $n - 1$, sia il cammino da q a q', dove $q' = q_f$ oppure il pacchetto viene scartato al nodo q' da $\mathcal{F}$. Non può essere che $[\![q]\!]^{\mathcal{F},I}_s(p) = \bot$ in quanto il cammino non sarebbe lungo n, per il caso base quindi anche $(\!|q|\!)^{\mathcal{F},I}_s(p)$ è diverso da $\bot$. $[\![q]\!]^{\mathcal{F},I}_s(p) = p'$ se e solo se $[\![c(q)]\!](s)(p) = (p'')$, $\delta(q, p'') = q'$ e $[\![q']\!]^{\mathcal{F},I}_s(p'') = p'$. Per il lemma 3 $[\![c(q)]\!](s)(p) = (p'')$ è vero se e solo se $(\!|c(q)|\!)(s)(p) = t$ con $t(p) = p''$ e per ipotesi induttiva $[\![q']\!]^{\mathcal{F},I}_s(p'') = p'$ è vero se e solo se $(\!|q'|\!)^{\mathcal{F},I}_s(p'') = t'$ e $t'(p'') = p'$. Infine si noti che vale $(\!|q|\!)^{\mathcal{F},I}_s(p)(p) = p'$ se e solo se $(\!|q|\!)^{\mathcal{F},I}_s(p) = t'$ e $t'(p) = p'$; dove $(\!|q|\!)^{\mathcal{F},I}_s(p) = t'$ è vero se e solo se $(\!|c(q)|\!)(s)(p) = t$, se $t(p) = p''$ e se $(\!|q'|\!)^{\mathcal{F},I}_s(p'') = t'$ e $t'(p'') = p'$.

□

A.3 Correttezza della pipeline di transcompilazione

Lemma 4. *Due firewall IFCL $\mathcal{F}$ e $\mathcal{F}'$ sono equivalenti secondo la semantica operazionale se e solo se la loro normalizzazione è equivalente secondo la semantica denotazionale, ovvero*

$$\forall s \in S, p \in \mathbb{P}.\ \forall p', s'.\ s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{\mathcal{F}'} s'$$

se e solo se

$$(\!|\mathbb{C}\ \mathcal{F}'\ \mathbb{D}|\!) \equiv (\!|\mathbb{C}\ \mathcal{F}'\ \mathbb{D}|\!)$$

Dimostrazione. Per il teorema 2.4 abbiamo che $s \xrightarrow{p,p'}_{\mathcal{F}} s'$ se e solo se $s \xrightarrow{p,p'}_{\mathbb{C}\ \mathcal{F}\ \mathbb{D}} s'$. Per il teorema 3 abbiamo che $s \xrightarrow{p,p'}_{\mathbb{C}\ \mathcal{F}\ \mathbb{D}} s'$ se e solo se $[\![\mathcal{F}]\!](s)(p) = p'$ e $s' = s \uplus (p, p')$. Per il teorema 4 abbiamo che $[\![\mathcal{F}]\!](s)(p) = p'$ se e solo se $(\!|\mathcal{F}|\!)(s)(p)(p) = p'$. Per definizione di $\equiv$, $(\!|\mathcal{F}|\!) \equiv (\!|\mathcal{F}|\!)$ se e solo se per ogni p e s, $(\!|\mathcal{F}|\!)(s)(p)(p) = (\!|\mathcal{F}|\!)(s)(p)(p)$. Il teorema segue per transitività. □

Teorema 5 (Correttezza della pipeline)**.** *Sia* `file.conf` *un firewall concreto in uno qualunque dei sistemi $k \in \{$* `iptables`, `ipfw`, `pf`*$\}$. Il firewall target* `file.conf'` *prodotto dalla pipeline di transcompilazione, per il sistema target k', ha la stessa semantica del firewall source per quanto riguarda pacchetti non appartenenti a connessioni stabilite. Formalmente vale:*

$$(\!|\mathbb{C}\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ \mathbb{D}|\!)(s_{NEW}) \equiv (\!|\mathbb{C}\ (\mathcal{C}_{k'}, for_{k'}(\texttt{file.conf}'))\ \mathbb{D}|\!)(s_{NEW})$$

Dimostrazione. Per definizione di $con_{k'}$, $\mathbb{C}\ (\mathcal{C}_{k'}, for_{k'}(\texttt{file.conf'}))\ \mathbb{D}$ è uguale a $(\mathcal{C}_{k'}, \Sigma')$ alla fine del passo 3.b della pipeline. Sia $\mathbb{C}\ (\mathcal{C}_k, for_k(\texttt{file.conf}))\ \mathbb{D}$ uguale a $(\mathcal{C}_k, \Sigma)$.

$$\begin{aligned}
&(\!|(\mathcal{C}_k, \Sigma)|\!) = \\
&\odot\,(\mathcal{C}_k, f) = && \text{dove } \forall q \in Q.\ f(q) = (\!|c(q)|\!)(s_{\texttt{NEW}}) \\
&\odot\,(\mathcal{C}_{k'}, f') = && \text{dove } \forall q \in Q'.\ f'(q) = (\!|c'(q)|\!)(s_{\texttt{NEW}}) \\
&(\!|(\mathcal{C}_{k'}, \Sigma')|\!)
\end{aligned}$$

Dove la seconda uguaglianza vale per ipotesi della pipeline, in particolare il fatto che $\mathcal{F}_4 = \mathcal{F}'_4$, e le altre due valgono banalmente in quanto i due termini sono sintatticamente identici. □

A.4 Correttezza della sintesi di un firewall

Teorema 6. *Sia $\mathcal{F}_2 = (\mathcal{C}, \Sigma)$ con $\Sigma = (\rho, c)$ il firewall normalizzato in input all'algoritmo di semiastrazione. L'algoritmo produce un firewall $\tilde{\mathcal{F}}_3 = (\mathcal{C}, \tilde{f})$ tale che*

- *$\tilde{\mathcal{F}}_3$ è un firewall semiastratto sintetizzato*
- *la funzione sintetizzata $\tilde{f}$ è tale che per ogni $q \in Q_k$ vale $i(\tilde{f})(q) = (\!|c(q)|\!)(s_{NEW})$*

Dimostrazione. $\tilde{\mathcal{F}}_3$ è un firewall semiastratto sintetizzato in quanto:

- $\forall (P, t) \in \tilde{f}(q).\ P \in \mathcal{M}(\mathbb{P})$, come abbiamo mostrato nel capitolo 5, in quanto le funzioni usate preservano la forma del parametro della funzione che inizialmente è un multicubo;
- $\forall (P, t), (P', t') \in \tilde{f}(q).\ P \cap P' = \emptyset$, per assurdo assumiamo che esista un p tale che $p \in P$ e $p \in P'$, allora esistono un t e un ϕ tali che $\phi(t(p), s) \wedge \neg\phi(t(p), s)$;
- $\bigcup_{(P,t)\in\tilde{f}(q)} P = \mathbb{P}$, ovvero $\forall p \in \mathbb{P}.\ \exists (P, t) \in \tilde{f}(q).\ p \in P$, per assurdo, se non fosse vero, allora avremmo che esistono un t e un ϕ tali che $\neg\phi(t(p), s) \wedge \neg\neg\phi(t(p), s)$;

- per quanto riguarda il fatto che $\forall (P,t) \in \tilde{f}(q).\ P \neq \emptyset$, abbiamo assunto di controllare ogni coppia prima di inserirla, il controllo non è mostrato nell'algoritmo per leggibilità.

$\forall q \in Q_k.\ i(\tilde{f})(q) = (\!|c(q)|\!)(s_{\texttt{NEW}})$ deriva dal fatto che per ogni ruleset R, trasformazione t e multicubo di pacchetti P, se $p \in P$ allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = i(\text{RULESET\_SYMTHESIS}(P,R,t))(p)$, che dimostriamo per induzione sulla lunghezza della ruleset R.

caso base: $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = t$,

$i(\text{RULESET\_SYMTHESIS(P, R, t)})(p) = i(\{(P,t)\})(p) = t$.

passo induttivo: si assuma che l'enunciato sia vero per ogni ruleset di lunghezza $n-1$, sia $R = (\phi, a) \cdot R'$ una ruleset lunga n: se $\neg\phi(p, s_{\texttt{NEW}})$ oppure $a = \texttt{CHECK-STATE}(X)$ e $p \not\vdash_s$, allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p)$ e $i(\text{RULESET\_SYMTHESIS}(P,R,t))(p)$ sono rispettivamente uguali a

$(\!|R'|\!)^t_{s_{\texttt{NEW}}}(p)$ e $i(\text{RULESET\_SYMTHESIS}(P_n,R,t))(p)$,

per un $P_n \in \mathbf{P}_n$ tale che $p \in P_n$, i quali sono uguali per ipotesi induttiva.

Altrimenti, se $\phi(p, s_{\texttt{NEW}})$ allora $p \in P_s$; analizziamo l'enunciato per casi su a:

- se $a = \texttt{ACCEPT}$ allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = t$,
  $i(\text{RULESET\_SYMTHESIS}(P,R,t))(p) = i(\{(P_s,t)\} \cup \dots)(p) = t$;
- se $a = \texttt{DROP}$ allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = \bot$,
  $i(\text{RULESET\_SYMTHESIS}(P,R,t))(p) = i(\{(P_s,\bot)\} \cup \dots)(p) = \bot$;
- se $a = \texttt{NAT}(d_n, s_n)$ allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = tr_{nat}(d_n, s_n)$,
  $i(\text{RULESET\_SYMTHESIS(P, R, t)})(p) = i(\{(P_s, tr_{nat}(d_n,s_n))\} \cup \dots)(p) = tr_{nat}(d_n,s_n)$;
- il caso in cui $a = \texttt{CHECK-STATE}(X)$ e $p \vdash_{s_{\texttt{NEW}}} \alpha$ non è contemplato in quanto impossibile per definizione nello stato $s_{\texttt{NEW}}$;
- se $a = \texttt{MARK}(m)$ allora $(\!|R|\!)^t_{s_{\texttt{NEW}}}(p) = (\!|R'|\!)^{(id:id,\ id:id,\ cost(m)) \ltimes t}_{s_{\texttt{NEW}}}(p[tag \mapsto m])$,
  mentre $\text{RULESET\_SYMTHESIS}(P_s, R, (id:id,\ id:id,\ cost(m)) \ltimes t)) \subseteq \text{RULESET\_SYMTHESIS}(P, R, t))$;
  quindi da $p \in P_s$ segue che $i(\text{RULESET\_SYMTHESIS}(P,R,t))(p)$ sia uguale a
  $i(\text{RULESET\_SYMTHESIS}(P_s, R, (id:id,\ id:id,\ cost(m)) \ltimes t))(p)$ e quindi l'enunciato è vero per ipotesi induttiva.

□

Teorema 7. *Se due firewall sono simili allora hanno semantica equivalente per quanto riguarda i pacchetti non ciclanti di $\mathcal{F}$, ovvero: $\mathcal{F} \trianglerighteq \mathcal{F}' \implies (\ \forall p \notin pc(\mathcal{F}), s \in S.\ (\!|\mathcal{F}|\!)(s)(p) = (\!|\mathcal{F}'|\!)(s)(p)\ )$.*

Dimostrazione. Il teorema segue dal predicato

$$(\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}', q', I') \Rightarrow (\forall p \notin pc(\mathcal{F}), s \in S.\ (\!|q|\!)^{\mathcal{F},I}_s(p) = (\!|q'|\!)^{\mathcal{F}',I'}_s(p))$$

che dimostriamo per induzione sulla lunghezza del cammino del pacchetto p in $\mathcal{F}$.

base induttiva: il cammino comprende un solo nodo, sono possibili due alternative

- $q = q_f$, allora assumendo l'antecedente abbiamo che $q' = q'_f$ e quindi $(\!|q_f|\!)^{\mathcal{F},I}_s(p) = (\!|q'_f|\!)^{\mathcal{F}',I'}_s(p) = id$;
- altrimenti l'unica alternativa è che $(\!|q|\!)^{\mathcal{F},I}_s(p) = \bot$, che è vero se e solo se $(\!|c(q)|\!)_s(p) = \bot$ (il pacchetto non può essere ciclante per ipotesi). Ma allora per l'antecedente vale che $c(q) = c'(q')$ e quindi $(\!|c'(q')|\!)_s(p) = \bot$ e $(\!|q'|\!)^{\mathcal{F}',I'}_s(p) = \bot$.

passo induttivo: assumiamo che l'enunciato sia vero per percorsi lunghi $n-1$, assumiamo l'antecedente dell'enunciato, sia q un nodo dal quale il pacchetto p impiega $n-1$ passi per essere accettato o scartato.

$(\!|c(q)|\!)_s(p)$ non può essere $\perp$ in quanto altrimenti il cammino non sarebbe lungo n; per l'antecedente vale $c(q) = c'(q')$. Chiamiamo t il risultato di $(\!|c(q)|\!)_s(p)$, $\bar{p} = t(p)$ e $q_1 = \delta(q, \bar{p})$. Vale che $(\!|q|\!)_s^{\mathcal{F},I}(p) = (\!|q_1|\!)_s^{\mathcal{F},I\cup\{q_1\}}(\bar{p})$. Da $(\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}', q', I')$ deriva che esiste un q_1' tale che $q_1' = \delta(q', \bar{p})$ e $(\mathcal{F}, q_1, I \cup \{q_1\}) \trianglerighteq (\mathcal{F}', q_1', I' \cup \{q_1'\})$ dal quale deriva per ipotesi induttiva che, per ogni p'' non ciclante in $\mathcal{F}$, $(\!|q_1|\!)_s^{\mathcal{F},I\cup\{q_1\}}(p'') = (\!|q_1'|\!)_s^{\mathcal{F}',I'\cup\{q_1'\}}(p'')$. Infine osserviamo che $(\!|q'|\!)_s^{\mathcal{F}',I'}(p) = (\!|q_1'|\!)_s^{\mathcal{F}',I'\cup\{q_1'\}}(\bar{p})$, e che $\bar{p}$ non è ciclante in $\mathcal{F}$.

□

Teorema 8. *Sia $\mathcal{F}$ un firewall IFCL, sia $\mathcal{F}_u$ il risultato dell'applicazione della funzione UNLOOP al firewall $\mathcal{F}$:*

1. *$\mathcal{F}_u$ è un firewall IFCL aciclico*
2. *$\mathcal{F} \trianglerighteq \mathcal{F}_u$*
3. *$\forall p \in pc(\mathcal{F}), s \in S.\ (\!|\mathcal{F}_u|\!)(s)(p) = \perp$*

Dimostrazione. Dimostriamo i punti uno per uno:

1. Per assurdo, assumiamo che un pacchetto percorre un percorso contenente un nodo ripetuto q. Distinguiamo due casi: se $q = q_i$ abbiamo una contraddizione perché nell'algoritmo il nodo q_i non compare mai come destinazione di un arco in A_u; altrimenti perché sia possibile un loop servono almeno due archi diversi (q_1, ψ_1, q) e (q_2, ψ_1, q), ma questo è impossibile per ogni nodo diverso da q_f e $q_\perp$ in quanto ogni nodo viene generato fresco ad una iterazione ed usato solo in quella come nodo destinazione di un unico arco, ed è impossibile anche per $q_\perp$ in quanto non ha archi uscenti e per $q_\perp$ il cui unico arco uscente ha come destinazione $q_\perp$.

2. Per prima cosa notiamo che a partire da UNLOOP_REC$(\mathcal{F}, q_i, q_i, \{q_i\})$, per ogni chiamata UNLOOP_REC$(\mathcal{F}, q, q_u, I)$, $q = q_f$ se e solo se $q_u = q_f$. Per induzione sul numero di chiamate ricorsive alla funzione UNLOOP_REC dimostriamo allora che

$$\text{UNLOOP\_REC}(\mathcal{F}, q, q_u, I) = (Q_u, A_u, c_u) \Rightarrow (\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}_u, q_u, I)$$
$$\text{dove } \mathcal{F}_u = (\mathcal{C}_u, \Sigma_u), \mathcal{C}_\sqcap = (Q_u, A_u, c_u, c_f), \Sigma_u = (c_u, \rho_u)$$

   caso base: È sufficiente una sola applicazione della funzione se non esistono archi uscenti da q verso nodi $q' \notin I$, quindi vale $(\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}_u, q_u, I)$ in quanto $q = q_f$ se e solo se $q_u = q_f$ e l'algoritmo setta $c_u(q_u)$ uguale a $c(q)$.

   passo induttivo: Assumiamo che sia vero per $n-1$ applicazioni della funzione, sia UNLOOP_REC$(\mathcal{F}, q, q_u, I)$ un'applicazione che richiede n chiamate, allora $(\mathcal{F}, q, I) \trianglerighteq (\mathcal{F}_u, q_u, I)$ in quanto:

   - l'algoritmo modifica c_u affinché $c_u(q_u) = c(q)$;
   - $q = q_f \iff q_u = q_f$;
   - dall'algoritmo risulta che per ogni $q' \notin I$ e ψ tali che $(q, \psi, q') \in A$, esiste un q_u' tale che $(q_u, \psi, q_u') \in A_u$ e tale che UNLOOP_REC$(\mathcal{F}, q', q_u', I \cup \{q'\}) = (Q_u', A_u', c_u')$ con $Q_u' \subseteq Q_u$, $A_u' \subseteq A_u$ e $\forall q''' \in Q_u'.\ c_u'(q''') = c_u(q''')$.
     Per ipotesi induttiva vale quindi che, per $\mathcal{F}_u' = (\mathcal{C}_u', \Sigma_u')$, dove $\mathcal{C}_u' = (Q_u', A_u', c_u', c_f)$ e $\Sigma_u' = (c_u', \rho_u')$, vale $(\mathcal{F}, q', I \cup \{q'\}) \trianglerighteq (\mathcal{F}_u', q_u', I \cup \{q'\})$.

Dato che per ogni q' tale che $(q, \psi, q') \in A$ si ha che $Q'_u \subseteq Q_u$, $A'_u \subseteq A_u$ e $\forall q''' \in Q'_u.\ c'_u(q''') = c_u(q''')$, possiamo scrivere $(\mathcal{F}, q', I \cup \{q'\}) \trianglerighteq (\mathcal{F}_u, q'_u, I \cup \{q'\})$ da cui la tesi.

□

Teorema 9 (Correttezza dell'algoritmo di sintesi)**.** *thm:sintesi Sia $\mathcal{F}_2$ un firewall IFCL normalizzato aciclico, sia $\tilde{\mathcal{F}}_4$ il firewall astratto sintetizzato restituito dall'algoritmo di sintesi, allora vale:*

$$i(\tilde{\mathcal{F}}_4) = (\!|\mathcal{F}_2|\!)(s_{NEW})$$

Dimostrazione. Dimostriamo per induzione sul numero di nodi massimi attraversati che

$$\forall I.\ \forall p \in \mathbb{P}.\ i(\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q))(p) = (\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p)$$

caso base: Se il numero massimo di nodi visitati è uno allora siamo nel nodo finale q_f, dunque vale che $(\!|q_f|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p) = id$ e $\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q_f = \{(\mathbb{P}, id)\}$, con $i(\{(\mathbb{P}, id)\})(p) = id$.

passo induttivo: Assumiamo vero l'enunciato per cammini lunghi fino a $n-1$, supponiamo che il cammino dal nodo q in poi sia lungo al più n passi.

- se $(\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p) = \bot$, allora $c(q)(p) = \bot$ e per il teorema 6, $i(\tilde{f}(q))(p) = \bot$ e quindi, per la semantica della funzione DROPPER, esiste una coppia $(P, \bot)$ in $\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q)$ tale che $p \in P$ e quindi

  $$i(\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q))(p) = \bot = (\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p)$$

- se invece $(\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p) \neq \bot$ allora

  $$(\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p) = (\!|q'|\!)^{\mathcal{F}_2,\{q\}}_{s_{\text{NEW}}}(p') \ltimes t$$

  dove $t = c(q)(p)$, $p' = t(p)$ e $q' = \delta(q, p')$. Essendo che q è appena stato visitato e che il grafo è aciclico per ipotesi, possiamo scrivere equivalentemente:

  $$(\!|q|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p) = (\!|q'|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p') \ltimes t$$

  Per il teorema 6, da $t = c(q)(p)$ abbiamo che $i(\tilde{f}(q))(p) = t$, dal quale, considerando anche che $q' = \delta(q, p')$, per $p' = t(p)$, deriviamo che esiste un P, tale che $(P, t) \in \text{FILTER}(\tilde{f}(q), \psi)$ con $p \in P$ e dove ψ è il predicato sull'arco fra q e q'.

  Per ipotesi induttiva

  $$(\!|q'|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p') = i(\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q'))(p')$$

  quindi abbiamo una coppia $(P', t') \in i(\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q'))$, tale che $p' \in P'$ e $t' = (\!|q'|\!)^{\mathcal{F}_2,\emptyset}_{s_{\text{NEW}}}(p')$.

  Ma allora, chiamando $\tilde{\lambda}_{(q,q')}$ l'insieme $\text{FILTER}(\tilde{f}(q), \psi)$ e $\tilde{\lambda}_{q'}$ l'insieme $i(\text{COMPOSITION\_REC}(\tilde{\mathcal{F}}_3, q'))$; vale che $i(\text{CONCAT}(\tilde{\lambda}_{(q,q')}, \tilde{\lambda}_{q'}))(p') = t' \ltimes t$ da cui la tesi.

□

A.5 Correttezza dell'espressività di un sistema firewall

Teorema 10. *L'insieme delle configurazioni semiastratte di un diagramma di controllo* $\mathcal{C} = (Q, A, q_i, q_f)$*, legali secondo un assegnamento di etichette* v*, è l'insieme delle configurazioni di firewall ottenute dalla semiastrazione di firewall IFCL normalizzati legali secondo* v*.*

$$\mathbb{M}_3(\mathcal{C}, v) = \{f : Q \to \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid f \models v\}$$

Dimostrazione. Formalmente

$$\mathbb{M}_3(\mathcal{C}, v) = \{f \mid \forall q \in Q.\ f(q) = (\!|c(q)|\!)(s_{\texttt{NEW}}) \wedge (c, \rho) \in \mathbb{M}_2\} =$$
$$\mathbb{M}_3(\mathcal{C}, v) = \{f \mid \forall q \in Q.\ c(q) \models v(q) \wedge f(q) = (\!|c(q)|\!)(s_{\texttt{NEW}})\}$$

Il teorema è vero in quanto $\forall L \subseteq \{SNAT, DNAT, DROP\}$ vale

$$(R \models L \Rightarrow (\!|R|\!) \models L) \ \wedge \ (\lambda \models L \Rightarrow \exists R.\ (\!|R|\!) = \lambda \wedge R \models L)$$

Per primo dimostriamo $R \models L \Rightarrow (\!|R|\!) \models L$; chiamiamo $\lambda = (\!|R|\!)$. Assumiamo per assurdo che $R \models L$, ma che non valga $\lambda \models L$. Allora per definizione esiste un $l \notin L$. $\lambda \in \Lambda_{\Box l}$.

- se $l = SNAT$ allora deve esistere un $p \in \mathbb{P}$, tale che $\lambda(p) = t$ con $t.sIP \neq id$ o $t.sPort \neq id$. Dalla semantica denotazionale è evidente che l'unico modo per avere una trasformazione del genere è usare un target $\texttt{NAT}(s_n, d_n)$ con $s_n \neq \star : \star$, ma allora $SNAT$ deve appartenere a L.
- il caso di $DNAT$ è identico, con $d_n \neq \star : \star$ al posto di s_n.
- se $l = DROP$ allora deve esistere un $p \in \mathbb{P}$, tale che $\lambda(p) = \bot$, ma dalla semantica denotazionale è evidente che l'unico modo per avere una trasformazione del genere è usare un target $\texttt{DROP}$, ma allora $DROP$ deve appartenere a L.

Passiamo a $\lambda \models L \Rightarrow \exists R.\ (\!|R|\!) = \lambda \wedge R \models L$. Definiamo, data λ, come costruire la ruleset R che verifica il predicato: semplicemente per ogni coppia (p', t) tale che $\lambda(p') = t$ scriviamo una regola (ϕ, a) dove $\phi(p, s)$ è vera se e solo se il pacchetto passato per parametro è p' e il target a è tale che $(\!|(\phi, a) \cdot R'|\!)(p) = t$. Il target a può essere calcolato in maniera banale: se $t = id$ allora $a = \texttt{ACCEPT}$, se $t = \bot$ allora $a = \texttt{DROP}$ altrimenti $a = anat(d_n, s_n)$ con d_n e s_n adeguati. Vale banalmente per la ruleset R costruita che $(\!|R|\!) = \lambda$.

Assumiamo dunque $\lambda \models L$ e che R sia costruita come detto a partire da λ; vogliamo provare che $R \models L$. Procediamo per contraddizione assumendo che $R \models L$ sia falso. Allora deve sussistere uno dei seguenti casi

- esiste una regola $r = (\phi, \texttt{DROP})$ nella ruleset e $DROP \notin L$;

  ma questo non è possibile per costruzione in quanto inseriamo un target $\texttt{DROP}$ solo se $\lambda(p) = \bot$ per qualche p, e questo è possibile solo se $\lambda \in \Lambda_{\Box DROP}$, ma allora $DROP \in L$.

- esiste una regola $r = (\phi, \texttt{NAT}(ip : port, \star : \star))$ nella ruleset e $SNAT \notin L$

  ma questo non è possibile per costruzione in quanto inseriamo un target $\texttt{NAT}(ip : port, ip' : port')$ con ip o $port$ diversi da $\star$ solo se $\lambda(p).sIP \neq id$ o $\lambda(p).sPort \neq id$ per qualche p, e questo è possibile solo se $\lambda \in \Lambda_{\Box SNAT}$, ma allora $SNAT \in L$.

- esiste una regola $r = (\phi, \texttt{NAT}(\star : \star, ip : port))$ nella ruleset e $DNAT \notin L$, questo caso è identico al precedente.

- esiste una regola $r = (\phi, \texttt{NAT}(ip_1 : port_1, ip_2 : port_2))$ nella ruleset e $DNAT \notin L$ oppure $SNAT \notin L$;

  anche questo non è possibile per costruzione in quanto inseriamo un target $\texttt{NAT}(ip : port, ip' : port')$ con ip o $port$ e ip' o $port'$ diversi da $\star$ solo se $\lambda(p).sIP \neq id$ o $\lambda(p).sPort \neq id$ per qualche p e $\lambda(p').dIP \neq id$ o $\lambda(p').dPort \neq id$ per qualche p', e questo è possibile solo se $\lambda \in \Lambda_{\Box SNAT} \cap \Lambda_{\Box DNAT}$, ma allora $SNAT$ e $DNAT$ sono entrambi in L.

□

Teorema 11. *Un firewall astratto λ è legale secondo un diagramma di controllo $\mathcal{C}$ e un assegnamento di etichette v se e solo se valgono $\epsilon_0(\lambda, C, v)$ e $\epsilon_1(\lambda, C, v)$.*

$$\mathbb{M}_4(\mathcal{C}, v) = \{\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\} \mid \epsilon_0(\lambda, C, v) \wedge \epsilon_1(\lambda, C, v)\}$$

Dimostrazione.

$$\mathbb{M}_4(\mathcal{C}, v) = \{\lambda \mid f \in \mathbb{M}_3(\mathcal{C}, v).\ \odot(\pi, f)(p) = \lambda(p)\}$$

$$\epsilon_0(\lambda, \mathcal{C}, v) = \forall p \in \mathbb{P}.\ \exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p)$$
$$\epsilon_1(\lambda, \mathcal{C}, v) = (\forall p \in \mathbb{P}.\ \exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p)) \Rightarrow (\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ \forall p \in \mathbb{P}.\ (\odot(\mathcal{C}, f))\ (p) = \lambda(p))$$

Abbiamo che $\epsilon_0(\lambda, \mathcal{C}, v) \wedge \epsilon_1(\lambda, \mathcal{C}, v)$ implica per *modus ponens* $\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f)) = \lambda$ e quindi che $\lambda \in \mathbb{M}_4(\mathcal{C}, v)$.

Viceversa abbiamo che $\exists f \in \mathbb{M}_3(\mathcal{C}, v).\ (\odot(\mathcal{C}, f)) = \lambda$ implica $\epsilon_0(\lambda, \mathcal{C}, v)$ e che i due implicano $\epsilon_1(\lambda, \mathcal{C}, v)$. □

A.6 Correttezza della generazione di configurazioni

Teorema 12. *Se il sistema k è uniterale e compatto, e se la funzione $\lambda : \mathbb{P} \to \mathcal{T}(\mathbb{P}) \cup \{\bot\}$ verifica la fattibilità locale, $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$, allora $(i) \iff (ii) \wedge (iii)$*

Dimostrazione.

$$f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \forall p \in \mathbb{P}.\ \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p) \quad \text{(i)}$$
$$\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \odot(\pi, f)(p) = \lambda(p) \qquad \text{dove } \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p)) \quad \text{(ii)}$$
$$\forall p \in \mathbb{P}.\ \lambda(p) = \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \bot \quad \text{(iii)}$$

Chiaramente $(i) \iff (ii') \wedge (iii)$, dove

$$\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \exists \pi \in \Pi(\mathcal{C}).\ \Delta((\mathcal{C}, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p) \quad \text{(ii')}$$

Il teorema quindi è vero se e solo se $(ii') \iff (ii)$ assumendo che il sistema sia uniterale e compatto e che valga $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$.

Poiché il sistema è uniterale e $\epsilon_0(\lambda, \mathcal{C}_k, v_k)$ vale, l'insieme $\mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p))$ è un singoletto e dunque (ii') è equivalente a

$$\forall p \in \mathbb{P}.\ \lambda(p) \neq \bot \Rightarrow f \in \mathbb{M}_3(\mathcal{C}, v) \wedge \Delta((\mathcal{C}_k, f), p) = \pi \wedge \odot(\pi, f)(p) = \lambda(p) \quad \text{dove } \{\pi\} = \mathcal{P}(\mathcal{C}_k, v_k, p, \lambda(p))$$

Essendo k compatto e uniterale possiamo inoltre trascurare il controllo $\Delta((\mathcal{C}_k, f), p) = \pi$ in quanto sussunto da $\odot(\pi, f)(p) = \lambda(p)$, ottenendo quindi (ii). □

Teorema 13. *Se un sistema è senza NAT ripetuti, allora è compatto.*

Dimostrazione. Assumiamo che il sistema sia senza NAT ripetuti e che data una configurazione semiastratta $f \in \mathbb{M}_3(\mathcal{C}_k, v_k)$ e due pacchetti $p, p' \in \mathbb{P}$, valga

$$\Delta((\mathcal{C}_k, f), p) = \Delta((\mathcal{C}_k, f), p') = \pi \wedge \odot(\pi, f)(p) = \odot(\pi, f)(p') = t \neq \bot$$

Vogliamo dimostrare che $p \underset{(\pi,f)}{\cong} p'$. Procediamo per casi:

- se $t = id$ allora banalmente per ogni nodo in π, $f(q)(p)$ e $f(q)(p')$ deve essere id, per le proprietà della composizione.
- se $t = (id : id, tdIP : tdPort)$ dove almeno uno fra $tdIP$ e $tdPort$ è diverso da id; allora deve esistere un nodo nel percorso tale che $DNAT \in v(q)$, poiché non ci sono NAT ripetuti non ce ne può essere più di uno.

  Allora banalmente per ogni nodo $q' \neq q$ in π, $f(q')(p)$ e $f(q')(p')$ deve essere id, e $f(q)(p) = f(q)(p') = t$ per le proprietà della composizione.
- se $t = (tsIP : tsPort, id : id)$ il caso è praticamente identico al precedente, ma con $SNAT$ al posto di $DNAT$.
- se $t = (tsIP : tsPort, tdIP : tdPort)$ allora esistono due nodi, potenzialmente identici, uno etichettato con $SNAT$ e l'altro etichettato con $DNAT$. Se il nodo è unico allora faccio $f(q)(p) = f(q)(p') = t$ in quel nodo q e id in tutti gli altri; altrimenti applico la prima trasformazione nel primo nodo e la seconda nel secondo nodo.

  In ogni caso i due pacchetti subiscono le stesse trasformazioni.

□

Teorema 14. *Per ogni sistema uniterale e compatto k con diagramma di controllo $\mathcal{C}_k$, etichettato secondo v_k, percorso $\pi \in \Pi(\mathcal{C}_k)$, multicubo P e trasformazione t vale che*

$$\forall \tilde{f}.\tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f}) \iff \tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$$

Dimostrazione. Per induzione sulla lunghezza di π

caso base: se $\pi = \epsilon$ allora entrambi i predicati sono veri se e solo se $t = id$.

passo induttivo: assumo che il teorema valga per percorsi lunghi $n-1$, sia $\pi = q \cdot \pi'$ lungo n.

- $\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$ è vero se e solo se
  $\exists t', t''.\ t = t' \ltimes t''$, $t' \in \nu(\ell(\pi, \mathcal{C}_k, v_k))$, $(P, t') \tilde{\in} \tilde{f}(q)$, $t'(p) = p'$ e $\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi', P', t', \tilde{f})$.
- $\tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$ è vero se e solo se
  $\forall p \in P.\ \exists t', t''.\ t = t' \ltimes t''$, $t' \in \nu(\ell(\pi, \mathcal{C}_k, v_k))$, $i(\tilde{f})(q)(p) = t'$, $t'(p) = p'$ e $\chi(\mathcal{C}_k, v_k, \pi', p', t', i(\tilde{f}))$.

Dato che il sistema è uniterale e compatto, e che a tutti i pacchetti in P è assegnata la stessa trasformazione, in ogni nodo essi subiranno tutti la stessa trasformazione. Pertanto in

$$\tilde{\chi}(\mathcal{C}_k, v_k, \pi, P, t, \tilde{f})$$

è possibile spostare e distribuire il quantificatore nella seguente maniera:

$$\exists t', t''.\ t = t' \ltimes t'' \wedge t' \in \nu(\ell(\pi, \mathcal{C}_k, v_k)) \wedge \forall p \in P.\ i(\tilde{f})(q)(p) = t' \wedge \forall p \in P.\ \chi(\mathcal{C}_k, v_k, \pi', t'(p), t', i(\tilde{f}))$$

$$\exists t', t''.\ t = t' \ltimes t'' \wedge t' \in \nu(\ell(\pi, \mathcal{C}_k, v_k)) \wedge (P, t') \tilde{\in} \tilde{f}(q) \wedge \tilde{\chi}(\mathcal{C}_k, v_k, \pi', t'(P), t', \tilde{f})$$

Dove per ipotesi induttiva $\tilde{\chi}(\mathcal{C}_k, v_k, \pi', t'(P), t', \tilde{f})$ è verificato se e solo se $\tilde{\tilde{\chi}}(\mathcal{C}_k, v_k, \pi', P', t', \tilde{f})$.

□

Teorema 15 (Correttezza del firewall generato)**.** *Se il sistema k è senza NAT ripetuti e la funzione sintetizzata $\tilde{\lambda}$ è disgiunta, se esiste una configurazione $\Sigma \in \Gamma_k$ tale che $i(\tilde{\lambda}) = \{\!|\mathbb{C}\ (\mathcal{C}_k, \Sigma)\ \mathbb{D}|\!\}$ e tale che non esistono pacchetti ciclanti in $(\mathcal{C}_k, \Sigma)$, allora l'algoritmo 4 restituisce una configurazione semiastratta sintetizzata $\tilde{f}$ tale che*

$$i(\tilde{f}) \in \mathbb{M}_3(\mathcal{C}_k, v_k) \wedge \forall p \in \mathbb{P}.\ \odot(\mathcal{C}_k, i(\tilde{f}))(p) = i(\tilde{\lambda})(p)$$

altrimenti l'algoritmo termina segnalando errore.

Dimostrazione. L'algoritmo gestisce correttamente i pacchetti accettati in quanto le ipotesi del teorema sono più forti di quelle dei teoremi 12 e 14, e in quanto la funzione CHI è una riscrittura fedele del predicato $\tilde{\tilde{\chi}}$. Dunque per il teorema 14 vale (ii) e per il teorema 12 se vale (iii), cioè i pacchetti scartati sono gestiti correttamente, allora vale (i) (e quindi la tesi).

Dato che il sistema è uniterale e compatto, per ogni nodo q del diagramma di controllo, esistono una serie di coppie (p'', t'') tali che se non vale $f''(q)(p'') = t''$ per qualche coppia, allora non è possibile che $\odot(\mathcal{C}_k, f'') = \lambda$ e tali che se $f''(q)(p'') = t''$ è verificato per ogni nodo q e per ogni coppia dell'insieme di coppie associato a q, allora f'' verifica (ii). Dal teorema 12, deriva che le coppie (p'', t'') associate al nodo q sono tutte e sole quelle che compaiono nella forma di $f''(q)(p'') = t''$ nello svolgimento di $\chi(\mathcal{C}_k, v_k, \pi, p, t, f'')$. Quindi dal teorema 14 deriva che, per ogni nodo q, gli insiemi di pacchetti $P_\#$ non trattati dalla configurazione astratta $\tilde{f}$ prodotta da CHI, e sfruttati da FILL, sono tali che se uno qualunque dei pacchetti che non è in $P_\#$ fosse trattato diversamente nel nodo q, allora non varrebbe più (ii).

Assumiamo dunque per assurdo che esista una configurazione semiastratta f' che scarta un pacchetto p tale che $i(\tilde{\lambda})(p) = \bot$ e che $f = i(\tilde{f})$ invece non lo scarti. Dato che non sono ammessi pacchetti ciclanti, allora esiste un percorso da q_i ad un certo nodo q_d tale che $DROP \in v(q_d)$; dato che abbiamo assunto che f' verifichi (ii), il pacchetto p non viene mai trasformato in un pacchetto che sta fuori da $P_\#$. Dimostriamo per induzione sulla lunghezza del percorso che allora anche f scarta il pacchetto.

caso base: se il percorso contiene solo un nodo allora vuol dire che il nono q_i è etichettato con $DROP$, e dato che $p \in P_\#$ abbiamo che sicuramente $f(q)(p) = \bot$.

passo induttivo: se il percorso è lungo n allora f trasforma p in p' e lo passa ad un nodo successivo q'; per ipotesi induttiva allora $p' \in P_\bot$ del nodo q', quindi f manda p in un $p'' \in P_\bot$, se $p \in P_\#$. L'unico caso in cui $p \notin P_\#$ è quello in cui un valore sia già assegnato a p in q, ma dato che p non viene accettato questo vuol dire che viene già scartato da un altro nodo.

□

Teorema 16. *Sia $\rho = \{R_{snat}, R_{dnat}, R_{nat}, R_{fil}, R_{snat} \cdot R_{fil}, R_{dnat} \cdot R_{fil}, R_{nat} \cdot R_{fil}, R_\epsilon\}$, dove R_{fil}, R_{snat}, R_{dnat} e R_{nat} sono prodotto dall'algoritmo 6 con input R_λ. Sia $c : Q \to \rho$ l'assegnamento di ruleset ai nodi del diagramma di controllo del sistema target $\mathcal{C}_k$, generato secondo v_k. Se il sistema target k è senza NAT ripetuti, se ogni percorso da q_i a q_f comprende almeno un nodo etichettato con DROP, se l'interpretazione di $\tilde{\lambda}$ è localmente fattibile dal sistema target, $\epsilon_0(i(\tilde{\lambda}), \mathcal{C}_k, v_k)$, e se le etichette sugli archi non predicano sul campo tag, allora la semantica del firewall $(\mathcal{C}_k, \Sigma)$ con $\Sigma = (\rho, c)$ per lo stato $s_{\texttt{NEW}}$ è identica all'interpretazione di $\tilde{\lambda}$.*

$$\{\!|(\mathcal{C}_k, \Sigma)|\!\}(s_{\texttt{NEW}}) = i(\tilde{\lambda})$$

Dimostrazione. Dato che ogni percorso da q_i a q_f passa per almeno un nodo con etichetta $DROP$, e dato che ogni pacchetto p tale che $i(\tilde{\lambda})(p) = \perp$ viene scartato se arriva ad un nodo etichettato con $DROP$ (qualunque sia il suo formato al momento dell'arrivo sul nodo), tutti i pacchetti da scartare sono gestiti correttamente dal firewall prodotto.

Dato che l'interpretazione è localmente fattibile e che non ci sono NAT ripetuti, esiste un'unica serie di trasformazioni che un pacchetto p può subire lungo il percorso per essere trattato secondo $i(\tilde{\lambda})(p) = t \neq \perp$. Dato che la funzione su pacchetti è localmente fattibile, per ogni pacchetto p esiste un percorso π nel diagramma di controllo, avente le etichette adeguate alla trasformazione assegnatagli $t = i(\tilde{\lambda})$. È banale che la composizione delle trasformazioni delle ruleset prodotte dall'algoritmo, in corrispondenza delle etichette del percorso, realizzano la trasformazione t, a meno del campo *tag*. Dato che il sistema è senza NAT ripetuti, il fatto che la composizione delle trasformazioni associate ai nodi corrisponda alla trasformazione attesa implica che anche nodo per nodo, le trasformazioni siano quelle attese a meno del campo *tag*, e quindi, dato che per ipotesi le etichette sugli archi non predicano sul campo *tag*, vale che $\Delta((\mathcal{C}_k, \Sigma), p) = \pi$. Vale dunque sia $\Delta((\mathcal{C}_k, \Sigma), p) = \pi$ che $\odot(\pi, f)(p) = t$, con $(\mathcal{C}_k, f)$ semiastrazione di $(\mathcal{C}_k, \Sigma)$; e quindi vale la tesi. □

Teorema 17. *Sia $\rho = \{R_{snat}, R_{dnat}, R_{nat}, R_{fil}, R_{snat} \cdot R_{fil}, R_{dnat} \cdot R_{fil}, R_{nat} \cdot R_{fil}, R_\epsilon\}$, dove R_{fil}, R_{snat}, R_{dnat} e R_{nat} sono prodotto dall'algoritmo 6 con input R_λ. Sia $c : Q \to \rho$ l'assegnamento di ruleset ai nodi del diagramma di controllo del sistema target $\mathcal{C}_k$, generato secondo v_k. Se tutti i percorsi del sistema target k, $\Pi(\mathcal{C}_k)$, sono tali che $\ell(\ddot{\pi}) = \{SNAT, DNAT, DROP\}$, e se nessun pacchetto a cui siano applicate trasformazioni SNAT e DNAT al massimo una volta percorre dei loop nel diagramma di controllo, allora la semantica del firewall $(\mathcal{C}_k, \Sigma)$ con $\Sigma = (\rho, c)$ per lo stato s_{NEW} è identica all'interpretazione di $\tilde{\lambda}$.*

$$(\!|(\mathcal{C}_k, \Sigma)|\!)(s_{NEW}) = i(\tilde{\lambda})$$

Dimostrazione. Si può notare che nessun pacchetto subisce più di una trasformazione SNAT e DNAT diversa, qualsiasi sia l'ordine e il numero delle ruleset visitate. Pertanto nessun pacchetto nel sistema prodotto sarà scartato per colpa di un ciclo nel diagramma di controllo e dato che ogni percorso verso q_f è etichettato con $SNAT$, $DNAT$ e $DROP$, qualunque sia $t = i(\lambda)(p)$ è banale verificare che il firewall prodotto associa al pacchetto p il destino t. □