UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDO HOROWITZ

Estudo sobre a Extração de Políticas de *Firewall* e uma Proposta de Metodologia

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Luís C. Lamb
Orientador

Porto Alegre, dezembro de 2007

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Horowitz, Eduardo

Estudo sobre a Extração de Políticas de *Firewall* e uma Proposta de Metodologia / Eduardo Horowitz. – Porto Alegre: PPGC da UFRGS, 2007.

72 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Orientador: Luís C. Lamb.

1. Segurança de Rede, *Firewalls*, Políticas de *Firewall*, Descorrelacionamento, Extração de Políticas. I. Lamb, Luís C.. II. Título.

AGRADECIMENTOS

Agradeço ao Prof. Lamb, por sempre acreditar na minha capacidade.

Agradeço ao CPD da UFRGS e a todos que ajudaram, direta ou indiretamente, a tornar este trabalho possível.

Agradeço à minha família, com destaque para minha sobrinha.

Em especial, gostaria de agradecer aos meus pais, por tudo.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

BDD	Binary Decision Diagram
CLP	Constraint Logic Programming
FA	Firewall Analyzer
ER	Entity-Relationship Model
FDD	Firewall Decision Diagram
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPSec	IP Security
ITVal	IP Tables Validator
LPP	Logic Programming with Priorities
MDD	Multi-way Decision Diagrams
MDL	Model Definition Language
NAT	Network Address Translation
OBDD	Ordered Binary Decision Diagram
SPD	Security Policy Database
TOS	Type Of Service

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

RESUMO

Com o aumento das ameaças na Internet, *firewalls* tornaram-se mecanismos de defesa cada vez mais utilizados. No entanto, sua configuração é notadamente complexa, podendo resultar em erros. Vários estudos foram realizados com o intuito de resolver tais problemas, mas a grande maioria deles se concentrou em trabalhar diretamente no nível de configuração, o que possui limitações. O presente trabalho investiga maneiras de extrair políticas em mais alto nível a partir de regras de *firewall* em baixo nível, o que é mais intuitivo. A fim de extrair as políticas reais a partir de regras de *firewall*, o problema do descorrelacionamento é estudado e algoritmos anteriormente propostos para resolvê-lo são apresentados e discutidos. É apresentado, também, um tipo de grafo para a melhor visualização e análise de correlacionamento entre regras. Além disso, é pesquisado o agrupamento de regras descorrelacionadas, que tem o objetivo de elevar o nível das mesmas. São apresentados dois algoritmos para realizar o agrupamento, sendo um deles novo. A seguir, é proposta uma nova metodologia de extração de políticas de *firewall*. A primeira parte desta consiste na utilização de um novo tipo de descorrelacionamento, o descorrelacionamento hierárquico. Este é acompanhado por uma nova maneira de agrupar regras descorrelacionadas hierarquicamente, o agrupamento hierárquico. A segunda parte é uma nova modelagem de regras de *firewall* que fazem parte de *blacklist* ou *whitelist*, separando-as das demais regras na extração de políticas. Algumas maneiras de realizar esta separação também são discutidas. Por fim, são relatadas as conclusões e possibilidades de trabalhos futuros.

Palavras-chave: Segurança de Rede, *Firewalls*, Políticas de *Firewall*, Descorrelacionamento, Extração de Políticas.

A Study about Firewall Policy Extraction and a Proposal for a Methodology

ABSTRACT

As the number of threats in the Internet grows, firewalls have become a very important defense mechanism. However, configuring a firewall is not an easy task and is prone to errors. Several investigations have been made towards solving these issue. However, most of them have focused on working directly at the configuration level and have a number of limitations. This work investigates methods to extract higher level policies from low level firewall rules. Aiming at extracting real policies from firewall rules, we analyse the firewall decorrelation problem and previously proposed algoritmhs to solve it. In addition, a new type of graph is presented aiming at better visualising and analysing rules' correlation. We also investigate the merging of decorrelated rules, with the goal of defining more abstract rules. Two algorithms are then presented and a new methodology for the extraction of firewall policies is proposed. This methodology is twofold. The first part consists of the use a new type of decorrelation: the hierachical decorrelation, which is introduced along with a new way of hierarchically merging decorrelated rules. The second part is a new model for *blacklist* or *whitelist* firewall rules, separating them from the other rules in the policy extraction. We also present alternatives for accomplishing this separation. Finally, we conclude and point out directions for future work.

Keywords: Network Security, Firewalls, Firewall Policies, Decorrelation, Policy Extraction.

1 INTRODUÇÃO

A Internet, que inicialmente era uma rede pequena e controlada, utilizada apenas pelos militares e instituições acadêmicas americanas, atravessa uma explosão exponencial de crescimento. Este aumento no número de computadores interconectados continua ocorrendo de forma descontrolada, e a Internet não possui autoridade central. Somando-se a isso, o nível de anonimidade dos usuários da rede é muito grande, tornando difícil descobrir a origem de dados que trafegam pela mesma, se esta for a vontade de seu autor. Por outro lado, a sociedade real está cada vez mais dependente da virtual. A infra-estrutura básica de muitos países é controlada por computadores conectados na Internet, o comércio eletrônico está em franco crescimento e um grande número de informações privadas são armazenadas em computadores pessoais com acesso à Internet. Estes são apenas alguns exemplos de quão importante a Internet é hoje em dia.

Devido aos fatores acima, a Internet mostra-se um meio no qual é difícil se defender e eventualmente punir os agressores e, por conseguinte, um excelente alvo para atacantes e malfeitores com os mais diversos interesses. O primeiro exemplo publicamente conhecido de ataque global pela Internet foi o Morris Worm em 1988 (EICHIN; ROCHLIS, 1989), que infectou boa parte da mesma. Máquinas infectadas por *worms* ou *bots* ficam constantemente varrendo o espaço de endereços da Internet em busca de novas vítimas. O *worm* Code-Red de 2001, por exemplo, infectou, em quatorze horas, 359 mil máquinas, causando um prejuízo de mais de dois bilhões de dólares (MOORE; SHANNON; BROWN, 2002). Já o worm Blaster de 2003 infectou 100 mil máquinas e causou um prejuízo de milhões de dólares (BAILEY et al., 2005). Hoje em dia, parte significativa do tráfego da Internet é maliciosa (SYMANTEC CORPORATION, 2006) e as ações ilegais estão se tornando um negócio lucrativo para criminosos profissionais.

Face aos diversos potenciais problemas de segurança, foi necessário o surgimento de técnicas defensivas para a redução de tais ameaças. Uma das técnicas mais bem sucedidas é a utilização de *firewall*. A comprovação da importância desta defesa é evidenciada pela sua presença nas mais diversas instituições. Além disso, ela existe em boa parte dos Sistemas Operacionais, como: Linux, FreeBSD, OpenBSD, Windows 2000, Windows XP e Mac OS X. Outro aspecto que ressalta a sua importância é o fato de boa parte dos "guias de segurança" para quem usa Internet, divulgados por publicações populares de Informática, recomendá-lo como essencial não apenas para servidores como para computadores pessoais.

Um *firewall* pode ser visto, analogicamente, como uma "parede corta-fogo", com o objetivo de isolar uma rede interna que está sendo protegida das ameaças externas e impedir que possíveis problemas internos se propaguem para a rede externa. Ingham e Forest em (INGHAM; FORREST, 2005) citam cinco motivos pelos quais *firewalls* são importantes para delimitar as fronteiras de uma rede:

- Proteger contra problemas de Segurança em Sistemas Operacionais;
- Evitar acesso à informação;
- Evitar vazamento de informação sigilosa;
- Reforçar políticas;
- Ajudar em auditorias.

Tecnicamente, um *firewall* é um filtro de pacotes que fica na fronteira de uma determinada rede e é o principal responsável pela segurança de seu perímetro. Quando, por exemplo, um pacote vai de *unknown.com* para *www.instituicao.br*, é o *firewall* que determina se este pacote será aceito ou não, isto é, se passará ou não para dentro da rede da instituição em questão. Assim, o que um *firewall* faz é implementar uma política de segurança do perímetro de uma rede. Esta política de segurança consiste nas definições e diretivas que devem ser adotadas na instituição no que diz respeito à segurança. A representação da política de segurança no *firewall* é chamada de política de *firewall*. Para exemplificar, uma diretiva da política de *firewall* pode ser "Permitir tráfego da instituição A para todos os serviços de email da rede B". Outra pode ser "Proibir que máquina maliciosa acesse a rede B".

É importante notar que, apesar de hoje em dia existirem outras definições de *firewall* (como *firewall* de aplicação e *firewall* pessoal), será considerada apenas a acima citada, isto é, *firewall* de rede.

1.1 Motivação

Firewalls foram projetados para especialistas e sua configuração pode ser uma tarefa um tanto complexa (GOUDA; LIU, 2004). A maior parte dos *firewalls* pode ser configurada através de regras em baixo nível (chamadas regras de filtragem) que definem ações sobre tráfegos bastante específicos. Quando um pacote chega no *firewall*, verifica-se qual regra de filtragem casa com ele (isto é, todos os seus campos estão contidos nos campos da regra de filtragem). Esta verificação normalmente segue uma ordem, visto que pode haver conflitos entre regras. Assim, um outro "meta-campo" da regra de filtragem é sua ordem (ou prioridade).

O administrador que configura o *firewall* deve, a partir da política de *firewall* da instituição, gerar estas regras de filtragem específicas em baixo nível. Esta conversão do domínio abstrato de políticas para as regras de filtragem não é trivial, especialmente quando há muitas diretivas na política. Uma diretiva da política pode gerar múltiplas regras.

Em adição a esta dificuldade de conversão, há uma outra de atualização e verificação das regras de filtragem e de garantia de que, ao longo do tempo, elas continuem representando as políticas que as definiram. No caso de *firewalls* legados em ambientes corporativos com vários anos de uso, quando há uma troca na sua administração, o novo administrador muitas vezes se depara com inúmeras regras em baixo nível. Não tendo o conhecimento completo sobre a configuração do *firewall*, a chance de erros serem introduzidos é bastante grande.

Várias destas dificuldades haviam sido mostradas por Chapman em (CHAPMAN, 1992) e quantitativamente comprovadas por Wool em (WOOL, 2004).

A partir de uma revisão da literatura, percebe-se que:

- O maior número de pesquisas sobre especificação de *firewall* é dirigida à tradução de políticas de *firewall* em alto nível para regras em baixo nível, como relatado em (BARTAL et al., 2004) e (GUTTMAN, 1997);

- A parte de análise de regras de filtragem se concentra em apontar possíveis problemas no *firewall*, como redundância e conflito (AL-SHAER et al., 2005) e (CHOMSIRI; PORNAVALAI, 2006).

No entanto, a fim de contribuir na solução do problema de atualização e verificação de regras, seria bastante interessante, do ponto de vista do administrador do *firewall* se houvesse métodos que convertessem regras de filtragem em baixo nível para políticas de *firewall* em um nível mais alto. Isto permitiria que fosse verificado se o *firewall* está funcionando de acordo com sua especificação original. Além disso, esta tradução eliminaria automaticamente qualquer redundância ou conflito, mostrando ao administrador a política que o *firewall* está aplicando.

O presente trabalho, ao invés de especular sobre erros no *firewall*, como boa parte da literatura de análise de conflitos, incluindo os acima citados, concentra-se exclusivamente em fatos.

1.2 Objetivos

O principal objetivo deste trabalho é estudar a extração de políticas de *firewall* em mais alto nível a partir de regras de *firewall* em baixo nível, ou seja, resolver o seguinte problema: "dado um conjunto de regras de filtragem R e um método de resolver conflitos (como priorização), extrair a política real que o *firewall* em questão implementa".

Para isto, o estudo sobre o descorrelacionamento de regras de *firewall*, isto é, como transformá-las em regras disjuntas, em que a ordem não é importante, mostrou-se fundamental. Desta forma, parte significativa deste trabalho se concentra no estudo do problema do descorrelacionamento. Além disso, tem-se como objetivo investigar os trabalhos relacionados a extração de políticas e propor novas soluções e abordagens para este problema.

1.3 Contribuições

As principais contribuições deste trabalho são:

- Um estudo sobre a extração de políticas de *firewall*, enfocando:
  - O problema do descorrelacionamento de regras de *firewall*, cuja discussão é pouco encontrada na literatura. Para resolvê-lo, foi apresentado um novo algoritmo baseado em outros já publicados;
  - A apresentação de uma estrutura para visualizar correlacionamentos, denominada grafo de correlacionamento;
  - Uma análise sobre o agrupamento de regras, com a elaboração de um algoritmo original para tal.

- A proposta de uma nova metodologia para extração de políticas de *firewall*, composta por:

- Um novo algoritmo hierárquico para realizar tanto extração de políticas quanto descorrelacionamento: o descorrelacionamento hierárquico;
- Um novo modelo de extração de políticas, que se baseia na diferenciação de regras relacionadas a *blacklists* e *whitelists*.

1.4 Organização

No capítulo 2 é realizada uma revisão da literatura a fim de embasar o desenvolvimento deste trabalho e são fornecidas as definições mais importantes utilizadas. No capítulo 3 é investigado o problema do descorrelacionamento, sua definição, sua complexidade, suas aplicações e algoritmos que o resolvem. No capítulo 4 são relatados métodos de agrupamento de regras descorrelacionadas, a fim de elevar o nível das mesmas tornando-as mais compreensíveis. No capítulo 5 é proposta uma nova metodologia de extração de políticas de *firewall*, composta pelo descorrelacionamento hierárquico e pela modelagem de regras de *blacklist* e *whitelist*. Por fim, no capítulo 6, são relatadas as considerações finais e as possibilidades de trabalhos futuros.

2 FUNDAMENTAÇÃO DO ESTUDO

Neste capítulo, é realizada uma revisão da literatura sobre *firewalls*, categorizando-se a pesquisa já realizada sobre este assunto. Esta revisão serviu de embasamento para o assunto deste trabalho. Além disso, são definidos os principais conceitos sobre *firewalls* utilizados no restante deste trabalho.

2.1 Revisão Bibliográfica

Muitas pesquisas foram realizadas a fim de tentar resolver os problemas mencionados anteriormente. Vários dos artigos originados nestas pesquisas são analisados brevemente abaixo. Para fins de organização, os artigos foram divididos em três áreas diferentes:

- Análise de Consistência de Regras de Filtragem;
- Especificação de Políticas de *Firewall*;
- Auditoria e Testes de *Firewalls*.

2.1.1 Análise de Consistência de Regras de Filtragem

A análise de consistência verifica se as regras de filtragem estão consistentes, principalmente no que diz respeito a conflitos e à redundância. Nas regras da Tabela 2.1 (onde "*" representa qualquer informação válida no dado domínio), por exemplo, assumindo-se *first-match wins* (se um pacote casar com duas ou mais regras, a que foi listada primeiro terá sua ação executada), as regras 1 e 4 são conflitantes, já que um pacote casaria com as duas, mas suas ações são diferentes. Já as regras 2 e 3 são redundantes, sendo que a regra 3 poderia ser descartada, já que a regra 2 é mais genérica e nenhum pacote casaria com a regra 3.

Tabela 2.1: Exemplo de regras de filtragem com conflito (1 e 4) e redundância (2 e 3)

Priority	Protocol	Source Address	Source Port	Destination Address	Destination Port	Action
1	TCP	malicious	*	servidor	80	Accept
2	TCP	cliente	*	servidor	*	Accept
3	TCP	cliente	*	servidor	53	Accept
4	TCP	malicious	*	servidor	80	Drop

Abaixo, alguns trabalhos significativos nesta área são comentados.

Hari, Suri, Parulkar em (HARI; SURI; PARULKAR, 2000) introduziram o estudo sobre conflitos em regras de filtragem. Nele, são apresentados algoritmos para detecção e resolução dos mesmos. A estrutura usada para representação é a árvore *trie*, para a qual é apresentado um algoritmo de detecção rápida de conflitos no caso de regras bidimensionais (isto é, que contém apenas endereços de origem e de destino). No caso mais geral de regras com cinco campos (endereços de origem e destino, portas de origem e destino e protocolo), a eficiência não é tão grande.

Talvez a mais influente pesquisa sobre análise de conflitos tem origem em (AL-SHAER; HAMED, 2003). Neste artigo é utilizada a *Teoria de Conjuntos* para formalizar as regras de filtragem, já que cada campo da mesma pode ser um intervalo e a representação como tupla é natural. Assim, primeiramente são definidas as relações entre regras arbitrárias R_x e R_y, podendo as mesmas serem: completamente disjuntas, casadas exatamente (*exactly matched*), casadas inclusivamente, casadas parcialmente ou correlacionadas. Além disso, o conjunto de regras é colocado em uma "árvore de política", onde cada nó é um campo que pode aparecer na regra (sendo as ações os nós terminais) e cada aresta representa uma possibilidade de valor utilizado. Assim, cada caminho da raiz até uma folha representa uma regra. Partindo desta base, é criada uma taxonomia de possíveis anomalias em regras de filtragem, quando analisadas duas a duas. As anomalias são:

- Anomalia de Sombreamento (*Shadowing anomaly*) - quando uma regra R_x com maior prioridade casa com todos os pacotes que casariam com uma regra R_y e suas ações são diferentes. Desta forma, R_y nunca será atingida;

- Anomalia de Correlação (*Correlation anomaly*) - quando as regras R_x e R_y possuem ações diferentes e no mínimo um pacote casaria com ambas. Neste caso a "anomalia" pode ser um artifício de configuração usado pelo administrador do *firewall*;

- Anomalia de Generalização (*Generalization anomaly*) - quando uma regra R_y é uma generalização (isto é, cada campo seu contém o campo da regra generalizada) de uma regra R_x que a precede. Esta anomalia também é, na maioria das vezes, utilizada pelos administradores para apontar algum tráfego de exceção (na regra R_x) em relação a regra geral (R_y);

- Anomalia de Redundância (*Redundancy anomaly*) - quando uma regra contém (ou possui igualdade em) todos os campos de outra e suas ações são idênticas. A regra redundante não acrescenta nenhuma informação, já que nunca terá um *match*. Redundâncias não alteram a política representada pelas regras de filtragem, mas poluem arquivos de configuração e ocupam espaço na memória do *firewall*;

- Anomalia de Irrelevância (*Irrelevance anomaly*) - apresentada em (AL-SHAER; HAMED, 2004), ela acontece quando a regra em questão não casa com nenhum pacote que passa pelo *firewall* onde ela se encontra.

O algoritmo de detecção de inconsistências refere-se à comparação de regras, de uma a uma, para detectar se elas possuem algum tipo de anomalia. Uma ferramenta, denominada *Policy Advisor*, foi implementada para detectar as anomalias e permitir que elas sejam testadas na inserção e remoção de regras. Em (AL-SHAER; HAMED, 2004) e (AL-SHAER et al., 2005) é apresentada uma classificação parecida de anomalias para

múltiplos *firewalls* e é dito que o *Policy Advisor* foi atualizado para considerar tais casos. Um grave defeito dos trabalhos de Shaer et al. é a consideração de apenas duas regras por vez. Além disso, em (AL-SHAER; HAMED, 2003) é descrita uma técnica para traduzir a política atual de uma árvore de política (*policy tree*) para um subconjunto da linguagem natural (*filtering policy translator*), que facilitaria a compreensão da mesma. No entanto, a técnica utilizada não parece satisfatória, necessitando de análises de muitas permutações da árvore. Provavelmente por isto, os trabalhos subsequentes dos autores não fazem menção à técnica.

As limitações das técnicas apresentadas em trabalhos de Shaer et al., isto é, o fato de apenas duas regras serem analisadas por vez, são exploradas e corrigidas em (CHOMSIRI; PORNAVALAI, 2006). Neste artigo, é utilizada a *Álgebra Relacional* para modelar as regras. Assim, apesar de as mesmas anomalias descritas acima serem utilizadas (Sombreamento, Correlação, Generalização e Redundância), elas são redefinidas para serem usadas através de Álgebra Relacional. Isto permite que sejam verificadas múltiplas regras a cada vez e não apenas duas. Os autores afirmam ter implementado uma aplicação baseada em suas técnicas de análise de conflitos em regras de filtragem. Um possível problema desta técnica pode ser na representação de intervalos de endereços grandes, visto que, para o intervalo 30.0.0.0/30, foi necessário, em um dado exemplo, expandi-lo para seus endereços individuais, formando uma tabela com 30.0.0.0, 30.0.0.1, 30.0.0.2 e 30.0.0.3. No caso de 30.0.0.0/16 (com 65536 endereços), esta expansão provavelmente se tornaria proibitiva tanto em espaço quanto em tempo de processamento.

Em Eppstein; Muthukrishnan (2001), são apresentados algoritmos para resolver dois problemas:

- Problema da Classificação de Pacote (*Packet Classification Problem*) - Dada uma base de dados F de regras de filtragem que podem ser pré-processadas, cada *query* é um pacote P e o objetivo é classificá-la, isto é, determinar a regra de maior prioridade que casa com P;

- Problema da Detecção de Conflitos de Filtro (*Filter Conflict Detection Problem*) - Dada uma base de dados de regras de filtragem F, determinar se existe algum pacote P tal que quaisquer duas regras de filtragem que se aplicam a P possuem ações conflitantes. Problemas relacionados são o de listar todas as regiões onde Ps conflitantes se encontram ou listar todos os pares de regras conflitantes.

A abordagem utilizada é resolver o problema abstrato que representa cada um dos problemas acima, que é naturalmente formulado como um problema de estruturas de dados geométricas. O problema resolvido trata de regras com duas dimensões (origem e destino), por terem sido as mais usadas na época do artigo. Uma interessante observação realizada é que se dois intervalos de IPs possuem uma interseção, um está completamente contido no outro. Através da utilização de algoritmos sobre estruturas relativamente avançadas (B-trees, kD-trees, priority queues, etc.), consegue-se resolver o problema de detecção de conflito de regras de filtragem em tempo $O(n^{3/2})$, com espaço $O(n)$. Isto representou um avanço em relação aos algoritmos usados anteriormente, com tempo $O(n^2 * logn)$. O maior problema do artigo é sua limitação ao caso específico de duas dimensões.

O trabalho de Zhang, Zhang e Wang em (ZHANG; ZHANG; WANG, 2005) também modela regras de *firewall* através de *Geometria*. O artigo usa a classificação de anomalias

feita por Shaer et al (AL-SHAER et al., 2005) e possui o objetivo de diminuir o número de regras, que são representadas como retângulos, visto que tratam apenas de endereço origem e destino (2D). A diminuição se dá através de um algoritmo para agrupar retângulos semelhantes, já que as regras com "accept" e "block" são representadas por retângulos de cores diferentes. Além disso, são fornecidos procedimentos para aumentar a eficiência do *firewall* através da reordenamento de regras ou inserção de regras de bloqueio (para que não seja necessário percorrer até o fim da lista de regras). Os procedimentos requerem a estatística de uso de cada regra. O artigo também mostra como verificar a política real de um *firewall* depois de plotar todas as suas regras, apesar de este não ser seu tema central. Em geral, os exemplos são fornecidos para apenas 13 regras e a visualização provavelmente seria mais complicada com algumas milhares de regras. O artigo também não mostra quais estruturas devem ser usadas para representar os retângulos e qual o espaço necessário para tal.

Grande parte dos estudos sobre análise de consistência, como (AL-SHAER; HAMED, 2003) e (ZHANG; ZHANG; WANG, 2005) revisados acima, se concentra em melhorar e trabalhar sobre as regras já em baixo nível. Apesar de interessantes, os resultados obtidos desta forma não permitem que se saiba se a política de *firewall* foi implementada corretamente ou se foi planejada corretamente em primeiro lugar. O que se consegue, neste caso, é a otimização ou detecção de inconsistências nas regras de baixo nível. Devido ao fato de alguns *firewalls* possuírem milhares de regras, trabalhando neste nível baixo, muitas vezes é difícil associar a política de *firewall* com o que estava errado nas regras.

2.1.2 Especificação de Políticas de *Firewall*

Especificação de políticas de *firewall* diz respeito ao estudo de como fazer para representar, de maneira correta e eficiente, as políticas de *firewall* que normalmente são pensadas e especificadas em linguagem natural. Além disso, esta área abrange técnicas de como converter o formalismo utilizado na representação das políticas de alto nível para regras de filtragem em baixo nível. Alguns artigos relacionados a esta área são brevemente revisados a seguir.

Em (GUTTMAN, 1997), foi abordado o problema da localização (*localization problem*), isto é, o problema de implementar uma política de *firewall* global de uma instituição através da composição das políticas dos *firewalls* distribuídos na instituição. Para tentar resolver o problema, é utilizada uma especificação das características da rede e da política global por meio de uma notação com sintaxe parecida com LISP (uma linguagem de programação). A modelagem da rede é feita por um grafo bipartido de *firewalls* e áreas da rede as quais estes se conectam. O funcionamento dos filtros é formalizado usando alguns conceitos de *Teoria dos Conjuntos* e de *Funções*. São descritos brevemente um algoritmo para verificação da política implementada em determinado *firewall* (percorrendo-se o grafo em busca por profundidade) e um algoritmo para geração de política de *firewall*. O autor deixa claro que o algoritmo de geração de políticas para múltiplos *firewalls* é bastante limitado. Em geral, este estudo é bastante importante por apresentar e abordar um problema que naquela época já era desafiador. O autor previu, corretamente, que *firewalls* seriam importantes no futuro.

Bartal et al descrevem em (BARTAL et al., 1999) e (BARTAL et al., 2004) o projeto e a implementação do Firmato, um *toolkit* para gerenciamento de *firewall* e estabelecem

os seguintes objetivos:

- Separar o design da política de *firewall* das especificidades do produto de *firewall* em questão - o administrador não precisa se preocupar com sintaxe, ordem de regras, complexidade das regras, etc.;
- Separar o design da política de *firewall* da topologia de rede - facilita na modularização e reaproveitamento de políticas, além de permitir flexibilidade;
- Gerar os arquivos de configuração do *firewall* automaticamente, a partir de uma política de *firewall* em alto nível, para múltiplos *firewalls*;
- Permitir um alto nível de depuração dos arquivos de configuração.

A idéia do Firmato é claramente espelhada no artigo descrito acima (GUTTMAN, 1997), visto que o Firmato também ataca o problema da localização e, para isso, utiliza a mesma modelagem de rede introduzida anteriormente: um grafo bipartido.

O Firmato é formado pelos seguintes componentes:

- *Entity-Relationship Model* (ER) - fornece *framework* para representar a política de *firewall* (independente do tipo de *firewall*) e da topologia da rede. Isto é conseguido através da utilização de "papéis" (*roles*), que são usados para definir as permissões da rede;
- *Model Definition Language* (MDL) - é usada para definir uma instância do modelo ER;
- *Model Compiler* - traduz uma instância do modelo para arquivos de configuração específicos de *firewalls*. O compilador gera uma base central de regras lógicas, distribui estas regras para vários dispositivos e transforma as regras lógicas em regras de filtragem;
- Rule Illustrator - transforma os arquivos de configuração específicos de determinado *firewall* em uma representação gráfica da política atual na topologia atual.

São apresentados também cada componente do Firmato e diversas decisões tomadas quanto a seu desenvolvimento, além de algoritmos utilizados pelo *toolkit*. No final, os autores afirmam que a separação entre a política de segurança e a topologia da rede, o nível maior de abstração e o uso e geração de arquivos texto (ao invés de usar uma GUI - Graphical User Interface) foram atributos bem-sucedidos. No entanto, a parte de ilustrações de regras, segundo os autores, ainda possui problemas. Outras limitações mais graves foram o fato de NAT não ser suportado devido a complicações que isto geraria no modelo, o fato de o usuário não poder controlar a ordem das regras de filtragem (geradas automaticamente) e o fato de "block" não ser utilizado e sim um outro artifício nada intuitivo (chamado *CLOSED*). Além disso, as regras de filtragem geradas podem conter redundância.

Em (GOUDA; LIU, 2007) são identificados três tipos de problemas em *firewalls*:

1. Problema da Consistência (*Consistency problem*): quando há conflito entre regras e a ordem das mesmas não gera o efeito esperado por quem configurou o *firewall*;

2. Problema da Completude (*Completeness problem*): devido a dificuldade de considerar todos os tipos de pacotes, a última regra normalmente tem como predicado uma tautologia (casa com qualquer pacote). Isto gera uma generalização que normalmente não contempla os interesses de quem controla o *firewall*;

3. Problema da Compactação (*Compactness problem*): quando há muitas regras redundantes (regras que poderiam ser removidas sem que o funcionamento do *firewall* mude), o *firewall* ocupa mais memória, que não será utilizada na prática.

Para solucionar estas deficiências, é proposto o "structured *firewall* design", que consiste em:

- Especificar um *firewall* de maneira "formal" usando um FDD (*Firewall Decision Diagram*) (GOUDA; LIU, 2004) - uma árvore (baseada em BDD - Binary Decision Diagram) com propriedades específicas que garante que o problema da Consistência e o problema da Completude não aconteçam;

- Gerar as regras do *firewall* a partir do FDD, que são compactadas usando algoritmos para compactação de FDD e resolvendo o problema da Compactação.

O artigo também mostra a definição formal de um FDD, como reduzir um FDD, como marcar um FDD (para compactação), como gerar as regras de *firewall* a partir de um FDD, como compactar um *firewall* e como "simplificar" um *firewall* (gerar várias regras simples a partir de regras mais complexas). Como limitação, não é mencionada a possibilidade de especificar uma política de *firewall* global para múltiplos *firewalls*.

2.1.3 Auditoria e Testes de *Firewalls*

Auditoria e testes de *firewalls* está relacionada à descoberta de problemas no que diz respeito ao design e/ou a implementação das políticas de *firewall*. Talvez o recurso mais comum utilizado para tal fim é o de *queries* (perguntas), que permitem que perguntas como: "Para onde são permitidos pacotes ICMP?" ou "O que o *host* malicioso pode enviar para o servidor Web?" sejam respondidas em relação a uma dada política de *firewall*.

Os artigos analisados abaixo dizem respeito a testes realizados de maneira *offline*, sem enviar pacotes ou alterar o estado da rede onde os *firewalls* estão implementados.

Hazelhurst em (HAZELHURST, 1999) foi um dos primeiros autores a possibilitar que auditorias e testes fossem realizados em *firewalls*. Para tal, foi utilizada uma modelagem através de OBDD (Ordered Binary Decision Diagram) reduzido, que é uma variante de BDD, sendo este uma representação de uma expressão booleana. Regras de filtragem podem ser representadas sem muita dificuldade como sentenças lógicas, sendo esta modelagem intuitiva. Além disso, BDDs podem representar expressões lógicas de maneira compacta e permitirem que a operação de verificação de equivalência de dois BDDs não seja custosa. Inteiros são transformados em binário (vetores de bits) a fim de serem representados. Apesar de poderosos, os BDDs também apresentam problemas, como o fato de nem sempre serem eficientes, visto que conseguem representar o problema NP-completo SAT (satisfazibilidade booleana) com facilidade. O artigo mostra como se converter regras de filtragem para expressões booleanas e como unir estas expressões para gerar uma expressão booleana usada para representar todo o conjunto de regras. Um protótipo foi implementado sobre o sistema Voss (usado para verificação de hardware), que possui

uma linguagem funcional chamada FL. A FL foi utilizada para a análise das regras, especialmente permitindo que sejam realizadas *queries*, que mudanças nas regras sejam analisadas e que algumas validações automáticas sejam realizadas. O autor cita a análise de redundância nas regras e menciona que outras validações são possíveis, mas não descreve como elas são implementadas. O autor também relata o resultado de algumas experiências com o protótipo, mas afirma outras são necessárias em ambientes reais. A modelagem através de BDDs e a possibilidade de se fazer perguntas sobre o tráfego do *firewall* foram inovadoras.

Eronen e Zitting (2001) introduzem uma ferramenta baseada em um sistema especialista que permite que sejam feitas *queries* sobre o *firewall*. No caso do artigo em questão, a base de conhecimento são as regras de filtragem do *firewall* e regras de produção criadas pelo especialista para analisar o *firewall*. Já o motor de inferência é uma ferramenta chamada Eclipse, que é um ambiente para programação lógica em CLP (*Constraint Logic Programming*). Na programação lógica, ao invés de se darem os passos para resolver um problema, são fornecidos os fatos lógicos e dependências que descrevem uma solução e um mecanismo de inferência resolve o problema. A parte de restrição (*constraint*), possibilita que se restrinjam os valores das variáveis da solução. Por exemplo, pode-se especificar que portas são valores entre 0 e 65535.

O estudo mostra como a representação do espaço de pacotes é realizada na base de conhecimento, assim como a representação das regras de filtragem como restrições sobre o espaço de pacotes e a representação da topologia da rede. A ferramenta permite que sejam verificadas propriedades da rede, problemas de configuração e propriedades de lista de acesso. Uma das principais vantagens da ferramenta, que usa programação lógica, é a possibilidade e facilidade com a qual são definidos novos predicados. No entanto, esta flexibilidade tem a desvantagem de que a detecção de problemas pela ferramenta é tão boa quanto o conhecimento dos especialistas que a estão utilizando. O artigo também não explica como é feita a conversão de regras de filtragem de equipamentos da Cisco, que é o único tipo suportado, para regras lógicas.

Uma abordagem mais recente que também utiliza Programação Lógica é a de Bandara et al. (2006). Nela, é utilizado um *framework* baseado em argumentação para LPP (*Logic Programming with Priorities*), através da ferramenta GORGIAS. Através da modelagem das regras de filtragem para este *framework*, é possível especificar e usar abstrações de alto nível para representar entidades de redes, além de especificar a ordem relativa entre as regras de filtragem. As regras de filtragem são traduzidas para a lógica através da utilização de predicados apresentados no artigo. Além de permitir análises através de *queries*, os autores do artigo mostraram como analisar, utilizando a programação lógica, os erros classificados por Shaer et al. (AL-SHAER et al., 2005): Sombreamento, Generalização, Correlação e Redundância. O trabalho mostra que a LPP captura de maneira natural a estrutura ordenada das regras de filtragem e permite sua análise de maneira flexível, assim como em (ERONEN; ZITTING, 2001). No entanto, (BANDARA et al., 2006) já apresenta análises que devem ser realizadas, e não exigem a participação direta de um especialista. Um potencial problema do artigo é que o ambiente da ferramenta GORGIAS é provadamente P-Completo, que é a classe dos problemas polinomiais mais difíceis (DIMOPOULOS; NEBEL; TONI, 2002). Além disso, a escalabilidade e o uso em um grande número de regras mais complexas não foram testados. Um dos trabalhos futuros propostos é a expansão desta técnica para *firewalls* distribuídos.

Em (MAYER; WOOL; ZISKIND, 2006), são apresentados a evolução, as características e o funcionamento de uma ferramenta de análise de *firewall* que foi comercializada e agora é denominada Algosec FA (*Firewall Analyzer*) (ALGOSEC, 2007). A ferramenta foi projetada para possuir as seguintes características:

- Usar de um nível adequado de abstração;
- Ser compreensiva, analisando completamente e não parcial ou estatisticamente;
- Não precisar modificar as configurações dos *firewalls*;
- Ser passiva, isto é, fazer testes e auditoria *offline*.

A primeira geração da ferramenta de análise *offline* de *firewall* foi o protótipo denominado Fang (*Firewall ANalyzer enGine*) (MAYER; WOOL; ZISKIND, 2000), criada na Bell Labs. A partir do retorno fornecido pelos usuários, a ferramenta foi aprimorada e denominada *Firewall Analyzer* (antes sendo chamada de Lumeta *Firewall Analyzer* em (WOOL, 2001)). A parte central da ferramenta é a *query engine*, que é responsável pela maior parte de computações feitas durante a análise. Esta obtém as partes relevantes de arquivos de configuração e as representa internamente, representando tanto as regras de filtragem como a topologia da rede. Dada uma *query*, a *query engine* simula o comportamento de vários *firewalls*, levando em consideração a topologia da rede, e computa quais partes da *query* original conseguiriam chegar da origem ao destino. Esta poderosa *query engine* consegue simular ataques de *spoofing* (no qual um atacante tenta esconder sua origem dizendo possuir outro endereço IP de origem), já que é possível especificar onde os pacotes serão injetados na rede. Além disso, também é possível simular as regras de filtragem que fazem NAT. Segundo os autores, a mais importante lição aprendida através do Fang foi que os usuários não sabem o que perguntar sobre o *firewall*. Assim, a grande mudança do FA foi que a interação humana foi limitada e a análise se tornou automatizada. Somando-se a isso, a saída do programa evoluiu, sendo feita através de uma série de páginas Web bastante detalhadas e fáceis de navegar. Atualmente são suportados vários tipos de *firewall* diferentes (PIX, FWSM, Checkpoint Firewall-1, Checkpoint Provider-1, Juniper-Netscreen e Cisco IOS Access List (ACL) segundo (ALGOSEC, 2007)).

A modelagem realizada pelo FA é baseada em grafos, sendo a topologia da rede representada através de um grafo bipartido específico apresentado inicialmente em (GUTTMAN, 1997). A análise é feita por simulação da seguinte maneira: a ferramenta simula a injeção de tráfego em determinado ponto da rede, colocando-o no ponto correspondente do grafo e propagando este tráfego para todas as direções possíveis. Se o tráfego passar por um *firewall*, as regras do mesmo são aplicadas (através de uma representação de lista encadeada) e o tráfego que passa continua sendo propagado por *flood filling*. Quando nenhum tráfego novo passar para nenhuma parte do grafo, o algoritmo pára. Pode-se, então, verificar qual tráfego chegou no destino. No algoritmo, cada regra ou *query* é vista como um hipercubo no espaço com quatro dimensões (IP de origem, porta de origem, IP de destino, porta de destino) e a verificação de que há *match* ou não em uma regra é realizada através da operação de intersecção entre hipercubos. Outro interessante algoritmo apresentado é como extrair a topologia da rede automaticamente através das tabelas de roteamento dos *firewalls*, o que retira outra responsabilidade do administrador ou analista de segurança.

Em geral, o FA representa o estado da arte no que diz respeito a auditoria *offline* de *firewalls*. No entanto, ele também apresenta alguns problemas. Primeiramente, ele funciona através de simulação e, desta forma, não analisa diretamente os problemas de regras (conflitos, redundância, etc.). Além disso, o algoritmo de grafo apresentado possui, no pior caso, complexidade exponencial em relação ao tamanho do grafo. Os autores, no entanto, afirmam que os grafos do mundo real são bastante esparsos e que o pior caso dificilmente será atingido, mas isto não retira a limitação teórica do algoritmo. Além isso, a linguagem interna utilizada é a do *Lucent VPN Brick Firewall* e não uma linguagem abstrata e formal. Desta maneira, boa parte do código da ferramenta é utilizado para converter a sintaxe e semântica dos múltiplos produtos para este e não há garantia formal de correção. O artigo menciona que os *firewalls* são considerados como possuindo semântica de *first-match wins*, o que restringe a flexibilidade de modelagem que a ferramenta possui (não podendo modelar o *firewall* pf, por exemplo, pois este utiliza *last-match wins* - se um pacote casar com duas ou mais regras, a que foi listada por último terá sua ação executada).

Em (LIU; GOUDA, 2004), a idéia, originalmente de Engenharia de Software, de N times de programadores resolverem o mesmo problema e depois comparar e selecionar o melhor programa é aplicada a especificação de políticas de *firewalls*, com o intuito de diminuir os erros de especificação. Boa parte do artigo, no entanto, se refere a como transformar as regras de filtragem de baixo nível, criadas pelas equipes, em uma abstração que permita comparações de maneira mais simples. Isto é feito utilizando-se o FDD (*Firewall Decision Diagram*) (GOUDA; LIU, 2004), uma árvore com características específicas que garantem que um *firewall* especificado por ela tenha as propriedades de Consistência (sem contradições), Completude (cobrindo todo o domínio de pacotes) e Compactação (sem redundância), como foi dito na seção 2.1.2. O algoritmo que faz esta transformação é denominado de construção. Além disso, são mostrados algoritmos para se moldar dois FDDs de maneira que fiquem parecidos a fim de que, posteriormente, sejam comparados. A complexidade de tempo e espaço no pior caso do algoritmo de construção é de $O(n^d)$, onde d é o número de campos que compõe uma regra de filtragem (normalmente 4 ou 5).

Em (LIU et al., 2004) é apresentada a SFQL (*Structured Firewall Query Language*), uma linguagem que permite o uso de *queries* do tipo "select...from...where", tentando imitar a sintaxe da SQL (*Structured Query Language*). A linguagem necessita a conversão do *firewall* para uma estrutura chamada FDT (*Firewall Decision Tree*), parecido com o FDD mencionado acima. Os resultados experimentais apresentados mostram que a estrutura de árvore permite uma velocidade de consultas bastante rápida. Nenhum dos artigos de Gouda e Liu tratam de múltiplos *firewalls*.

Em (MARMORSTEIN; KEARNS, 2005a) é apresentada a ferramenta ITVal (*IP Tables Validator*), uma ferramenta em software livre criada especialmente para a análise de *firewalls* projetados usando *iptables/Netfilter*. A ferramenta utiliza uma linguagem simples que permite que *queries* sejam realizadas sobre a política do *firewall*. O processamento todo se dá através da utilização de MDDs (*Multi-way Decision Diagrams*). Em (MARMORSTEIN; KEARNS, 2005b), a ferramenta ITVal é estendida para suportar NAT e análise de múltiplos *firewalls* conectados.

2.2 Definições sobre *Firewalls*

Nesta seção, são definidos os principais conceitos relativos a *firewall* usados no restante deste trabalho. As definições se baseiam em (LIU; GOUDA, 2004), (CONDELL; SANCHEZ, 2004) e (QIAN; HINRICHS; NAHRSTEDT, 2001).

- Um **pacote** P é um ponto em um espaço d-dimensional. Ele pode ser representado pela conjunção

  $$P_1 \wedge P_2 \wedge \ldots \wedge P_d$$

  onde cada P_i é um inteiro não negativo e representa o valor da i-ésima dimensão do pacote em questão. O domínio de todos os pacotes é denotado como $D(\textit{Pacotes}) = D(P_1) \times D(P_2) \times \ldots \times D(P_d)$.

- Uma **regra de filtragem**, **regra de baixo nível** ou simplesmente **regra** é representada pela seguinte implicação lógica:

  $$\textbf{Condições} \rightarrow \textbf{Ação}$$

  Condições é uma tupla d-dimensional cujos elementos são conjuntos. Neste trabalho, inicialmente, as regras de filtragem são representada como simples tuplas de intervalos. Um intervalo é representado por $[a, b]$, onde a e b são inteiros não negativos e $a < b$. O intervalo é equivalente ao conjunto com todos os valores entre valores entre a e b. Se $a = b$, o valor é representado por $[a]$.

  Após o uso de descorrelacionamento (capítulo 3), os elementos das tuplas serão vistos como conjuntos de intervalos. Quando houver mais de um intervalo em uma determinada dimensão da tupla, esta será representada por $\{a_1, a_2, \ldots, a_n\}$, onde cada a_i é um intervalo. Caso haja apenas um intervalo na dimensão, ela será representada por $[a]$.

  Para uma dada regra R, uma condição é representada da seguinte maneira

  $$\textbf{Condições} = R.c_1 \wedge R.c_2 \wedge \ldots \wedge R.c_d$$

  onde cada $R.c_i$ denota o valor da i-ésima dimensão da tupla **Condições**. Cada dimensão $R.c_i$ de uma regra é chamada de **campo** ou **seletor** e cada campo tem um nome representado por $nome(c_i)$. O domínio da tupla **Condições** é considerado o mesmo de *Pacotes*, sendo $\forall_{1 \leq i \leq d}, D(c_i) = D(P_i)$. O domínio $D(c_i)$ de um campo também pode ser representado pelo domínio do nome desse campo, isto é $D(c_i) = D(nome(c_i))$. Quando o valor de um campo é igual a seu domínio ($R.c_i = D(c_i)$), ao invés de representá-lo por seu valor, ele é representado pelo símbolo *****. Quando for importante mostrar o nome de campo junto com seu valor, usa-se a seguinte notação neste trabalho:

  $$\textbf{Condições} = nome(c_1) = R.c_1 \wedge nome(c_2) = R.c_2 \wedge \ldots \wedge nome(c_d) = R.c_d$$

  Neste caso, se um campo não aparece em uma regra, seu valor é *****. Por exemplo, quando a regra R está no espaço tridimensional com $nome(c_1) = a$, $nome(c_2) = b$ e $nome(c_3) = c$, e se sua condição é representada por $a = [1, 10]$, esta é equivalente a $a = [1, 10] \wedge b = * \wedge c = *$.

  Ação é a ação ou decisão tomada pela dada regra quando suas condições são satisfeitas. A ação de uma regra R é denotada $R.$**ação**. Neste trabalho, o domínio $D($**Ação**$)$ é {*aceitar*, *bloquear*} ou {*accept*, *block*}.

- Diz-se que um pacote P **casa** (ou ***matches***) com a regra R, se e somente se, $P \in R$. Formalmente

$$P \in R \iff \forall_{1 \leq i \leq d}, P_i \in R.c_i$$

Um exemplo de um pacote que casa com uma regra de filtragem é mostrado na Figura 2.1

(a) $[7] \wedge [7]$
(b) $[1, 10] \wedge [1, 10] \rightarrow$ **accept**

Figura 2.1: Exemplo de pacote (a) que casa com regra de filtragem (b).

- O **conjunto de regras** ou **configuração** de um *firewall* é uma sequência ordenada de regras de filtragem, sendo a ordem baseada na prioridade associada a cada regra. A necessidade de prioridade existe, pois, como visto na seção 2.1.1, pode acontecer de mais de uma regra casar com um pacote e estas regras podem ter ações conflitantes. Desse modo, um ***firewall*** é uma função $f : D(\textit{Pacotes}) \longmapsto D(\textbf{\textit{Ação}})$ baseado em um conjunto de regras de filtragem de tal forma que, dado um pacote P, o *firewall* o mapeia para a ação da regra de maior prioridade que casa com ele.

- Uma **política de *firewall***, neste trabalho, é definida como uma representação em mais alto nível de uma ou mais regras de filtragem que se encontram em um conjunto de regras ordenadas por prioridade. Desta forma, qualquer representação que mostre a semântica das regras de maneira mais real (por exemplo, sem a necessidade de priorização) já é considerada política. Assim, define-se **extração de políticas de *firewall*** como o ato de se transformar regras de filtragem em baixo nível em uma representação de mais alto nível.

- Um conjunto de regras A é **semanticamente equivalente** a um conjunto de regras B se a função do *firewall* baseada em A e a função do *firewall* baseada em B são iguais.

- Duas regras A e B são ditas **correlacionadas** ou **sobrepostas** se elas possuem, no mínimo, um elemento em comum em todos seus campos. Formalmente,

$$\forall_{1 \leq k \leq d} \qquad A.c_k \cap B.c_k \neq \emptyset$$

Nos algoritmos abaixo, o correlacionamento das regras A e B será representado por $A \cap B \neq \emptyset$

- Duas regras A e B são ditas **descorrelacionadas** ou **disjuntas** se, em no mínimo um campo, elas não possuam nenhum elemento em comum. Formalmente

$$\exists k, (1 \leq k \leq d) \qquad A.c_k \cap B.c_k = \emptyset$$

- Um conjunto de regras é chamado de **conjunto de regras descorrelacionado** se todas as suas regras são disjuntas. Neste caso, este não precisa ser ordenando por prioridade.

3 DESCORRELACIONAMENTO DE REGRAS

O **problema do descorrelacionamento** pode ser definido como: "dado um conjunto de regras C de regras correlacionadas ordenadas por prioridade, gerar um conjunto de regras U, semanticamente equivalente, tal que nenhuma regra está correlacionada com qualquer outra" (CONDELL; SANCHEZ, 2004).

O descorrelacionamento remove conflitos e, desta forma, a necessidade de priorização das regras de filtragem. Além disto, remove redundâncias, mostrando a política real do *firewall*, que é o objetivo deste trabalho e foi pouco explorado na literatura.

A Figura 3.1 apresenta um exemplo de conjunto de regras correlacionadas, enquanto a Figura 3.2 mostra uma versão descorrelacionada e semanticamente equivalente do mesmo conjunto de regras, ambos em duas dimensões. Note-se que R_2 na Figura 3.1 é **fragmentada** em S_2 e S_3 na Figura 3.2.

R_1: $[2,3] \wedge [2,3] \rightarrow$ **accept**
R_2: $[1,5] \wedge [1,5] \rightarrow$ **block**

Figura 3.1: Exemplo de conjunto de regras correlacionadas.

S_1: $[2,3] \wedge [2,3] \rightarrow$ **accept**
S_2: $\{[1],[4,5]\} \wedge [1,5] \rightarrow$ **block**
S_3: $[2,3] \wedge \{[1],[4,5]\} \rightarrow$ **block**

Figura 3.2: Exemplo de conjunto de regras descorrelacionadas, semanticamente equivalente ao da Figura 3.1.

3.1 Complexidade

Infelizmente, o problema do descorrelacionamento é difícil no pior caso. De acordo com Gupta e Mckeown em (GUPTA; MCKEOWN, 1999), o número de regiões criadas por n regras d-dimensionais é $O(n^d)$ no pior caso. Isto significa que o descorrelacionamento pode gerar, no pior caso, um conjunto de regras de tamanho $O(n^d)$.

Apesar desta dificuldade teórica, vários trabalhos práticos mostraram que o número de regras correlacionadas em um conjunto de regras real é muito menor do que no pior caso, como relatado a seguir.

3.1.1 Realidade

Visando a evitar o tratamento puramente teórico dos problemas relacionados a *firewall*, alguns pesquisadores analisaram conjuntos de regras reais e apontaram suas características. Isto é importante principalmente, porque o pior caso de complexidade teórica de vários problemas algorítmicos relacionados a *firewalls* é bastante desanimador. Desta forma, a análise de casos práticos de uso de *firewall* serve, na maioria das vezes, para demonstrar que o pior caso praticamente nunca será atingido e que o caso médio é muito mais tratável.

Infelizmente, nem todos pesquisadores conseguem obter conjuntos de regras reais e os que conseguem normalmente não podem divulgá-los. Um conjunto de regras pode revelar muitas informações internas sobre a estrutura de uma organização o que é, na maioria das vezes, considerado confidencial. Assim, tanto os testes realizados por pesquisadores, quanto as características divulgadas pelos mesmos dos conjuntos em questão, não podem ser reproduzidos facilmente. No caso das características, felizmente, há uma grande uniformidade entre diferentes pesquisadores, o que sugere que um *firewall*, na realidade, tende a possuir uma estrutura comum mesmo quando usado por instituições diferentes.

Em (GUPTA; MCKEOWN, 1999) e (GUPTA, 2000) foi realizada a pesquisa mais abrangente publicada. Nela, foram coletados 793 conjuntos de regras de 101 provedores de serviços de Internet e redes corporativas, totalizando 41505 regras de filtragem. Dentre outras, foram encontradas as seguintes características:

- Os conjuntos não continham um grande número de regras. Apenas 0.7% dos conjuntos possuía mais de 1000 regras, com média de 50 regras;
- A sintaxe usada permitia 7 campos: endereço de origem e destino (32 bits cada), portas de origem e destino (16 bits cada), tipo de serviço (Type Of Service - TOS) (8 bits), protocolo (8 bits), flags do protocolo (8 bits); 17% das regras possuíam apenas um campo especificado, 23% possuíam três campos e 60% possuíam quatro campos especificados;
- Quatorze porcento das regras nos *firewalls* eram redundantes;
- Analisando-se a estrutura dos conjuntos de regras, chegou-se a conclusão que o número de regras correlacionadas (ou regiões que se sobrepõe) é muito menor do que o pior caso. No maior conjunto analisado, de 1733 regras, havia 4316 correlações em quatro dimensões (o pior caso seria 1733^4, que é aproximadamente 10^{13}). Em geral, o número de correlações nos conjuntos de regras se mostrou relativamente pequeno. A explicação disso, segundo os autores, é que as regras são originadas de políticas específicas dos operadores de rede e de acordos entre redes diferentes. Por exemplo, os operadores de redes diferentes podem especificar várias políticas relacionadas a interação entre *hosts* destas redes. Isto sugere que regras tendem a estar aglomeradas em pequenos grupos e não distribuídas aleatoriamente.

Em (QIU; VARGHESE; SURI, 2001), são analisados cinco conjuntos de regras de *firewalls* industriais, com um total de 953 regras, sendo o maior conjunto de 279 regras

e a média de 190 regras. Dentre outras características, foi descrito que nenhum campo possuía mais de quatro correlações em cada dimensão, sendo que a maioria continha uma, duas ou três correlações apenas. Isto implica, pela definição de correlação, que nenhuma regra de filtragem estava correlacionada com mais de quatro outras;

Em (BABOESCU; VARGHESE, 2001), quatro conjuntos de regras variando com entre 158 e 266 regras foram analisados. O resultado mais importante deste trabalho é que, nesses conjuntos, nenhum pacote casa com mais de quatro regras, indicando que cada regra está correlacionada com no máximo quatro outras. Baboescu et al em (BABOESCU; SINGH; VARGHESE, 2003), são analisados quatro conjuntos de regras, sendo que cada conjunto contém de 85 a 2000 regras. Confirmando o que foi descrito em (GUPTA; MCKEOWN, 1999), Baboescu et al afirmam que cada regra está correlacionada com entre três e cinco outras regras.

Kounavis et al em (KOUNAVIS et al., 2003) analisou quatro conjuntos de regras com tamanhos variando de 157 a 2399 regras. Dentre várias das análises, foi também realizada uma análise da sobreposição de regras. O resultado a que se chegou, como nos trabalhos anteriores, foi de que o máximo de regras com as quais um pacote casa é de sete, com média de quatro. Isto é, cada regra está correlacionada em média com apenas 4 outras regras.

Taylor e Turner em (TAYLOR; TURNER, 2005) revisaram doze conjuntos de regras com tamanhos variando entre 68 e 4557 regras. Eles afirmam que, em geral, sua análise concorda com as publicadas anteriormente, em que o número de regras com as quais um pacote qualquer casa é normalmente menor ou igual a cinco.

A experiência dos autores deste trabalho concorda com a da literatura, sendo que os conjuntos de regras normalmente estão pouco correlacionados. No entanto, há uma exceção não abordada na literatura e que será tratada na seção 5.2.

Desta forma, pela análise de conjuntos de regras reais, pode-se ver que o pior caso não acontece normalmente e que o descorrelacionamento é viável.

3.2 Aplicações

A fim de mostrar a importância de soluções para o problema do descorrelacionamento, são expostas, abaixo, várias utilizações práticas de algoritmos de descorrelacionamento.

A primeira publicação encontrada que faz uso do termo "descorrelacionamento" (*decorrelation*) foi (SANCHEZ; CONDELL, 2000). O algoritmo de descorrelacionamento de Sanchez e Condell surgiu para resolver possíveis inconsistências no *caching* de políticas do IPSec (a terminologia, para esta trabalho, é regra de filtragem). O exemplo dado pelo artigo, que mostra o problema e a necessidade de descorrelacionamento, é descrito a seguir em detalhes.

Tabela 3.1: Exemplo de políticas do IPSec

	origem	**destino**	**protocolo**	**direção**	**ação**
1	*	H_2	*	inbound	accept
2	*	*	*	inbound	block

As duas políticas do exemplo estão correlacionadas, já que todos os seus campos (que não incluem a ação) se sobrepõem. Sem entrar em detalhes sobre o funcionamento da

arquitetura do IPSec, suponha-se que um servidor de políticas PS_2 possua as políticas da Tabela 3.1. Suponha-se, também, que a seguinte transação aconteça:

1. PS_1 requisita política para *host* 3 (H_3);
2. PS_2 retorna a política número 2 a PS_1, que então coloca a política 2 em seu *cache*;
3. PS_1 agora procura por H_2 e descobre que já possui uma política que casa com H_2 (a política 2) e então a usa.

Este procedimento está incorreto, já que a política 2 indica que a comunicação com H_2 deve ser proibida, mas a política 1 em PS_2 afirma, contrariamente, que ela deve ser permitida.

A solução é remover as ambiguidades que podem existir entre as políticas. Assim, as políticas precisam estar descorrelacionadas antes de entrar em uso. A Tabela 3.2 mostra a versão descorrelacionada da Tabela 3.2.

Tabela 3.2: Exemplo de políticas do IPSec descorrelacionadas

	origem	**destino**	**protocolo**	**direção**	**ação**
1	*	H_2	*	inbound	accept
2	*	$not\ H_2$	*	inbound	block

Agora a transação torna-se correta:

1. PS_1 requisita política para *host* 3 (H_3);
2. PS_2 retorna a política número 2 a PS_1, que então coloca a política 2 em seu *cache*;
3. PS_1 agora procura por H_2, não possui uma política que casa com H_2 e requisita para PS_2 a política para H_2;
4. PS_2 retorna a política 1 para PS_1, que então coloca a política 1 em seu *cache*.

A partir da pesquisa de Sanchez e Condell, outras aplicações de descorrelacionamento sugiram. Eronen e Zitting em (ERONEN; ZITTING, 2001), inicialmente, tentaram representar as regras de filtragem do modo convencional, utilizando prioridade para resolver conflitos. No entanto, esta abordagem se mostrou não intuitiva e ineficiente para a CLP utilizada (descrita na seção 2.1.3). Assim, foi adotado o descorrelacionamento, que gerou uma representação mais simples e elegante das regras de filtragem. Eles utilizaram o algoritmo de Sanchez e Condell (SANCHEZ; CONDELL, 2000), detalhado a seguir.

Mierlutiu em (MIERLUTIU, 2003) utilizou descorrelacionamento para um mecanismo de *cache* de regras de filtragem para o *firewall iptables/Netfilter* do Linux, a fim de melhorar seu desempenho. A idéia básica do mecanismo é colocar regras que causariam uma diminuição no tempo de resposta do *firewall* em um *cache*. Como a estrutura do Netfilter é de uma lista encadeada com *first-match wins*, onde para cada pacote a lista é percorrida até alguma casar com ele, o *cache* ficaria antes do início da lista de regras. No entanto, como regras podem estar correlacionadas, não seria possível simplesmente colocar uma regra de filtragem bastante requisitada diretamente no *firewall*, visto que isto

pode alterar a política do *firewall*. Desta forma, a solução do autor foi colocar apenas versões descorrelacionadas das regras na *cache*, o que garante que a política se manterá. O algoritmo utilizado é o de Sanchez et al.

Em (ACHARYA et al., 2006), é utilizada a idéia de otimização da ordem das regras de filtragem em um dado conjunto de regras baseado na análise do tráfego que está chegando ao *firewall*. A fim de possuir flexibilidade total no reordenamento das regras, é necessário eliminar qualquer dependência entre elas, isto é, é necessário descorrelacioná-las. A técnica utilizada pelos autores foi de descorrelacionar e depois tentar agrupar regras similares (discutida no capítulo 4). O descorrelacionamento é baseado no apresentado por Sanchez e Condell (CONDELL; SANCHEZ, 2004).

Zhao e Bellovin em (ZHAO; BELLOVIN, 2007) descrevem uma álgebra híbrida para manipulação de políticas de *firewall*, isto é, tanto para manipular *firewalls* de rede (usados neste trabalho) quanto *firewalls* de *host*. Com o objetivo de conseguir manipular *firewalls* de rede, os autores utilizam o descorrelacionamento proposto em (SANCHEZ; CONDELL, 2000), já que, desta maneira, as regras não dependem mais de ordem.

3.3 Grafo de correlacionamento

Nesta seção é introduzida um estrutura auxiliar que permite uma melhor visualização do correlacionamento em um conjunto de regras e uma melhor manipulação do mesmo. Esta estrutura é denominada de *grafo de correlacionamento*. A utilização de um grafo para representar o correlacionamento entre regras se justifica, pois grafos são ideais para representar relações binárias e a relação de correlação (definida na seção 2.2) é binária.

Grafo de correlacionamento é um grafo $G = (V, E)$, onde:

- V é conjunto R de regras de filtragem possivelmente correlacionadas;
- E é a relação binária de $correlacao$ ($correlacao \subseteq R \times R$). Assim, $(v, v') \in E \iff correlacao(v, v')$. Dois vértices (que representam regras de filtragem) possuem uma aresta os ligando se e somente se eles estão correlacionados.

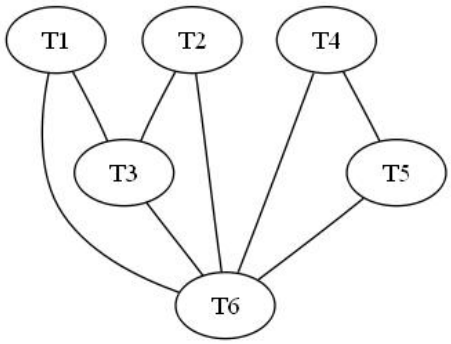

Figura 3.3: Grafo de correlacionamento das regras na Figura 3.4.

Como pode ser visto na Figura 3.3, esta representação do correlacionamento é bastante intuitiva e, por isso, será utilizada neste trabalho. Além disso, como relatado na seção 3.1.1 sobre características de filtros reais, uma regra, normalmente, está correlacionada com poucas outras. O grafo, que explicita estas correlações, é útil para segmentar

o conjunto de regras em subconjuntos de regras relacionadas, possibilitando a execução do algoritmo de descorrelacionamento apenas nestes subconjuntos e desta forma, o otimizando.

O Algoritmo 1 mostra uma simples maneira de gerar tal grafo. O algoritmo aplica diretamente a definição de grafo de correlacionamento da seção acima. Sua complexidade é quadrática $O(n^2)$ em relação ao tamanho de C ($n = |C|$). A complexidade pode ser melhorada utilizando-se uma estrutura mais adequada para manipular as regras de filtragem, como a do estudo ((QIAN; HINRICHS; NAHRSTEDT, 2001)). Em geral, os algoritmos deste trabalho serão mostrados da maneira mais didática e simples, o que normalmente não é a maneira mais otimizada.

Entrada: Regras de filtragem correlacionadas C.
Saída: Grafo de correlacionamento $G = (V, E)$ do C dado.

```
para k ← 1 até |C| faça
    para l ← i até k − 1 faça
        se C_k ∩ C_l ≠ ∅ então
            acrescenta aresta (k, l) em E
        fim
    fim
fim
```

Algoritmo 1: Algoritmo de Geração de Grafo de Correlacionamento

3.4 Algoritmos

Dada a viabilidade de resolver o problema do descorrelacionamento em tempo aceitável, como evidenciado nas seções 3.1 e 3.2, nesta seção são descritos alguns algoritmos para resolvê-lo. O contexto no qual cada algoritmo aparece na literatura será brevemente mencionado, o pseudo código do algoritmo e sua descrição serão apresentados. Pelo fato de o pior caso do descorrelacionamento raramente acontecer, como reforçado na seção 3.1.1, em geral, a complexidade de pior caso dos algoritmos não será mostrada.

Um exemplo motivacional, que mostra como deve ser a saída dos algoritmos no caso de um conjunto de regras correlacionadas unidimensionais (Figura 3.4) encontra-se na Figura 3.5.

T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $[3, 10] \rightarrow$ **block**
T_4: $[25] \rightarrow$ **block**
T_5: $[20, 30] \rightarrow$ **accept**
T_6: $[1, 1000] \rightarrow$ **block**

Figura 3.4: Conjunto de regras correlacionadas.

T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $\{[4], [6, 10]\} \rightarrow$ **block**
T_4: $[25] \rightarrow$ **block**
T_5: $\{[20, 24], [26, 30]\} \rightarrow$ **accept**
T_6: $\{[1, 2], [11, 19], [31, 1000]\} \rightarrow$ **block**

Figura 3.5: Conjunto de regras descorrelacionadas obtido a partir das regras da Figura 3.4.

3.4.1 Algoritmo de Descorrelacionamento de Sanchez e Condell

Sanchez e Condell em (SANCHEZ; CONDELL, 2000) e (CONDELL; SANCHEZ, 2004) apresentaram o primeiro algoritmo de descorrelacionamento para bancos de dados de políticas de segurança (*Security Policy Databases* - SPD) do protocolo IPSec. Pelo fato de políticas de segurança serem modeladas exatamente como regras de filtragem de *firewalls*, o algoritmo pode ser usado para descorrelacionar regras de filtragem sem qualquer modificação. A motivação de Sanchez e Condell foi a de retirar a necessidade de manter as regras ordenadas, visto que a ordem causava inconsistências no mecanismo de *caching* de políticas utilizado (detalhes na seção 3.2). Abaixo, o algoritmo é descrito em alto nível e todos os seus passos são mostrados.

O algoritmo possui como entrada um conjunto C de regras correlacionadas e gera em sua saída um conjunto U de regras descorrelacionadas construídas a partir de C e semanticamente equivalentes a este. Para tal, ele utiliza uma estrutura de árvore para cada regra $C_i \in C$ que está sendo descorrelacionada. A árvore possui, como nodos, os nomes dos campos e cada aresta ou ramo da árvore possui um valor assumido pelo campo do nó de onde ela é originada. Os seguintes são os passos do algoritmo, como representado nos artigos originais:

1. Colocar C_1, a primeira regra de C, no conjunto U como U_1.

Para cada regra subsequente $C_i (2 \leq i \leq |C|) \in C$:

2. Determinar se C_i está correlacionada com qualquer regra em U. Se C_i não está correlacionada com nenhuma regra em U, acrescentar C_i a U.

3. Se C_i está correlacionada com alguma regra em U, criar uma árvore para a regra C_i.

   (a) Selecionar um campo c_j da regra C_i. Se esta é a primeira vez dentro do loop, qualquer campo pode ser escolhido. Para as demais iterações, qualquer campo que ainda não foi usado no ramo sendo processado pode ser utilizado.

   (b) Criar um nó usando o campo c_j. Se esta é a primeira iteração do loop, este nó é o nó raiz, se não, ele é criado no final do ramo sendo processado atualmente.

   A regra neste nó é a regra formada pelos valores dos campos de cada ramo entre a raiz e este nó. Quaisquer valores de campo que ainda não foram representados no ramo assumem o valor do campo em $C_i.c_j$, já que os valores em C_i representam o conjunto universo para cada campo. T é o conjunto de regras em U que estão correlacionadas com a regra até este nó.

(c) Adicionar um ramo à árvore para cada valor do campo c_j que aparece em qualquer das regras em T e para o valor de c_j em C_i. Se o valor de c_j de qualquer regra em T inclui o valor de c_j em C_i, então usa-se o valor de c_j em C_i, já que este valor representa o conjunto universo.

(d) Acrescentar um ramo para o complemento da união de todos os valores do campo c_j em T. Quando gerar o complemento, lembrar que o conjunto universo é o valor do campo c_j em C_i. Um ramo não precisa ser criado para o conjunto vazio.

(e) Para cada ramo criado, ir para (a) até que a árvore esteja completa (isto é, que não haja campos restantes para processar em nenhum dos ramos).

4. Acrescentar a U todas as regras nas folhas da árvore que não estão correlacionadas com U.

5. Pegar o próximo C_i e ir para 2. Quando todas as políticas em C tiverem sido processadas, então U vai conter a versão descorrelacionada de C.

Como pode ser visto, o algoritmo é apresentado de maneira completamente descritiva. A fim de analisá-lo melhor ou implementá-lo, uma listagem em estilo de pseudo-código é mais útil. Desta forma, no Algoritmo 2 é apresentada uma versão em pseudo-código do algoritmo descrito acima. As funções auxiliares usadas no algoritmo são listadas abaixo:

- `insereRegra`(r, C) - insere a regra r no conjunto C ;
- `descorrelacionado`(r, C) - retorna $TRUE$ se a regra r está descorrelacionada em relação as regras do conjunto C e $FALSE$ caso contrário ;
- `selecionaCampo`(*r*,*ramoAtual*) - seleciona um campo ainda não utilizado na regra r que está sendo processada ;
- `insereArvore`(s,*ramoAtual*,*arvore*) - insere o nó s como destino de *ramoAtual* na *arvore*. Se *arvore* ainda não estiver criada, s é inserido como raiz ;
- `acrescentaRamo`(*ramo*, *ramoAtual*) - acrescenta *ramo* no nó destino de *ramoAtual*, se *ramo* $\neq nil$;
- `complemento`(*conj*,*univ*) - retorna o complemento de *conj* tendo *univ* como conjunto universo ;
- `selecionaProximoRamo()` - seleciona próximo ramo a ser processado.

O algoritmo começa inserindo a primeira regra de C em U, visto que esta regra possui maior prioridade e, por isso, não está correlacionada com nenhuma outra (já que não há nenhuma outra regra no momento). Ele continua nas regras subsequentes ($C_i, 1 < i \leq |C|$), analisando regra a regra em relação ao conjunto U de regras já descorrelacionadas. Se a regra já se encontra descorrelacionada com U, basta inseri-la. Se não, é necessário descorrelacioná-la. Isto é realizado colocando-se as regras de U correlacionadas com o C_i atual (que formam o conjunto T) em uma árvore, campo a campo. O nome do primeiro campo selecionado forma a raiz da árvore.

```
1  U ← ∅
2  insereRegra (C_1, U)
3  arvore ← nil
4  ramoAtual ← nil
5  para i ← 2 até |C| faça
6      se descorrelacionado (C_i, U) então
7          insereRegra (C_i, U)
8      senão
9          enquanto A árvore não estiver completa faça
10             j ←selecionaCampo (C_i)
11             insereArvore (nome(c_j),ramoAtual,arvore)
               /* T é o conjunto de regras em U que estão
                  correlacionadas com a regra até este nó    */
12             uniaoValores ← ∅
13             para cada valorDiferente do campo nome(c_j) em T faça
14                 acrescentaRamo (valorDiferente ∩ C_i.c_j,ramoAtual)
15                 uniaoValores = uniaoValores ∪ valorDiferente
16             fim
17             acrescentaRamo (C_i.c_j, ramoAtual)
18             acrescentaRamo (complemento (uniaoValores, C_i.c_j),
               ramoAtual)
               /* seleciona ramo ainda não processado        */
19             ramoAtual = selecionaProximoRamo ()
20         fim
21         para cada regra em arvore faça
22             se descorrelacionado (regra, U) então
23                 insereRegra (regra, U)
24             fim
25         fim
26     fim
27 fim
```

Algoritmo 2: Algoritmo de Sanchez et al

Para cada valor diferente deste campo encontrado em T, cria-se um ramo, tendo como rótulo este valor, saindo do nó atual e indo para um nó com o nome do próximo campo a ser selecionado. O valor de cada ramo é "normalizado" em relação ao conjunto universo, que é o valor que C_i possui no campo em questão, já que as regras que surgiram serão partes de C_i. Além disso, é acrescentado um ramo com o complemento da união dos valores diferentes acrescentados na árvore para o campo atual (caso este seja não nulo) e um ramo com o valor do campo em questão de C_i (caso este já não tenha sido incluído). Seleciona-se o próximo campo, se ainda houver, e seu nó é acrescentado na outra extremidade de cada uma das arestas criadas. Para cada novo nó, um novo T é calculado e repetem-se os passos acima, a partir do início deste parágrafo. Caso não haja mais nada a ser acrescentado na árvore, ela está completa. Basta, agora, percorrer da raiz até cada uma das folhas e verificar se cada regra de filtragem gerada por estes caminhos está descorrelacionada em relação a U. Em caso positivo, esta regra é acrescentada a U e, caso contrário, ela é ignorada. O mesmo processo se repete para todas as regras e, no final, U

conterá regras descorrelacionadas equivalentes às regras de C.

A seguir, é apresentado um exemplo da execução do algoritmo de Sanchez e Condell 2. O conjunto de regras utilizado neste e nos demais exemplos deste capítulo encontra-se na Figura 3.6.

C_1: $[2] \wedge [2] \rightarrow$ **accept**
C_2: $[4,5] \wedge [4,5] \rightarrow$ **accept**
C_3: $[3,8] \wedge [3,8] \rightarrow$ **block**

Figura 3.6: Exemplo de conjunto de regras correlacionadas.

Inicialmente, C_1 é inserido em U. No laço, para $i = 2$, C_2 já está descorrelacionado com $U = \{C_1\}$ e, por isso, é inserido no mesmo. Para $i = 3$, C_3 está correlacionado com $U = \{C_1, C_2\}$, sendo $T = \{C_2\}$. Desta forma, é construída a árvore como descrita no algoritmo e mostrada Figura 3.7, sendo os campos escolhidos na ordem em que aparecem (isto é, c_1 e depois c_2). Os fragmentos de regra acrescentados em U estão, na figura, mostrados por caminhos pontilhados e serão chamados de $U_3 = [4,5] \wedge \{[3],[6,8]\} \rightarrow$ **block** e $U_4 = \{[3],[6,8]\} \wedge [3,8] \rightarrow$ **block**. No final, $U = \{C_1, C_2, U_3, U_4\}$.

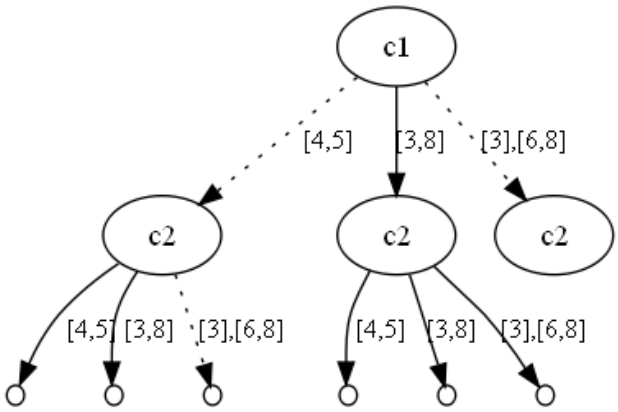

Figura 3.7: Árvore gerada pelo algoritmo de Sanchez e Condell para C_3.

Algumas otimizações foram citadas pelos autores do algoritmo:

1. O processamento de um ramo pode ser terminado se a regra formada por ele já está descorrelacionada com U, isto é, se $T = \emptyset$. No caso, a regra formada por este ramo é então incluída em U;

2. O processamento de um ramo pode ser terminado se a regra formada por ele encontra-se sobreposta por qualquer regra em $C_1, \ldots, C_{i-1}$. Isto é válido, pois estas mesmas regras são representadas de maneira equivalente em U (mas possivelmente fragmentadas);

3. O algoritmo não determina qual campo é selecionado a cada vez. Uma seleção criteriosa pode diminuir o número de ramos formados ou até formar um ramo descorrelacionado sem a necessidade de mais processamento;

4. Se o valor do campo $C_i.c_j$ estiver contido ou for igual a todos os valores deste mesmo campo em T, não é necessário processá-lo. Isto, pois apenas um ramo será criado, com o valor do conjunto universo e o ramo com o complemento será o conjunto vazio.

O motivo do funcionamento do algoritmo acima pode ser descrito da seguinte maneira: para uma regra C_i correlacionada com outras regras já descorrelacionadas que formam o conjunto T, são testados todos os valores possíveis para cada campo das regras de T (tendo como valor universo de cada campo o valor do campo em C_i), além dos valores do universo que não aparecem em T (complemento da união dos valores dos campos de T e o próprio valor do campo em C_i). Desta forma, faz-se uma busca exaustiva pelo espaço dos campos de C_i tentando-se todas as combinações de valores de campo em T e tentando-se "cortar" esta regra, através do uso de complemento, gerando assim, as regras descorrelacionadas.

Como o algoritmo realiza, então, uma forma de força-bruta com heurística, é natural implementá-lo utilizando-se backtracking. Como mostrado no próximo e último pseudo-código do algoritmo de Sanchez e Condell, a implementação torna-se mais simples e intuitiva, visto que a estrutura de árvore fica implícita nas chamadas recursivas à função que realiza o descorrelacionamento. No Algoritmo 3 é mostrado este método de resolução recursivo. Os parâmetros da função `recursaoDescorrelaciona` são:

- *regraBase* - regra a ser descorrelacionada;
- *regraAtual* - regra formada pelo caminho da raiz até o ramo atual da árvore. Os campos que ainda não foram selecionados possuem o valor da regra base (universo);
- *nivel* - nível da árvore onde está sendo realizado o processamento.

3.4.2 Algoritmo de Descorrelacionamento de Liu e Gouda

Como já mencionado na seção 2.1.3, Liu e Gouda em (LIU; GOUDA, 2004) mostraram uma maneira de transformar regras de baixo nível, em uma abstração baseada em árvores de decisão chamada FDD (*Firewall Decision Diagram*). No artigo em questão os autores tinham como objetivo utilizar FDDs para comparar configurações de *firewalls* criados por pessoas diferentes, mas com a mesma finalidade. Essa transformação das regras de baixo nível garante algumas propriedades, sendo uma delas o descorrelacionamento entre as regras.

Abaixo, são expostas as definições relevantes utilizadas por Liu e Gouda, como aparecem no artigo (LIU; GOUDA, 2004). Estas definições são usadas nesta seção apesar de parecidas com as da seção 2.2, pois foram criadas especificamente para tratar de FDDs.

- Um *pacote* sobre os campos $F_1, \ldots, F_n$ é uma tupla d-dimensional $(p_1, \ldots, p_n)$ onde cada p_i é um elemento no domínio $D(F_i)$ do campo F_i e cada $D(F_i)$ é um intervalo de inteiros não negativos. O conjunto de todos os pacotes possui tamanho $|D(F_1)| \times \ldots \times |D(F_d)|$ e é chamado de Σ
- Um *firewall* consiste de uma sequência de regras no formato

$$(F_1 \in S_1) \wedge \ldots \wedge (F_d \in S_d) \rightarrow \textbf{ação}$$

onde cada S_i é um subconjunto não-vazio de $D(F_i)$ e ação é aceitar ou bloquear. Note-se que, nas definições da seção 2.2, $F_i = nome(c_i)$ e para uma regra R, $S_i = R.c_i$.

- Um *Firewall Decision Diagram (FDD)* f sobre os campos $F_1, \ldots, F_n$ é um grafo acíclico e dirigido (ou seja, uma árvore) com as seguintes 5 propriedades:

  1. Existe exatamente um nó em f que não possui arestas de entrada, sendo este nó denominado de *raiz* de f. Os nós em f que não possuem arestas de saída são denominados nós *terminais* de f.
  2. Cada nó v em f possui um rótulo, chamado de $F(v)$, tal que

$$F(v) \in \begin{cases} \{F_1, \ldots, F_n\} & \text{se } v \text{ é não terminal.} \\ \{\text{accept, block}\} & \text{se } v \text{ é terminal.} \end{cases}$$

  3. Cada aresta e em f possui um rótulo, denotado $I(e)$, tal que se e é uma aresta de saída do nó v, então
$$I(e) \subseteq D(F(v)).$$
  4. Um caminho dirigido em f da raiz até um nó terminal é denominado *caminho de decisão* de f. Dois nós em um caminho de decisão não podem ter o mesmo rótulo.
  5. O conjunto de todas as arestas de saída de um nó v em f, chamado de $E(v)$, satisfaz duas condições:
     (a) *Consistência*: $I(e) \cap I(e') = \emptyset$ para quaisquer duas arestas diferentes e e e' em $E(v)$
     (b) *Completude*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$

- Um caminho de decisão em um FDD f é representado por $(v_1 e_1 \ldots v_k e_k v_{k+1})$, onde v_1 é a raiz e v_{k+1} é um nó terminal de f, e cada e_i é uma aresta dirigida do nó v_i para o nó v_{i+1}. Um caminho de decisão $(v_1 e_1 \ldots v_k e_k v_{k+1})$ em um FDD sobre os campos $F_1, \ldots, F_n$ define a seguinte regra:

$$(F_1 \in S_1) \wedge \ldots \wedge (F_d \in S_d) \rightarrow F(v_{k+1})$$

  onde

$$S_i = \begin{cases} I(e_j) & \text{se há um nó } v_j \text{ no caminho de decisão com rótulo igual a } F_i, \\ D(F_i) & \text{se nenhum nó no caminho de decisão possui como rótulo } F_i. \end{cases}$$

Abaixo é mostrado, no Algoritmo 4, como construir um FDD a partir de uma sequência de regras, sendo que cada regra está no formato $F_1 \in S_1 \wedge \ldots \wedge F_d \in S_d \rightarrow$ **ação**.

A construção do FDD é mostrada por indução matemática. Inicialmente, um FDD parcial, isto é, sem a propriedade de completude, é construído a partir da primeira regra R_1. O FDD construído a partir da primeira regra contém apenas o caminho de decisão desta regra. Suponha que para as primeiras i regras, de R_1 até R_i, foi construído um FDD parcial, cuja raiz v é rotulada como F_1 e suponha que v possui k arestas de saída,

$e_1, \ldots, e_k$. Seja R_{k+1} a regra $(F_1 \in S_1) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ **ação**. A seguir, é necessário analisar como anexar R_{i+1} a este FDD parcial.

Inicialmente, examina-se se é necessário acrescentar outra aresta de saída a v. Se $S_1 - (I(e_1) \cup \ldots \cup I(e_k)) \neq \emptyset$, será necessário acrescentar uma nova aresta de saída com rótulo $S_1 - (I(e_1) \cup \ldots \cup I(e_k))$ a v, pois qualquer pacote cujo campo F_1 pertence a $S_1 - (I(e_1) \cup \ldots \cup I(e_k))$ não casa com qualquer das primeiras i regras, mas casa com R_{i+1} se o pacote satisfaz também $(F_2 \in S_2) \wedge \ldots \wedge (F_d \in S_d)$. Então, deve ser construído um caminho de decisão a partir de $(F_2 \in S_2) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ **ação** e deve-se fazer com que a nova aresta de saída do nó v aponte para o primeiro nó deste novo caminho de decisão.

Depois disso, é necessário comparar S_1 e $I(e_j)$ para cada $j, 1 \leq j \leq k$. A comparação leva a três possibilidade:

1. $S_1 \cap I(e_j) = \emptyset$: Neste caso, pula-se a aresta e_j, pois nenhum pacote cujo valor do campo F_1 pertence ao conjunto $I(e_j)$ casa com R_{k+1};

2. $S_1 \cap I(e_j) = I(e_j)$: Neste caso, um pacote cujo valor do campo F_1 está no conjunto $I(e_j)$ pode casar uma das primeiras i regras ou a regra R_{i+1}. Desta maneira, é necessário acrescentar a regra $(F_2 \in S_2) \ldots F_d \in S_d) \rightarrow$ **ação** ao subgrafo que possui como raiz o nó apontado por e_j;

3. $S_1 \cap I(e_j) \neq \emptyset$ e $S_1 \cap I(e_j) \neq I(e_j)$: Aqui, é necessário dividir a aresta e em duas: e' com o rótulo $I(e_j) - S_1$ e e'' com rótulo $I(e_j) \cap S_1$. Então, são feitas duas cópias do subgrafo que tem como raiz o nó apontado por e_j, sendo que e' e e'' apontam para uma cópia cada. Agora, trata-se e' utilizando o caso 1 e e'' utilizando-se o caso 2.

No pseudo-código que segue, $e.t$ é o nó destino ao qual a aresta e aponta.

Após construído o FDD, basta extrair todos seus caminhos de decisão para regras, que já estão descorrelacionadas. Isto acontece, pois pela propriedade de Consistência, os caminhos de decisão já se encontram descorrelacionados.

Um exemplo de FDD gerado a partir das regras da Figura 3.6 se encontra na Figura 3.8. Inicialmente, cria-se a árvore baseada em C_1. Depois, C_2 é acrescentado em ramos separados, já que não há correlação entre C_1 e C_2. Por último, C_3 é inserida, sendo fragmentada tanto no primeiro quanto no segundo campos (F_1 e F_2, pela terminologia usada em FDDs). As novas regras são $U_3 = [4,5] \wedge \{[3],[6,8]\} \rightarrow$ **block** e $U_4 = \{[3],[6,8]\} \wedge [3,8] \rightarrow$ **block**. No final, $U = \{C_1, C_2, U_3, U_4\}$.

3.4.3 Algoritmo de Descorrelacionamento baseado em Cuppens et al e Qian et al

O algoritmo de Cuppens, Cuppens e García em (CUPPENS; CUPPENS; GARCÍA, 2005) tem como objetivos notificar quais regras possuem as anomalias de sombreamento e redundância (como definidas por Shaer e Hamed e mencionadas na seção 2.1.1) e, durante este processo, descorrelacionar as regras. Segundo os autores, o descorrelacionamento pode ajudar o usuário perceber os erros que foram detectados.

A contribuição deste algoritmo é a maneira simples e ordenada na qual duas regras são descorrelacionadas. O Algoritmo 5 mostra o pseudo-código da função denominada `Exclusion` em (CUPPENS; CUPPENS; GARCÍA, 2005) e aqui chamada de `descorrelacionaRegra`. O algoritmo, basicamente, descorrelaciona uma regra B

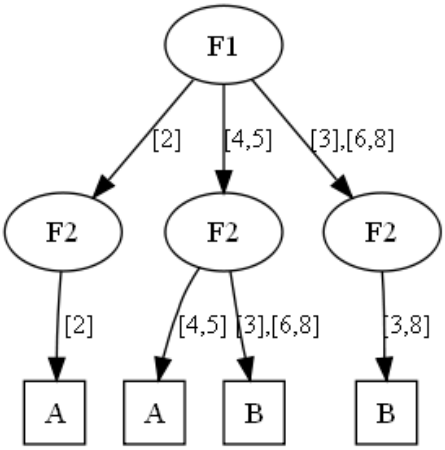

Figura 3.8: FDD gerado pelo algoritmo de Liu e Gouda.

em relação a uma regra de maior prioridade A em cada dimensão, retornando, no pior caso, d fragmentos da regra B. A Figura 3.2 foi obtida através da aplicação deste algoritmo a R_1 e R_2 da Figura 3.1.

O algoritmo de descorrelacionamento de Cuppens et al, consiste em percorrer as regras e, uma a uma, descorrelacionando-as usando a função dada no Algoritmo 5. No entanto, o algoritmo é descrito em alto nível e não há informação de como armazenar os fragmentos de regras gerados e de como manipulá-los.

O algoritmo *filter* apresentado por Qian, Hinrichs e Nahrstedt et al em (QIAN; HINRICHS; NAHRSTEDT, 2001), tem o objetivo de transformar um conjunto de regras com múltiplos tipos de ação (no nosso caso, accept e block) para um conjunto de regras explícitas com apenas um tipo de ação e uma regra genérica implícita que casa com todo tráfego não casado pelas regras explícitas. Para isso, é necessário descorrelacionar as regras de ações diferentes e é nesta parte do algoritmo que há uma contribuição: o uso de uma fila para armazenar os fragmentos da regras descorrelacionadas. O algoritmo de (QIAN; HINRICHS; NAHRSTEDT, 2001) e de (CUPPENS; CUPPENS; GARCÍA, 2005) se complementam: um deles mostra como armazenar os fragmentos, sem mostrar como descorrelacionar as regras e o outro mostra como descorrelacionar as regras, mas omite onde armazenar seus fragmentos.

Com base nestas observações, foi criado, no presente trabalho, um algoritmo de descorrelacionamento que é uma colagem entre os algoritmos de (QIAN; HINRICHS; NAHRSTEDT, 2001) e de (CUPPENS; CUPPENS; GARCÍA, 2005). O pseudo-código deste algoritmo é mostrado no Algoritmo 6, que faz uso da função no Algoritmo 5.

O Algoritmo 6 percorre cada regra da entrada (um conjunto de regras correlacionadas C ordenado por prioridade) e verifica se ela está correlacionada. Se a regra está descorrelacionada, ela pode ser inserida na saída (o conjunto de regras descorrelacionadas U). Se não, a regra é descorrelacionada (em relação a regra de maior prioridade que já foi processada) usando o Algoritmo 5 e seus fragmentos (cada *fragRegra*) são colocados em uma fila. Enquanto há fragmentos na fila, cada fragmento passa pelo processo acima descrito e é inserido em U apenas se ele está descorrelacionado. Quando a fila ficar vazia, a regra atual de C está certamente descorrelacionada e representada em U. É importante notar que o descorrelacionamento pode eliminar a regra atual, se ela estiver contida em regras de maior prioridade. Depois de todas as regras de C serem analisadas, U torna-se uma versão semanticamente equivalente, mas descorrelacionada, de C.

Abaixo é desenvolvido um exemplo de aplicação do algoritmo apresentado nesta seção às regras da Figura 3.6. Inicialmente, C_1 é colocado e retirado da fila diretamente para U, já que $U = \emptyset$. Para $i = 2$, o mesmo acontece com C_2, já que ele não está correlacionado com C_1. Quando $i = 3$, C_3 está correlacionado a função `encontraCorrelacionada` atribui a j o valor 2, pois C_2 é a primeira (e única) regra correlacionada com C_3. Após isso, a função `descorrelacionaRegra` do Algoritmo 5 é chamada, descorrelacionado C_3 em relação a C_2. Os fragmentos retornados por `descorrelacionaRegra` são $U_3 = \{[3], [6, 8]\} \wedge [3, 8] \rightarrow$ **block** e $U_4 = [4, 5] \wedge \{[3], [6, 8]\} \rightarrow$ **block** e estes são colocados na fila. O algoritmo prossegue retirando U_3 da fila e, como ele não está correlacionado com nenhuma regra, inserindo-o em U. O mesmo acontece com U_4. No final, $U = \{C_1, C_2, U_3, U_4\}$.

```
1  insereRegra (C_1, U)
2  para i ← até |C| faça
3      se descorrelacionado (C_i, U) então
4          insereRegra (C_i, U)
5      senão
6          recursaoDescorrelaciona (C_i, C_i,nivel)
7      fim
8  fim

9  recursaoDescorrelaciona (regraCorrelacionada,regraAtual,nivel)
   início
       /* regra processada por inteiro                          */
10     se nivel = d + 1 então
11         se descorrelacionado (regraAtual, U) então
12             insereRegra (regraAtual, U)
13         fim
14         retorna
15     fim
16     j = selecionaCampo (C_i)
       /* T é o conjunto de regras em U que estão
          correlacionadas com a regra até este ramo.           */
17     uniaoValores ← ∅
18     para cada valorDiferente do campo nome(c_j) em T faça
19         temp = valorDiferente ∩ C_i.c_j
20         se temp não foi investigado ainda então
21             uniaoValores ← uniaoValores ∪ temp
22             regraAtual.c_j ← temp
23             recursaoDescorrelaciona (regraCorrelacionada, regraAtual,
               nivel +1)
24             regraAtual.c_j ← C_i.c_j
25         fim
26     fim
27     se C_i.c_j não foi investigado ainda então
28         regraAtual.c_j ← C_i.c_j
29         recursaoDescorrelaciona (regraCorrelacionada, regraAtual,
           nivel +1)
30         regraAtual.c_j ← C_i.c_j
31     fim
32     tmp ← complemento (uniaoValores, C_i.c_j)
33     se tmp ≠ ∅ então
34         regraAtual.c_j ← tmp
35         recursaoDescorrelaciona (regraCorrelacionada, regraAtual,
           nivel +1)
36         regraAtual.c_j ← C_i.c_j
37     fim
38 fim
```

Algoritmo 3: Algoritmo de Sanchez e Condell - versão recursiva

Entrada: Um conjunto de regras de filtragem f $(R_1, \ldots, R_n)$
Saída: Um FDD f' semanticamente equivalente a f

1. construir um caminho de decisão com raiz v a partir de R_1
2. **para** $i \leftarrow 2$ **até** n **faça** APPEND (v, R_i)

APPEND (v, $(F_m \in S_m) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ **ação**)
início
/* $F(v) = F_m \in E(v) = \{e_1, \ldots, e_k\}$ */
se $S_m - (I(e_1) \cup \ldots \cup I(e_k)) \neq \emptyset$ **então**

1. inserir uma aresta de saída e_{k+1} com rótulo $S_m - (I(e_1) \cup \ldots \cup I(e_k))$ em v
2. construir um caminho de decisão da regra $(F_{m+1} \in S_{m+1}) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ ***ação***, e fazer e_{k+1} apontar para o primeiro nó deste caminho

fim
se $m < d$ **então**
para $j \leftarrow 1$ **até** k **faça**
se $I(e_j) \subseteq S_m$ **então**
APPEND ($e_j.t$, $(F_{m+1} \in S_{m+1}) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ ***ação***)
senão se $I(e_j) \cap S_m \neq \emptyset$ **então**

1. acrescentar uma aresta de saída e a v, e colocar $I(e_j) \cap S_m$ como rótulo de e
2. fazer uma cópia do subgrafo com raiz $e.t$ e fazer e apontar para a raiz da cópia
3. substituir o rótulo de e_j por $I(e_j) - S_m$
4. APPEND ($e.t$, $(F_{m+1} \in S_{m+1}) \wedge \ldots \wedge (F_d \in S_d) \rightarrow$ *ação*)

fim
fim
fim
fim

Algoritmo 4: Algoritmo de Liu et al

Entrada: Regras B e A
Saída: Fragmentos da regra B, que foi cortada em torno de A

descorrelacionaRegra (B, A)
início
  $C \leftarrow$
    $\{(B_1 - A_1) \wedge B_2 \ldots \wedge B_d$,
    $(B_1 \cap A_1) \wedge (B_2 - A_2) \ldots \wedge B_d$,
    $(B_1 \cap A_1) \wedge (B_2 \cap A_2) \wedge (B_3 - A_3) \ldots \wedge B_d$,
    $\vdots$
    $(B_1 \cap A_1) \wedge (B_2 \cap A_2) \wedge (B_3 \cap A_3) \ldots \wedge (B_d - A_d)\}$
  retorna C
fim

Algoritmo 5: Função que descorrelaciona regra B da regra A (CUPPENS; CUPPENS; GARCÍA, 2005)

Entrada: Um conjunto de regras correlacionadas C, ordenado por prioridade.
Saída: Um conjunto de regras descorrelacionadas U, semanticamente equivalente a C.

$U \leftarrow \emptyset$
para $i \leftarrow 1$ **até** $|C|$ **faça**
  Fila.`push` (C_i)
  enquanto Fila.`size` > 0 **faça**
    regraAtual = Fila.`pop` ()
    $j \leftarrow$`encontraCorrelacionada` (regraAtual,i, C)
    se $j = -1$ **então**
      insere regraAtual em U
    senão
      tempList $\leftarrow$ descorrelacionaRegra (regraAtual, C_j)
      para cada fragRegra $\in$ tempList **faça**
        Fila.`push` (fragRegra)
      fim
    fim
  fim
fim

`encontraCorrelacionada` (regraAtual , i, C)
início
  para $j \leftarrow 1$ **até** $i - 1$ **faça**
    se regraAtual $\cap\, C_j \neq \emptyset$ **então**
      retorna j
    fim
  fim
  `/*` regraAtual `está descorrelacionada` `*/`
  retorna -1
fim

Algoritmo 6: Algoritmo de Descorrelacionamento baseado em (QIAN; HINRICHS; NAHRSTEDT, 2001)

4 AGRUPAMENTO DE REGRAS DESCORRELACIONADAS

Vários algoritmos de descorrelacionamento foram previamente descritos neste trabalho. Também já foi estabelecido que, após a aplicação de um dos algoritmos citados, a política real se torna visível. No entanto, ela ainda é representada em regras de baixo nível fragmentadas, que normalmente são difíceis de entender. O próximo passo lógico é tentar agrupar regras similares, com o objetivo de compactar a representação das mesmas, facilitando a sua compreensão, o que será feito a seguir.

Pensando-se superficialmente, é possível cometer o equívoco do exemplo mostrado na Figura 4.1. O resultado do agrupamento de regras contíguas em todas as dimensões ou contíguas em mais de uma dimensão e igual nas outras, não equivale às regras separadamente. Isto pode ser visto no exemplo da seguinte maneira: com as duas regras separadas, o pacote $[1] \wedge [10] \wedge [10]$ não casaria com nenhuma das regras, no entanto, a regra resultante do agrupamento aceitaria este pacote.

$R_1 : [1,5] \wedge [1,5] \rightarrow$ **accept**
$R_2 : [6,10] \wedge [6,10] \rightarrow$ **accept**
$R_{1_2} \neq [1,10] \wedge [1,10] \rightarrow$ **accept**

Figura 4.1: Erro de agrupamento: regras contíguas em todas as dimensões.

Assim, é necessário permitir o agrupamento apenas de regras que cumpram condições bem específicas. Para isto, é necessário definir similaridade entre regras. Duas regras A e B são similares, se e somente, apenas um de seus campos forem diferentes e elas possuírem ações iguais, como mencionado em (ACHARYA et al., 2006). Formalizando: duas regras A e B são consideradas **similares** (e podem, assim, ser agrupadas) se, e somente se,

$$(\exists i \;\; A.c_i \neq B.c_i) \wedge (\forall_{1 \leq j \leq d}, (j \neq i) \wedge (A.c_j = B.c_j)) \wedge (A.\text{ação} = B.\text{ação})$$

Esta definição funciona, pois quando as regras estão descorrelacionadas, elas podem ser vistas como uma disjunção, já que um pacote casará necessariamente apenas com uma delas. Da definição de regra de filtragem e do fato de $\wedge$ ser distributivo em relação a $\vee$ $(((a \wedge b \wedge c) \vee (a \wedge b \wedge d)) \iff (a \wedge b \wedge (c \vee d)))$, segue a definição de similaridade acima.

O problema a ser resolvido é, então, o seguinte: dado um conjunto de regras descorrelacionadas U, agrupar as regras similares de U em uma ordem que gere um conjunto

de regras compacto. O conceito de "compacto", neste trabalho, será utilizado de maneira simples: um conjunto de regras é **compacto** quando não houver regras similares nele.

A fim de trabalhar o problema de agrupamento de regras, a estrutura de grafo se mostra, mais uma vez, bastante útil. Abaixo é definido o grafo 1-dif, que explicita quando duas regras possuem apenas um campo diferente (ou seja, as duas regras são similares).

4.1 O Grafo 1-dif

O grafo 1-dif é um grafo G1dif $= (V, E)$, sendo que V é um conjunto de regras e E é a relação binária na qual dois vértices $(v_i, v_j) \in E$, se e somente se, as regras v_i e v_j são similares. O rótulo da aresta $e = (v_i, v_j)$ é o nome do campo que difere entre as regras v_i e v_j. O Algoritmo 7 mostra como gerar este tipo de grafo a partir de um conjunto de regras descorrelacionadas U. A função `contaDiferencas`(v_i, v_j) usada no algoritmo computa a quantidade de campos diferentes entre as regras representadas pelos vértices v_i e v_j.

Entrada: Conjunto de regras descorrelacionadas U.
Saída: Grafo 1-dif G1dif $= (V, E)$ do U dado.

```
1 para k ← 1 até |U| faça
2     para l ← 1 até k − 1 faça
3         se contaDiferencas (k, l) = 1 então
4             acrescenta aresta (k, l) em E
5             rotula aresta (k, l) com o campo onde há diferença
6         fim
7     fim
8 fim
```

Algoritmo 7: Algoritmo de Geração de Grafo 1-dif

$$R_1 : [1] \wedge [80] \rightarrow \textbf{accept}$$
$$R_2 : [1] \wedge [21] \rightarrow \textbf{accept}$$
$$R_3 : [2] \wedge [80] \rightarrow \textbf{accept}$$
$$R_4 : [3] \wedge [80] \rightarrow \textbf{accept}$$

Figura 4.2: Exemplo de conjunto de regras com múltiplas possibilidades de agrupamento.

Pelo exemplo das Figuras 4.2 e 4.3, pode-se perceber claramente que, como mencionado anteriormente, há mais de uma possibilidade de agrupamento de regras. No caso, R_1 pode ser agrupada tanto com R_2 (opção destacada na Figura 4.3 por linha pontilhada) quanto com R_3 e R_4.

Com o grafo 1-dif, o problema pode ser redefinido da seguinte maneira: dado um grafo 1-dif, agrupar vértices adjacentes de forma a gerar um conjunto compacto de regras. Uma observação de fundamental importância ao resolver este problema é que, ao se agrupar uma regra com outra em um campo c_i, todas as outras possibilidades de agrupamento com campos c_j diferentes de c_i são eliminadas de ambas as regras, visto que elas mudaram. No grafo, isto significa que o novo nó gerado a partir do agrupamento mantém

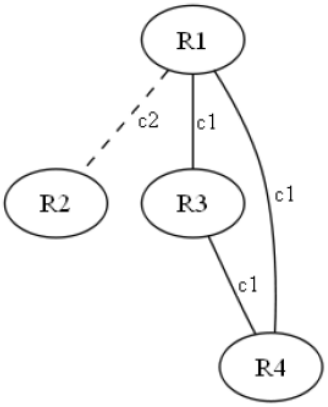

Figura 4.3: Grafo 1-dif das regras na Figura 4.2.

apenas as arestas rotuladas com $nome(c_i)$. Como podem surgir novas possibilidades de agrupamento do nó agrupado, o ideal é recalcular estas possibilidades para o novo nó. A seguir, são descritos dois algoritmos diferentes para, a partir de regras descorrelacionadas, gerar um conjunto de regras compacto.

4.2 Agrupamento Sequencial

Nesta seção, é descrito um algoritmo de agrupamento parecido com o utilizado em (ACHARYA et al., 2006) e (ABEDIN et al., 2006). Ele é mostrado no Algoritmo 8. Como pode ser visto, para cada regra U_i do conjunto de regras descorrelacionadas U de entrada, o algoritmo, enquanto possível, tenta agrupar U_i às regras $U_j (j \neq i)$ similares, na ordem sequencial em que elas se encontram em U (que pode ser arbitrária, já que U está descorrelacionado). Como se pode notar pelo uso da variável *haSimilares*, sempre que houver um agrupamento, todo o conjunto U deve ser percorrido novamente a fim de encontrar novas similaridades. Assim, a importante observação realizada acima é respeitada pelo algoritmo.

Entrada: Conjunto de regras descorrelacionadas U.
Saída: Conjunto de regras em mais alto nível.

```
1  para i ← 1 até |U| faça
2      haSimilares ← TRUE
3      enquanto haSimilares ≠ FALSE faça
4          haSimilares ← FALSE
5          para j ← 1 até |U| faça
6              se (i ≠ j) ∧ similar(U_i, U_j) ∧ (U_k.acao = U_l.acao) então
7                  agrupar U_j a U_i
8                  haSimilares ← TRUE
9              fim
10         fim
11     fim
12 fim
```

Algoritmo 8: Algoritmo Sequencial de Agrupamento

Aplicando este algoritmo ao conjunto de regras da Figura 3.5 resulta no conjunto de regras agrupado na Figura 4.4. Pode-se notar que este exemplo unidimensional é o melhor caso possível para o algoritmo, pois todas as regras com mesma ação são similares. A aplicação deste algoritmo ao conjunto de regras da Figura 4.2 é mais realístico e seu resultado se encontra na Figura 4.5.

Antes:
T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $\{[4], [6, 10]\} \rightarrow$ **block**
T_4: $[25] \rightarrow$ **block**
T_5: $\{[20, 24], [26, 30]\} \rightarrow$ **accept**
T_6: $\{[1, 2], [11, 19], [31, 1000]\} \rightarrow$ **block**

Depois:
T_1: $\{[3], [5], [20, 24], [26, 30]\} \rightarrow$ **accept**
T_3: $\{[1, 2], [4], [6, 19], [25], [31, 1000]\} \rightarrow$ **block**

Figura 4.4: Exemplo de geração de conjunto de regras compacto.

Entrada:
$R_1 : [1] \wedge [80] \rightarrow$ **accept**
$R_2 : [1] \wedge [21] \rightarrow$ **accept**
$R_3 : [2] \wedge [80] \rightarrow$ **accept**
$R_4 : [3] \wedge [80] \rightarrow$ **accept**

Saída:
$R_{1,2} : [1] \wedge \{[21], [80]\} \rightarrow$ **accept**
$R_{3,4} : \{[2], [3]\} \wedge [80] \rightarrow$ **accept**

Figura 4.5: Exemplo de aplicação do Algoritmo Sequencial de Agrupamento.

4.3 Agrupamento Guloso

Com o objetivo de agrupar o maior número de regras similares com mesmo campo, a seguir, é apresentado um algoritmo guloso. O Algoritmo 9 processa as regras a partir de um grafo 1-dif. Enquanto houver arestas, isto é, enquanto for possível agrupar regras, o algoritmo procura o vértice com maior número de possibilidades de agrupamento com o mesmo campo. Todos estes agrupamentos com o mesmo campo são realizados, e o vértice resultante tem suas arestas recalculadas. A função `contaArestas`(v,*campo*) computa quantas arestas rotuladas com campo *campo* saem do vértice v.

```
   Entrada: Conjunto de regras R e seu grafo 1-dif G1dif = (V, E).
   Saída: Conjunto de regras em mais alto nível.
   /* Enquanto há arestas                                          */
 1 enquanto |E| > 1 faça
 2     maior ← −1
 3     para cada v ∈ V faça
 4         para cada campo em R_v faça
 5             tmp ← contaArestas (v,campo)
 6             se tmp > maior então
 7                 tmp ← maior
 8                 vEscolhido ← v
 9                 campoEscolhido ← campo
10             fim
11         fim
12         para cada aresta a = (vEscolhido , x) rotulada com campoEscolhido
           faça
13             agrupar vEscolhido e x
14             eliminar vértice x com todas suas arestas
15         fim
16         eliminar todas as arestas de vEscolhido
17         recalcular arestas de vEscolhido
           /* o cálculo é realizado como no loop interno do
              Algoritmo 7, mas percorrendo todos os vértices
              */
18     fim
19 fim
```

Algoritmo 9: Algoritmo Guloso de Agrupamento

A Figura 4.6 mostra o resultado do agrupamento guloso do conjunto de regras na Figura 4.2. Pode-se perceber que, neste exemplo, o agrupamento faz uma escolha diferente do algoritmo sequencial de como agrupar R_1. O conjunto gerado possui uma regra com o maior agrupamento possível.

O algoritmo guloso apresentado pode ser útil quando se deseja agrupar ao máximo regras similares. Uma possível aplicação poderia ser a criação de *chains* em *firewalls* que suportam este recurso como o *iptables/Netfilter*.

4.4 Descorrelacionamento e Agrupamento de Regras

Após a apresentação do descorrelacionamento (capítulo 3) e do agrupamento de regras descorrelacionadas (capítulo atual), torna-se possível descrever, de forma resumida, a metodologia para extração, em mais alto nível, de políticas de *firewall* utilizada atualmente. O procedimento abaixo é usado, de modo indireto, em (ABEDIN et al., 2006). Tanto (QIAN; HINRICHS; NAHRSTEDT, 2001) quanto (TONGAONKAR; INAMDAR; SEKAR, 2007) possuem uma abordagem bastante parecida, que será discutida na seção 5.1.2.

A metodologia condensa o que foi discutido até o momento e consiste de duas etapas:

1. Dado um conjunto de regras correlacionadas C ordenado por prioridade, um dos

Entrada:
$R_1 : [1] \wedge [80] \rightarrow$ **accept**
$R_2 : [1] \wedge [21] \rightarrow$ **accept**
$R_3 : [2] \wedge [80] \rightarrow$ **accept**
$R_4 : [3] \wedge [80] \rightarrow$ **accept**

Saída:
$R_{1,3,4} : \{[1], [2], [3]\} \wedge [80] \rightarrow$ **accept**
$R_2 : [1] \wedge [21] \rightarrow$ **accept**

Figura 4.6: Exemplo aplicação do Algoritmo Guloso de Agrupamento.

algoritmos de descorrelacionamento da seção 3.4 (como o Algoritmo 6), é aplicado a ele, gerando um conjunto de regras descorrelacionadas U semanticamente equivalente a C;

2. As regras de U são agrupadas, quando possível. Isto gera regras compactas representando a política real e, por conseguinte, de mais alto nível.

Um exemplo de aplicação desta metodologia encontra-se na Figura 5.14.

5 PROPOSTA DE METODOLOGIA PARA EXTRAÇÃO DE POLÍTICAS DE *FIREWALL*

A seguir, é proposta uma metodologia original para extração de políticas de *firewall*, baseado no descorrelacionamento hierárquico e na modelagem de *blacklist* e *whitelist*.

5.1 Extração Hierárquica

A metodologia de extração mostrada na seção 4.4, apesar de mostrar a política real de maneira mais legível, ainda trabalha exclusivamente no nível de regras de filtragem. Nesta seção, será mostrado um algoritmo de extração que, através do uso de um simples modelo, traduz a política resultante para um nível mais alto. O modelo é baseado na hierarquia entre regras que, por sua vez, é baseado na relação de "estar contido" entre regras.

5.1.1 Motivação

A motivação para a criação deste modelo vem de observações sobre a estrutura de conjuntos de regras utilizados na realidade. Quando um administrador configura um *firewall*, ele normalmente utiliza grupos de regras (ou *clusters*, como em (KOUNAVIS et al., 2003)) para realizar determinada política de *firewall*. Normalmente, ao final do grupo, há uma regra genérica que contém todas as regras deste grupo e que determina o que deve ser realizado com os pacotes relacionados a política de *firewall* representada por este grupo, mas que não são especificados nas regras do mesmo. Por exemplo, a Figura 5.1 mostra a tradução para baixo nível da política "bloquear todo tráfego para a rede cujo endereço das máquinas está entre 1 e 10, com exceção da porta 80 para os *hosts* 1,5 e 9". Desta forma, esta estruturação de regras é bastante comum, especialmente quando o *firewall* protege múltiplas redes e estas possuem vários serviços. A utilização de uma regra genérica para cada grupo de regras (e não uma única para todos), se dá por dois motivos básicos:

- O uso de uma regra genérica por grupo previne erros, pois se for usada apenas uma regra genérica global, caso a ação desta regra genérica seja modificada por engano, vários "furos" aparecerão no *firewall*, deixando a rede interna desprotegida;

- Em boa parte dos *firewalls* como *iptables/Netfilter*, as regras são testadas sequencialmente, por ordem de prioridade. Desta forma, quanto antes uma regra casar com o dado pacote, menos comparações precisarão ser feitas e mais eficiente será o *firewall* (como discutido em (ZHANG; ZHANG; WANG, 2005)). Assim, neste caso,

é melhor colocar as regras genéricas o quanto antes no conjunto de regras ordenado e não apenas no final do mesmo.

Somando-se a isso, o descorrelacionamento das regras genéricas como R_4 na Figura 5.1 normalmente gera vários fragmentos, ou no caso, uma regra com um campo fragmentado ($\{[2,4],[6,8],[10]\} \wedge [80] \rightarrow$ **block**). Este tipo de fragmentação dificulta a leitura. O modelo hierárquico apresentado aqui resolve este problema.

$R_1 : [1] \wedge [80] \rightarrow$ **accept**
$R_2 : [5] \wedge [80] \rightarrow$ **accept**
$R_3 : [9] \wedge [80] \rightarrow$ **accept**
$R_4 : [1,10] \wedge [80] \rightarrow$ **block**

Figura 5.1: Exemplo de grupo de regras que representa políticas de *hosts* liberados na porta 80 da rede composta por *hosts* de endereço de 1 a 10.

Como exemplo, na Figura 3.4, as regras T_1 e T_2 foram, provavelmente, configuradas pelo administrador como exceções à regra mais genérica representada em T_3. No entanto, seria interessante representar estas regras como mostrado na Figura 5.2, com T_1 e T_2 sendo hierarquicamente inferiores a T_3. É importante observar que as regras de "exceção" (chamadas de regras de subconjunto) possuem maior prioridade do que sua regra raiz, já que elas não estão descorrelacionadas.

Antes:
T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $[3,10] \rightarrow$ **block**

Depois:
T_3: $[3,10] \rightarrow$ **block**
{
    T_1: $[3] \rightarrow$ **accept**
    T_2: $[5] \rightarrow$ **accept**
}

Figura 5.2: Representação hierárquica do conjunto de regras: a regra T_3 bloqueia $[3,10]$ com exceção de 3 e 5.

5.1.2 Algoritmos

Diz-se que uma regra A **é subconjunto** da regra B, se e somente se, cada campo de A é um subconjunto do campo correspondente de B. Mais formalmente,

$$A \subset B \iff \forall_{1 \leq i \leq d}, A.c_i \subset B.c_i$$

Usando esta definição, se A e B possuem ações diferentes A pode ser vista como uma *exceção* da regra B. Quando as ações são iguais, as regras em questão são redundantes. O uso desta relação é intuitiva para o administrador que configura o *firewall* e, muito provavelmente, muitas regras correlacionadas são o resultado deste caso, como discutido acima.

O Algoritmo 10 mostra como computar o **descorrelacionamento hierárquico**, que é uma nova maneira de se fazer descorrelacionamento (e, consequentemente, extração de políticas) explorando a relação de "subconjunto" definida acima. Para este modelo, cada regra possui uma lista de regras de subconjunto associada a ela própria. Em um *conjunto de regras hierarquicamente descorrelacionado*, todas as regras estão descorrelacionadas. No entanto, regras que contêm outras regras na lista de subconjuntos são mantidas, propositalmente, correlacionadas. Neste caso, as regras na lista de subconjuntos possuem maior prioridade que a regra raiz. É importante notar que as regras que estão na lista de subconjuntos de outra regra podem, também, possuir uma lista de subconjuntos, o que origina uma estrutura de árvore.

O Algoritmo 10 se inicia simplesmente copiando a primeira regra do conjunto correlacionado C ao conjunto de regras descorrelacionadas de saída U, já que ela é a regra com maior prioridade e, desta forma, não está correlacionada. A seguir, cada regra subsequente em C é processada, com uma fila sendo utilizada para armazenar os fragmentos da regra de C atual. Se o fragmento atual já está descorrelacionado, ele pode ser adicionado a U. Caso contrário, a primeira regra com a qual ele está correlacionado (U_k) é encontrada e descorrelacionada com ele em uma de três maneiras possíveis:

1. Se U_k está contida na regra atual e suas ações são diferentes, U_k é adicionada à lista de subconjuntos da regra atual;

2. Se U_k está contida na regra atual, mas suas ações são iguais, U_k é redundante e pode ser descartada. No entanto, no algoritmo mostrado, a regra atual, se possui uma lista de subconjuntos, é marcada como redundante e adicionada à lista de subconjuntos da regra atual. Caso contrário, ela é descartada;

3. Se U_k não está contida na regra atual, esta é descorrelacionada em relação U_k (usando a função no Algoritmo 5, por exemplo) e seus fragmentos são colocados na fila.

Depois de a fila ser esvaziada para a regra atual, a próxima regra em C é processada. Após todas as regras de C terem sido processadas, o conjunto de saída U resultante está hierarquicamente descorrelacionado.

Há algumas sutilezas neste algoritmo. Em primeiro lugar, a verificação de correlacionamento e de "estar contido" é realizada em relação a U, que já está certamente descorrelacionado. Isto evita muitos problemas, especialmente aqueles relacionados ao fato de manter a lista de subconjuntos de cada regra descorrelacionada internamente. Em segundo lugar, como mencionado, há duas maneiras básicas de tratar redundância de regras: marcando-as ou descartando-as. Aqui, preferiu-se apenas marcar regras redundantes que tenham lista de subconjuntos, a fim de não perder nenhuma informação. É importante notar que, se fosse escolhida a opção de eliminar regras redundantes, seria necessário manter suas listas de subconjuntos, já que estas se manteriam relevantes. Em terceiro lugar, existe um caso que o algoritmo apresentado não trata explicitamente. Isto ocorre quando as regras da lista de subconjuntos, se agrupadas, possuírem a mesma condição que a regra raiz.

```
   Entrada: Conjunto de regras correlacionadas C, ordenado por prioridade.
   Saída: Conjunto de regras hierarquicamente descorrelacionadas U.
 1 U_1 ← C_1
 2 para j ← 2 até |C| faça
 3     Fila.push (C_j)
 4     enquanto Fila.size > 0 faça
 5         regraAtual ← Fila.pop ()
 6         se estaDescorrelacionada (regraAtual, U) então
 7             adiciona regraAtual a U
 8         senão
 9             k ← encontraCorrelacionada (regraAtual, U)
10             se U_k ⊂ regraAtual então
                   /* Redundância                                    */
11                 se U_k.ação = regraAtual.ação então
12                     se U_k.subconjuntos.size > 0 então
13                         marca U_k como redundante
14                         regraAtual.subconjuntos.add (U_k)
15                         Fila.push (regraAtual)
16                     fim
17                 senão
18                     regraAtual.subconjuntos.add (U_k)
19                     Fila.push (regraAtual)
20                 fim
21                 remove U_k de U
22             senão
23                 tmp ←descorrelacionaRegra (regraAtual, U_k)
24                 para cada fragRegra ∈ tmp faça
25                     Fila.push (fragRegra)
26                 fim
27             fim
28         fim
29     fim
30 fim
```

Algoritmo 10: Algoritmo de Descorrelacionamento Hierárquico

Neste caso, a regra raiz deveria ter sua ação alterada para a mesma das regras da sua lista de subconjuntos e a lista descartada. Um exemplo onde este caso problemático ocorre é a Figura 5.3. Este caso é resolvido no Algoritmo 11.

Um exemplo da aplicação do Algoritmo 10 de Descorrelacionamento Hierárquico ao conjunto de regras da Figura 3.4 é mostrado na Figura 5.4. No exemplo, é possível perceber claramente a hierarquia de regras genéricas assim como, provavelmente, pensou o administrador ao formular a política do *firewall*.

A fim de agrupar um conjunto de regras descorrelacionadas hierarquicamente, os algoritmos do capítulo 4 não são suficientes, pois com as listas de subconjuntos, a estrutura que está sendo tratada é recursiva. Desta forma, o algoritmo de agrupamento também deve ser recursivo. O Algoritmo 11 agrupa recursivamente as regras e suas listas de subconjuntos. Após o agrupamento das regras na lista de subconjuntos de uma dada regra, o caso

Entrada:
T_1: $[3] \rightarrow$ **accept**
T_2: $[4] \rightarrow$ **accept**
T_3: $[3, 4] \rightarrow$ **block**

Saída:
T_3: $[3, 4] \rightarrow$ **block**
{
    T_1: $[3] \rightarrow$ **accept**
    T_2: $[4] \rightarrow$ **accept**
}

Figura 5.3: Caso especial do Algoritmo 10 resolvido pelo Algoritmo 11.

da condição da lista de subconjuntos ser igual a da regra raiz é tratado. Este caso acontece quando a lista agrupada de subconjuntos for uma regra com exatamente os mesmos campos que sua regra raiz.

O uso de um agrupamento recursivo é justificável, pois, após o agrupamento de duas regras, suas listas de subconjuntos são unidas e pode haver novas possibilidades de agrupamento entre estas regras na lista de subconjuntos unida. Desta forma, a função `agrupa` utilizada no algoritmo pode ser como uma das descritas no capítulo 4, com o acréscimo de que, quando duas regras são agrupadas, suas respectivas listas de subconjuntos são unidas, se existentes.

Entrada: Conjunto de regras hierarquicamente descorrelacionadas U.
Saída: Conjunto de regras hierarquicamente descorrelacionadas agrupado U.

```
1  agrupamentoRecursivo (H)
2  início
3      se |H| = 0 então return
4      agrupa (H)
5      para k ← 1 até |H| faça
6          se H_k.subconjuntos.size > 1 então
7              agrupamentoRecursivo (H_k.subconjuntos)
8              se H_k.subconjuntos.size = 1 então
9                  tmp ← H_k.subconjuntos
10                 se ∀_{1≤i≤d}, H_k.c_i = tmp.c_i então
11                     H_k ← tmp
12                 fim
13             fim
14         fim
15     fim
16 fim
```

Algoritmo 11: Algoritmo de Agrupamento Recursivo

A Figura 5.5 mostra uma aplicação do descorrelacionamento hierárquico seguida de

Entrada:
T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $[3, 10] \rightarrow$ **block**
T_4: $[25] \rightarrow$ **block**
T_5: $[20, 30] \rightarrow$ **accept**
T_6: $[1, 1000] \rightarrow$ **block**

Saída:
T_6: $[1, 1000] \rightarrow$ **block**
{
    T_3: $[3, 10] \rightarrow$ **block** !Redundante!
    {
        T_1: $[3] \rightarrow$ **accept**
        T_2: $[5] \rightarrow$ **accept**
    }
    T_5: $[20, 30] \rightarrow$ **accept**
    {
        T_4: $[25] \rightarrow$ **block**
    }
}

Figura 5.4: Exemplo de aplicação do Algoritmo 10 de Descorrelacionamento Hierárquico.

uma aplicação do agrupamento recursivo. Sem o uso de recursão, as regras R_1 e R_3 não teriam sido agrupadas.

Como mencionado anteriormente, (QIAN; HINRICHS; NAHRSTEDT, 2001) e (TONGAONKAR; INAMDAR; SEKAR, 2007) convertem o conjunto de regras de forma que as resultantes possuem apenas um tipo de ação (aceitar ou bloquear), com uma regra genérica implícita que possui a ação oposta para todo tráfego não mostrado nas regras explícitas. Este procedimento pode facilitar o entendimento de o que é bloqueado ou o que é aceito pelo *firewall*. Por exemplo, aplicando esta transformação (que consiste em descorrelacionar regras com ações diferentes e descartar, depois, as regras que não possuem a ação desejada) às regras da Figura 3.4, produz-se uma representação (Figura 5.6) bastante reduzida de apenas o que é aceito . No entanto, esta abordagem não é tão genérica quanto a proposta aqui. Isto ocorre por vários motivos:

1. A conversão para um conjunto de regras com apenas um tipo de ação descarta informações úteis. Regras com a ação oposta são removidas , o que faz com que seja difícil verificar o efeito que elas possuem no *firewall*. Em contraste, a abordagem apresentada aqui mantém o máximo de informação possível, fazendo com que seja mais fácil para o administrador perceber a associação entre regras de baixo nível e regras na representação hierárquica;

2. A conversão de regras com múltiplas ações para regras com apenas um tipo de ação nem sempre produz uma representação clara do conjunto de regras original. Isto

Entrada:
R_1: $[20] \wedge [80] \rightarrow$ **accept**
R_2: $[1, 100] \wedge [80] \rightarrow$ **block**
R_3: $[20] \wedge [21] \rightarrow$ **accept**
R_4: $[1, 100] \wedge [21] \rightarrow$ **block**

Após descorrelacionamento hierárquico:
R_2: $[1, 100] \wedge [80] \rightarrow$ **block**
{
    R_1: $[20] \wedge [80] \rightarrow$ **accept**
}
R_4: $[1, 100] \wedge [21] \rightarrow$ **block**
{
    R_3: $[20] \wedge [21] \rightarrow$ **accept**
}

Após agrupamento recursivo:
$R_{2,4}$: $[1, 100] \wedge \{[21], [80]\} \rightarrow$ **block**
{
    $R_{1,3}$: $[20] \wedge \{[21], [80]\} \rightarrow$ **accept**
}

Figura 5.5: Exemplo de aplicação do Descorrelacionamento Hierárquico e do Agrupamento Recursivo.

ocorre principalmente quando há muitas regras específicas com a ação oposta da desejada. Por exemplo, se na Figura 3.4 T_6 fosse uma regra de aceitação, após a transformação ela se tornaria $\{[1, 2], [11, 24], [31, 1000]\} \rightarrow$ **accept**. Em contraponto a isso, a saída da representação hierárquica não mudaria. Este problema fica pior quando o número de campos aumenta;

3. O descorrelacionamento de regras com múltiplas ações para regras com apenas uma ação está em um nível mais baixo do que a representação hierárquica. Isto pode ser justificado da seguinte maneira: a saída do descorrelacionamento hierárquico proposto aqui pode ser facilmente convertida para a representação com apenas um tipo de ação, mas o oposto não é verdadeiro. Em geral, não é possível converter a representação com apenas uma ação para a representação hierárquica, pois, como mencionado acima, informações são perdidas quando regras de ações diferentes são removidas;

4. Bartal et al em (BARTAL et al., 2004), após extensa pesquisa como descrita em 2.1.2, afirma que o método mais natural para um administrador de *firewall* excluir *hosts* ou serviços é utilizando "block". Quando se convertem todas as regras para regras com apenas a ação "accept", as regras de bloqueio são eliminadas, o que não acontece no descorrelacionamento hierárquico.

A seguir são modelados *hosts* em *blacklists* (listas negras) e *whitelists* (listas brancas),

Antes:
T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_3: $[3, 10] \rightarrow$ **block**
T_4: $[25] \rightarrow$ **block**
T_5: $[20, 30] \rightarrow$ **accept**
T_6: $[1, 1000] \rightarrow$ **block**

Depois:
T_1: $[3] \rightarrow$ **accept**
T_2: $[5] \rightarrow$ **accept**
T_5: $\{[20, 24], [26, 30]\} \rightarrow$ **accept**

Figura 5.6: Exemplo de descorrelacionamento para apenas um tipo de ação (aceitação).

a fim de melhorar o descorrelacionamento e extração de políticas através da redução de fragmentação dentro de conjuntos de regras.

5.2 Modelagem de Regras de *Blacklist* e *Whitelist* em *Firewall*

Um problema com algoritmos de descorrelacionamento em geral ocorre quando regras estão correlacionadas com um grande número de outras regras e, especificamente do descorrelacionamento hierárquico, quando estas regras correlacionadas não possuem a relação de "está contido" entre elas. Neste caso, muitas regras são quebradas, criando fragmentos que fazem a visualização da política ser mais difícil. Atualmente, um dos casos mais comuns para a existência destas regras correlacionadas está relacionado a *blacklist* e *whitelist*. Este caso problemático, é a exceção mencionada na seção 3.1.1.

No contexto deste trabalho, uma *blacklist* é uma lista que contém *hosts* ou redes (possivelmente maliciosos) que sempre deveriam ser bloqueados no *firewall*. Por outro lado, uma *whitelist* contém *hosts* ou redes que sempre deveriam possuir permissão de acesso.

A fim de definir regras de *blacklist* e regras de *whitelist* em um dado conjunto de regras correlacionadas, assume-se que um dos campos em uma regra é denominado *src*, representando o endereço de origem do pacote e outro campo é denominado *dst*, representando o endereço de destino do pacote. Em geral, uma **regra de *blacklist*** (**regra de *whitelist***) pode ser definida de maneira heurística como segue:

- Uma regra que possui um endereço de origem (src) ou de destino (dst) especificado e que pode possuir qualquer valor do domínio nos outros campos (isto é, possui * nos outros campos). Enquanto regras de *blacklist* possuem como ação **bloquear**, regras de *whitelist* possuem como ação **aceitar**;

- A regra encontra-se na metade superior do conjunto de regras ordenado por prioridade.

Nota-se que a heurística "na metade superior" é uma aproximação um pouco grosseira, mas funciona na maioria dos casos, pois para uma regra ser parte de uma *blacklist* ou

whitelist, ela precisa estar antes das demais (isto é, a regra precisa possuir maior prioridade). Este fato torna-se importante, pois as regras que fazem parte destas listas não serão necessariamente descorrelacionadas. Em caso de necessidade, esta heurística do posicionamento da regra pode ser alterada sem afetar o restante da modelagem mostrada.

Na Figura 5.7, as regras B_1 e B_2 representam um típico exemplo de uma regra que coloca o *host* $[3]$ em uma *blacklist*. É importante notar que quaisquer destas regras abaixo que contém * nos campos *src* ou em *dst* estarão correlacionadas com elas, como a regra B_{50}, que mostra uma tipica liberação de serviço (porta 80) para um servidor com o endereço $[8]$.

B_1: $src = [3] \wedge dst =$* $\wedge\ dport =$* $\rightarrow$ **block**
B_2: $src =$* $\wedge\ dst = [3] \wedge\ dport =$* $\rightarrow$ **block**
$\vdots$
B_{50}: $src =$* $\wedge\ dst = [8] \wedge\ dport = [80] \rightarrow$ **accept**
$\vdots$
mais regras

Figura 5.7: Exemplo de um *host* (com endereço 3) em uma *blacklist*.

A fim de mostrar na prática os benefícios da extração de regras de *blacklist* ou *whitelist*, a Figura 5.8 mostra o grafo de correlacionamento do conjunto de regras na Figura 5.15. Pode-se notar que o grafo é bastante denso e que várias regras precisariam ser descorrelacionadas. A Figura 5.9, em contrapartida, mostra o mesmo grafo de correlacionamento, mas gerado após a extração das regras com *blacklist* ou *whitelist*. O grafo gerado é muito mais tratável com uma quantidade muito menor de correlacionamentos.

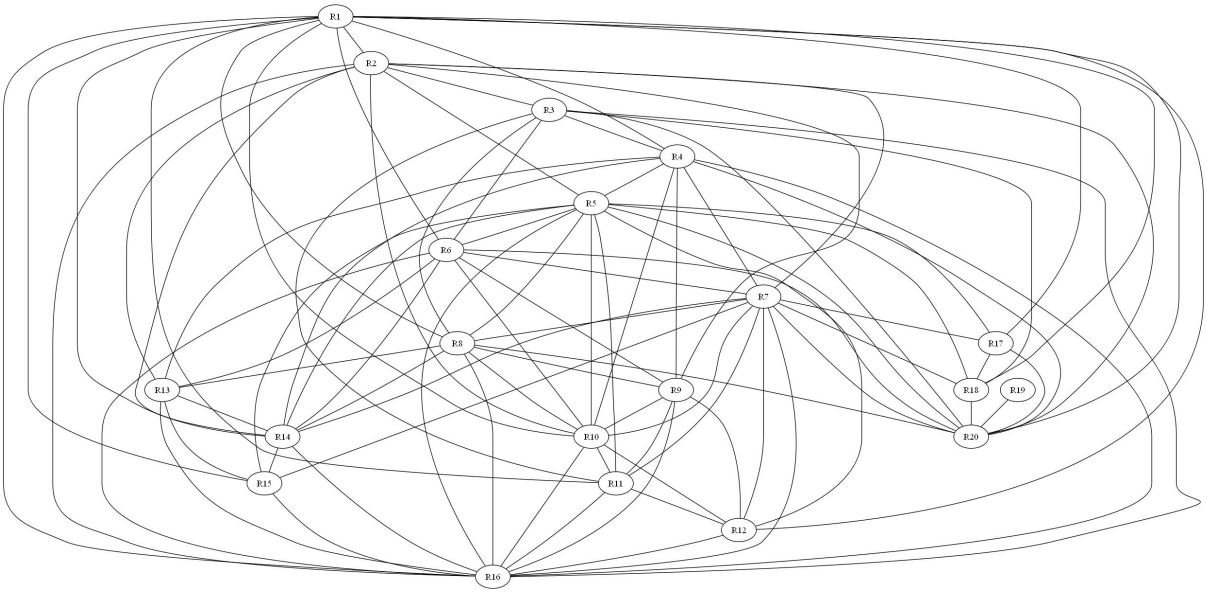

Figura 5.8: Grafo de correlacionamento das regras na Figura 5.15

Uma vez identificadas, as regras que fazem parte de *blacklist* ou *whitelist* serão separadas das demais, não participando do descorrelacionamento. Desta forma mantém-se

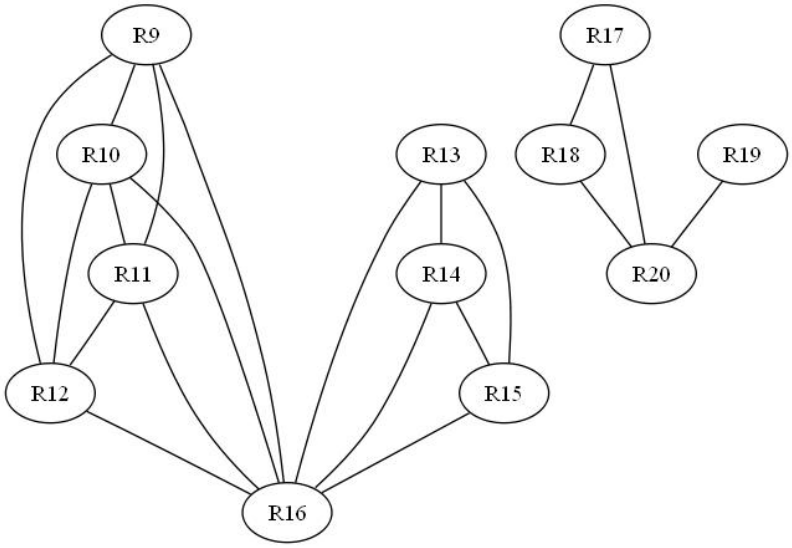

Figura 5.9: Grafo de correlacionamento com separação de *blacklist* e *whitelist* das regras na Figura 5.15

o correlacionamento entre as regras de *blacklist* ou *whitelist* e o resto das regras com o intuito de aumentar o nível da política resultante. Para resolver possíveis conflitos, utiliza-se a maneira usual: regras de *blacklist* ou *whitelist* possuem maior prioridade do que o resto das regras. Esta separação ou extração das regras de *blacklist* ou *whitelist* pode ser realizada de maneiras diferentes. A seguir serão mostradas duas maneiras alternativas de como extrair regras de *blacklist* ou *whitelist*, sendo que suas vantagens e desvantagens serão discutidas.

5.2.1 Descorrelacionando *Blacklist* e *Whitelist*

O primeiro modo de extrair regras de *blacklist* é separá-las das demais, descorrelacionando-as das regras de maior prioridade. Isto mantém as regras de *blacklist* e de *whitelist* descorrelacionadas (isto é, mostrando somente os pacotes que realmente serão casados com as respectivas regras). É importante ressaltar que, neste caso, o descorrelacionamento precisa ser realizado apenas em regras com ações diferentes. No entanto, esta abordagem pode tornar a visualização da política mais difícil. Um exemplo encontra-se na Figura 5.11.

R_1: $src = [0, 10] \wedge dst =$* $\wedge\ dport = [80] \rightarrow$ **accept**
R_2: $src = [5] \wedge dst =$* $\wedge\ dport =$* $\rightarrow$ **block**
R_3: $src =$* $\wedge\ dst = [7] \wedge dport =$* $\rightarrow$ **accept**
$\vdots$
mais regras

Figura 5.10: Conjunto de regras utilizado nos exemplos de extração de *blacklist* e *whitelist*.

***Blacklist*:**
$src = [5] \wedge dst =$* $\wedge\ dport = \{[0, 79], [81, 2^{32} - 1]\} \rightarrow$ **block**

***Whitelist*:**
$src = [11, 2^{32} - 1] \wedge dst = [7] \wedge dport =$* $\rightarrow$ **accept**
$src = \{[0, 4], [6, 10]\} \wedge dst = [7] \wedge dport = \{[0, 79], [81, 2^{32} - 1]\} \rightarrow$ **accept**

Figura 5.11: Exemplo de extração descorrelacionando *blacklist* e *whitelist* a partir das regras da Figura 5.10.

5.2.2 Mantendo a Correlação entre *Blacklist* e *Whitelist*

Outra abordagem para extração de regras de *blacklist* ou *whitelist* é, ao separá-las, mantê-las correlacionadas. A fim de não gerar uma política em mais alto nível incorreta e não intuitiva, é necessário acrescentar mais duas restrições à extração:

1. As regras extraídas não podem estar contidas em nenhuma regra ou agrupamento de regras de maior prioridade. Se uma regra de *blacklist* ou *whitelist* estiver contida em outra regra qualquer de maior prioridade, ela simplesmente não deve ser representada na política real, já que nenhum pacote casará com ela. Conceitualmente, esta restrição é necessária exatamente pelo fato de o descorrelacionamento não ser utilizado;

2. Regras de *whitelist* e *blacklist* são extraídas apenas quando não possuem nenhuma outra regra como subconjunto. Isto evita que regras mais genéricas sejam tratadas como regras de *blacklist* ou *whitelist*.

São descritas, agora, duas opções de extração de regras de *blacklist* ou *whitelist* mantendo-as correlacionadas.

A primeira é não separar as regras de *blacklist* das de *whitelist* e manter ambas as regras na ordem de prioridade em que aparecem, sem descorrelacioná-las. Isto pode permitir mais legibilidade, mas o preço é manter *whitelist* e *blacklist* misturadas. Um exemplo é mostrado na Figura 5.12.

***Blacklist e Whitelist*:**
R_2: $src = [5] \wedge dst =$* $\wedge\ dport =$* $\rightarrow$ **block**
R_3: $src =$* $\wedge\ dst = [7] \wedge dport =$* $\rightarrow$ **accept**

Figura 5.12: Exemplo de extração mantendo a correlação de *blacklist* e *whitelist* a partir das regras da Figura 5.10.

A segunda é separar as regras de *whitelist* das de *blacklist* e não descorrelacioná-las. Isto permite maior legibilidade, mas compromete, de uma certa forma, a extração da política real, visto que correlacionamentos entre regras de *blacklist* e *whitelist* são simplesmente ignorados. Nesta abordagem, é viável realizar mais um refinamento. Com

base na observação de que um endereço em uma *blacklist* deve ser sempre bloqueado, é natural que os administradores criem, para cada endereço, duas regras de filtragem: uma que possui apenas o campo src especificado e, logo depois, uma que possui apenas o campo dst especificado. Nesta situação, a partir destas duas regras A e B é formada outra em mais alto nível que afirma $src = A.c_{src} \vee dst = B.c_{dst} \rightarrow$ ação. Um exemplo desta abordagem pode ser visto na Figura 5.13.

***Blacklist*:**
R_2: $src = [5] \wedge dst =$* $\wedge$ $dport =$* $\rightarrow$ **block**

***Whitelist*:**
R_3: $src =$* $\wedge$ $dst = [7] \wedge dport =$* $\rightarrow$ **accept**

Figura 5.13: Outro exemplo de extração mantendo a correlação de *blacklist* e *whitelist* a partir das regras da Figura 5.10.

5.3 Resumo da Metodologia Proposta

Nesta seção, as maiores contribuições propostas neste trabalho são condensadas para facilitar a sua compreensão. Os passos da metodologia proposta de extração de políticas de *firewall* são mostrados no Algoritmo 12.

Entrada: Conjunto de regras correlacionadas C, ordenado por prioridade.
Saída: Representação em mais alto nível da política implícita em C.

1. Gerar *blacklist* e *whitelist*

blacklist $\leftarrow \emptyset$ e whitelist $\leftarrow \emptyset$

para cada *regra R na metade superior de C* **faça**
  se *R é uma regra de blacklist ou whitelist* **então**
    extrai regra de acordo com critérios da seção 5.2.1 ou da seção 5.2.2
  fim
fim

2. aplicar o algoritmo de descorrelacionamento hierárquico ao resto de C
(Algoritmo 10)

3. aplicar o algoritmo de agrupamento recursivo no U resultante
(Algoritmo 11)

4. mostrar os resultados da seguinte maneira:

- blacklist
- whitelist
- Regras Hierárquicas (como mostradas na Figura 5.4)

Algoritmo 12: Sumário da Metodologia de Extração de Políticas

A fim de exemplificar a metodologia proposta e contrastá-la com a abordagem anteriormente usada, dois conjuntos de regras mais completos (baseados no exemplo de Shaer et al (AL-SHAER et al., 2005)) são mostrados nas Figuras 5.14 e 5.15. Quanto a estes exemplos, são feitas algumas suposições em relação ao nome dos campos e seus domínios, como mostrado abaixo:

- $nome(c_1) = prot$, $D(c_1) = [0, 255]$;
- $nome(c_2) = src$, $D(c_2) = [0, 2^{32} - 1]$;
- $nome(c_3) = sport$, $D(c_3) = [0, 65535]$;
- $nome(c_4) = dst$, $D(c_4) = [0, 2^{32} - 1]$;
- $nome(c_5) = dport$, $D(c_5) = [0, 65535]$.

Entrada:
$R_1 : src = [7777] \rightarrow$ **block**
$R_2 : dst = [7777] \rightarrow$ **block**
$R_3 : prot = [6] \wedge src = [20] \wedge dport = [80] \rightarrow$ **block**
$R_4 : prot = [6] \wedge src = [0, 255] \wedge dport = [80] \rightarrow$ **accept**
$R_5 : prot = [6] \wedge dst = [1003] \wedge dport = [80] \rightarrow$ **accept**
$R_6 : prot = [6] \wedge src = [0, 255] \wedge dst = [1003] \wedge dport = [80] \rightarrow$ **block**
$R_7 : prot = [6] \rightarrow$ **block**

Saída:
$U_1 : src = [7777] \rightarrow$ **block**
$U_2 : src = \{[0, 7776], [7778, 2^{32} - 1]\} \wedge dst = [7777] \rightarrow$ **block**
$U_3 : prot = [6] \wedge src = [20] \wedge dst = \{[0, 7776], [7778, 2^{32} - 1]\} \rightarrow$ **block**
$U_4 : prot = [6] \wedge src = \{[0, 19], [21, 255]\} \wedge dst = \{[0, 7776], [7778, 2^{32} - 1]\} \wedge$
$\wedge dport = [80] \rightarrow$ **accept**
$U_5 : prot = [6] \wedge src = \{[256, 7776], [7778, 2^{32} - 1]\} \wedge dst = [1003] \wedge$
$\wedge dport = [80] \rightarrow$ **accept**
$U_6 : prot = [6] \wedge src = \{[0, 19], [21, 255]\} \wedge dst = \{[0, 7776], [7778, 2^{32} - 1]\} \wedge$
$\wedge dport = \{[0, 79], [81, 65535]\} \rightarrow$ **block**
$U_7 : prot = [6] \wedge src = \{[256, 7776], [7778, 2^{32} - 1]\} \wedge$
$\wedge dst = \{[0, 7776], [7778, 1003339], [1003341, 2^{32} - 1]\} \rightarrow$ **block**
$U_8 : prot = [6] \wedge src = \{[256, 7776], [7778, 2^{32} - 1]\} \wedge dst = [1003] \wedge$
$\wedge dport = \{[0, 79], [81, 65535]\} \rightarrow$ **block**

Figura 5.14: Saída gerada pela aplicação de descorrelacionamento (Algoritmo 6) seguido de agrupamento (Algoritmo 8).

Na Figura 5.14, é mostrado um exemplo da aplicação da abordagem descrita na seção 4.4, na qual, dado um conjunto de regras correlacionadas, primeiramente se descorrelaciona as mesmas e, depois, se agrupa as regras resultantes. Como pode ser visto na figura, o resultado não é de fácil visualização. As regras R_1 e R_2, que são regras de

blacklist segundo a definição da seção 5.2 provocam uma grande quantidade de descorrelacionamentos nos campos src e dst. Além disso, a regra R_7, que é mais genérica, gera três novas regras quando descorrelacionada (U_6, U_7 e U_8). Deve-se notar, também, que mesmo se fossem mantidas regras com apenas uma ação (como em (TONGAONKAR; INAMDAR; SEKAR, 2007)), a política resultante ainda seria complicada.

A Figura 5.15 mostra um conjunto de regras correlacionadas com tamanho 20, sendo algumas delas as regras da Figura 5.14. A saída da aplicação do Algoritmo 12 ao conjunto de regras da Figura 5.15 é mostrada na Figura 5.16. Pode-se perceber que muitos descorrelacionamentos são evitados pelo fato de se manter o correlacionamento entre regras de *blacklist* (e *whitelist*) e as demais regras. Além disso, com o uso do descorrelacionamento hierárquico, são evitados descorrelacionamentos com regras genéricas, o que poderia gerar muita fragmentação (como foi visto no exemplo da Figura 5.14). Desta forma, o modelo apresentado neste trabalho ajuda a extrair políticas de maneira mais legível e, assim, mais úteis para o administrador do *firewall*.

$R_1 : src = [50] \rightarrow$ **block**
$R_2 : dst = [50] \rightarrow$ **block**
$R_3 : src = [7777] \rightarrow$ **block**
$R_4 : dst = [7777] \rightarrow$ **block**
$R_5 : src = [70] \rightarrow$ **accept**
$R_6 : dst = [70] \rightarrow$ **accept**
$R_7 : src = [121] \rightarrow$ **accept**
$R_8 : dst = [121] \rightarrow$ **accept**
$R_9 : prot = [6] \wedge src = [20] \wedge dport = [80] \rightarrow$ **block**
$R_{10} : prot = [6] \wedge src = [0, 255] \wedge dport = [80] \rightarrow$ **accept**
$R_{11} : prot = [6] \wedge dst = [1003] \wedge dport = [80] \rightarrow$ **accept**
$R_{12} : prot = [6] \wedge src = [0, 255] \wedge dst = [1003] \wedge dport = [80] \rightarrow$ **block**
$R_{13} : prot = [6] \wedge src = [30] \wedge dport = [21] \rightarrow$ **block**
$R_{14} : prot = [6] \wedge src = [0, 255] \wedge dport = [21] \rightarrow$ **accept**
$R_{15} : prot = [6] \wedge src = [0, 255] \wedge dst = [1003] \wedge dport = [21] \rightarrow$ **accept**
$R_{16} : prot = [6] \rightarrow$ **block**
$R_{17} : prot = [17] \wedge src = [0, 255] \wedge dst = [1003] \wedge dport = [53] \rightarrow$ **accept**
$R_{18} : prot = [17] \wedge dst = [1003] \wedge dport = [53] \rightarrow$ **accept**
$R_{19} : prot = [17] \wedge src = [256, 511] \wedge dst = [2003] \rightarrow$ **accept**
$R_{20} : prot = [17] \rightarrow$ **block**

Figura 5.15: Exemplo mais completo de conjunto de regras correlacionadas.

Blacklist
$R_{1,2} : src = [50] \vee dst = [50] \rightarrow$ **block**
$R_{3,4} : src = [7777] \vee dst = [7777] \rightarrow$ **block**

Whitelist
$R_{5,6} : src = [70] \vee dst = [70] \rightarrow$ **accept**
$R_{7,8} : src = [121] \vee dst = [121] \rightarrow$ **accept**

Regras Hierárquicas
$R_{16,20} : prot = \{[6], [17]\} \rightarrow$ **block**
{
$R_{10,14} : prot = [6] \wedge src = [0, 255] \wedge dport = \{[21], [80]\} \rightarrow$ **accept**
{
$R_9 : prot = [6] \wedge src = [20] \wedge dport = [80] \rightarrow$ **block**
$R_{13} : prot = [6] \wedge src = [30] \wedge dport = [21] \rightarrow$ **block**
}
$R_{11} : prot = [6] \wedge src = [256, 2^{32} - 1] \wedge dst = [1003] \wedge dport = [80] \rightarrow$ **accept**
$R_{18} : prot = [17] \wedge dst = [1003] \wedge dport = [53] \rightarrow$ **accept**
$R_{19} : prot = [17] \wedge src = [256, 511] \wedge dst = [2003] \rightarrow$ **accept**
}

Figura 5.16: Saída gerada pela aplicação da Metodologia Proposta (Algoritmo 12) ao conjunto de regras da Figura 5.15

6 CONCLUSÃO

No estudo sobre descorrelacionamento realizado neste trabalho, constatou-se que, apesar de a complexidade teórica ser desanimadora, na realidade, há fortes evidências de que este caso não ocorre e o problema é tratável. Pôde-se perceber, também, que vários trabalhos se baseiam na resolução deste problema. Foram apresentados dois algoritmos utilizados na literatura (de Sanchez e Condell e de Liu e Gouda), além de um novo algoritmo, que é uma colagem dos algoritmos de Cuppens et al e de Qian et al. No estudo sobre agrupamento de regras descorrelacionadas, foi apresentada uma nova estrutura denominada grafo 1-dif, que explicita as possibilidades de agrupamento das regras. Foi mostrado, também, o algoritmo sequencial de agrupamento, que já havia sido mencionado na literatura e um algoritmo guloso de agrupamento, elaborado neste trabalho. O único modelo empregado de extração de políticas de *firewall*, até o presente trabalho, foi o de descorrelacionamento de regras e o agrupamento de regras descorrelacionadas.

Com base neste trabalho, pode-se concluir que a extração de políticas de *firewall* de mais alto nível a partir de regras de filtragem em baixo nível é viável. Através do uso do descorrelacionamento de regras e do subsequente agrupamento de regras semelhantes, é possível extrair políticas de nível mais elevado. Todavia, esta abordagem, que já havia sido apresentada na escassa literatura sobre extração de políticas de *firewall*, possui a limitação de não sair do modelo de regras de filtragem.

Assim, a fim de produzir políticas de mais alto nível, este trabalho propôs uma nova metodologia para extração de políticas. Esta é baseada em duas observações:

- As regras de filtragem em baixo nível possuem, em boa parte, uma estrutura hierárquica, isto é, algumas regras contêm outras e, por isso, são mais genéricas. Isto ocorre pela maneira como o administrador do *firewall* pensa em alto nível e pela necessidade de garantir a proteção para determinados tipos de tráfego (com regras genéricas);

- Regras de filtragem que representam *blacklist* ou *whitelist* estão correlacionadas, por definição, com várias outras regras.

Estas observações, em conjunto com o uso de descorrelacionamento, formam a nova metodologia proposta. A primeira observação resulta no uso do descorrelacionamento hierárquico, que, em conjunto com um agrupamento hierárquico gera, quando possível, regras organizadas hierarquicamente e compactas. A segunda observação resulta na modelagem de regras de *blacklist* ou *whitelist* como regras separadas das demais, o que seria bastante útil, pois evitaria uma grande quantidade de descorrelacionamentos e, consequentemente, de fragmentação de regras. O principal diferencial da metodologia proposta em relação à abordagem anterior (de descorrelacionamento seguido de agrupamento) é o

fato de que as regras são extraídas para um modelo. Este é formado por regras de *blacklist*, regras de *whitelist* e regras hierárquicas, todas representadas separadamente. Desta forma, trabalha-se em um nível mais alto e estruturado. Além disso, no pior caso o descorrelacionamento hierárquico e o agrupamento hierárquico funcionarão sem hierarquia, como a abordagem já existente.

As principais contribuições deste trabalho foram:

- Um estudo sobre a extração de políticas de *firewall*, enfocando:
  - O problema do descorrelacionamento de regras de *firewall*, cuja discussão é pouco encontrada na literatura. Para resolvê-lo, foi apresentado um novo algoritmo baseado em outros já publicados;
  - A apresentação de uma estrutura para visualizar correlacionamentos, denominada grafo de correlacionamento;
  - Uma análise sobre o agrupamento de regras, com a elaboração de um algoritmo original para tal.
- A proposta de uma nova metodologia para extração de políticas de *firewall*, composta por:
  - Um novo algoritmo hierárquico para realizar tanto extração de políticas quanto descorrelacionamento: o descorrelacionamento hierárquico;
  - Um novo modelo de extração de políticas, que se baseia na diferenciação de regras relacionadas a *blacklists* e *whitelists*.

Visando a aprimorar a extração de políticas de *firewalls*, trabalhos futuros poderiam:

- Testar a metodologia proposta com conjuntos de regras de *firewalls* reais a fim de validá-la;
- Utilizar modelos mais abstratos na extração de políticas de *firewalls* para facilitar sua legibilidade;
- Verificar a possibilidade de extrair as políticas de múltiplos *firewalls* ou *firewalls* distribuídos;
- Explorar o uso do grafo de correlacionamento como estrutura para analisar e visualizar conflitos, tanto em *firewalls* únicos, como em *firewalls* distribuídos.

Este é um tema atual e instigante. Certamente, novos enfoques surgirão colaborando para o seu aprofundamento e consequente avanço.

REFERÊNCIAS

ABEDIN, M. et al. Detection and Resolution of Anomalies in Firewall Policy Rules. In: IFIP CONF. ON DATA AND APPLICATIONS SECURITY, 20., 2006. **Proceedings...** [S.l.]: Springer, 2006. p.15–29.

ACHARYA, S. et al. Traffic-aware firewall optimization strategies. In: INTL. CONFERENCE ON COMMUNICATIONS, 2006. **Proceedings...** [S.l.: s.n.], 2006.

AL-SHAER, E. S.; HAMED, H. H. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT, IM, 8., 2003, Colorado Springs. **Integrated Network Management VIII**: managing it all. Boston: Kluwer Academic, 2003. p.17–30.

AL-SHAER, E. S.; HAMED, H. H. Discovery of Policy Anomalies in Distributed Firewalls. In: INFOCOM, 2004. **Proceedings...** [S.l.: s.n.], 2004.

AL-SHAER, E. et al. Conflict Classification and Analysis of Distributed Firewall Policies. **IEEE Journal on Selected Areas in Communications (JSAC)**, [S.l.], v.23, n.10, p.2069–2084, Oct. 2005.

ALGOSEC. **Firewall Analyzer**. Disponível em <http://www.algosec.com/Products/FA/>. Acesso em: nov. 2007.

BABOESCU, F.; SINGH, S.; VARGHESE, G. Packet classification for core routers: is there an alternative to cams ? In: ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES, INFOCOM, 2003. **Proceedings...** [S.l.]: IEEE, 2003, v.1, p.53–63

BABOESCU, F.; VARGHESE, G. Scalable packet classification. **Computer Communication Review**, New York, v.31, n.4, p.199–210. Trabalho apresentado na ACM SIGCOMM Conference, 2001, San Diego.

BAILEY, M. et al. A Quantitative Study of Firewall Configuration Errors. **IEEE Security & Privacy Magazine**, [S.l.], v.3, n.4, p.26–31, 2005.

BANDARA, A. K. et al. Using Argumentation Logic for Firewall Policy Specification and Analysis. In: IFIP/IEEE INTERNATIONAL WORKSHOP ON DISTRIBUTED SYSTEMS: OPERATIONS AND MANAGEMENT, DSOM, 17., 2006, Dublin, Ireland. **Large Scale Management of Distributed Systems**: proceedings. Berlin: Springer, 2006. p.185–196. (Lecture Notes in Computer Science, v.4269).

BARTAL, Y. et al. Firmato: a novel firewall management toolkit. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.17–31.

BARTAL, Y. et al. *Firmato*: a novel firewall management toolkit. **ACM Transactions on Computer Systems**, [S.l.], v.22, n.4, p.381–420, 2004.

CHAPMAN, D. B. Network (In)Security Through IP Packet Filtering. In: USENIX UNIX SECURITY SYMPOSIUM, 3., 1992, Baltimore, MD., USA. **Proceedings...** [S.l.]: Usenix, 1992.

CHOMSIRI, T.; PORNAVALAI, C. Firewall Rules Analysis. In: INTERNATIONAL CONFERENCE ON SECURITY& MANAGEMENT, SAM, 2006, Las Vegas, Nevada, USA. **Proceedings...** [Las Vegas]: CSREA Press, 2006. p.213–219.

CONDELL, M.; SANCHEZ, L. **On the Deterministic Enforcement of Un-ordered Security Policies**. [S.l.]: BBN Technologies, 2004. (BBN Technical Memorandum No. 1346)

CUPPENS, F.; CUPPENS, N.; GARCÍA, J. Detection and Removal of Firewall Misconfiguration. In: INTERNATIONAL CONFERENCE ON COMMUNICATION, NETWORK AND INFORMATION SECURITY, CNIS, 2005, Phoenix, AZ, USA. **Proceedings...** [S.l.]: IASTED, 2005.

DIMOPOULOS, Y.; NEBEL, B.; TONI, F. On the computational complexity of assumption-based argumentation for default reasoning. **Artificial Intelligence**, Essex, UK, v.141, n.1, p.57–78, 2002.

EICHIN, M. W.; ROCHLIS, J. A. With microscope and tweezers: analysis of the internet virus of november 1988. In: IEEE COMPUTER SOCIETY SYMPOSIUM ON SECURITY AND PRIVACY, 1989. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1989.

EPPSTEIN, D.; MUTHUKRISHNAN, S. Internet packet filter management and rectangle geometry. In: SYMPOSIUM ON DISCRETE ALGORITHMS, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.827–835.

ERONEN, P.; ZITTING, J. An expert system for analyzing firewall rules. In: NORDIC WORKSHOP ON SECURE IT SYSTEMS, NORDSEC, 6., 2001, Copenhagen, Denmark. **Proceedings...** [S.l.]: Technical University of Denmark, 2001. p.100–107. (Technical Report IMM-TR-2001-14).

GOUDA, M. G.; LIU, A. X. Firewall Design: consistency, completeness, and compactness. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, ICDCS, 24., 2004, Hachioji, Tokyo, Japan. **Proceedings...** [S.l.]: IEEE Computer Society, 2004. p.320–327.

GOUDA, M. G.; LIU, A. X. Structured firewall design. **Computer Networks**, New York, NY, USA, v.51, n.4, p.1106–1120, 2007.

GUPTA, P. **Algorithms for Routing Lookups and Packet Classification**. 2000. Tese (Doutorado em Ciência da Computação) — Department of Computer Science, Stanford University.

GUPTA, P.; MCKEOWN, N. Packet Classification on Multiple Fields. In: SIGCOMM, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.147–160.

GUTTMAN, J. D. Filtering Postures: local enforcement for global policies. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 1997, Oakland, CA, USA. **Proceedings...** [S.l.]: IEEE Computer Society, 1997. p.120–129.

HARI, H. A.; SURI, S.; PARULKAR, G. M. Detecting and Resolving Packet Filter Conflicts. In: INFOCOM, 2000. **Proceedings...** [S.l.: s.n.], 2000. p.1203–1212.

HAZELHURST, S. **Algorithms for analysing firewall and router access lists**. [S.l.]: University of the Witwatersrand, 1999. (Technical Report TR-WitsCS -1999-5)

INGHAM, K.; FORREST, S. **Enhancing Computer Security with Smart Technology**. [S.l.]: CRC Press, 2005. p. 9-35.

KOUNAVIS, M. E. et al. Directions in Packet Classification for Network Processors. In: WORKSHOP ON NETWORK PROCESSORS, NP, 2., 2003. **Proceedings...** [S.l.: s.n.], 2003.

LIU, A. X.; GOUDA, M. G. Diverse Firewall Design. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2004, Florence, Italy. **Proceedings...** [S.l.]: IEEE Computer Society, 2004. p.595–604.

LIU, A. X. et al. Firewall Queries. In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DISTRIBUTED SYSTEMS, OPODIS, 8., 2004, Grenoble, France. **Revised Selected Papers**. Berlin: Springer, 2004. p.197–212. (Lecture Notes in Computer Science, v.3544).

MARMORSTEIN, R.; KEARNS, P. A Tool for Automated IPtables Firewall Analysis. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2005, Anaheim, CA, USA. **Proceedings...** [S.l.: s.n.], 2005. p.71–81.

MARMORSTEIN, R.; KEARNS, P. An Open Source Solution for Testing NAT'd and Nested iptables Firewalls. In: LARGE INSTALLATION SYSTEM ADMINISTRATION CONFERENCE, 19., 2005, San Diego, CA, USA. **Proceedings...** [S.l.: s.n.], 2005. p.103–112.

MAYER, A.; WOOL, A.; ZISKIND, E. Fang: a firewall analysis engine. In: IEEE SYMP. ON SECURITY AND PRIVACY, 2000, Oakland, CA, USA. **Proceedings...** [S.l.]: IEEE Computer Society Press, 2000. p.177–187.

MAYER, A.; WOOL, A.; ZISKIND, E. Offline firewall analysis. **Int. J. Inf. Secur.**, Berlin, v.5, n.3, p.125–144, 2006.

MIERLUTIU, A. P. A Rule Cache for iptables in Linux. In: ROEDUNET INTERNATIONAL CONFERENCE, 2., 2003. **Proceedings...** [S.l.: s.n.], 2003.

MOORE, D.; SHANNON, C.; BROWN, J. Code-Red: a case study on the spread and victims of an internet worm. In: INTERNET MEASUREMENT WORKSHOP, IMW, 2002. **Proceedings...** [S.l.: s.n.], 2002.

QIAN, J.; HINRICHS, S.; NAHRSTEDT, K. ACLA: a framework for access control list (ACL) analysis and optimization. In: IFIP TC6/TC11 INTERNATIONAL CONFERENCE ON COMMUNICATIONS AND MULTIMEDIA SECURITY ISSUES OF THE NEW CENTURY, 2001, Deventer, The Netherlands. **Proceedings...** [S.l.]: Kluwer BV, 2001. p.4.

QIU, L.; VARGHESE, G.; SURI, S. Fast firewall implementations for software-based and hardware-based routers. In: SIGMETRICS/PERFORMANCE, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.344–345.

SANCHEZ, L.; CONDELL, M. **Security Policy Protocol**. [S.l.]: IETF, 2000. n.2.

SYMANTEC CORPORATION. **Symantec Internet Security Threat Report**: trends for january 06-june 06. Disponível em: <http://www.symantec.com/specprog/threatreport/ent-whitepaper_ symantec_internet_ security_threat_report_x_09_2006.en-us.pdf>. Acesso em: nov. 2007.

TAYLOR, D. E.; TURNER, J. S. Scalable packet classification using distributed crossproducing of field labels. In: ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES, INFOCOM, 24., 2005, Miami, FL,USA. **Proceedings...** [S.l.: s.n.], 2005. p.269–280.

TONGAONKAR, A.; INAMDAR, N.; SEKAR, R. Inferring Higher Level Policies from Firewall Rules. In: LARGE INSTALLATION SYSTEM ADMINISTRATION CONFERENCE, LISA, 21., 2007. **Proceedings...** [S.l.: s.n.], 2007.

WOOL, A. Architecting the Lumeta firewall analyzer. In: USENIX SECURITY SYMPOSIUM, 10., 2001, Washington, D.C., USA. **Proceedings...** [S.l.]: Usenix, 2001. p.85–97.

WOOL, A. A Quantitative Study of Firewall Configuration Errors. **IEEE Computer**, [S.l.], v.37, n.6, p.62–67, 2004.

ZHANG, Y.; ZHANG, Y.; WANG, W. Optimization of Firewall Filtering Rules by a Thourough Rewritten. In: LANOMS, 4., 2005, Porto Alegre, RS, BR. **Proceedings...** Porto Alegre: UFRGS, 2005.

ZHAO, H.; BELLOVIN, S. M. **Policy Algebras for Hybrid Firewalls**. [S.l.]: Columbia University, 2007. (CUCS-017-07).