UNIVERSIDADE DE CAXIAS DO SUL

LEONARDO SCUSSIATTO

AVALIAÇÃO DE REGRAS DE *FIREWALL* UTILIZANDO UM MOTOR DE INFERÊNCIA

CAXIAS DO SUL

2017

LEONARDO SCUSSIATTO

AVALIAÇÃO DE REGRAS DE *FIREWALL* UTILIZANDO UM MOTOR DE INFERÊNCIA

Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação apresentado à Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientadora: Prof. Dr. Maria de Fátima Webber do Prado Lima.

CAXIAS DO SUL

2017

LEONARDO SCUSSIATTO

AVALIAÇÃO DE REGRAS DE *FIREWALL* UTILIZANDO UM MOTOR DE INFERÊNCIA

Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação apresentado à Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.
Orientadora: Prof. Dr. Maria de Fátima Webber do Prado Lima.

Aprovado em: ____/____/_____

Banca Examinadora

_____
Profa. Dra. Maria de Fátima Webber do Prado Lima
Universidade de Caxias do Sul – UCS

_____
Profa. Dra. Carine Geltrudes Webber
Universidade de Caxias do Sul – UCS

_____
Prof. Me. Marcos Eduardo Casa
Universidade de Caxias do Sul – UCS

AGRADECIMENTOS

Agradeço primeiramente a minha família por todo o suporte na minha formação pessoal e intelectual, em especial aos meus pais e minha tia Ermida.

Agradeço a todos os colegas e professores com quem convivi neste período, compartilhando conhecimentos necessários a minha formação.

Por fim, agradeço a minha orientadora Fátima, por todo o apoio prestado na realização deste trabalho.

RESUMO

Este trabalho apresenta o desenvolvimento de uma abordagem de verificação de problemas de segurança em regras de *firewall* utilizando um motor de inferência. *Firewalls* são um dos principais meios de segurança de redes computacionais. Eles são configurados por meio de um conjunto de regras definidas por um administrador de rede. Porém, estas regras podem ser definidas de maneira errada, sendo de configuração complexa dependendo do tamanho da rede. A má configuração do *firewall* pode comprometer a segurança dos dados ou tornar indisponíveis os serviços de rede. Utilizando o código fonte do sistema de verificação de regras de *firewall* FRChecker (CARBONERA, 2009), este trabalho modificou-o, adicionando uma arquitetura de Sistema Baseado em Conhecimento. Deste modo, as regras de *firewall* passaram a serem analisadas pelo motor de inferência do sistema SWI-Prolog. Foram testados cinco arquivos de regras de *firewall* diferentes. Em comparação com a implementação original do FRChecker, o motor de inferência obteve uma taxa verificação de inconsistências em *firewalls* mais confiável e com melhor desempenho computacional.

Palavras chaves: Regras de *firewall*. Sistema Baseado em Conhecimento. Motor de Inferência. Prolog. JPL.

ABSTRACT

This work presents the development of a firewall rules verification approach, using an inference engine. Firewalls are one of the main means of security in computer networks. They are set through a group of rules defined by a network administrator. However, these rules may be defined in an incorrect way, and are complex to manage depending on the network size. Bad firewall configuration can compromise data security or make network services unavailable. Using the source code of the FRChecker (CARBONERA, 2009) firewall rule verification system, this work has modified it by including a Knowledge-based System architecture. By doing so, the firewall rules started to be analyzed by the SWI-Prolog inference engine. Five diferent firewall rule files were tested. Comparing to the original FRChecker implementation, the inference engine obtained a more reliable firewall issues check, and better computational performance.

Keywords: Firewall rules. Knowledge-based System. Inference Engine. Prolog. JPL.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
BC	Base de Conhecimento
CAIS	Centro de Atendimento a Incidentes de Segurança
CIDR	*Classless Inter-Domain Routing*
DMZ	*Demilitarized Zone*
DNAT	*Destination Network Address Translation*
HTTP	*Hypertext Transfer Protocol*
ICMP	*Internet Control Message Protocol*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
JDK	*Java Development Kit*
JVM	*Java Virtual Machine*
MCD	Módulo Coletor de Dados
ME	Módulo de Explicações
MI	Motor de Inferência
MIT	*Massachusetts Institute of Technology*
MT	Memória de Trabalho
NSBC	Núcleo do Sistema Baseado em Conhecimento
NAPT	*Network Address and Port Translation*
NAT	*Network Address Translation*
OSI	*Open Systems Interconnection*
PCI-DSS	*Payment Card Industry Data Security Standard*
RAM	*Random Access Memory*

RPC	*Remote Procedure Call*
SBC	Sistema Baseado em Conhecimento
SNAT	Tradução de Endereços de Rede de Origem
SOX	*Sarbanes-Oxley*
SQL	*Structured Query Language*
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
TCPMSS	*Transmission Control Protocol Maximum Segment Size*
TOS	*Type of Service*
UDP	*User Datagram Protocol*

SUMÁRIO

1 INTRODUÇÃO

Vivencia-se um mundo de constantes avanços e mudanças, e muito disso se deve a evolução das tecnologias da informação. Seja para a comunicação, realização de negócios, ou adquirir conhecimento, faz-se uso diário e ubíquo das redes de computadores. A medida que estes serviços se tornam cada vez mais cruciais para pessoas e empresas, também cresce a necessidade de tomar medidas que garantam a segurança das informações, e dos equipamentos que as geram e transmitem.

O Centro de Atendimento a Incidentes de Segurança (CAIS), que atua na segurança da Rede Nacional de Ensino e Pesquisa, relatou que apenas em 2014 ocorreram mais de vinte e sete mil incidentes nas redes que monitora. Isso evidencia a necessidade de prevenir ao máximo estes problemas, seja a nível administrativo, de estrutura de rede, ou de aplicação.

Diversas políticas e tecnologias tem sido empregadas em conjunto para reduzir os riscos do roubo de informações e invasão de sistemas computacionais, indo desde a criptografia dos dados, a sistemas que detectam tráfego de rede suspeito e programas executados indevidamente. Dentre este conjunto de ferramentas, pode-se citar como relevante o uso de *firewalls*.

Assim como as pontes levadiças medievais, que atuavam como única rota e ponto de verificação de quem desejasse sair ou entrar dos castelos, os sistemas de *firewall* atuam como uma entrada e saída única, ligando uma rede a outra, aonde cada pacote de rede que deseja sair ou entrar deve ser verificado (TANENBAUM, 2003). A política de verificação de pacotes é definida através de um conjunto de regras que são configuradas no *firewall*, as quais descrevem que tipos de dados podem trafegar, e de que local podem partir ou chegar. Dentre as técnicas de filtragem de pacotes de redes, a de controle de serviços é a mais utilizada. Através dela, delimitam-se serviços oferecidos pela rede, como por exemplo servidores de arquivos e de email. Esta filtragem é baseada nos endereços IP (*Internet Protocol*) de destino e origem, bem como nas portas de rede utilizadas. As outras três técnicas são o controle de direção, de usuário, e de comportamento, que filtra os pacotes de acordo com conteúdo do protocolo de aplicação (STALLINGS, 2007).

Embora os sistemas de *firewall* ofereçam um nível a mais de proteção, eles também apresentam problemas, principalmente com o avanço das tecnologias de rede. Hoje, dispositivos móveis possuem acesso facilitado a Internet, e deste modo, acabam interferindo na função do *firewall* de ser o único ponto de acesso a uma rede externa, comprometendo a

segurança de rede em empresas e instituições. O aumento do tamanho das redes, e o número de serviços e protocolos utilizados também tornou a configuração das regras de *firewall* mais complexa. De acordo com Anderson et al. (1997), especialistas em segurança de rede demonstravam preocupação com o fato dos *firewalls* serem configurados manualmente, o que poderia ocasionar em falhas humanas de segurança. Essa preocupação ficou evidente na pesquisa de Wool (2010), onde 84 conjuntos de regras de *firewalls* comerciais foram avaliados. Destes, mais de quarenta e cinco por cento apresentavam algum problema de configuração com tráfego de entrada, e mais de oitenta por cento com tráfego de saída da rede.

Visando resolver estes problemas de má configuração, algumas pesquisas e abordagens tem sido empregadas, como a utilização de sistemas multiagentes (SREELAJA, 2010), ou sistemas de álgebra relacional (KHAN, 2010). Wool (2004) identificou os problemas de configuração mais comuns e implementou uma solução comercial de verificação de regras de *firewall*. Utilizando a pesquisa de Wool, Carbonera (2009) implementou um sistema gratuito de verificação chamado FRChecker. O FRChecker foi codificado na linguagem de programação Java, e analisa inconsistências no sistema de *firewall* nativo do *kernel* Linux, chamado Netfilter. Esta verificação é realizada através do arquivo de regras de *firewall*, exportado pela ferramenta iptables.

1.1 Objetivos

Com os serviços de rede sendo cada vez mais necessários ao nosso dia a dia, fica evidente a importância de um sistema de *firewall* para a filtragem de dados irrelevantes ou prejudiciais. Mais do que isso, também deve-se ressaltar que o *firewall* apenas cumpre o seu papel se devidamente configurado. Porém, soluções gratuitas que ajudem a identificar problemas nas regras de filtragem são escassas.

Este trabalho se propõe a incrementar o sistema FRChecker, através da integração de um mecanismo de inferência na linguagem Prolog. Para isto, a implementação do FRChecker foi analisada, e foram determinados os problemas de configuração que se beneficiam da análise de um mecanismo de inferência. Outros sistemas semelhantes foram analisados, em busca de identificar outros tipos de problemas que podem receber a mesma análise.

Após selecionar um conjunto de problemas de configuração, estes foram modelados como regras de inferência. Estas regras foram implementadas na linguagem Prolog, e o sistema FRChecker, implementado em Java, foi modificado para integrar o motor de inferência do Prolog.

Ao final, arquivos com regras de *firewall* foram avaliados pelo FRChecker utilizando o

mecanismo de inferência do Prolog, e os resultados analisados para verificar a consistência da aplicação.

1.2 Organização do Trabalho

A estrutura do trabalho está dividida em cinco capítulos.

O Capítulo 2 aborda o tema de *firewalls*, explicando suas características, topologias e aplicações. O enfoque maior é o *firewall* Netfilter e sua interface com o usuário, a ferramenta iptables. Os componentes deste filtro de pacotes, a organização das suas regras e de que modo são acessadas foram estudadas.

O terceiro Capítulo descreve o sistema FRChecker, explicando de que modo obtém as regras de *firewall* e que tipos de abordagens utiliza na verificação de regras de *firewall*.

O Capítulo 4 relata a análise de sistemas semelhantes ao FRChecker, descrevendo suas semelhanças e diferenças, a fim de encontrar funcionalidades distintas.

O Capítulo 5 realiza uma breve introdução aos Sistemas Baseados em Conhecimento, descrevendo seus conceitos e elementos da arquitetura.

O sexto Capítulo apresenta uma proposta de solução, citando e justificando os elementos selecionados para implementação do sistema, e esquematizando sua estrutura.

O Capítulo 7 detalha os passos para implementação do sistema, em especial a modelagem da base de fatos, das regras de inferência e do fluxo de dados.

O oitavo Capítulo apresenta os resultados do sistema implementado, a fim de validar este estudo.

E por fim, o último Capítulo conclui o trabalho, sugerindo futuros estudos relacionados.

2 *FIREWALLS*

Um *firewall* pode ser descrito como o elemento que separa uma rede segura de uma não segura. Através de um conjunto de regras, definidas de acordo com uma política de segurança, pacotes de dados obtém autorização para entrar e sair de uma rede. Deste modo, o *firewall* é um conjunto de software e hardware que permite a um administrador controle o acesso a seus recursos de rede. Kurose (2010) define claramente os três objetivos dos sistemas de *firewall*:

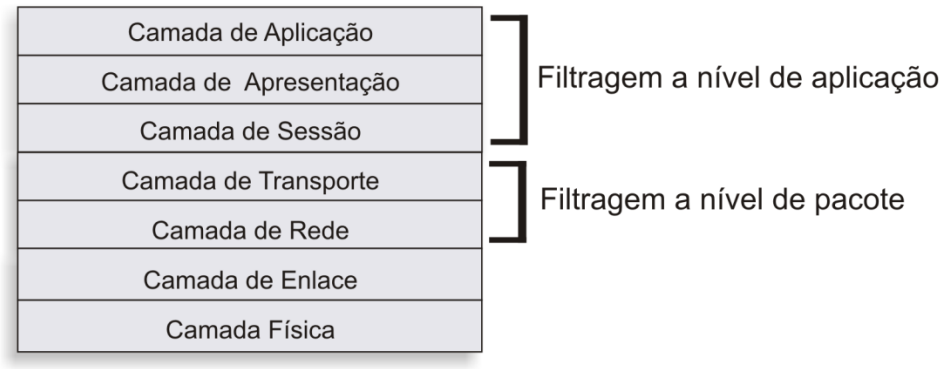
- Em uma rede com *firewall* implementado, todo o tráfego que entra ou sai passa pelo *firewall*. Deste modo, os *firewalls* delimitam uma rede interna e administrada, de uma rede externa e potencialmente não segura.
- Somente o tráfego autorizado pelas políticas de segurança obtém o direito de passar pelo *firewall*.
- O *firewall* deve ser projetado e configurado de modo a ser imune a ação de intrusos.

Firewalls utilizam quatro tipos de controle para implementar sua política de segurança. O mais comum é o controle de serviço, que especifica que tipos de serviço de rede podem transitar em uma rede. Estes serviços são especificados pelo endereço IP (*Internet Protocol*) e porta de destino ou origem, embora em alguns casos também é realizada uma filtragem de conteúdo através de um *proxy*. O controle de direção determina que *host* deve iniciar a solicitação de um serviço, enquanto o controle de usuário define quem dentro de uma rede tem acesso a determinado serviço. Já o controle de comportamento serve para delimitar o modo como os serviços são utilizados, e pode deste modo, impedir o acesso a determinadas funcionalidades de um serviço.

2.1 Tipos de Filtragem

De acordo com Hunt (1998), o modo como serviços e dados podem ser permitidos e bloqueados é classificado de acordo com a pilha de protocolos do modelo OSI (*Open Systems Interconnection*) (Figura 1).

Figura 1 - Tipos de filtragem de acordo com o modelo OSI.



Fonte*: HUNT (1998)*

Na arquitetura TCP/IP (*Transport Control Protocol/Internet Protocol*), *firewalls* com filtragem a nível de pacote atuam nas camadas de Rede e Transporte, enquanto que *firewalls* com filtragem a nível de aplicação lidam com as informações da camada de Aplicação, já que esta arquitetura não implementa as camadas de Sessão e Apresentação.

2.1.1 **Filtragem a nível de pacote**

Também chamados de filtros de pacote tradicionais (KUROSE, 2010), os *firewalls* realizam a filtragem baseados em um conjunto de regras, que são aplicadas a cada pacote IP que chega a rede, encaminhando-o ou descartando-o. Geralmente, os *firewalls* deste tipo operam sobre duas políticas padrão: descartar tudo o que não for permitido, ou aceitar tudo o que não for proibido.

As principais informações utilizadas para verificação dos pacotes são o endereço IP e a porta TCP/UDP (*Transport Control Protocol/User Datagram Protocol*) de origem e destino. No entanto, a filtragem também pode se estender a informações complementares, como por exemplo o tipo de protocolo de rede e transporte, os bits SYN e ACK do protocolo TCP/IP, e também o tipo de mensagem ICMP (*Internet Control Message Protocol*).

Através deste conjunto de informações, pode-se definir diferentes políticas de segurança, de acordo com as necessidades da organização. Por exemplo, para impedir o acesso a qualquer página HTTP (*Hypertext Transfer Protocol*), deve-se bloquear todo pacote com porta número 80. Já para impedir que requisições TCP externas adentrem a rede interna, pode-se bloquear todo pacote TCP com bit SYN ativado (KUROSE, 2010).

Sempre que se inicia uma conexão TCP, uma porta no endereço de origem é

dinamicamente definida, com valores de 1024 a 65535. O *firewall* necessita estar habilitado a permitir a passagem de pacotes originários deste enorme intervalo de portas, o que muitas vezes pode significar um risco para a rede. Este problema é amenizado utilizando-se filtragem de pacote com controle de estado. Através deste tipo de filtragem, todas as conexões TCP ativas são registradas em uma tabela, e o *firewall* habilita apenas a passagem de pacotes cuja as portas estão definidas nesta tabela.

2.1.2 Filtragem a nível de aplicação

A filtragem a nível de aplicação, também chamada de *gateway* ou *proxy* de aplicação, atua na camada superior do protocolo TCP/IP, não limitando-se ao endereço IP e protocolos de transporte. Geralmente, cada aplicação que faz uso deste tipo de filtragem utiliza um *firewall* (*gateway*) próprio, pois sua implementação depende do tipo de serviço oferecido. Por exemplo, um *gateway* de Telnet pode limitar o uso da aplicação para determinados usuários, ao invés de endereços IP. Outra utilização típica é impedir o acesso a determinadas páginas HTTP, ou o bloqueio de anexos inseguros em emails.

A filtragem a nível de aplicação pode ser aplicada em conjunto com a filtragem a nível de pacote, garantindo maior segurança, embora torne o trânsito de pacotes mais demorado. A principal vantagem de um *gateway* de aplicação é possuir uma configuração mais simples, já que as regras são definidas por aplicação e não sobre um conjunto de portas e endereços IP. Também torna-se mais fácil auditar o tráfego gerado pela rede. Como desvantagens, está a necessidades de *gateways* diferentes para cada aplicação, e o processamento maior necessário para verificação dos dados.

2.1.3 Filtragem a nível de circuito

Stallings (2007) também define um terceiro tipo de filtragem, que igualmente pode ser implementada em conjunto com as filtragens já descritas. A filtragem a nível de circuito realiza a comunicação entre um *host* e um *gateway*, fazendo com que o *gateway* se encarregue de definir o destino dos dados. Para isto, o *gateway* mantém aberta duas conexões, uma para o *host* origem e outra para o *host* destino, transitando os pacotes entre elas, sem necessariamente efetuar uma averiguação dos dados e cabeçalho dos protocolos. O servidor SOCKS é um dos filtros a nível de circuito mais conhecidos.

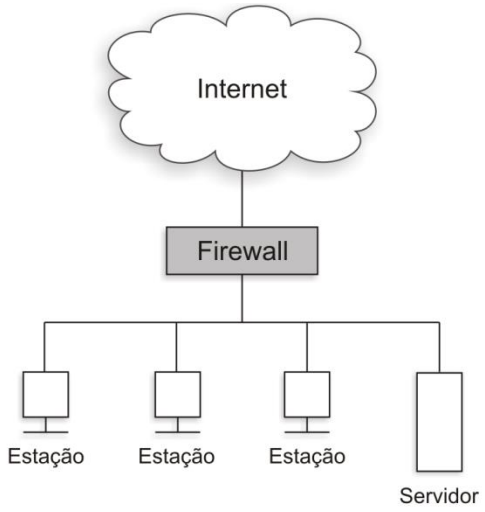
2.2 Topologias de *firewall*

De nada adianta definir corretamente as políticas de segurança de acesso, se a estrutura física da rede interna não estiver com o sistema de *firewall* configurado como único ponto de entrada e saída de dados. Três topologias são definidas como mais utilizadas para atingir este fim, e podem ser implementadas com *firewalls* tanto a nível de pacote como de aplicação: Screened Subnet, One-legged DMZ, e True DMZ (ALDER et al., 2007).

2.2.1 *Screened Subnet*

Conforme ilustrado na Figura 2, utiliza um único *firewall*, com duas interfaces de rede, uma conectada a rede local, e outra a rede externa (ou Internet). Deste modo, todo o tráfego que entra e sai da rede necessariamente deve passar pelo *firewall*. É a topologia mais simples existente, de baixo custo e fácil configuração, e deste modo, popular em residencias e pequenas empresas. Sua desvantagem está no fato de que os serviços de rede são acessados pela rede interna sem qualquer controle de *firewall*. Levando em conta que 70% dos ataques a sistemas se originam da rede interna (TANENBAUM, 2003), este pode ser um problema crítico.

Figura 2 - Arquitetura Screened Subnet



Fonte: ALDER et al. (2007).

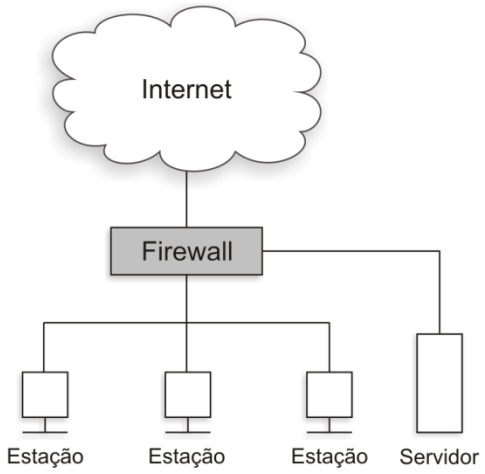
2.2.2 *One-legged DMZ*

Assim como a topologia *Screened Subnet*, esta também utiliza apenas um *firewall*, porém com três interfaces de rede (Figura 3). Deste modo, isola-se os serviços de rede em uma terceira subrede, chamada de zona desmilitarizada (DMZ). Isto não apenas impede que *hosts*

da rede interna tenham acesso irrestristo aos serviços, como também impede que os serviços, que geralmente são alvos de ameaças externas, caso comprometidos, acessem livremente os *hosts* internos.

Uma rede *One-legged DMZ* pode conter múltiplas interfaces contemplando diferentes serviços e subredes internas, aumentando ainda mais a segurança oferecida pelo *firewall*.

Figura 3 - Arquitetura One-legged DMZ



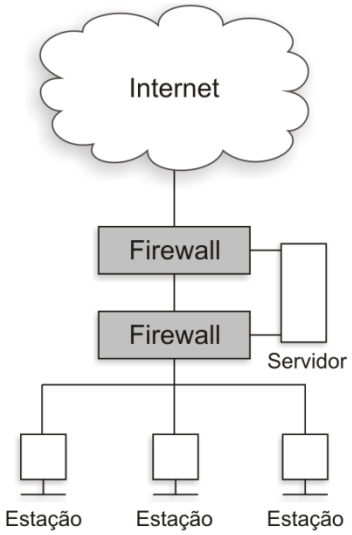
Fonte: ALDER et al. (2007).

2.2.3 ***True DMZ***

Esta topologia utiliza dois *firewalls*, um ligando os serviços de rede à rede externa, e outro ligando os serviços de rede à rede interna (Figura 4). É a arquitetura mais cara de ser implementada, dentre as três apresentadas, mas garante o melhor nível de segurança. Os *hosts* locais necessitam passar por dois *firewalls* para acessar redes externas, e os serviços de rede não são conectados diretamente a rede interna, evitando problemas, caso os serviços sejam afetados por algum malfeitor.

É comum que sejam utilizados diferentes sistemas em cada *firewall*. Caso de um destes seja alvo de um ataque por conta de uma vulnerabilidade, o mesmo ataque não surtirá efeito no outro *firewall*.

Figura 4 - Arquitetura True DMZ



Fonte: ALDER et al. (2007).

2.3 Netfilter

Dada a importância do uso de *firewalls* para melhorar a segurança de rede, inúmeras ferramentas foram implementadas com a funcionalidade de bloquear e permitir tráfego em rede. A grande maioria são sistemas comerciais, mas existem alternativas gratuítas, e dentre estas se destaca o Netfilter[1].

O Netfilter é um *framework* incluso no *kernel* Linux, que realiza a chamada de funções do *kernel* para cada pacote de rede recebido ou enviado. Segundo o site oficial do projeto Netfilter, as principais características deste *framework* são a filtragem de pacotes a nível de rede com e sem controle de estado, tradução de endereços e portas de rede, e sua flexibilidade e customização, através de extensões que podem ser acopladas ao Netfilter durante a compilação do *kernel*.

É comum associar os sistemas ipchains e ipfwadm ao Netfilter, pois são versões anteriores do *framework*. Porém, o Netfilter passou por uma grande reestruturação e não é compatível com estes dois sistemas. Já o sistema iptables, embora muitas vezes confundido com o Netfilter, é apenas a ferramenta para realizar a manutenção das regras de *firewall* aplicadas ao Netfilter. O iptables será abordado na Seção 2.4.

Para organizar melhor o seu conjunto de funcionalidades, o Netfilter atua com uma estrutura de tabelas, cada uma com uma função própria. Para cada tabela, são definidas regras, agrupadas em conjuntos chamados *chains*. Como diferentes tabelas possuem *chains* com as

[1] www.netfilter.org

mesmas funções, estas serão descritas em seções separadas. Há ainda o conceito de targets, que indicam a ação a ser tomada quando uma regra for aplicada. A seguir, estes três conceitos são detalhados.

2.3.1 **Tabelas**

A estrutura de tabelas do Netfilter possui quatro tabelas definidas: *raw*, *mangle*, NAT (*Network Address Translation*) e *filter*. Cada tabela (*table*) realiza uma funcionalidade específica, e possui uma ordem de precedência. A primeira tabela pela qual um pacote passa é a *raw*, seguida por *mangle*, NAT e *filter* (CARBONERA, 2009).

A tabela *raw* é a menos utilizada, sendo inclusive não descrita em algumas bibliografias. Alder et al.(2007), a define como a tabela utilizada para rastreamento, caso necessário. Deste modo, o pacote recebe uma marcação, para que seja localizado nas outras tabelas do *firewall*.

A tabela *mangle*, realiza alterações nas informações contidas no cabeçalho do pacote, a fim de controlar o seu comportamento, como mudar sua prioridade, o tamanho permitido para os pacotes em uma conexão, ou até mesmo seu destino.

NAT é a terceira tabela por ordem de precedência, e é utilizada quando é necessário mudar o destino ou origem de um pacote. É comum seu uso em redes que possuem apenas um endereço IP para se conectar a Internet. Deste modo, diversos *hosts* na rede interna podem fazer uso deste endereço, e o *firewall* além de realizar a conversão de endereços, também mantém registro das conexões, para transmitir os pacotes aos *hosts* corretos (PETERSEN, 2008).

Já a tabela *filter*, é a mais utilizada, pois realiza a filtragem de pacotes, fator primordial para o funcionamento do *firewall*. Através dela é que são definidas as políticas de permissão ou negação de tráfego, de acordo com a política de segurança implementada.

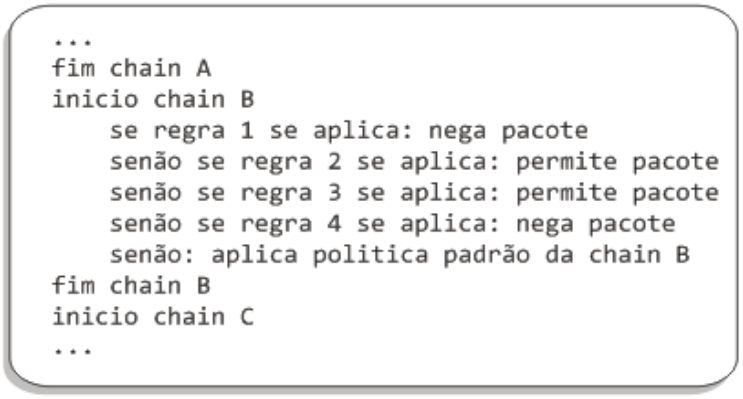
2.3.2 ***Chains***

Conforme já descrito, *chains* (cadeias) são conjuntos de regras, pelas quais os pacotes são avaliados. De acordo com Purdy (2004), diferentes *chains* são aplicadas aos pacotes, dependendo de sua origem e destino (pacotes com origem do *firewall*, com destino ao *firewall*, que apenas são encaminhados pelo *firewall*, ou ainda com destino e origem do *firewall*, utilizado para comunicação entre processos).

Petersen (2008) define as *chains* como um conjunto de regras em uma estrutura if-then-else. Se um pacote não atender a primeira regra, a próxima é verificada, e assim por diante

(Figura 5). Se nenhuma regra naquela *chain* atender as características do pacote, a política padrão da *chain* é utilizada. As *chains* padrão do Netfilter possuem duas políticas: aceitar ou negar o pacote. Já *chains* definidas por usuários podem apenas implementar a política de repassar o pacote para outra *chain* (PURDY, 2004).

Figura 5 - Exemplo de fluxo de pacotes entre chains.



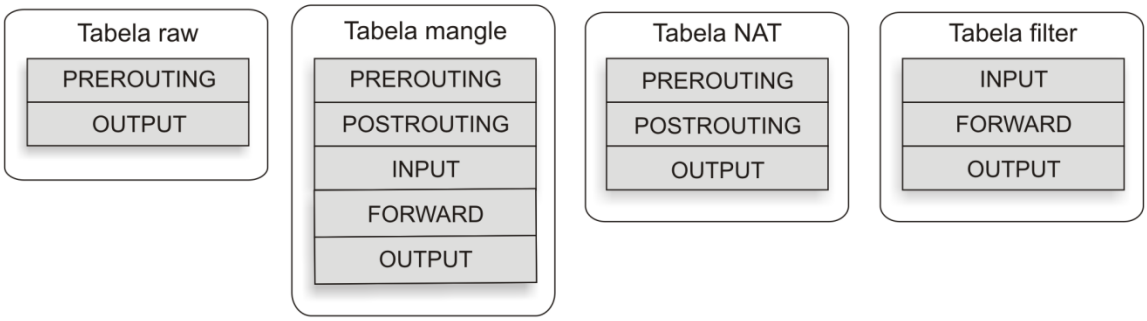
Fonte: Baseado em PETERSEN (2008).

Chains podem ser criadas, modificadas e removidas de acordo com as necessidades do administrador do *firewall*. Por padrão, o Netfilter oferece as seguintes *chains* já criadas:

- INPUT: Verifica pacotes originários de um *host*, com destino ao *firewall*. OUTPUT: Verifica pacotes originários do *firewall*, com destino a algum *host* qualquer.
- FORWARD: Verifica pacotes que não se originam ou destinam do *firewall*, e serão repassados a outro *host*.
- PREROUTING: Realiza o pré processamento do pacote, alterando campos do cabeçalho se necessário, antes de encaminhá-lo a outra *chain*.
- POSROUTING: Verifica e altera se necessário campos do cabeçalho, antes que o pacote saia do *firewall*. É utilizado principalmente para conversão de endereços NAT.

A *chain* INPUT está presente nas tabelas *filter* e *mangle*; a *chain* OUTPUT nas tabelas *filter*, NAT, *mangle* e *raw*; FORWARD nas tabelas *filter* e *mangle*; a *chain* PREROUTING nas tabelas NAT, *mangle* e *raw*; e a *chain* POSTROUTING nas tabelas NAT e *mangle* (Figura 6).

Figura 6 - As quatro tabelas e suas chains padrão.



Fonte: Autoria própria (2017).

2.3.3 **Targets**

Os *targets* (alvos) definem uma ação a ser realizada quando o pacote é atendido por uma regra. Também são utilizados para definir a política padrão de uma *chain* caso nenhuma regra da *chain* se aplique ao pacote. Alder et al. (2007) e Petersen (2008) definem os *targets* mais utilizados pelo Netfilter:

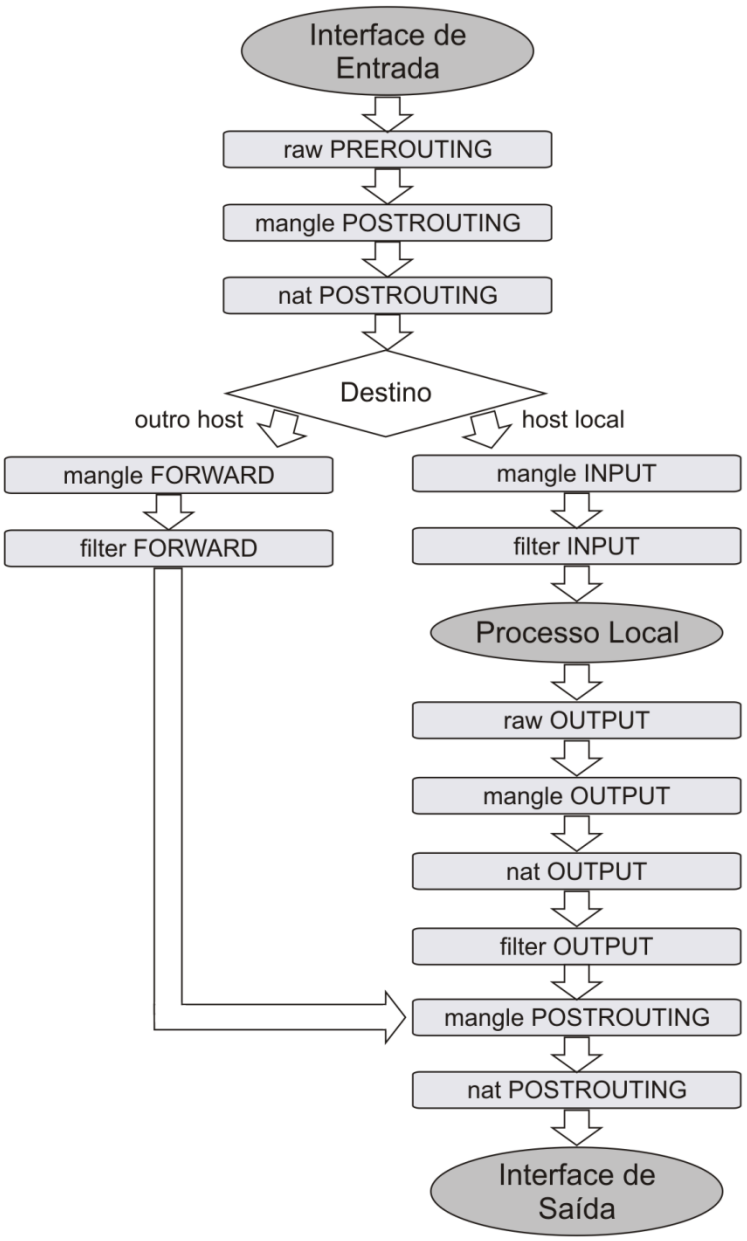
- ACCEPT: O pacote é autorizado a passar pelo *firewall*
- DROP: O pacote é bloqueado pelo *firewall*, sem qualquer notificação ao *host* de origem
- REJECT: O pacote é bloqueado, e o *host* de origem é notificado sobre o bloqueio.
- QUEUE: Deixa o pacote inativo, para ser acessado por algum processo do sistema operacional.
- RETURN: No caso de *chains* criadas pelo usuário, o pacote volta para a *chain* anterior. Para *chains* definidas pelo sistema, é aplicado o target padrão.
- LOG: A verificação do pacote é documentada no log do sistema (syslog).

Existem ainda outros *targets*, para situações mais específicas. O Netfilter permite que diversos módulos sejam adicionados, garantindo novos *targets* e funcionalidades. Alguns exemplos de *targets* específicos são SNAT (*Source Network Address Translation*) e DNAT (*Destination Network Address Translation*) que modificam o endereço IP para realizar NAT, TOS (*Type of Service*) para alterar o tipo de serviço do pacote, e TCPMSS (*TCP Maximum Segment Size*) para alterar o tamanho permitido para pacotes em uma conexão (Petersen, 2008).

2.3.4 **Fluxo de pacotes no Netfilter**

Entendendo os conceitos de tabelas, *chains* e *targets*, pode-se sumarizar como ocorre a passagem dos pacotes pelo Netfilter através de um fluxograma. A Figura 7 mostra o fluxo de pacotes através das *chains* padrão.

Figura 7 - Fluxo de pacotes no Netfilter



Fonte: PETERSEN (2008)

2.4 Iptables

O iptables é a ferramenta de código aberto, criada para alterar o funcionamento do Netfilter, aplicando a ele as políticas definidas para garantir a segurança e controle do fluxo de dados em uma rede. Deste modo, o iptables trabalha como uma interface do usuário com o Netfilter, realizando a manutenção de tabelas e *chains*, mas não efetuando nenhum controle sobre os pacotes, função esta do Netfilter.

A ferramenta iptables foi criada, e é mantida pelos mesmos desenvolvedores do Netfilter. Seu *site* oficial (netfilter.org) define como principais recursos a listagem, adição, remoção e modificação do conjunto de regras para filtragem de pacotes. É importante observar que o iptables lida apenas com endereços IPv4. Para endereços IPv6, deve-se utilizar a ferramenta ip6tables.

É através da linha de comando do sistema operacional que o iptables é executado, sucedido por um conjunto de *flags* que definem os comandos que configurarão o *firewall*. Por padrão, todos os comandos operam sobre a tabela *filter*. Para definir ações em outras tabelas, deve-se utilizar a *flag* –t. Por exemplo, o comando "*iptables –t mangle –N TOS_SMTP*" cria uma nova *chain* chamada "TOS_SMTP" na tabela *mangle*.

As *flags* podem ser dividas em comandos e opções. Comandos definem ações no *firewall*, como adição e remoção de regras e *chains*. Já as opções são parâmetros utilizados pelos comandos. Comandos são identificados com *flags* de letras maiúscula, e opções com *flags* de letras minúsculas (PETERSEN, 2008).

Embora este conjunto básico de *flags* realize boa parte das operações de filtragem de pacotes, ainda existem outras para situações mais específicas, como definição de tipo de pacote TCP e ICMP, para filtragem por controle de estado, ou para NAT.

As Tabelas 1 e 2 sumarizam os comandos e opções mais comuns do iptables:

Tabela 1 - Lista de comandos do iptables.

Comando	**Função**
-A chain	Adiciona uma regra a uma *chain*.
-D chain [num_regra]	Remove regra de número especificado de uma *chain*.
-I chain [num_regra]	Insere regra de número especificado em uma *chain*. Se nenhum número for informado, insere como primeira regra.
-R chain num_regra	Substitui regra de número especificado em uma *chain*.
-L [chain]	Lista as regras de uma *chain* (Ou de todas, se nenhuma for especificada).
-E chain chain2	Renomeia uma *chain*.
-F [chain]	Remove todas as regras de uma *chain*.
-N chain	Cria uma nova *chain*.
-X chain	Remove uma *chain* criada por usuário.
-P chain target	Muda o target padrão de uma *chain*.

Fonte: http://ipset.netfilter.org/iptables.man.html

Tabela 2 - Lista de opções do iptables.

Opções	**Função**
-p [!] protocolo	Especifica um protocolo. Exemplo: TCP, UDP, ICMP.
-s [!} endereço [!] [porta[:porta]]	Especifica um endereço de origem e opcionalmente uma porta, ou intervalo de portas. Exemplo: 1050:1065.
--sport [!] porta[:porta]	Especifica uma porta ou intervalo de portas de origem.
-d [!] endereço [!] [porta[:porta]]	Especifica um endereço de destino e opcionalmente uma porta, ou intervalo de portas.
--dport [!] porta[:porta]	Especifica uma porta ou intervalo de portas de destino.
-i [!] nome	Especifica a interface de entrada de pacotes (Exemplo: eth0). É utilizado apenas pela *chain* INPUT.
-j target	Especifica o *target* da regra.
-n	Especifica que os endereços e portas sejam mostrados na forma numérica, evitando o uso de nomes de domínio. É utilizado em conjunto com o comando –L.
-o [!] nome	Especifica a interface de saída de pacotes. Apenas utilizado pelas *chains* FORWARD e OUTPUT.
-t tabela	Especifica a tabela onde o comando será executado. Caso esta opção não seja utilizada, o comando executará na tabela *filter*.
!	Operador de negação. Por exemplo: “-o ! eth1” aplicará a regra a todas as interfaces de saída, exceto eth1.

Fonte: http://ipset.netfilter.org/iptables.man.html

Para exemplificar o uso do iptables, abaixo são apresentadas alguns comandos que podem ser executados, acompanhados de um breve comentário sobre seu significado.

- iptables –N SERVIDOR
  Cria uma *chain* chamada “SERVIDOR” na tabela *filter*.
- iptables –A SERVIDOR –p ICMP –i eth0 –j DROP
  Cria uma regra na *chain* SERVIDOR que não permite a entrada de pacotes ICMP pela interface eth0.
- iptables –A SERVIDOR –p TCP -s 192.168.0.5 --sport ! 23 –j REJECT
  Cria uma regra na *chain* SERVIDOR, que nega pacotes TCP do endereço 192.168.0.5 que não sejam originados da porta 23, enviando uma mensagem de recusa a origem.
- iptables –P INPUT DROP
  Altera a política padrão da *chain* INPUT para rejeitar pacotes que não se enquadrem em nenhuma de suas regras.

3 O SISTEMA FRCHECKER

Desenvolvido por Carbonera (2009), o FRChecker (Firewall Rules Checker) é um protótipo de um sistema para verificação de inconsistências nas regras do *firewall* Netfilter. Este Capítulo descreve o funcionamento deste sistema, e as abordagens que utiliza para encontrar problemas de configuração.

Com o amplo uso dos serviços de rede, a necessidade de manter um *firewall* devidamente configurado é cada vez maior. Porém, tal tarefa pode ser exaustiva, dependendo do tamanho do conjunto de regras de *firewall* e a frequência de alteração nas mesmas. Com o intuito de facilitar essa manutenção, diversas ferramentas foram implementadas para auxílio ao administrador de rede.

Moore (2007) categorizou os diferentes tipos de ferramentas para manter o *firewall* bem configurado:

- Verificadores de vulnerabilidades: realiza tarefas como descobrir portas abertas, e também problemas de segurança no sistema de *firewall*.
- Farejadores de pacotes: analisa os pacotes, podendo assim determinar que tipos de conteúdo estão transitando pela rede.
- Analisadores de *logs*: coletam e analisam *logs* gerados pelo *firewall*, rastreando os acessos negados e permitidos.
- Verificadores de desempenho: organizam as regras para tornar o trânsito de pacotes pelo *firewall* mais eficaz.
- Analisadores dos conjuntos de regras: verificam regras que estão se sobrepondo, são redundantes, ou não utilizadas, e também regras que oferecem riscos a segurança da rede.

A ferramenta FRChecker se enquadra na categoria dos analisados dos conjuntos de regras. Deste modo, realiza uma análise nas regras utilizadas pelo iptables e informa ao usuário os possíveis conflitos que podem ocorrer entre elas, além de verificar possíveis problemas na política de segurança. O sistema utiliza apenas o conjunto de regras da tabela *filter*, primeiro por se tratar de um protótipo, e segundo pois a grande maioria das regras costumam ser gerenciadas nesta tabela (CARBONERA, 2009).

O FRChecker foi desenvolvido na linguagem de programação Java, sendo multiplataforma, podendo ser executado em qualquer sistema que possua uma máquina virtual

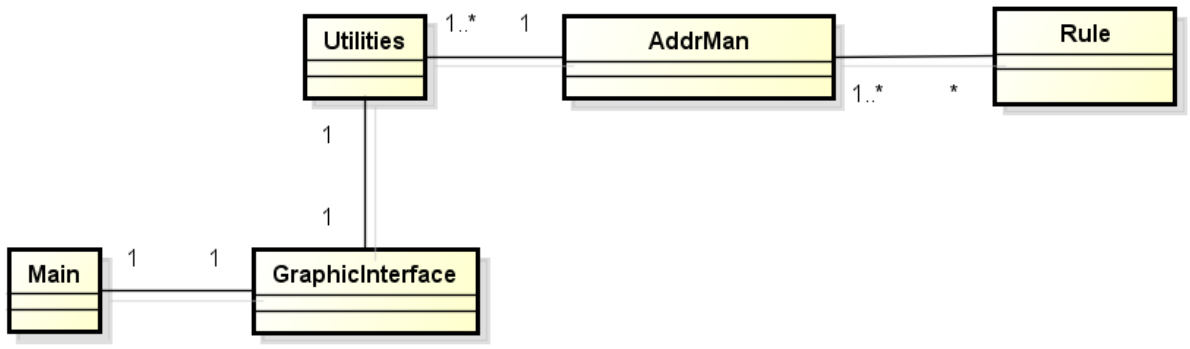
Java (JVM) instalada. É o caso do Linux, sistema operacional aonde o Netfilter opera.

A execução do sistema se divide em etapas:

1. Importação do arquivo com as regras de *firewall*, obtido pelo iptables.
2. Organização das regras em objetos Rule, contidos em uma classe Vector.
3. Verificação das regras, buscando conflitos e problemas na política de segurança.
4. Cálculo da complexidade do *firewall*.

A estrutura do sistema está definida pelo diagrama de classes (Apêndice A). O modelo de domínio (Figura 8) sumariza a relação entre as classes.

Figura 8 - Modelo de domínio do FRChecker.



Fonte: Baseado em CARBONERA, 2009

3.1 Importação e Estruturação das Regras

Para verificar as regras do *firewall*, é necessário gravá-las em um arquivo texto. Para gerar este arquivo, a ferramenta iptables é executada utilizando as *flags* –L e –n, aonde deve ser também informado o caminho de saída. Por exemplo: "*iptables –L –n > /caminho/do/arquivo.txt*".

Para que o FRChecker analise as regras, é necessário carregar na memória o arquivo texto gerado através da opção de importação das regras. Ao carregar o arquivo, o FRChecker verifica linha a linha, ignorando linhas em branco, e linhas de cabeçalho. Um *parser* (componente que analisa sequências de entrada de texto de acordo com sua estrutura) organiza as linhas válidas, removendo espaços em branco, e inserindo o conteúdo em objetos da classe Rule. A Figura 9 ilustra um trecho de um arquivo obtido pelo iptables..

Figura 9 - Trecho de arquivo exportado pelo iptables.

```
Chain VOIP@INPUT (1 references)
target     prot opt source          destination
ACCEPT     tcp  --  0.0.0.0/0       200.175.102.42      multiport dports 8000,5060,1571

Chain VOIP@OUTPUT (1 references)
target     prot opt source          destination

Chain VPN@FORWARD (1 references)
target     prot opt source          destination
ACCEPT     all  --  172.16.255.25   192.168.0.0/16
ACCEPT     all  --  172.16.255.35   192.168.0.2
ACCEPT     all  --  172.16.255.31   192.168.0.0/16
ACCEPT     tcp  --  172.16.255.33   192.168.0.10        tcp dpt:3389
ACCEPT     tcp  --  172.16.255.48   192.168.0.7
ACCEPT     tcp  --  172.16.255.48   192.168.0.5
ACCEPT     all  --  172.16.255.46   0.0.0.0/0
ACCEPT     all  --  172.16.255.47   192.168.2.0/24
ACCEPT     tcp  --  172.16.255.49   0.0.0.0/0
ACCEPT     tcp  --  172.16.255.30   192.168.0.2         tcp dpt:3389
ACCEPT     all  --  172.16.255.20   0.0.0.0/0
ACCEPT     all  --  172.16.255.21   0.0.0.0/0
ACCEPT     all  --  172.16.255.22   0.0.0.0/0
ACCEPT     all  --  172.16.255.28   192.168.0.10
ACCEPT     tcp  --  172.16.255.23   0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0       172.16.255.0/24
ACCEPT     47   --  0.0.0.0/0       0.0.0.0/0
ACCEPT     tcp  --  0.0.0.0/0       200.175.102.42      tcp dpt:1723
LOG        all  --  172.16.255.0/24 192.168.0.0/16      LOG flags 0 level 7 prefix `logfirewall (LOG):'
DROP       all  --  172.16.255.0/24 0.0.0.0/0
```

Fonte: Autoria própria (2017).

3.1.1 **Classe *Rule***

A classe *Rule* define os atributos que caracterizam uma regra de *firewall*. É composto por:

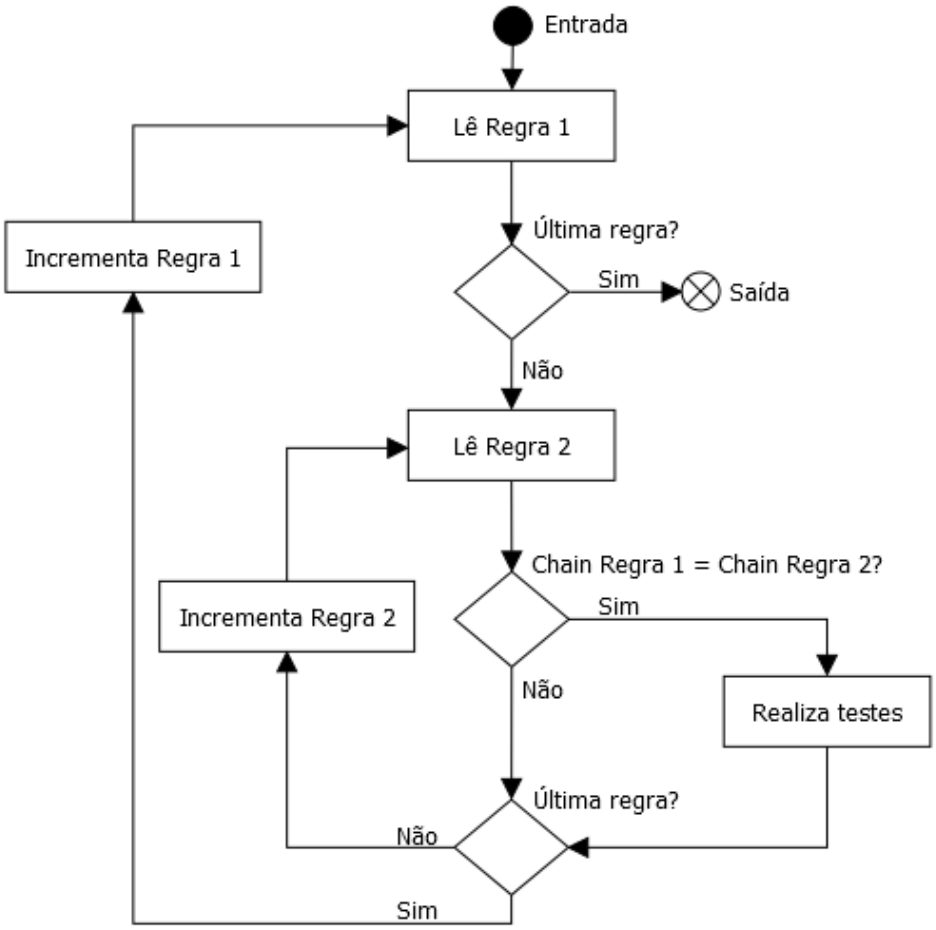
- *Order*: especifica a ordem de execução no conjunto de regras.
- *Protocol*: O protocolo do pacote (TCP, UDP, etc).
- *Source IP*: o endereço IP de origem.
- *Source Port*: a porta de origem.
- *Destination IP*: o endereço IP de destino.
- *Destination Port*: a porta de destino
- *Action*: define como o *firewall* se comportará se a regra for aplicada (*target* da regra).
- *Original Chain*: a *chain* em que a regra está contida.
- *Policy*: a política padrão da *chain*.
- *Advices*: neste atributo são armazenados os avisos de possíveis problemas detectados durante a verificação da regra, caso houverem.
- *Conflict Rules*: aqui são definidas outras regras que estão gerando conflitos.

Quando criado, o objeto da classe *Rule* é inserido em um conjunto nativo da linguagem Java, do tipo *Vector*. Deste modo, é a classe *Vector* quem gerencia a organização da lista de regras.

3.2 Verificação das Regras

Após as regras serem carregadas e organizadas no sistema, pode-se realizar a aferição das mesmas. através de uma análise de possíveis problemas de configurações e também boas práticas em políticas de segurança. Carbonera (2009) utilizou as pesquisas de Wool (2004), Al-Shaer (2005) e Capretta et al (2007) para este fim. Todas as regras são comparadas umas com as outras, desde que estejam na mesma *chain* (Figura 10).

Figura 10 - Diagrama de verificação de regras



Fonte: CARBONERA (2009)

As operações executadas pelo FRChecker para aferição são definidas nas seções a seguir. As funções de conflito de IPs, portas e protocolos são utilizadas como funções auxiliares para as demais.

3.2.1 **Conflito de IPs**

Esta função compara duas regras, verificando a relação entre os endereços IP da primeira e da segunda regra. Como possíveis resultados, tem-se o conjunto de endereços IP da regra 1 sendo um subconjunto ou superconjunto dos endereços da regra 2, os dois conjuntos sendo equivalentes, totalmente distintos, ou ainda, intercalados. Esta verificação ocorre através da conversão dos endereços IP em números binários, que são inseridos em uma tabela ordenada. O modo como os endereços das regras 1 e 2 ficam contidos nessa tabela definem a sua relação. As Tabelas 3, 4 e 5 exemplificam esta abordagem, aonde na primeira coluna está expresso o endereço IP em formato decimal, na segunda coluna o mesmo endereço em formato binário, e na terceira coluna, a qual regra o intervalo de endereços pertecem.

Tabela 3 - Verificação de conflito de endereços IP. Regra 2 é um subconjunto da regra 1.

Endereço IP	**Endereço IP (binário)**	**Regra**
192.168.0.2	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	1
192.168.0.15	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	2
192.168.0.18	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0	2
192.168.0.50	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0	1

Fonte: Autoria própria (2017).

Tabela 4 - Verificação de conflito de endereços IP. Regras distintas.

Endereço IP	**Endereço IP (binário)**	**Regra**
192.168.0.2	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	1
192.168.0.50	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0	1
192.168.0.127	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1	2
192.168.0.254	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0	2

Fonte: Autoria própria (2017)

Tabela 5 - Verificação de conflito de endereços IP. Regras intercaladas.

Endereço IP	**Endereço IP (binário)**	**Regra**
192.168.0.2	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	1
192.168.0.15	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	2
192.168.0.50	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0	1
192.168.0.127	1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1	2

Fonte: Autoria própria (2017).

3.2.2 Conflito de Portas

Verifica se as mesmas portas ou intervalos de portas estão sendo utilizados por duas regras diferentes. É implementado utilizando um vetor de inteiros de *n* posições, aonde *n* é o número total de portas que podem ser utilizadas (65535). O vetor é inicalizado com valor 0, e as portas utilizadas pela primeira regra são setadas com valor 1. Na verificação da segunda regra, é apenas checado se algum valor de suas portas no vetor equivale a 1.

3.2.3 Conflito de Protocolos

Determina se há conflito de protocolo caso o protocolo utilizado por duas regras seja o mesmo (TCP, UDP, ICMP, ALL, etc). A exceção ocorre quando qualquer uma das regras possui protocolo definido como “ALL”. Nesta situação o FRChecker também determina conflito.

3.2.4 Número de Conexões SSH

Esta função retorna o número de conexões SSH[2] (*Secure Shell*) autorizadas a acessarem o *firewall* (*chain* INPUT, porta 22). Tendo em vista a necessidade do *firewall* ser configurado por um número limitado de administradores para manter sua consistência, se mais de cinco conexões são encontradas, o FRChecker retorna um aviso ao usuário.

3.2.5 Porta RPC

Verifica se o *firewall* está aceitando a chamada de procedimentos remotos (RPC[3]) através da porta 111. Esta configuração pode originar operações mal intencionadas ao *firewall*.

3.2.6 Qualquer Origem e Destino

Verifica se alguma regra está permitindo o tráfego de qualquer IP de origem a qualquer IP de destino. Na pesquisa de Wool (2004), este foi o erro de segurança mais encontrado, presente em mais de 90% dos *firewalls* verificados. Esta análise é realizada verificando endereços IP de origem e destino de número “0.0.0.0”.

3.2.7 Porta Telnet

O Telnet é um protocolo de comunicação utilizado para acessar remotamente uma

[2] Protocolo de comunicação criptografada entre computadores

[3] Protocolo de comunicação que possibilita a uma máquina remota executar procedimentos e subrotinas.

máquina (TANENBAUM, 2003). Por não implementar criptografia de dados, o Telnet pode oferecer riscos a rede, por isso o FRChecker informa ao usuário se conexões Telnet estão sendo permitidas no *firewall*. Para isto, é verificado cada pacote com destino ao *firewall*, cuja a porta de destino seja 23.

3.2.8 **Teste de Política Padrão**

Verifica se a política padrão de alguma das *chains* padrão da tabela *filter* (INPUT, FORWARD, OUTPUT) está definida como “ACCEPT”. Isto pode significar um erro de configuração ou a adoção de uma política de segurança muito branda.

3.2.9 **Verificação de Correlação**

Verifica o *target*, o IP destino e origem, a porta de destino, e o protocolo de duas regras, retornando se estão correlacionadas. Regras correlacionadas podem gerar problemas ao *firewall*, já que a primeira regra pode anular a verificação da segunda. Ocorre correlação entre regras se as seguintes relações são confirmadas:

- Mesmo protocolo entre duas regras.
- Mesmas portas de destino entre duas regras.
- Endereço IP de origem da primeira regra está intercalado com o da segunda ou é um superconjunto dela, e endereço IP de destino da primeira regra está intercalado com o da segunda ou é um subconjunto dela; ou o endereço de origem da primeira regra está intercalado com o da segunda ou é um subconjunto dela, e endereço de destino da primeira regra está intercalado com o da segunda ou é um superconjunto dela.

3.2.10 **Verificação de Exceções**

Verifica o IP destino e origem, a porta de destino, e o protocolo de duas regras, retornando um aviso caso ocorrer exceção entre elas. Segundo Al-Shaer (2005), uma exceção ocorre quando uma regra permite algum tráfego que posteriormente é bloqueado por outra regra. O FRChecker verifica este problema quando ocorrem os seguintes conflitos:

- Mesmo protocolo entre duas regras.
- Mesmas portas de destino entre duas regras.
- Mesmo endereço IP de origem, e o endereço de destino da primeira regra é um subconjunto da segunda; ou endereço de origem da primeira regra é um

subconjunto da segunda, e o endereço de destino é o mesmo para ambas; ou se ambos os endereços de destino e origem da primeira regra são subconjuntos da segunda regra.

3.2.11 Verificação de Redundâncias

Define se duas regras possuem as mesmas características. A verificação de redundâncias não denota um erro de configuração ou uma mudança de política de segurança, existindo apenas para deixar as regras de *firewall* mais coesas. As regras devem possuir as seguintes características:

- Mesmo protocolo entre duas regras.
- Mesmas portas de destino entre duas regras.
- Os endereços IP de origem e destino da primeira regra devem ser iguais ou subconjuntos da segunda regra.

3.2.12 Verificação de Sombreamento

O sombreamento (*shadowing*) ocorre quando há *targets* diferentes entre duas regras, e a primeira regra é um supergrupo da segunda regra. Deste modo, a segunda regra nunca é executada. Segundo Al-Shaer (2005), isto indica um problema sério de configuração. No FRChecker esta aferição é realizada comparando-se todos os conflitos abaixo:

- Mesmo protocolo entre duas regras.
- Mesmas portas de destino entre duas regras.
- Os endereços IP de origem e destino da primeira regra devem ser iguais ou superconjuntos da regra 2.

3.3 Cálculo de Complexidade do *Firewall*

Em seu estudo, Wool (2004) concluiu que o modo de determinar a complexidade de configuração de *firewalls* era até então apenas subjetivo, e definiu um cálculo para mensurar as características do *firewall* e suas regras. Estipulando que o número de diferentes interfaces de rede em um sistema de *firewall* aumenta quadraticamente a complexidade de configuração, a fórmula de complexidade foi definida como

$$CF = Regras + Objetos + Interfaces \times \left(\frac{Interfaces - 1}{2}\right)$$

onde regras é a soma do conjunto de regras existentes no *firewall*, Objetos o número de diferentes *hosts* presentes nas regras, e Interfaces o número de interfaces de rede que se conectam ao sistema de *firewall*.

O cálculo de complexidade de *firewall* foi implementado no FRChecker, e assim, o administrador pode ter um valor que define a complexidade de seu sistema, e saber de que modo as alterações nas regras e políticas de segurança a afetam.

4 OUTRAS FERRAMENTAS DE VERIFICAÇÃO DE REGRAS

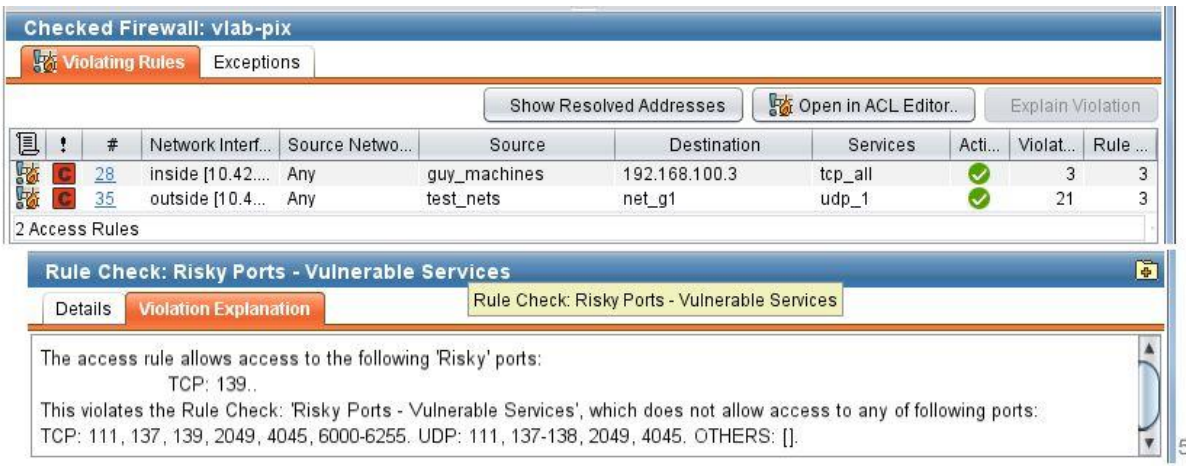
Buscando sistemas semelhantes ao FRChecker, outras ferramentas que se enquadram na categoria dos analisadores de regras foram pesquisadas, a fim de determinar abordagens diferentes de verificação. As seções a seguir descrevem estas ferramentas.

4.1 Skybox Firewall Assurance

O Skybox Firewall Assurance[4] está disponível para os sistemas operacionais Microsoft Windows Server 2012, 7, 8 e 10; Linux RedHat e CentOS. Este sistema realiza a detecção e importação de regras de mais de vinte modelos de *firewalls* diferentes, alguns de modo automático e outros através de *scripts* configuráveis. No sistema, os *firewalls* ficam catalogados, em conjunto com as interfaces de redes referentes a cada um. É possível definir políticas de segurança para cada *firewall*, sendo estas customizadas, ou disponíveis por padrão pelo próprio sistema. O Skybox também analisa se os *firewalls* estão devidamente configurados, identificado por exemplo, se estão utilizando a senha padrão de fábrica.

Quanto a verificação das regras, identifica acesso a serviços que podem oferecer riscos a rede, endereços de destino ou origem indefinidos, ou regras que permitem o acesso de múltiplos endereços e portas. O sistema também classifica os riscos de acordo com a severidade, e já conta com um conjunto de verificações pré-determinadas, sendo que outras podem ser adicionadas pelo usuário. A Figura 11 ilustra a verificação de regras de *firewall* pelo sistema.

Figura 11 - Verificação de regras no Firewall Assurance.



Fonte: http://www.skyboxsecurity.com

[4] www.skyboxsecurity.com

Para otimizar a execução do *firewall* e auxiliar na limpeza de regras, o sistema identifica sombreamento (quando uma regras impede o acesso a uma ou mais regras) e redundância (quando duas regras possuem a mesma função), ilustrando que elemento da regra está causando estes problemas. Esta ferramenta também mostra o índice de uso de cada regra, e assim o usuário pode priorizar a execução de regras mais acessadas, e eliminar regras sem uso.

O Skybox Firewall Assurance também é composto por um gerenciador de mudanças. Deste modo, guarda registro das mudanças efetuadas nas regras dos *firewalls*, e também de requisições de mudanças pendentes.

4.2 AlgoSec Firewall Analyzer

A ferramenta AlgoSec Firewall Analyzer[5] suporta 17 modelos de diferentes fabricantes de *firewall*.

Esta ferramenta mostra a topologia da rede automaticamente, sendo possível também adicionar dispositivos através de seus endereços IP. Também realiza auditoria de *firewalls* de acordo com normas técnicas da PCI-DSS (*Payment Card Industry Data Security Standard*), ISO/IEC 27001, dentre outras. Estas normas definem tanto o modo como o *firewall* é configurado, como artefatos auxiliares (como documentação da topologia de rede e descrição de cada regra). O AlgoSec Firewall Analyzer possui funções para cumprir alguns requisitos destas normas, e aponta quais regras e configurações estão interferindo para que as normas não sejam atendidas.

Dentre outros recursos, o AlgoSec Firewall Analyzer faz análise de risco, identificando acesso a serviços que podem comprometer a segurança de rede, ou também quando uma regra habilita um intervalo muito grande de portas. Este conjunto de itens está definido como padrão pelo sistema, mas outros podem ser definidos pelo usuário, que também pode atribuir um grau de severidade a cada risco.

Com relação as funções de otimização e limpeza de regras, este sistema identifica regras que estão em desuso (comparando relatórios por certo intervalo de tempo) ou sombreadas (sendo possível identificar um conjunto de regras que está impedindo acesso a outras). Também identifica *hosts* que não estão listados em nenhuma regra e mostra as regras mais utilizadas pelos *firewalls*, sugerindo reordenações, calculando inclusive a possível melhoria de desempenho caso a reordenação seja feita. O AlgoSec Firewall Analyzer também realiza a remoção de regras diretamente no *firewall*, simplificando sua manutenção.

[5] www.algosec.com

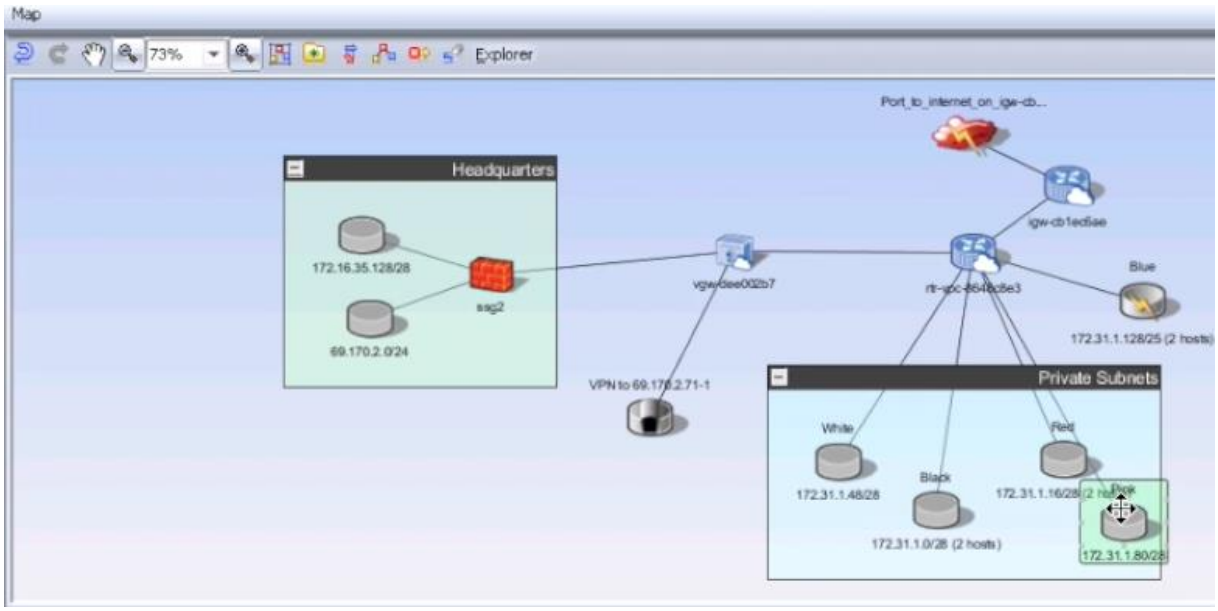
Ainda é possível procurar por regras de acordo com suas características e controlar quais regras sofreram mudanças, através de um *log* de alterações nos *firewalls*.

4.3 RedSeal Networks

O sistema RedSeal Networks[6] é um sistema de segurança de rede que suporta 14 modelos de diferentes fabricantes de *firewall*.

Dentre suas funções, esta ferramenta apresenta a topologia gráfica da rede, dividida em subredes (Figura 12). Através dessa topologia, pode-se verificar de que modo cada *host* se conecta com outros, e que serviços podem acessar. Também informa qual regra de *firewall* está habilitando cada acesso.

Figura 12 - Topologia de rede exibida pelo RedSeal Networks.



Fonte: www.redseal.net

Assim como o AlgoSec, o RedSeal também implementa a auditoria de *firewall*, de acordo com normas técnicas como a PCI-DSS e SOX (*Sarbanes-Oxley*), identificando quais regras e dispositivos estão interferindo na segurança. Adicionalmente, também verifica a configuração dos *firewalls*, de acordo com boas práticas definidas pelos fabricantes. Por exemplo, identifica senhas de administração muito fracas, ou a adoção de políticas de segurança permissivas.

Assim como os sistemas anteriores, a ferramenta também identifica regras inativas ou

[6] www.redseal.net/

redundantes. No entanto, nada na documendação foi citado a respeito de sombreamento de regras. Também realiza a análise da frequência de uso de cada regra, auxiliando o administrador na reordenação do conjunto de regras para melhorar o desempenho de verificação.

4.4 FireMon Security Manager

Atua em 15 modelos diferentes de *firewall*. O FireMon Security Manager[7] mapeia a topologia de rede, apresentando os caminhos que oferecem riscos, e classificando-os de acordo com o grau de severidade. Simulações de correções podem ser feitas para verificar o impacto das mudanças na segurança da rede.

O FireMon Security Manager realiza o gerenciamento de mudanças nas configurações de *firewall*, mantendo histórico de alterações e relacionando regras com observações a respeito de suas inserções ou alterações. Na necessidade de criação de uma nova regra, o sistema determina os possíveis riscos de segurança e também identifica regras já existentes, que podem ser modificadas para atender os requisitos da nova regra.

Com relação a limpeza e otimização dos *firewalls*, assim como os outros sistemas, determina quais regras são mais utilizadas, bem como identifica redundancias e sombreamento.

O sistema FireMon Security Manager ainda realiza auditoria de *firewall*. Vem previamente configurado com a análise de boas práticas, como as da norma PCI-DSS. Novas análises podem ser inseridas pelo usuário.

4.5 Ferramentas específicas para iptables

Com relação a soluções de avaliação de regras de *firewall* específicas para iptables e de código aberto, duas ferramentas foram analisadas.

A ferramenta ITVal (MARMORSTEIN, KEARNS, 2005) foi implementada para uso exclusivo com arquivos exportados pelo iptables, do mesmo modo que o FRChecker. Essa ferramenta consiste em um analisador sintático, aonde são carregados *scripts* criados pelo usuário, que consultam as regras de *firewall* em uma linguagem estilo SQL. Desta forma, o usuário pode consultar por exemplo, pacotes autorizados a chegarem a determinado *host*, ou quais *hosts* estão permitidos a acessarem determinado serviço, definido pelo protocolo e número da porta de rede.

O iptables analyzer (ipta)[8] opera através do arquivo de *logs* que é gerado pelo iptables.

[7] www.firemon.com
[8] http://ichimusai.org/projects/ipta/

O usuário deve inserir regras no *firewall* que gerem *logs* de acordo com o tipo de tráfego de rede a ser analisado. O arquivo de *logs* é posteriormente importado a um banco de dados MySQL, que é consultado através de funções predefinidas pelo ipta, ou ainda por consultas definidas pelo usuário. Pode, por exemplo, mostrar as portas de rede que tem mais acessos negados, ou as interfaces de rede mais acessadas por tipo de *target*.

4.6 Considerações Finais

A análise destes produtos mostrou que todos possuem características bem similares (Tabela 6). Embora ofereçam recursos avançados, como o suporte a múltiplos modelos de *firewall*, e funções de gerenciamento de risco e mudanças, suas funções de verificação e limpeza de regras são semelhantes ao FRChecker.

Tabela 6 - Comparativo das ferramentas de verificação de regras de *firewall*.

Empresa	Skybox	AlgoSec	RedSeal	FireMon
Produto	Firewall Assurance	Firewall Analyzer	RedSeal Networks	Security Manager
Modelos de *firewalls* suportados	20	17	14	15
Topologia visual da rede	Não	Sim	Sim	Sim
Auditoria das regras	Sim	Sim	Sim	Sim
Limpeza das regras e otimização	Sim	Sim	Sim	Sim
Gerenciamento de mudanças	Sim	Sim	Sim	Sim

Fonte: Autoria própria (2017).

Embora se conheçam as funções destes sistemas, não foi possível averiguar de que modo elas são implementadas, pois tratam-se de produtos comerciais, sem código fonte ou documentações sobre sua codificação disponíveis.

Todos os sistemas analisados avaliam o índice de utilização das regras, o que seria um incremento relevante ao FRChecker. Este recurso é possível pois o Netfilter mantém *log* do número de pacotes que passaram por cada regra, e traria os seguintes benefícios:

1. Avaliação de regras que não estão sendo utilizadas, seja por um erro de configuração ou por não serem mais necessárias ao negócio;
2. Classificação das regras por ordem de utilização, podendo-se otimizar a execução do *firewall*, melhorando seu desempenho;

3. Análise de mudança nas regras, possibilitando ao administrador identificar alterações indevidas na política de segurança.

No entanto, seria necessária a implementação de um banco de dados, aonde as regras seriam periodicamente salvas, e comparadas em determinados intervalos de tempo. Esta função fugiria do escopo desta pesquisa, que é a verificação de regras por um mecanismo de inferência.

Com relação as ferramentas de código aberto, ambas são voltadas a verificação manual dos conflitos ou políticas de segurança, ou exigem alterações nas regras já existentes para melhor avaliação de seus resultados. Nenhuma das ferramentas pesquisadas efetua de maneira automatizada as verificações, e também não levam em conta a avaliação integral de todas as regras obtidas através do *firewall*.

5 SISTEMAS BASEADOS EM CONHECIMENTO

Sistemas Baseados em Conhecimento (SBC) são programas que modelam conhecimentos sobre determinada área, geralmente obtidos por especialistas humanos, que informam as condições para se resolver um problema. Tipicamente eles são programas computacionalmente complexos e que exigem uma grande quantidade de conhecimento específico.

Um SBC por definição deve solicitar ao usuário para que receba as informações necessárias ao seu processamento (fatos), lidar com estas informações, realizando inferências de acordo com um conjunto de regras estabelecidas até encontrar soluções satisfatórias, detalhando ao usuário de que modo encontrou as soluções. Tal qual um especialista humano, pode estar suscetível a erros. Os resultados obtidos são geralmente não determinísticos, pois lidam com a limitação de recursos computacionais ou de base de conhecimentos (REZENDE, 2005).

Os SBCs têm sido utilizados nas últimas décadas em variadas áreas, incluindo negócios, medicina e engenharia. Um importante exemplo é o sistema MYCIN, para diagnóstico de doenças, desenvolvido nos anos 70. Contendo um conjunto de regras e um motor de inferência, foi o precursor na utilização de inteligência artificial na medicina, além de implementar conceitos novos para a época, como o módulo de explicação de resultados, e a separação entre base de conhecimento e código do programa (GIARRATANO, 2005).

Giarratano (2005) explica que SBCs e Sistemas Especialistas são utilizados como sinônimos. Já Genaro (1986) os separa, sendo os Sistemas Especialistas um subgrupo de SBCs. SBCs podem se basear em conhecimento adquirido de livros e pesquisas, enquanto que Sistemas Especialistas contam sempre com o conhecimento adquirido por um humano com domínio na área para a qual o sistema se destina.

A característica que mais diferencia SBCs de sistemas convencionais é a utilização de busca heurística para obtenção de resultados. Isto resulta em respostas não probabilísticas, mas que podem ser suficientemente úteis em problemas aonde não existam algoritmos conhecidos para resolução, ou os existentes demandem um número excessivo de processamento. Genaro (1986) faz uma analogia entre sistemas convencionais e SBCs, comparando-os com o combate ao contrabando em um país. Pela lógica dos sistemas convencionais, seria necessário vigiar todo o perímetro do território, checando cada pessoa e cada bagagem que ali entrasse. Esta tarefa seria inviável, dada a quantidade de recursos e pessoas para efetivá-la. Do ponto de vista dos SBCs que utilizam mecanismos de heurística, através da análise de eventos de contrabando

e de dados obtidos por profissionais, pode-se focar a vigilância a pontos de maior probabilidade destas ocorrências. Assim, o contrabando não seria integralmente impedido, mas dado o limitado número de recursos para a tarefa, seria satisfatoriamente combatido.

5.1 Estrutura de um SBC

O funcionamento de um SBC pode ser melhor compreendido através da divisão de suas funcionalidades em módulos bem definidos.

O principal módulo é o Núcleo do SBC (NSBC), que é responsável por obter e processar o conhecimento, além de mostrar de que modo os resultados foram obtidos. O núcleo pode ser dividido em submódulos:

- Módulo Coletor de Dados (MCD): é a interface de comunicação com o usuário, pela qual os dados que serão processados são obtidos e validados.
- Motor de Inferência (MI): através de um conjunto de regras, realiza o processamento das informações obtidas pelo MCD, de acordo com a base de conhecimento estabelecida.
- Módulo de Explicações (ME): é responsável em retornar uma explicação ao usuário sobre os resultados obtidos. Geralmente informam como tal resultado foi obtido ou porque determinado resultado não ocorreu.

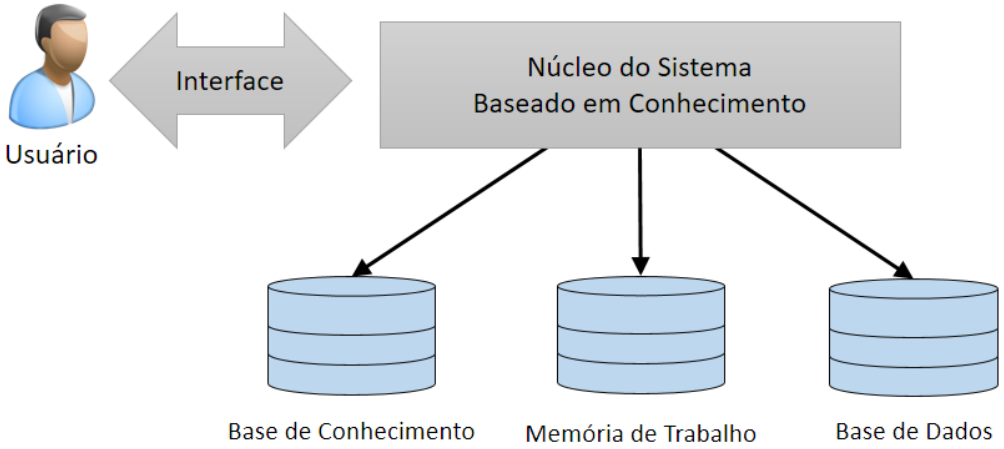
Outro módulo importante é a Base de Conhecimento (BC), que contém o conjunto de informações necessário a obtenção de resultados pelo NSBC. Este módulo é construído através de um especialista que tem o domínio sobre a solução do problema, podendo definir uma sequência de regras para determinadas situações. Cada representação do conhecimento é descrita na forma de uma sentença expressada através de regras de produção, redes semânticas, lógica, etc.

A Memória de Trabalho (MT) é o terceiro módulo. É composto pelas informações (fatos) que serão confrontadas com a BC através do motor de inferência. Também armazena informações preliminares ao resultado final, podendo guardar conclusões parciais. Tem também o objetivo de evitar a repetição de operações e informar os passos utilizados durante o processamento da informação.

O módulo de Base de Dados trata da interação com um banco de dados, obtendo e salvando informações necessárias a execução do SBC.

A Interface é o último módulo, e lida com a comunicação bidirecional entre o usuário e o SBC. A Figura 13 ilustra a estrutura de um SBC e sua interação com dados e usuário.

Figura 13 - Estrutura de um SBC



Fonte: REZENDE (2005)

5.2 Métodos de Inferência

Dentre as abordagens para verificação das regras de inferências, destacam-se o encadeamento regressivo (*back chaining*) e encadeamento progressivo (*forward chaining*). Eles são caracterizados como um conjunto de regras interligadas que começam por um problema e terminam numa solução. Estas abordagens determinam a ordem de inferência sobre as regras, e deste modo, comportam-se diferente ao processar o conhecimento.

O encadeamento progressivo utiliza-se dos fatos obtidos para determinar um resultado, enquanto que o encadeamento regressivo faz o caminho inverso. Ou seja, tenta provar que os fatos encaixam-se em alguma das respostas possíveis.

Giarratano (2005) cita as principais diferenças destes métodos. Encadeamento progressivo é mais aplicado a situações que envolvam planejamento e controle, baseia-se em informações antecedentes para encontrar consequentes, e tem dificuldades em manter um módulo de explicação. Encadeamento regressivo é mais voltado para aplicações de diagnóstico, tem o fluxo de análise que se inicia no resultado, procurando informações que o antecedem, e tem mais facilidade em explicar o porquê de seus resultados.

Prolog é um exemplo de linguagem que utiliza um motor de inferência com encadeamento regressivo, e a linguagem Lisp, utiliza encadeamento progressivo.

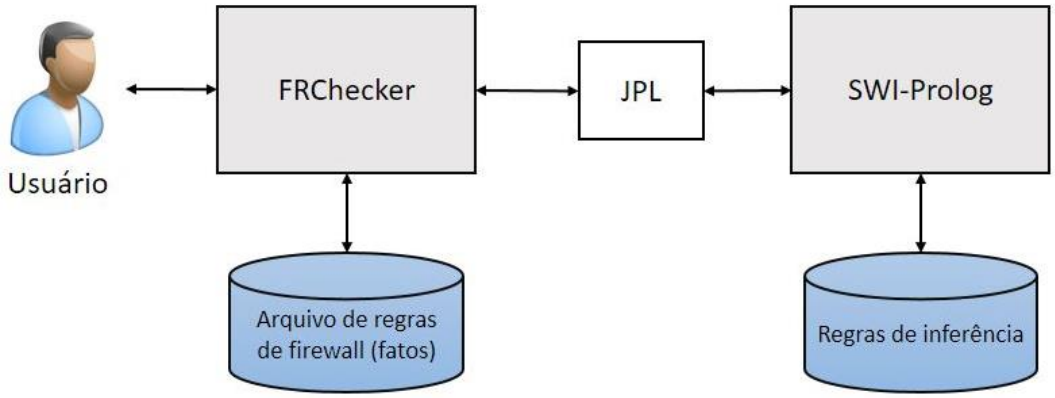
5.3 Considerações Finais

Analisando as características dos SBCs, nota-se que seus conceitos podem ser úteis na análise de regras de *firewall*. Primeiramente, a aferição de regras de *firewall* necessita da consulta de um especialista ou de referencial teórico específico para identificar e modelar as abordagens possíveis de verificação. É interessante manter este conhecimento específico em separado do restante da lógica do sistema. Em segundo, a análise do conjunto de regras, dependendo da abordagem, necessita que todas as regras sejam comparadas umas com as outras. Em *firewalls* com um número grande de regras, esta operação pode ser impossível na prática, devido a quantidade de processamento envolvido. A busca heurística provida pelo motor de inferência seria uma alternativa a esta limitação computacional.

6 PROPOSTA DE SOLUÇÃO

A proposta de solução deste trabalho é baseada na estrutura de um SBC. O FRChecker realiza a interação com o usuário, trata o arquivo com as regras de *firewall*, e comunica-se com a biblioteca JPL. A biblioteca JPL, conforme descrita na Seção 6.5, realiza a comunicação com o motor de inferência, que processa as regras de inferência escritas na linguagem Prolog (Figura 14).

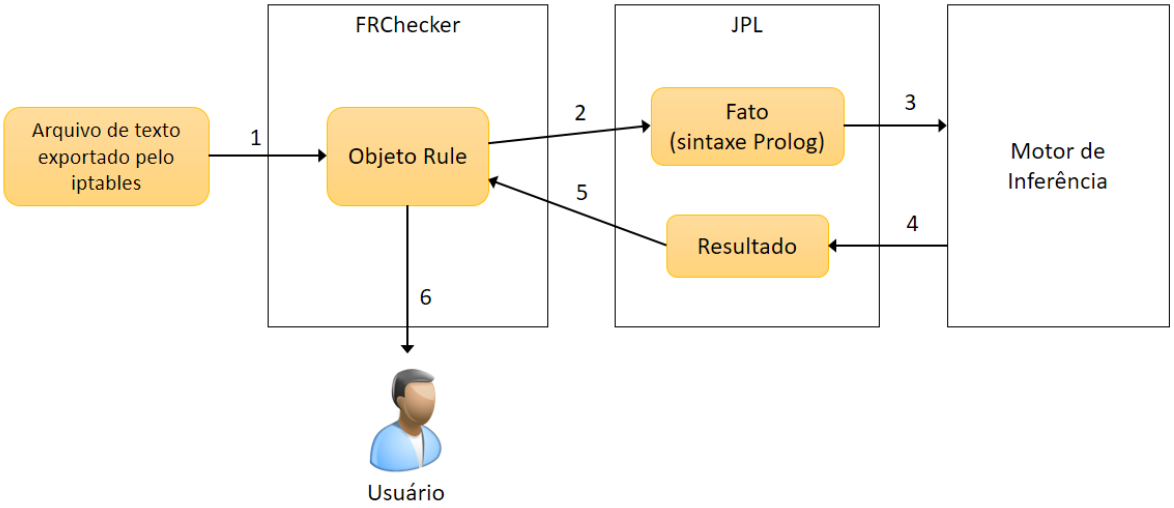
Figura 14 - Arquitetura do sistema proposto.



Fonte: Autoria própria (2017).

Deste modo, o fluxo de dados (Figura 15) começa pela importação do arquivo de regras obtido pelo iptables. Cada regra é organizada em um objeto Rule e transformado em um fato na linguagem Prolog. Esta informação é transferida ao motor de inferência pela biblioteca JPL. O resultado da inferência é retornado ao JPL, que o transfere ao objeto Rule do FRChecker. Por fim, esta informação é tratada e informada ao usuário.

Figura 15 - Fluxograma de dados do sistema proposto.



Fonte: Autoria própria (2017).

As seções a seguir descrevem os componentes correspondentes a cada módulo do SBC, justificando sua escolha.

6.1 Definição do Núcleo do Sistema Baseado em Conhecimento

O Núcleo do Sistema Baseado em Conhecimento é constituído pelo motor de inferência da linguagem Prolog. Esta linguagem foi proposta no início dos anos 70 e, desde então, vem se destacando em diversos problemas envolvendo o processamento de informação, como geração automática de código, verificação de programas, e concepção de linguagens de especificação de alto nível (LUGER, 2004)

Um interpretador Prolog é baseado em declarações lógicas, a partir das quais são realizadas inferências. Conforme Rezende (2005), ao contrário de linguagens imperativas, utiliza encadeamento regressivo.

Como interpretador Prolog é utilizado o SWI-Prolog[9]. O SWI-Prolog possui código livre (Lesser GNU Public License), está em desenvolvimento ativo desde 1986, e oferece um grande número de bibliotecas auxiliares, além de possuir implementações para múltiplos sistemas operacionais. Este interpretador é utilizado em projetos de pesquisa, educacionais e inclusive produtos comerciais.

[9] www.swi-prolog.org/

6.2 Definição da Base de Conhecimento

A Base de Conhecimento está descrita na forma de declarações lógicas na sintaxe da linguagem Prolog, a serem interpretadas pelo SWI-Prolog. É a representação lógica das abordagens de verificação de regras de *firewall*, e em resumo, a BC define as funções da versão modificada do FRChecker.

Através do estudo de sistemas de *firewall* e do FRChecker, as seguintes funções são identificadas e modeladas, formando a BC:

- Identificação de sombreamento entre regras de *firewall*;
- Identificação de exceções entre regras de *firewall*;
- Identificação de redundância entre regras de *firewall*;
- Identificação de correlação entre regras de *firewall*.

Estas funções são selecionadas por serem as de maior custo computacional, sendo necessária a comparação de cada regra de *firewall* com suas sucessoras.

Funções como a identificação de portas que ofereçam riscos a segurança, ou *chains* do Netfilter que utilizem políticas padrão permissivas podem ser implementadas, mas não se beneficiarão das vantagens de um SBC.

6.3 Definição da Memória de Trabalho e Base de Dados

A Base de Dados e Memória de Trabalho são compostas pelo conjunto de regras obtidas por Carbonera (2009), referentes ao *firewall* Netfilter instalado em uma empresa do setor metal-mecânico. O arquivo contém 354 regras organizadas em 38 diferentes *chains* do Netfilter. Deste modo, os resultados obtidos poderão ser comparados com os da versão original do FRChecker.

Os dados obtidos originalmente através do arquivo texto exportado pelo iptables, são carregados no FRChecker, e convertidos em fatos da linguagem Prolog, para em seguida serem analisados pelo motor de inferência do SWI-Prolog.

6.4 Definição da Interface com Usuário

Toda a interação com usuário se dará pela interface gráfica do FRChecker, ficando deste modo, transparente a execução do motor de inferência. Todos os resultados obtidos pelo SWI-Prolog são exibidos na tela, do mesmo modo que ocorre na versão anterior do FRChecker. Assim, quando for identificado algum problema em determinada regra, este problema será informado no campo Comentário, referente a regra.

6.5 Definição da Integração entre Java e Prolog

Dada a eficiência da utilização de Prolog na resolução de problemas específicos, e a popularização da linguagem Java, algumas bibliotecas foram implementadas, procurando integrar programas em Java com interpretadores Prolog. Não apenas bibliotecas foram criadas, como também implementações de interpretadores Prolog inteiramente em Java, e também linguagens novas que misturam as duas sintaxes. Esta Seção analisa algumas implementações que realizam esta integração.

A justificativa de integrar duas linguagens tão distintas dá-se pelos pontos fortes de cada uma. A utilização de um motor de inferência pode muito bem simplificar a resolução de um problema complexo. E a estruturação dos dados em classes e objetos, aliada a enorme variedade de bibliotecas Java, simplifica a organização e manutenção do conjunto de informações.

O InterProlog[10] é uma interface de comunicação capaz de invocar métodos Java numa aplicação Prolog, e executar declarações Prolog num programa Java. Esta interface trabalha em conjunto com XSB Prolog, uma implementação Prolog para sistemas Unix e Microsoft Windows.

A *Application Programming Interface* (API) tuProlog[11] é uma implementação de Prolog em Java. Logo, não é necessaria a utilização de um interpretador, apenas da Java Virtual Machine (JVM) para sua execução. Esta API oferece um motor de inferência customizável, originalmente com recursos mínimos, podendo ser executado em dispositivos móveis.

GNU Prolog for Java[12] também é uma API que implementa o motor de inferência do Prolog na linguagem Java, oferecendo funções básicas.

A biblioteca JPL[13] é uma interface bidirecional de comunicação entre aplicações Java e Prolog. É a biblioteca padrão para integração Java do SWI-Prolog, presente em sua instalação padrão. Devido ao fato do SWI-Prolog ser a implementação de Prolog mais completa, e o JPL ser sua biblioteca padrão para comunicação com aplicações Java, esta interface foi definida para ser utilizada na alteração do FRChecker.

A JPL possui um conjunto de classes que realizam a conversão de tipos de dados em Java e Prolog. A classe *Term* é abstrata, e representa qualquer tipo de variável. Já a classe *Compound* é composta de uma *string* descritiva e um *array* com objetos do tipo *Term*. As demais classes de dados representam tipos distintos, descritos na Tabela 7.

---

[10] http://interprolog.com/java-bridge/
[11] http://apice.unibo.it/xwiki/bin/view/Tuprolog/
[12] http://www.gnu.org/software/gnuprologjava/
[13] http://jpl7.org/

Tabela 7 - Tipos de dados da biblioteca JPL

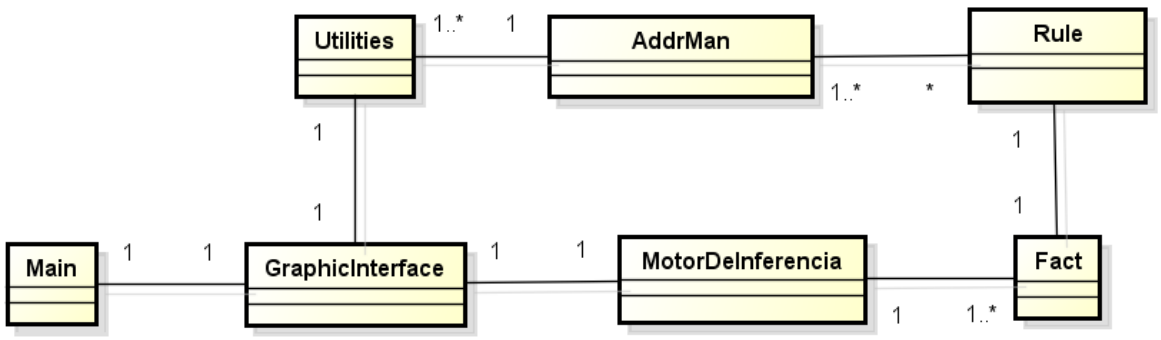
JPL	JAVA	PROLOG
Variable	String	variável
Integer	Long	inteiro de 32 bits
Float	Double	float de 64 bits

Fonte: http://jpl7.org/740/PrologApiOverview.jsp

A inicialização do mecanismo de inferência é realizado pela classe JPL, utilizando o método *setDefaultInitArgs* para alterar os valores padrão que são passados ao SWI-Prolog. Para realizar a inferência dos fatos no SWI-Prolog, é criado um objeto *Query*, passando como parâmetro o fato e o arquivo Prolog contendo as regras de inferência. O retorno da classe *Query* é um objeto *Map*, e deste modo pode conter um conjunto de resultados que podem ser acessados pelos métodos *hasMoreElements* e *nextElement*.

Um esboço do diagrama de classes atualizado para integração com o SWI-Prolog está definido no Apêndice B. A Figura 16 ilustra o modelo de domínio do FRChecker modificado.

Figura 16 - Modelo de domínio da versão modificada do FRChecker.



Fonte: Autoria própria (2017)

6.6 Considerações finais

Neste capítulo foi definida a proposta para solução do problema de inconsistência em regras de *firewall*. A lista de regras a serem analisadas, bem como os tipos de inconsistências entre regras serão obtidos através da pesquisa de Carbonera (2009).

Foram avaliadas implementações que integrem o FRChecker a um motor de inferência Prolog, e a implementação selecionada foi a SWI-Prolog, por possuir um desenvolvimento ativo e uma vasta biblioteca de funções nativas.

Por fim, um modelo de domínio e um diagrama de classes foram modelados de maneira a guiar o desenvolvimento da aplicação.

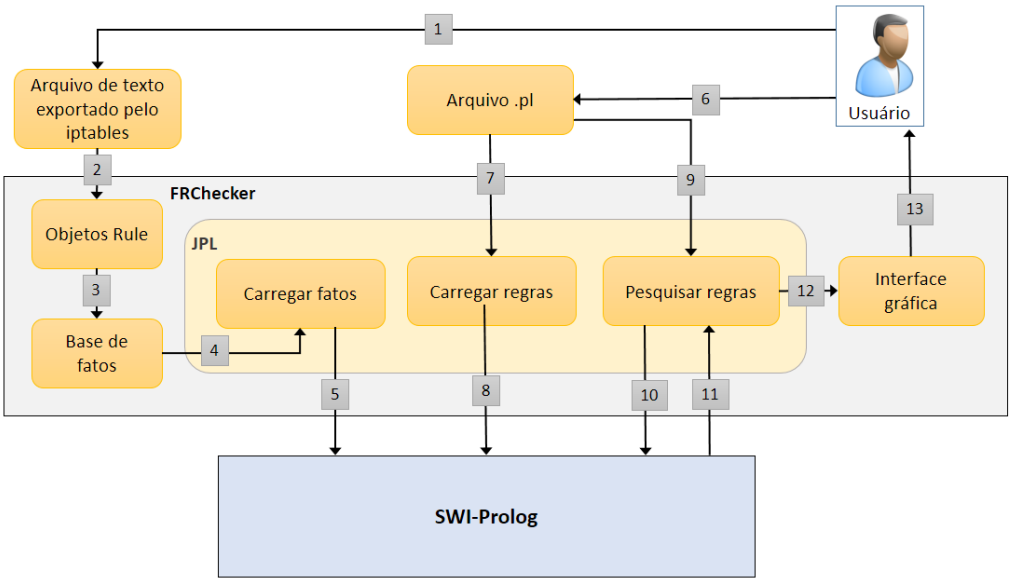
7 IMPLEMENTAÇÃO DO SISTEMA

De acordo com o estudo realizado e a solução proposta, um sistema integrando o sistema FRChecker com o motor de inferência SWI-Prolog foi implementado. Essa atividade foi dividida em etapas. Inicialmente os dados foram obtidos dos objetos Rule, da classe de liguagem Java do sistema FRChecker, e processados de modo a formarem a base de fatos. Esta base foi inserida no motor de inferência do SWI-Prolog em execução. Na etapa seguinte, as regras de inferência foram modeladas para atenderem aos critérios da solução. Estas então são consultadas pelo mecanismo de consulta, que também armazena as repostas obtidas. Por fim, as respostas são carregadas na interface para visualização do usuário. Nas seções deste capítulo, estas etapas são detalhadas.

O trabalho foi realizado em sistema operacional Windows 10 versão 1703, utilizando o motor de inferência SWI-Prolog versão 7.4.2 de arquitetura 64 *bits*, e a máquina virtual Java versão JDK 8u144, também de 64 *bits*. Como ambiente de desenvolvimento foi utilizado o *software* Netbeans IDE 8.2. Os arquivos da implementação original do FRChecker foram importados a um novo projeto no Netbeans. O arquivo jpl.jar, encontrando na pasta de instalação do SWI-Prolog foi adicionado como biblioteca a este projeto. Para compilação do projeto foi necessário adicionar os caminhos “C:\Program Files\swipl\bin“ e “C:\Program Files\swipl\lib\jpl.jar” a variável de ambiente de sistema “Path”.

O diagrama da Figura 17 detalha o fluxo de dados utilizado pela aplicação. O diagrama de classes atualizado (Apêndice C) ilustra as mudanças realizadas na implementação.

Figura 17 - Fluxograma de dados do programa implementado.



Fonte: Autoria própria (2017)

7.1 Obtenção da Base de Dados

Conforme descrito na seção 6.2, os dados são obtidos através de arquivo de texto exportado direto do *firewall* pelo iptables. O arquivo contém as *chains* cadastradas, junto com suas políticas padrão de permissão. Abaixo de cada *chain*, as regras correspondentes são listadas, com as seguintes informações: *target*, protocolo de rede, endereço ou máscara IP de origem e de destino. Opcionalmente pode receber informações adicionais como especificação do número das portas de rede, ou um intervalo em específico de endereços IP.

Estas informações são carregadas linha a linha pelo FRCheker, e são armazenadas em objetos Rule, sendo cada objeto incluído em uma lista do tipo Vector.

O objeto Rule armazena as informações na forma de s*trings*, tal qual obtido pelo arquivo texto. Portando, é necessário o pré-processamento dos endereços IP e de outros dados antes de passá-los para o JPL.

7.1.1 Preprocessamento das informações

Os endereços IP tanto de origem como de destino são armazenados no objeto Rule como *strings*, de três maneiras distintas: endereço único, intervalo de endereços e notação CIDR.

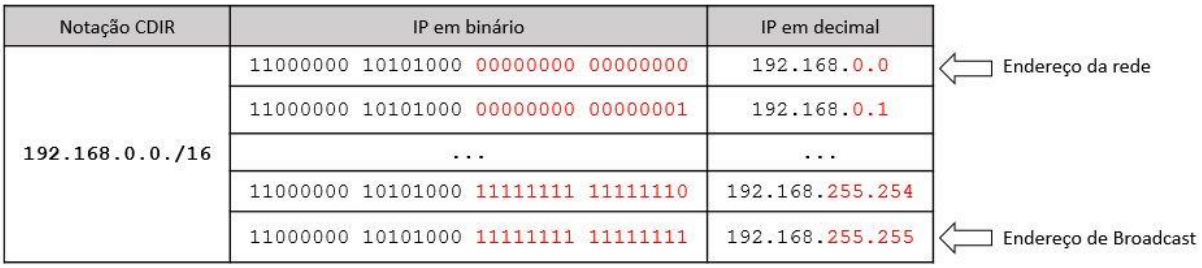
O endereço único é identificado por quatro conjuntos de numerais separados pelo caractere ponto. Para fins de normatização dos fatos na sintaxe do Prolog, endereços únicos serão tratados como conjuntos de IP cujo endereço inicial é igual ao final.

Um intervalo de endereços é identificado pelo caractere hífen (-) separando dois endereços únicos distintos, sendo o endereço a esquerda o inicio, e o endereço a direita o final do intervalo.

Um endereço com notação CIDR está identificado pelo caractere barra (/), separando o endereço IP do número que indica o tamanho da rede em questão. Este número representa a quantidade de bits que compõem o endereço IP que não sofrerão modificação, sendo que os bits restantes vão compor o intervalo de endereços IP da subrede, podendo receber valores 0 ou 1. O primeiro valor do intervalo identifica o endereço da rede, e o último valor o endereço de *broadcast*, conforme ilustrado na Figura 18. Para conversão de endereços IP com esta notação, foi utilizada a biblioteca CIDRUtils[14], de licença MIT.

[14] https://github.com/edazdarevic/CIDRUtils/

Figura 18 - Exemplo de notação CIDR.



Fonte: Autoria própria (2017).

Todos os endereços IP foram convertidos de s*trings* para decimais do tipo *long*, utilizando o método ipStringToLong, para deste modo facilitar a comparação entre pares de regras.

Com relação ao conjunto de portas de origem e destino, por ser um campo com quantidade variável de dados, intervalos de números ou ainda contendo a designação "*any*" para representar qualquer porta, os dados foram mantidos dentro de uma lista em sintaxe Prolog, delimitada por colchetes. Para fácil detecção entre números e intervalos de números, os intervalos foram alocados em um predicado *range/2* contendo o intervalo inicial e final.

As demais informações, tais como protocolo e *target* continuam identificadas como s*trings*, dada a característica de tipagem dinâmica da linguagem Prolog. Porém, os dados são convertidos de maneira a não possuírem caracteres especiais e também iniciarem em letras minúsculas para obedecerem a semântica dos átomos em Prolog

7.2 Geração da Base de Fatos

Com os dados tratados para se adequarem a semântica da linguagem Prolog e facilitarem as operações de comparação, estes são organizados em uma *String* que engloba todos os dados, na forma de um fato em Prolog. A implementação seguiu a semântica definida na Figura 19. Na Figura 20 são mostrados exemplos de elementos da fase de fatos implementada.

Figura 19 - Formato do fato na sintaxe Prolog que representa uma regra de *firewall*.

```
regra(ordem, target, protocolo, IP origem inicial, IP origem
final, [portas origem], IP destino inicial, IP destino final,
[portas destino], política padrão).
```

Fonte: Autoria própria (2017).

Figura 20 - Exemplos de base de fatos

```
regra(1, drop, tcp, 3232256257, 3232256510, [any], 3232235771, 3232235771, [any],
input).
regra(2, accept, all, 1, 4294967294, [any], 1, 4294967294, [80,8080], input).
regra(3, drop, all, 3232235521, 3232301054, [any], 3232235771, 3232235771,
[range(1,1024),range(64000,65000)], forward).
```

Fonte: Autoria própria (2017).

Após a formalização dos fatos, estes são inseridos em um objeto FactsBase de modo a serem a gerenciados como uma lista. A lista é passada como parâmetro ao método assertFacts. Este método é que alimenta o SWI-Prolog com os fatos representando regras de *firewall*, através do objeto Query, utilizando a função assertz/1, nativa do SWI-Prolog. A partir da execução da primeira Query, o motor de inferência é automaticamente inicializado.

Não foi feito uso das classes Term e Compound, descritas na seção 6.5, tendo em vista que as mesmas seriam instanciadas apenas para inicializarem os objetos Query, não havendo necessidade de mantê-las em memória. Como o Prolog utiliza tipagem dinâmica, os parâmetros foram passados na forma de s*trings.*

7.3 Implementação das regras de inferência

As regras de inferência caracterizam a lógica da implementação do sistema especialista. É por onde os fatos gerados são avaliados em busca dos resultados requisitados. É nesta etapa que entra a figura do especialista, que tendo o conhecimento do problema, modela logicamente a solução de acordo com as informações e a sintaxe da base de fatos.

A implementação foi realizada na linguagem Prolog, onde são realizadas operações de comparação de dados a fim de avaliar sua veracidade ou contradição. As principais operações de comparação utilizadas são listadas na Tabela 8.

Tabela 8 - Operações lógicas na sintaxe Prolog

Operação	Sintaxe Prolog
Operador “e”	,
Operador “ou”	;
Comparação de igualdade	==
Comparação de diferença	\==
Comparação “maior ou igual a”	>=
Comparação “menor ou igual a”	=<

Fonte: Baseado em http://www.swi-prolog.org/pldoc/man?section=compare (2017).

A cabeça das regras foi modelada a receber e retornar dois argumentos, os identificadores de cada par de regras avaliado que atendam aos critérios. Em seguida o par de fatos é listado de modo a identificar os argumentos que serão avaliados. A função segue com a avaliação de cada argumento relevante a solução do problema, e se todos forem verdadeiros, a regra retorna ao FRChecker o identificador das duas regras avaliadas, de modo a exibir a relação entre elas no sistema.

A Figura 21 exemplifica a sintaxe e lógica para verificar se em dado par de regras, os IPs da regra precedente são um subconjunto da segunda regra, e os protocolos de ambas são conflitantes. Os Apêndices D a F apresentam as regras implementadas.

Figura 21 - Exemplo de função modelada em Prolog.

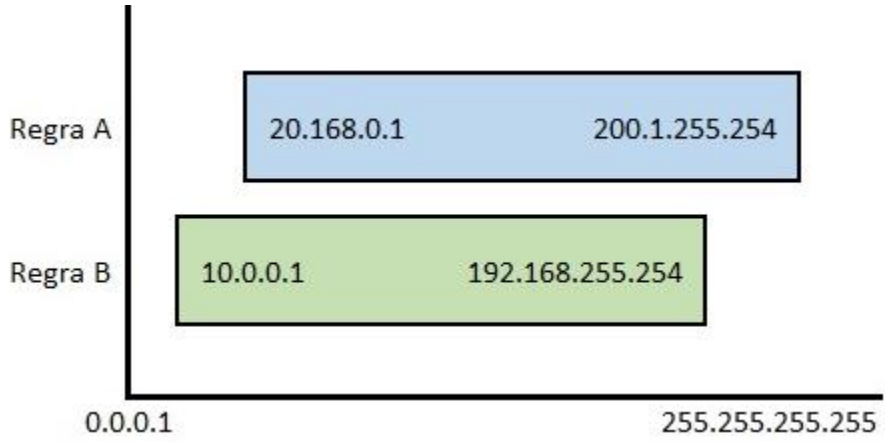
```
conflito(RegraA,RegraB) :-
regra(RegraA,ProtoA,IPInicialA,IPFinalA) ,
regra(RegraB,ProtoB,IPInicialB,IPFinalB) ,
RegraA < RegraB ,
ProtoA == ProtoB ,
subconjunto(IPInicialA,IPFinalA,IPInicialB,IPFinalB).
```

Fonte: Autoria própria (2017).

Verificou-se que ao usar a operação lógica de OU (;), o motor de inferência estava retornando resultados duplicados caso mais de uma condição fosse atendida. Para evitar isso utilizou-se a função once/1, nativa do SWI-Prolog, que evita que a operação posterior seja analisada caso a anterior seja verdadeira.

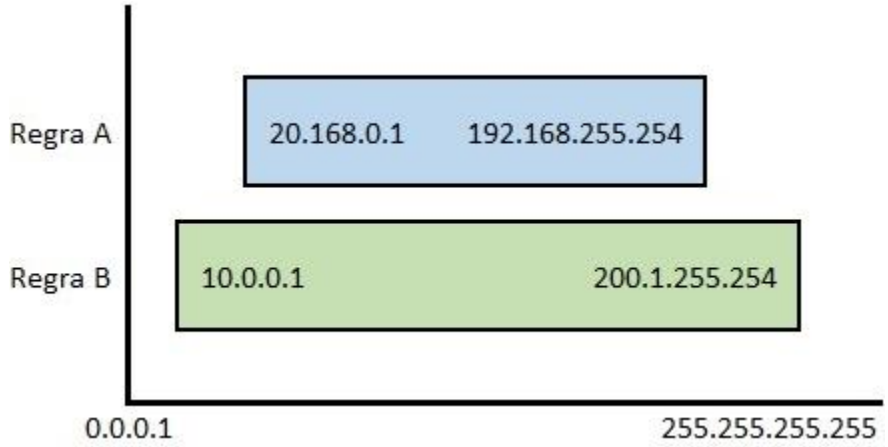
Para auxiliar a modelagem das funções com objetivo de identificar os quatro conflitos de regras descritos nas Seções 3.2.9 a 3.2.12, também foram modeladas as funções que identificam relações de subconjuntos, superconjuntos e intersecção entre pares de intervalos de endereços IP. Essas funções comparam o intervalo inicial e final do endereço IP entre uma regra e outra, e retornam verdadeiro caso os parâmetros atendam aos requisitos. Com os endereços IPs convertidos em decimais conforme descrito na Seção 7.1.1, simples operações de comparação identificam estas relações, representadas graficamente nas Figura 22, 23 e 24. As funções implementadas foram isintersection/4, issubset/4 e issuperset/4. Estas funções estão expostas no Apêndice H.

Figura 22 - Endereços IP das regras A e B formam uma intersecção.



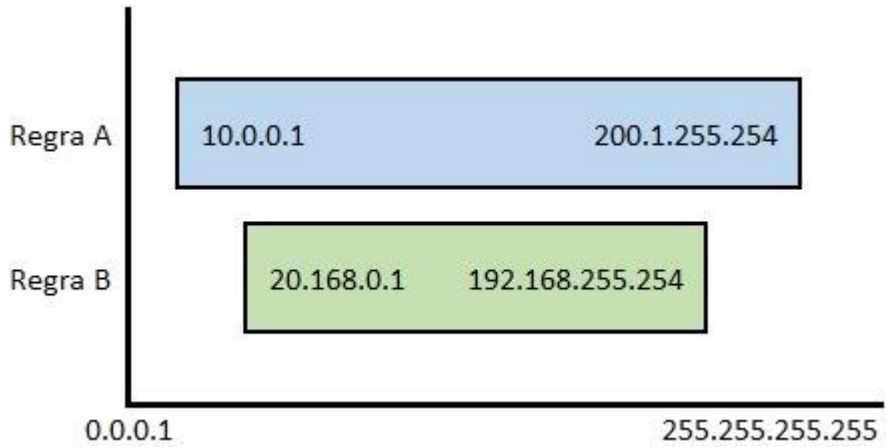
Fonte: Autoria própria (2017).

Figura 23 - Endereços IP da Regra A são um subconjunto da regra B.



Fonte: Autoria própria (2017).

Figura 24 - Endereços IP da Regra A são um superconjunto da regra B.



Fonte: Autoria própria (2017).

Para identificação de conflitos entre portas de rede, foi modelada a função portconflict/2, a qual recebe as duas listas de números de portas e realiza iteração recursiva entre seus elementos, comparando-os de acordo com os tipos de dados inseridos nas listas (Apêndice I).

O arquivo com as regras de inferência, geralmente caracterizado com a extensão .pl é carregado no sistema pela interface gráfica. Ao clicar no botão “Aferir por Inferência”, o usuário deve selecionar na estrutura de pastas o arquivo contendo as regras.

O caminho para o arquivo é passado para a função loadInferenceRules, que executa um comando consult/1 no SWI-Prolog, incluindo as regras implementadas para dentro da memória do motor de inferência.

7.4 Mecanismo de pesquisa

O mecanismo de pesquisa é responsável por invocar as funções desenvolvidas na etapa anterior. Através das funções, a memória do motor de inferência é consultada, inferindo as regras a base de fatos. As respostas com resultado positivo são retornadas pelas variáveis informadas ao efetuar a operação.

Para que o FRChecker tenha conhecimento das funções disponíveis, o especialista deve informá-las no arquivo .pl que contém as regras de inferência. São nas funções conflictFunction/2 que o especialista lista cada regra modelada. No primeiro argumento deve ser informada a descrição da regra; e no segundo argumento, a cabeça da regra e seus argumentos. Ambos os argumentos devem ser cadeias de caracteres (*strings*), delimitadas por aspas duplas, conforme ilustra a Figura 25.

Para análise das informações obtidas nas funções conflictFunction/2, um analisador sintático simples foi desenvolvido, para validar a formatação dos dados e identificar os argumentos a serem utilizados na consulta.

Figura 25 - Funções conflictFunction/2 utilizadas pelo mecanismo de pesquisa.

```
conflictFunction("Correlação","correlation(RuleA, RuleB)").
conflictFunction("Exceção","exception(RuleA, RuleB)").
conflictFunction("Redundância","redundancy(RuleA, RuleB)").
conflictFunction("Sombreamento","shadowing(RuleA, RuleB)").
```

Fonte: Autoria própria (2017).

A informação obtida pelo analisador sintático é passada como parâmetro para um objeto do tipo Query da biblioteca JPL, que realiza a consulta. Os resultados retornam por meio de um objeto do tipo Map, contendo n resultados da consulta, associados aos parâmetros das funções

consultadas. Este objeto é percorrido pelo método hasMoreSolution e os resultados são avaliados. As informações são inseridas em um objeto do tipo Result, cujo o qual é composto por:

- Uma *String* contendo o tipo de relação entre as regras de *firewall*, informado pelo especialista no primeiro parâmetro da função conflictFunction/2.
- Um objeto Rule representando a primeira regra da relação;
- E outro objeto Rule representando a segunda regra da relação.

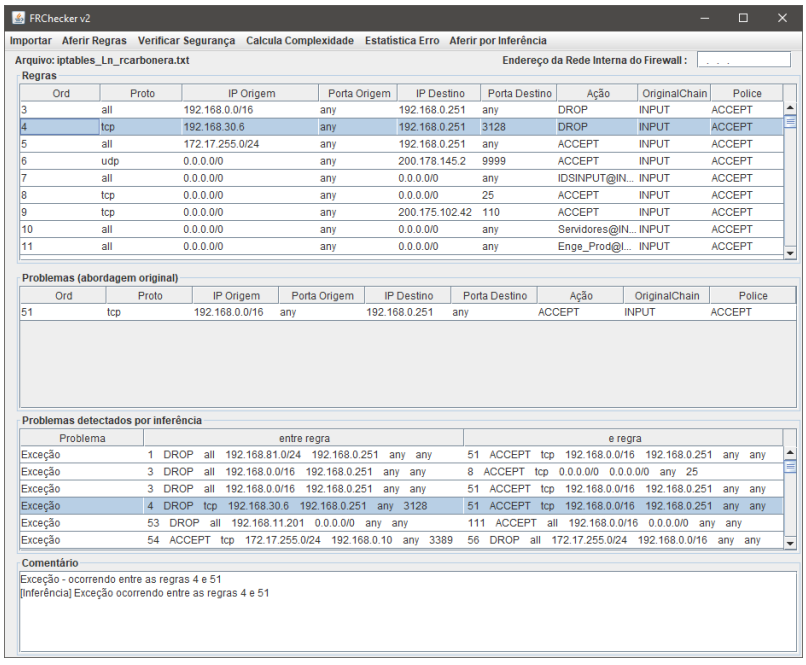
Os objetos Result são armazenados em lista ResultList, a qual estende a classe ArrayList, para posterior exibição dos resultados na interface gráfica.

7.5 Exibição dos resultados

Os resultados do motor de inferência vem ao usuário por meio da tabela “Problemas detectados por inferência”, exibida logo abaixo dos resultados realizados pela implementação procedural do FRChecker, e desta forma, facilitando análise dos resultados (Figura 26). A aferição dos problemas também é exposta no painel “Comentário”, de modo a facilitar a comparação entre as duas implementações.

Cada célula da tabela “Problemas detectados por inferência” traz uma síntese das regras conflitantes. Ao clicar na célula correspondente a regra, esta é selecionada na tabela “Regras”, e desta forma o usuário pode consultar integralmente as informações referentes a aquela regra.

Figura 26 - Interface do FRChecker em execução.

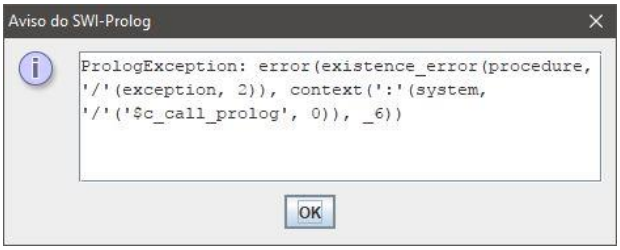


Fonte: Autoria própria (2017).

De modo a comportar a nova tabela na interface gráfica, o painel “Arquivo Importado” foi suprimido, pois suas informações já estavam representadas no painel “Regras”, tornando-o redundante. O painel “Comentário” com as observações do processamento de regras foi deslocado para o rodapé da janela. Foi adicionado um rótulo contendo o nome do arquivo carregado para identificação do mesmo.

As exceções geradas pelo motor de inferência pela classe PrologException do SWI-Prolog também são trazidas a interface do usuário, facilitando depurar problemas na base de fatos, ou no arquivo .pl importado ao sistema (Figura 27).

Figura 27 - Tela mostrando erros do motor de inferência.



Fonte: Autoria própria (2017).

7.6 Considerações

Neste capítulo foi documentada a implementação do motor de inferência na linguagem Prolog, bem como dos componentes que integram a solução. Foi detalhado o processo de transformação dos dados para serem inseridos na base da fatos. Foi realizada uma análise dos principais operadores utilizados pelo Prolog, especificado como a base de fatos é consultada, e como os resultados são obtidos e exibidos para o usuário.

O próximo capítulo detalha os testes realizados a fim de validar a implementação do sistema.

8 TESTES E AVALIAÇÃO DO SISTEMA

Após a implementação do sistema de verificação de regras de *firewall* por instrumento de um mecanismo de inferência, a aplicação foi testada para validação dos resultados.

Para fins de terminologia, a implementação orientada a objetos realizada por Carbonera (2009) será definida como “original”. A implementação descrita neste trabalho, implementada em linguagem funcional Prolog com uso da biblioteca JPL será definida como “por inferência”.

Os conjuntos de dados foram testados primeiro no software FRChecker original e depois na versão do software que utiliza o motor de inferência. A Tabela 9 detalha o caso de teste geral utilizado para testar o software.

Tabela 9 - Passos do teste de verificação de regras de *firewall*

Descrição	Verificação de inconsistências em regras de *firewall*.
Pré-condições	- Arquivo com a lista de regras de *firewall* obtido com o comando iptables –L –n. - Arquivo com as regras de inferência em linguagem Prolog.
Passos	1. Selecionar o arquivo de regras de *firewall* na opção “Importar”. 2. Clicar em “Aferir Regras” para executar a verificação de regras pela implementação original. 3. Clicar em “Aferir por Inferência” para selecionar o arquivo Prolog. 4. Clicar em cada regra na lista para verificar os possíveis conflitos entre regras de *firewall*.
Resultados esperados	A listagem de inconsistências encontradas ao analisar o conjunto de regras.
Comentários	A interface gráfica vai habilitando as opções conforme as opções pre-requisitadas são executadas. As analises são separadas em dois painéis distintos na parte inferior da janela, possibilitando comparação dos resultados.

Fonte: Autoria própria (2017).

8.1 Conjuntos de dados de teste

Para avaliar a implementação foram utilizados cinco conjuntos de regras de *firewall* distintos.

A amostra “A” refere-se ao arquivo exportado do iptables por Carbonera (2009), para validação da implementação original do FRChecker. Contém 354 regras de *firewall*, divididas em 38 *chains*.

Os conjuntos restantes (B, C, D e E) foram obtidos através do sítio “Public collection of firewall dumps”[15], o qual armazena arquivos relacionados a ferramenta iptables, tais como *scripts* de configuração, arquivos de estatísticas e listas de regras.

A amostra “B” é composta por 40 regras obtidas no diretório “config_networking.ringofsaturn.com “, arquivo “iptables_Ln”.

A amostra “C” foi obtida no diretório “configs_synology_diskstation_ds414” arquivo “iptables_Ln_jun_2015_legacyifacerules”, contendo 30 regras de *firewall*.

Já as amostras “D” e “E” são carregadas ao sistema pelos arquivos “iptables_Ln_29.11.2013” e “iptables-Ln-2016-06-27_16-29-01”, contendo respectivamente 2803 e 5528 regras de *firewall*. Ambas são obtidas no diretório **“**configs_chair_for-_Network_Architectures_and_Services”, disponível no sítio supracitado.

8.2 Avaliação dos resultados

Para avaliar a implementação, a cada execução do sistema foi carregada uma amostra de regras, conforme o passo 1 da Tabela 9. Após o processamento das amostras pela implementação original, foi carregado no sistema o arquivo .pl contendo as regras de inferência. Atendendo ao escopo definido por Carbonera (2009), as regras de inferência foram modeladas de modo a validarem apenas as regras de *firewall* que pertencentes a mesma *chain,* e com *target* igual a ACCEPT, DROP ou REJECT.

Na amostra “A”, com relação a avaliação de redundância entre regras de *firewall*, a implementação original do FRChecker encontrou ocorrência de redundância entre 89 pares de regras. O motor de inferência integrado ao software identificou 85 pares de regras redundantes. A implementação original informou redundância a mais entre as regras 57 e 61, 58 e 61, 59 e 61, 60 e 61. Em análise humana dos dados, verificou-se que não é um caso de redundância. Depurando o código, verificou-se que a função de verificação de conflitos entre IPs da implementação original identifica erroneamente os endereços IP de origem das regras 57 a 60 como idênticos aos da regra 61, quando na verdade há uma relação de superconjunto entre estas regras. Uma relação de superconjunto ocorre quando o intervalo de endereços IP da segunda regra está inteiramente contido no intervalo de endereços IP da primeira regra.

O FRChecker executando a verificação da implementação original identificou 39 eventos de sombreamento entre regras da amostra “A”, enquanto que a implementação por inferência detectou 38 sombreamentos. Verificou-se como diferença a detecção de

[15] https://github.com/diekmann/net-network

sombreamento entre os pares de regras 61 e 111. Depurando a aplicação, verificou-se que os IPs de origem estão erroneamente sendo identificados como idênticos, embora haja ali uma relação de subconjunto entre as regras analisadas. Ou seja o intervalo de endereços IP da primeira regra está integralmente contido no intervalo de endereços da segunda regra.

Ainda com relação a amostra “A”, tanto a verificação de exceção, como de correlação entre regras trouxeram os mesmos resultados na implementação original como na utilizando o motor de inferência em Prolog. Foram detectados os mesmos 74 eventos de exceção e os mesmos 2 eventos de correlação.

As amostras “B”, “C”, “D” e “E”, contendo respectivamente 40, 30, 2803 e 5528 regras distintas de *firewall*, ao serem aferidas pelo FRChecker receberam o mesmo resultado nas duas abordagens de verificação. Ou seja, tanto a implementação original como a implementação por inferência retornaram o mesmo número de ocorrências com relação a redundância, sombreamento, exceção e correlação entre as regras, com os mesmos pares de regras associados a estes eventos.

A Tabela 10 sumariza os resultados obtidos pelas duas implementação para cada uma das amostras utilizadas.

Tabela 10 - Comparação dos testes com as duas implementações.

Implementação	**Original**					**Por inferência**				
Amostra	**A**	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
Redundância	89	40	43	235	664	85	40	43	235	664
Exceção	74	45	48	14	15	74	45	48	14	15
Sombreamento	39	39	48	3	3	38	39	48	3	3
Correlação	2	0	0	0	84	2	0	0	0	84
Total	204	124	139	252	766	199	124	139	252	766

Fonte: Autoria própria (2017).

8.3 Avaliação de desempenho computacional

Tendo em vista o crescente número de serviços providos via rede e que devem ser filtrados por *firewalls* para garantir a segurança e integridade das informações, o número de regras inclusas no *firewall* de grandes instituições tende a crescer continuamente. Portanto, mostrou-se relevante avaliar com que velocidade as implementações do FRChecker processam todas as regras em busca de inconsistências.

Como medida de desempenho, determinou-se o tempo médio em segundos que as

verificações, tanto na implementação original como a realizada por inferência demoram para efetuar a etapa de aferição das regras. A medida é feita por meio do método System.currentTimeMillis() da linguagem Java. O tempo inicial é medido após as regras de *firewall* serem inseridas na memória do programa, e o tempo final é medido antes dos resultados serem apresentados à interface gráfica. Deste modo, o saldo do tempo final e inicial é delimitado apenas a etapa de comparação entre as regras. A avaliação de desempenho foi efetuada em um microcomputador, com processador operando a 3200MHz e memória RAM de 4 GB. Os resultados após cinco execuções do sistema para cada amostra estão expostos na Tabela 11.

Tabela 11: Tempos médios de cada implementação

Amostras	**Quantidade total de regras**	**Tempo médio (original) [1]**	**Tempo médio (por inferência) [1]**
Amostra “A”	354	0,48s	0,38s
Amostra “B”	40	0,34s	0,06s
Amostra “C”	30	0,18s	0,05s
Amostra “D”	2803	44,41s	20,97s
Amostra “E”	5528	239,30s	76,45s

(1) Tempo médio em 10 execuções do sistema, expresso em segundos.

Fonte: Autoria própria (2017).

Verificou-se desempenho mais rápido na implementação por motor de inferência, com velocidades médias 211% e 313% superiores a implementação original nas amostras “D” e “E” respectivamente. Analisando as implementações, ambas possuem complexidade computacional aproximada de $O(n^2/2)$, devido ao fato de testarem as inconsistências de *firewall* avaliando todos os pares de regras possíveis e não repetindo os pares já avaliados. Tal diferença de desempenho se deve principalmente ao processamento dos dados durante a aferição das regras de *firewall*. A implementação via inferência armazena os dados pré-processados, facilitando a comparação entre variáveis, enquanto que a implementação original armazena os dados brutos obtidos pela importação do arquivo de regras de *firewall*, e portanto precisa tratar as informações a cada nova verificação de regras efetuada.

8.4 Considerações

Este Capítulo apresentou como os testes para validação do estudo foram realizados. Comparou-se o resultado de aferição de regras de *firewall* entre a implementação original proposta por Carbonera (2009) com a implementação descrita neste trabalho, utilizando o motor de inferência da linguagem SWI-Prolog.

Do total de 8755 regras analisadas, a implementação original detectou 1485 inconsistências, e a implementação por inferência detectou 1480. Verificou-se que as cinco inconsistências a mais se devem a falsos-positivos detectados pela implementação original. Desta forma, a implementação por inferência mostrou uma qualidade de detecção superior ao FRChecker original.

Com relação ao desempenho computacional, a implementação por inferência processou as regras de maneira mais rápida que a implementação orientada a objetos. No entanto, isso se deve principalmente ao processamento não otimizado dos dados pela implementação original. Para uma avaliação mais apurada, seria adequado reimplementar parte do mecanismo original, especialmente quanto ao pré-processamento dos dados e dos métodos que detectam conflitos entre endereços IP e portas de rede.

9 CONCLUSÃO

Neste trabalho foi estudado o funcionamento de mecanismos de *firewall*, em especial a solução Netfilter para filtragem de pacotes de rede. O sistema FRChecker foi analisado, junto com outros sistemas comerciais de verificação de regras de *firewall*, a fim de elencar diferentes métodos de aferição. A estrutura de Sistemas Baseados em Conhecimento foi pesquisada para integrá-la a um sistema de verificação de regras de *firewall*.

Com base nos estudos realizados, foi proposta uma arquitetura integrando o sistema FRChecker a um motor de inferência para verificação de regras, utilizando as principais metodologias de verificação de conflitos implementadas pelo FRChecker. O motor de inferência SWI-Prolog foi integrado junto ao FRChecker, e as regras de inferência modeladas na linguagem Prolog. O sistema implementado neste trabalho apresentou melhor taxa de detecção de conflitos entre regras de *firewall* e maior velocidade de processamento, em comparação com o sistema orginal.

Firewalls representam um elemento primordial para garantir acesso seguro a sistemas e recursos de rede, em âmbito local ou de Internet. A medida que estes recursos aumentam e se tornam essenciais aos negócios, também cresce em complexidade a gestão das regras que os garantem acesso, ocasionando em possíveis falhas de segurança ou atrasos de comunicação. Em face disto, o sistema implementado facilita a manutenção do crescente e complexo conjunto de regras, sendo uma ferramenta de auxílio ao administrador de rede ao determinar incosistências nas políticas de segurança e no acesso a recursos.

A integração do sistema FRChecker a um motor de inferência não apenas mostrou-se eficiente, como trouxe uma nova abordagem de customização. Com a inclusão do SBC, um especialista pode modelar novas regras de verificação dos conjuntos de regras de *firewall* em um arquivo na linguagem Prolog, sem a necessidade de alterar os códigos fonte da aplicação em Java, e desde que obedeça a sêmantica definida para a base de fatos.

Infelizmente não foram encontrados outros métodos de avaliação envolvendo inter-relação entre conjuntos de regras além dos definidos neste trabalho. Uma pesquisa mais ampla poderá trazer novas abordagens a serem modeladas como regras de inferência. Através da arquitetura criada, métodos de aferição que levem em conta regras de *firewall* individuais são triviais de serem implementados, necessitando no entanto que sejam realizadas pesquisas para identificar melhores práticas em políticas de segurança. Com relação a métodos envolvendo estatísticas do tráfego do *firewall* que é registrado pelo Netfilter, o sistema necessitaria de diversos ajustes, especialmente com relação a interpretação do arquivo gerado pelo iptables, e

também pela implementação de estruturas a armazenar e tratar tais informações.

Por fim, o sistema se beneficiaria do suporte ao protocolo IPv6, em ampla implantação atualmente. Para tal, tanto o sistema desenvolvido em Java como os fatos e regras em Prolog devem ser modelados de forma a comportarem números inteiros de 128 *bits*, ou através de outras estruturas de dados que possibilitem a comparação entre endereços de rede IP.

REFERÊNCIAS BIBLIOGRÁFICAS

ALDER, R. et al. **How to cheat at configuring open source security tools**. Burlington: Syngress Publishing, 2007. ISBN-10 1-59749-170-5

AL-SHAER, E.; HAMED, H.; MASUM, R. B.; **Conflict classification and analysis of distributed firewall policies**. IEEE Journal on Selected Areas in Communications, v. 23, n. 10, p. 2069-2084, out. 2005.

AL-SHAER, E.; HAMED, H.; **Taxonomy of conflicts in network security policies.** IEEE Communications Magazine, v. 44, n. 3, p. 134-141, mar. 2006.

ANDERSON, James P. et al. **Firewalls: an expert roundtable**. IEEE Software Magazine, v. 14, n. 5, p. 60-66, set./out. 1997.

CAPRETTA V. et al. **Formal correctness of conflict detection for firewalls**, ACM Workshop on Formal Methods in Security Engineering, p. 22-30, nov. 2007.

CARBONERA, Rogério. **Ferramenta de interpretação de regras de configuração de *firewalls***. 2009. Trabalho de Conclusão de Curso - Universidade de Caxias do Sul, Bacharelado em Ciência da Computação. Caxias do Sul, 2009.

GENARO, Sergio. **Sistemas especialistas:** o conhecimento artificial. Rio de Janeiro: LTC – Livros Técnicos e Científicos S.A., 1986. ISBN 85-216-0501-3.

GIARRATANO, Joseph; RILEY, Gary. **Expert systems:** principles and programing. Boston: Course Technology, 2005. ISBN 0-534-38447-1.

HUNT, Ray. **Internet/Intranet firewall security** – policy, architecture and transaction services. Computer Communication, n. 21, p. 1107-1123, abr. 1998.

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a internet**. São Paulo: Pearson Addison Wesley, 2006. ISBN 85-88639-18-1.

LUGER, George F.. **Inteligência artificial:** estruturas e estratégias para a resolução de problemas complexos. 4. ed. Porto Alegre: Bookman, 2004. ISBN 85-36303-96-4.

MARMORSTEIN, Robert; KEARNS, Phil. **A tool for automated iptables firewall analysis.** Usenix Annual Technical Conference, p. 71-81, 2005.

MOORE, John. **5 firewall tests and supporting tool – IT Security**. 2007. Disponível em: <https://web.archive.org/web/20161125134752/http://www.itsecurity.com/features/5-firewall-tests-091107>. Acesso em: 21 ago. 2017.

PETERSEN, Richard. **Linux:** the complete reference. 6. ed. New York: McGraw-Hill, 2008. ISBN 0-07-159664-X.

PURDY, Gregor N. **Linux iptables**. 1. ed. Sebastopol: O'Reilly Media Inc., 2004. ISBN 0-596-00569-5

REDE NACIONAL DE ENSINO E PESQUISA. **Estatísticas**. Disponível em: <http://www.rnp.br/cais/estatisticas/index.php> Acesso em: 28 ago. 2017.

REZENDE, Solange O. **Sistemas inteligentes:** fundamentos e aplicações. Barueri: Editora Manole, 2005. ISBN 85-204-1683-7.

SREELAJA, N.K.; VIJAYALAKSHMI, G.A. **Ant colony optimization based approach for efficient packet filtering in firewall.** Applied Soft Computing, p. 1222–1236, 2010.

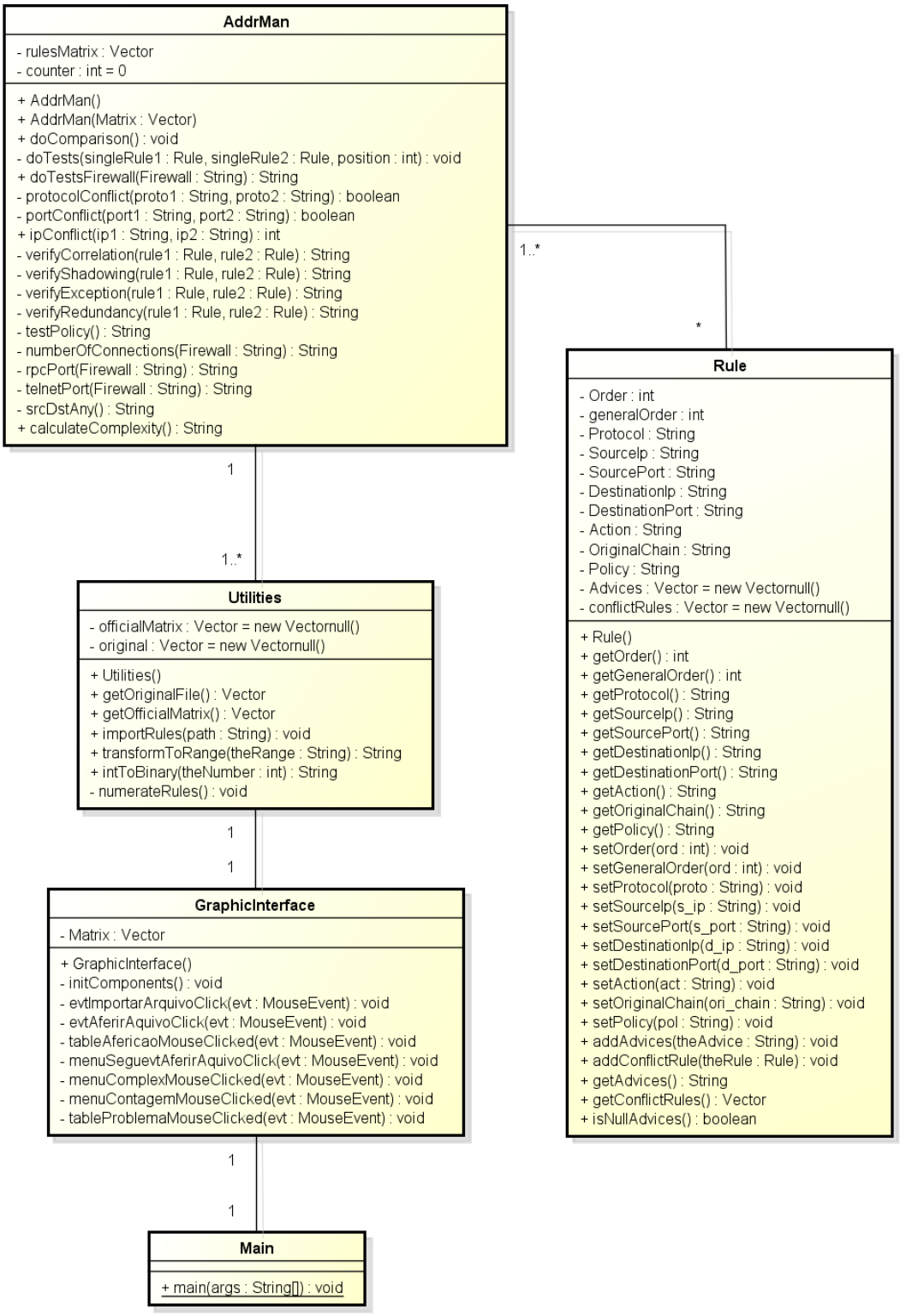
STALLINGS, W. **Criptografia e segurança de redes**. São Paulo: Pearson Prentice Hall, 2007. ISBN 978-85-7605-119-0.

TANENBAUM, Andrew S. **Redes de computadores**. 4. ed. Editora Campus (Elsevier), 2003. ISBN 8535211856.

WOOL A. **A quantitative study of firewall configuration errors**. IEEE Computer Magazine, v. 37, n. 6, p. 62-67, jun. 2004.

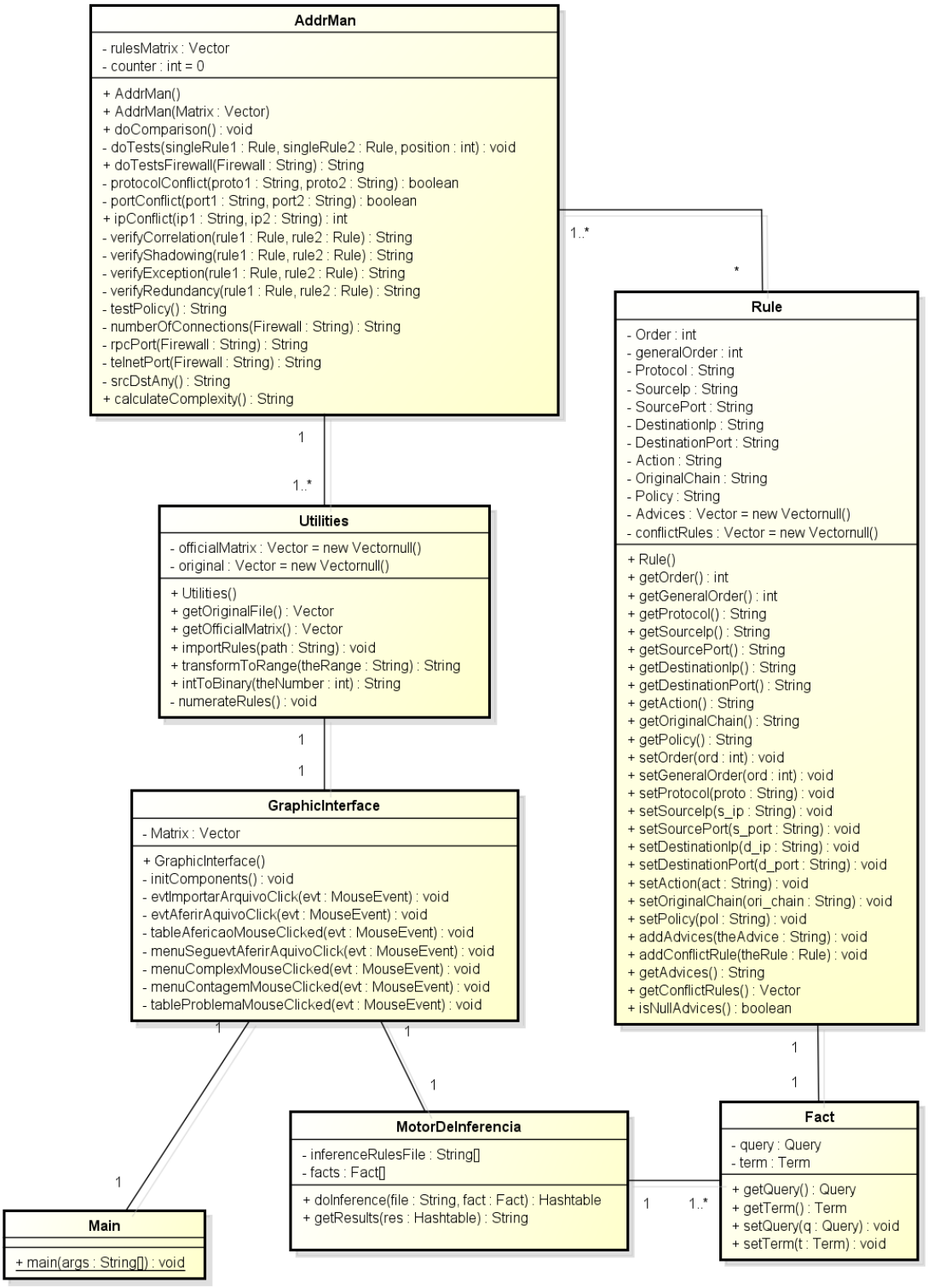
WOOL A. **Trends in firewall configuration errors:** measuring the holes in swiss cheese. IEEE Internet Computing, v. 14, n. 4, p. 58-65, jul/ago, 2010.

APÊNDICE A: DIAGRAMA DE CLASSES DO FRCHECKER.



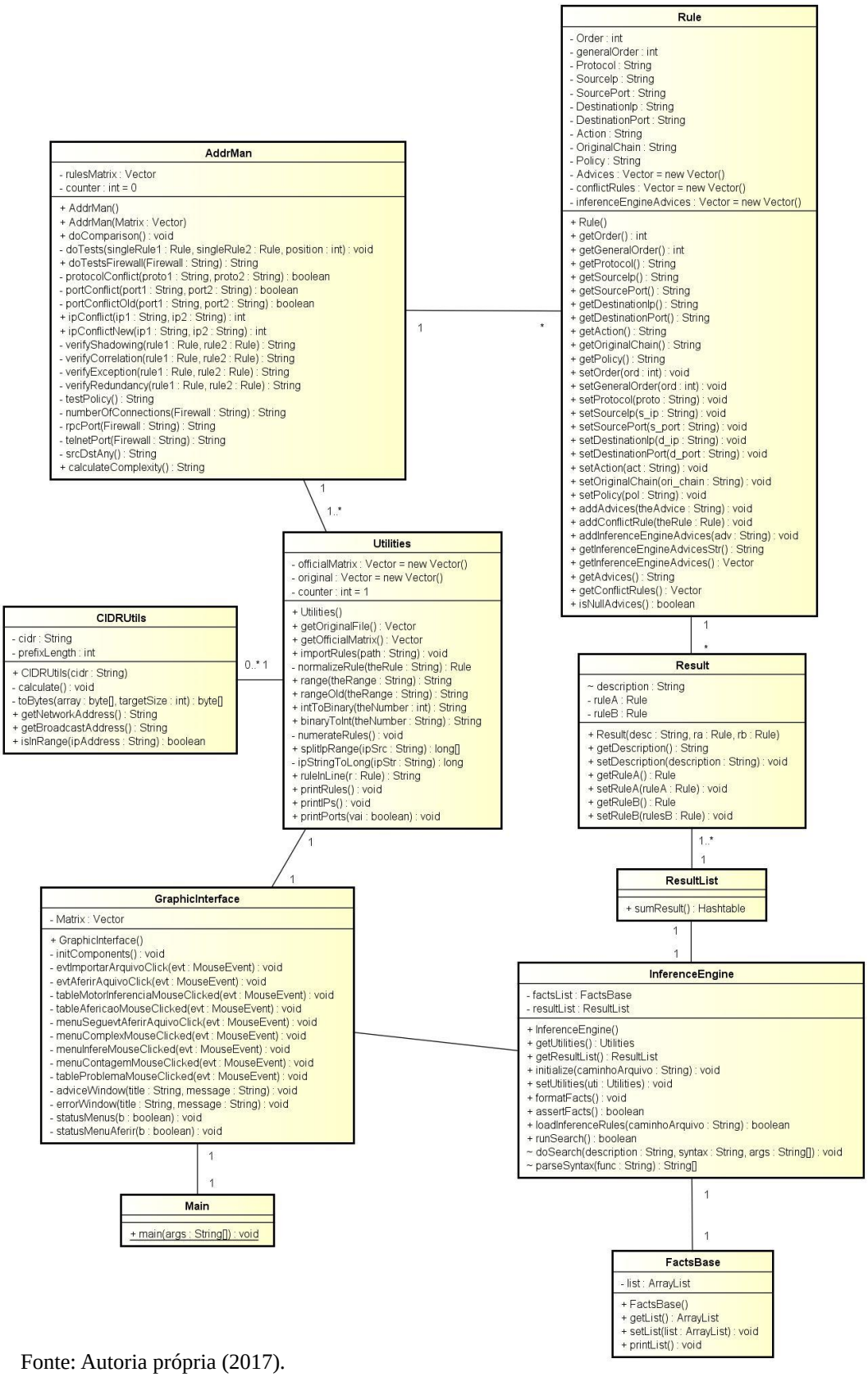
Fonte: CARBONERA (2009).

APÊNDICE B: PROPOSTA DE DIAGRAMA DE CLASSES DO FRCHECKER ATUALIZADO.



Fonte: Baseado em CARBONERA (2009).

APÊNDICE C: DIAGRAMA DE CLASSES COM MOTOR DE INFERÊNCIA INTEGRADO.



Fonte: Autoria própria (2017).

APÊNDICE D: REGRA EM PROLOG PARA VERIFICAÇÃO DE REDUNDÂNCIA

```
% verifica redundância entre regras A e B.
redundancy(RuleA, RuleB) :-
     regra(RuleA, ActionA, ProtoA, IpSrcIA, IpSrcFA, PortSrcA,
           IpDestIA, IpDestFA, PortDestA, ChainA),
     regra(RuleB, ActionB, ProtoB, IpSrcIB, IpSrcFB, PortSrcB,
           IpDestIB, IpDestFB, PortDestB, ChainB),
     RuleA < RuleB,
     ActionA == ActionB,
     (ActionA == accept ; ActionA == drop ; ActionA == reject),
     once( ProtoA == ProtoB ; ProtoA == all ; ProtoB == all ) ,
     issubset(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB),
     issubset(IpDestIA,IpDestFA,IpDestIB,IpDestFB),
     once(portconflict(PortDestA,PortDestB)),
     ChainA == ChainB.
```

Fonte: Autoria própria (2017).

APÊNDICE E: REGRA EM PROLOG PARA VERIFICAÇÃO DE SOMBREAMENTO

```
% verifica sombreamento entre regras A e B.
shadowing(RuleA, RuleB) :-
     regra(RuleA, ActionA, ProtoA, IpSrcIA, IpSrcFA, PortSrcA,
           IpDestIA, IpDestFA, PortDestA, ChainA),
     regra(RuleB, ActionB, ProtoB, IpSrcIB, IpSrcFB, PortSrcB,
           IpDestIB, IpDestFB, PortDestB, ChainB),
     RuleA < RuleB,
     ActionA \== ActionB,
     (ActionA == accept ; ActionA == drop ; ActionA == reject),
     (ActionB == accept ; ActionB == drop ; ActionB == reject),
     once( ProtoA == ProtoB ; ProtoA == all ; ProtoB == all ),
     issuperset(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB),
     issuperset(IpDestIA,IpDestFA,IpDestIB,IpDestFB),
     once(portconflict(PortDestA,PortDestB)),
     ChainA == ChainB.
```

Fonte: Autoria própria (2017).

APÊNDICE F: REGRA EM PROLOG PARA VERIFICAÇÃO DE EXCEÇÃO

```
% verifica exceção entre regras A e B.
exception(RuleA, RuleB) :-
     regra(RuleA, ActionA, ProtoA, IpSrcIA, IpSrcFA, PortSrcA,
           IpDestIA, IpDestFA, PortDestA, ChainA),
     regra(RuleB, ActionB, ProtoB, IpSrcIB, IpSrcFB, PortSrcB,
           IpDestIB, IpDestFB, PortDestB, ChainB),
     RuleA < RuleB,
     ActionA \== ActionB,
     (ActionA == accept ; ActionA == drop ; ActionA == reject),
     (ActionB == accept ; ActionB == drop ; ActionB == reject),
     once(ProtoA == ProtoB ; ProtoA == all ; ProtoB == all),
     issubset(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB),
     issubset(IpDestIA,IpDestFA,IpDestIB,IpDestFB),
     once(portconflict(PortDestA,PortDestB)),
     ChainA == ChainB.
```

Fonte: Autoria própria (2017).

APÊNDICE G: REGRA EM PROLOG PARA VERIFICAÇÃO DE CORRELAÇÃO

```
% verifica correlação entre regras A e B.
correlation(RuleA, RuleB) :-
     regra(RuleA, ActionA, ProtoA, IpSrcIA, IpSrcFA, PortSrcA,
           IpDestIA, IpDestFA, PortDestA, ChainA),
     regra(RuleB, ActionB, ProtoB, IpSrcIB, IpSrcFB, PortSrcB,
           IpDestIB, IpDestFB, PortDestB, ChainB),
     RuleA < RuleB,
     ActionA \== ActionB,
     (ActionA == accept ; ActionA == drop ; ActionA == reject),
     (ActionB == accept ; ActionB == drop ; ActionB == reject),
     once( ProtoA == ProtoB ; ProtoA == all ; ProtoB == all ),
     once((isintersection(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB) ;
       isintersection(IpDestIA,IpDestFA,IpDestIB,IpDestFB)
          ) ;
       ( issubset(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB) ,
         issuperset(IpDestIA,IpDestFA,IpDestIB,IpDestFB),
         IpSrcIA =\= IpSrcIB , IpSrcFA =\= IpSrcFB ,
         IpDestIA =\= IpDestIB , IpDestFA =\= IpDestFB
       ) ;
       ( issuperset(IpSrcIA,IpSrcFA,IpSrcIB,IpSrcFB) ,
         issubset(IpDestIA,IpDestFA,IpDestIB,IpDestFB),
         IpSrcIA =\= IpSrcIB , IpSrcFA =\= IpSrcFB ,
         IpDestIA =\= IpDestIB , IpDestFA =\= IpDestFB
       )
     ),
     once(portconflict(PortDestA,PortDestB)),
     ChainA == ChainB.
```

Fonte: Autoria própria (2017).

APÊNDICE H: REGRAS EM PROLOG PARA TESTES DE RELAÇÃO ENTRE ENDEREÇOS IP

```
% testa se intervalos do IP A são um subconjunto de intervalos
% do IP B.
issubset(Ainicial,Afinal,Binicial,Bfinal) :-
     Ainicial >= Binicial,
     Afinal =< Bfinal.


% testa se intervalos do IP A são um superconjunto de
% intervalos do IP B.
issuperset(Ainicial,Afinal,Binicial,Bfinal) :-
     Ainicial =< Binicial,
     Afinal >= Bfinal.


% testa se intervalos do IP A formam uma intersecção com
% intervalos do IP B
isintersection(Ainicial,Afinal,Binicial,Bfinal) :-
    (Ainicial < Binicial, Afinal < Bfinal, Afinal > Binicial);
    (Ainicial > Binicial, Afinal > Bfinal, Ainicial < Bfinal).
```

Fonte: Autoria própria (2017).

APÊNDICE I: REGRAS EM PROLOG PARA COMPARAÇÃO ENTRE PORTAS DE REDE

```
% compara lista de portas A com lista de portas B.
portconflict([Ha|Ta],[Hb|Tb]) :-
     Ha == any ; Hb == any;
     comparePorts(Ha,[Hb|Tb]);
     portconflict(Ta,[Hb|Tb]),!.


% compara um elemento da lista de portas com lista de portas.
comparePorts(A,[Hb|Tb]) :-
     A == Hb ;
     ( compareRangeRange(A,Hb); compareRangeRange(Hb,A) );
     compareNumRange(A,Hb);
     compareNumRange(Hb,A);
     comparePorts(A,Tb),!.


% compara dois intervalos de portas
compareRangeRange(range(Xa,Ya),range(Xb,Yb)) :-
     between(Xa,Ya,Xb) ; between(Xa,Ya,Yb),!.


% compara número de porta com intervalo de portas.
compareNumRange(A,range(Xb,Yb)) :-
     number(A),
     between(Xb,Yb,A),!.
```

Fonte: Autoria própria (2017).