JACSON RODRIGUES CORREIA DA SILVA

REDES NEURAIS ARTIFICIAIS PARA SISTEMAS DE DETECÇÃO DE INTRUSOS

BACHARELANDO EM CIÊNCIA DA COMPUTAÇÃO

FIC/CARATINGA

2007

JACSON RODRIGUES CORREIA DA SILVA

REDES NEURAIS ARTIFICIAIS PARA SISTEMAS DE DETECÇÃO DE INTRUSOS

Monografia apresentada à banca examinadora da Faculdade de Ciência da Computação das Faculdades Integradas de Caratinga, como exigência parcial para obtenção do grau de bacharel em ciência da computação, sob orientação do professor Msc. Paulo Eustáquio dos Santos

FIC/CARATINGA

2007

AGRADECIMENTOS

A Deus que sempre me acompanhou em todos os momentos de minha vida, que sempre me mostrou luz em caminhos escuros, que sempre me amparou e que sempre soube guiar minha vida.

A toda minha família, avó, pais, irmãs, cunhados e sobrinhas, que sempre souberam me dizer os palavras sábias, que sempre souberam me educar, que sempre demonstraram amor, carinho, companheirismo em tudo de minha vida. E especialmente ao meu pai, grande apoio e amigo que sempre esteve ao meu lado em todas as decisões difíceis que tive de fazer e em todos os momentos que precisei.

A minha noiva, que sempre soube me acompanhar em meus dias ruins, entender meus pensamentos, me acalmar, me fazer sorrir sem nem mesmo ter motivos para tal, que sempre me trouxe alegrias e que sempre demonstrou características únicas de alguém que ama e quer amar.

Aos meus primos e amigos que sempre aceitaram minha companhia, que sempre me fizeram lembrar que também devemos curtir a vida e vivê-la, antes que os momentos bons passem, que a distância chegue e que a gente se arrependa.

Aos amigos de minha sala, com quem sempre aprendi, com quem pude viver quatro anos seguidos sem enjoar de ninguém... rs, nunca sairão de minha mente, de minhas lembranças, meus pensamentos. Tenho a vocês como amigos para sempre.

Aos professores que sempre tiveram paciência em me ensinar a matéria difícil, que quiseram compartilhar além de conhecimento, amizade. Que sempre demonstraram confiança em minha pessoa e em meu aprender.

Aos meus colegas de trabalho, por agüentar meus momentos de cara chato e por sempre demonstrarem o quando a amizade é importante e o quanto amigos são.

“Amei a sabedoria e a busquei desde a minha juventude...”

Sabedoria 8:2a

RESUMO

As redes de computadores, espalhadas por todo o planeta, modificam a vida do ser humano, alterando sua forma de trabalhar, comunicar, compartilhar dados, etc. Porém com seu crescimento, as informações importantes passaram a trafegar e pessoas mal intencionadas passaram a procurar por falhas e vulnerabilidades para obter dinheiro, dados importantes, ou simplesmente brincar com as informações dos outros.

Métodos modernos de segurança existem, mas a falha sempre ocorre e sistemas denominados detectores de intrusos são muito utilizados. Porém o aprendizado de uma pessoa e o raciocínio lógico da mesma não incorpora um sistema desses. Áreas de inteligência artificial vem sendo estudadas e trabalhadas para melhorar o desempenho dessas ferramentas.

Este trabalho apresenta um estudo sobre a utilização das Redes Neurais Artificiais em Sistemas de Detecção de Intrusos. Sendo implementada uma ferramenta que permite a criação de redes neurais recorrentes em um simulador gráfico para seu uso em um software de detecção de intrusos.

Para a implementação foram escolhidas ferramentas com código fonte aberto, que permitem ser modificados e evoluídos constantemente por pessoas de todo o mundo, além de permitir uma melhor aprendizagem dos algoritmos utilizados em redes neurais artificiais e em sistemas de redes de computadores. Essas ferramentas foram o *JavaNNS* e o *Snort*. Sendo o primeiro um simulador de redes neurais mantido pela universidade de Tübingen, e o segundo um projeto mantido e atualizado constantemente pelo grupo SourceFire, que apresenta uma estrutura bem organizada em seu projeto.

Este trabalho também apresenta estudos sobre as redes neurais que devem ser utilizadas em gerência de redes de computadores e propondo o uso de um tipo de rede neural artificial para os sistemas de detecção de invasão.

ABSTRACT

Networks, scattered throughout the world, change the lives of human beings, changing their way of work, communicate, share data, etc.. But with its growth, important information came to traffic and badly intentioned people began to look for flaws and vulnerabilities wanting money, important data, or simply playing with the information of others.

Security moderns methods exist, but there is always existent fails, and systems called intrusion detection are being widely used. But the learning of a person in the same logical reasoning does not incorporate these systems. Areas of artificial intelligence is being studied and worked to improve the performance of these tools.

This paper presents a study on the use of Artificial Neural Networks in the Intrusos Detection Systems. Being implemented a tool that allows the creation of recurrent neural networks in a graphic simulator for its use in a software for detecting intruders.

For the implementation were chosen tools with open source code, enabling be modified and evolved constantly by people around the world, in addition to enabling a better learning of algorithms used in artificial neural networks and systems of computer networks. These tools were the JavaNNS and Snort. As the first is a simulated neural networks maintained by the University of Tübingen, and the second is a project maintained and updated constantly by the group SourceFire, which presents a project well-organized.

This paper also presents research on the neural networks to be used in management of computer networks and proposing the use of a type of artificial neural network systems for the detection of invasion.

Lista de Figuras

Lista de Tabelas

LISTA DE SIGLAS

.net	– Extensão dos arquivos de rede neural artificial do SNNS e JavaNNS
.pat	– Extensão dos arquivos de treinamento artificial do SNNS e JavaNNS
ANN	– Artificial neural network (Redes neurais artificiais)
Apt-get	– Gerenciador de programas de algumas distribuições Linux
Baseline	– Estado normal de uma rede de computadores
Firewall	– Dispositivo de segurança utilizado em redes de computadores
IDS	– Intrusion detection system (Sistemas de detecção de intrusos)
IRC	– Protocolo utilizado na internet para troca de mensagens (bate-papo)
Hacker	– Pessoas com habilidade de penetrar em sistemas de computadores
JavaNNS	– Simulador de redes neurais artificiais sucessor de SNNS
Kernel	– Núcleo responsável pelas principais funções de um sistema
Libpcap	– Biblioteca para captura de pacotes em sistemas Unix
Loop	– Repetição de rotinas conhecidas
Matlab	– Software que agrega funções prontas e simuladores
Patterns	– conjunto de entradas e saídas de uma rede neural artificial
Ping	– Programa responsável por enviar um pedido de resposta a determinada máquina em uma rede de computadores
Plugin	– Acessório que fornece um suporte adicional a determinado programa
RNA	– Redes neurais artificiais
SDI	– Sistemas de detecção de Intrusos
SNNS	– Simulador de redes neurais artificiais
Spam	– Mensagem de e-mail com fins publicitários
Trojan	– Vírus de computador com várias características complementares
Webmail	– Cliente de e-mails acessado por navegadores
Wget	– programa para obtenção de conteúdo na Internet
Winpcap	– Biblioteca para captura de pacotes em sistemas Windows

Sumário

1. INTRODUÇÃO

Desde o início dos tempos a comunicação foi uma preocupação e uma necessidade do ser humano. Caminhando por sinais de fumaça à redes de fibras óticas, consegue-se enxergar uma grande evolução na forma de trocar mensagens. Atualmente as informações conseguem trafegar rapidamente atingindo seu objetivo e levando consigo muito mais que mensagens, levam conhecimento, diversão e cultura (TANENBAUM, 2003).

Redes de Computadores tornaram-se, com o passar dos tempos, fundamentais e imprescindíveis na sociedade visto que a troca de informações é cada vez mais constante e crucial para seu desenvolvimento. Para suprir o crescente fluxo de informações, tecnologias antigas foram aprimoradas e novos padrões e periféricos foram criados. Milhares de dados de todos os tipos passam por essas redes a cada minuto, o número de transações confidenciais e importantes em meio a estes dados é incalculável, e o número de pessoas que tentam usufruir de falhas para obter estas informações também cresce de forma exponencial.

Formas de deter ataques, que visam desde a captura de informações confidenciais nas redes até mesmo a simples danificação dos dados em determinado computador ou seu uso para demais ataques, são desafios enfrentados todos os dias por administradores de redes. Para combater as metodologias de ataque, são utilizados *firewalls*, lista de regras para o tráfego de pacotes, em roteadores como primeira linha de defesa e sistemas de detecção de intrusos (IDS), capazes de detectar tentativas de invasão, junto à *firewalls* dedicados como segunda linha de defesa (CISCO, 2003).

Esses sistemas são refinados por administradores que visam manter sua rede segura, porém a cada dia surgem novas formas de atacar. Devido a demanda de sistemas capazes de enfrentar este problema preventivamente de forma mais eficaz, novas metodologias vem sendo estudadas e implementadas, até mesmo com o uso de inteligência artificial em sistemas de detecção de intrusos.

Dotado de inteligência, um administrador de redes é capaz de corrigir falhas e pensar em possíveis vulnerabilidades. Através do acúmulo de conhecimento prévio

adquirido em anos de profissão e a capacidade de aprender ao invés de repetir continuamente em ações conhecidas, provoca a descoberta de novos horizontes e assim a evolução, protegendo melhor sua rede.

Uma inteligência artificial em um sistema de detecção de intrusos poderia melhorar a performance da detecção e ajudar de forma mais eficaz o administrador do sistema. Estruturas paralelas que possuem a habilidade de aprender e generalizar resultados, respondendo a problemas não conhecidos ainda, ou seja, que antes não estavam em sua aprendizagem, são redes neurais artificiais (LUDWIG, 2007).

A gerência de falhas e a confiança dos usuários com a rede de computadores tornam-se maiores quando o administrador de redes possui ferramentas que o auxilie a detectar problemas e iniciar correções de configuração antes de ataques ou de perdas de informações (FRANCESCHI, 2003).

Embora as redes neurais artificiais possam ajudar no combate a ataques de computadores, a rede adequada deve ser encontrada, podendo variar devido ao ambiente e a forma de trabalho.

Sendo ideal moldar a rede neural artificial de forma gráfica e adequá-la facilmente em uma ferramenta de IDS para treiná-la e verificar seus resultados, desenvolveu-se uma ferramenta que permite criar a rede neural graficamente no simulador *JavaNNS* e adequá-la ao *Snort*(IDS) modificando apenas um código fonte do programa e um arquivo de configuração. Assim a modificação da rede neural torna-se fácil e seu aprendizado pode ocorrer direto em uma rede de computadores, em um IDS.

Também foram efetuados estudos sobre os tipos de redes neurais que poderiam melhor satisfazer a detecção de intrusos, apontando caminhos para futuros trabalhos nessa área. Foi demonstrado um exemplo da incorporação de uma RNA no *Snort.*

2. OBJETIVOS

Desenvolvimento de uma ferramenta que facilite a implementação de diversos tipos de redes neurais artificiais em um sistema de detecção de intrusos.

Estudos e proposta de uma rede neural artificial com melhor desempenho nos sistemas de detecção de intrusos.

3. REDES DE COMPUTADORES

A rede de computadores é o mecanismo mais utilizado e moderno para trafegar informações através de diferentes pontos do planeta, comunicações através da Internet tornaram-se cada vez mais comuns e fluentes no cotidiano de uma pessoa. O propósito inicial de uma rede foi a troca de mensagens e o compartilhamento de dispositivos em uma pequena região, permitindo que funcionários de uma empresa trocassem mensagens e utilizassem a mesma impressora. O crescimento desta tecnologia foi formidável e ganhou um enorme significado na vida humana, alterando modos de agir, pensar, trabalhar e estudar. Com trabalhos, eventos, ensino a distância e muitos outros recursos, o ser humano encontrou cada vez mais uma ligação à comunicação mundial pelas redes de computadores (CISCO, 2003).

Várias formas de envio de informações auxiliam a comunicação atual, ao invés de longos tempos esperando por cartas e por contatos, o simples envio de um e-mail, ou uma vídeo conferência, ou outro recurso de uma rede, podem levar todas as informações necessárias com muito mais velocidade e de forma mais eficiente (KUROSE, 2006). O acesso as informações distantes e a capacidade de obtenção de dados facilita um trabalho colaborativo.

Segundo (TANEMBAUM, 2003) o objetivo de uma rede de computadores deve ser: tornar os dados, informações, alcançáveis a todas as pessoas de uma rede. Atualmente, em várias empresas menores os computadores se encontram em uma mesma sala, em um mesmo espaço, porém em empresas maiores esse modelo modifica muito, os funcionários e as máquinas se distribuem entre diferentes países em diversas áreas do mundo. Mas o estado de um funcionário estar a milhares de quilômetros longe de seus dados, não deve significar que não possa usá-los como se fossem locais.

A Internet conecta as redes de computadores espalhadas por todos os pólos do mundo e define-se dificilmente, por estar sempre em modificações, referindo-se a componentes de hardware, software e recursos fornecidos pelos mesmos. Por esse

conjunto complexo de peças que se interagem através de protocolos, as redes são capazes de permitir que as informações atravessem todos os horizontes do mundo até atingir seu destino (KUROSE, 2006).

Embora o enfoque e as preocupações iniciais fossem voltados ao hardware (peças mecânicas) da rede, os softwares encontrados apresentam atualmente uma grande estruturação, sendo organizados em camadas ou níveis. Cada camada é responsável por uma parte da comunicação entre duas máquinas. A união dessas camadas torna capaz a troca de mensagens entre processos de sistemas localizados nos pontos da rede.

As camadas se agrupam em uma pilha, como pode ser visto na Figura 1.

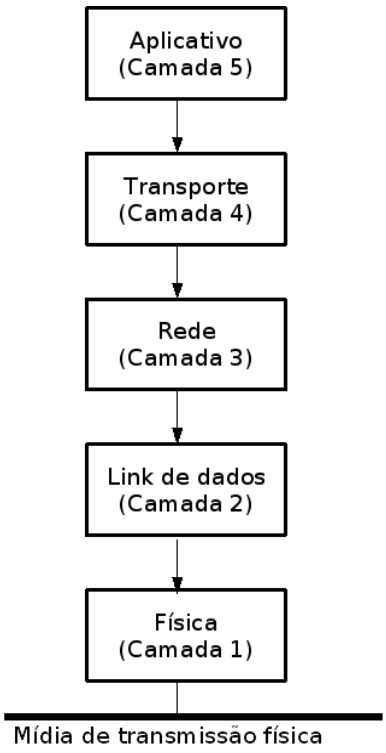

Figura 1: Modelo TCP/IP

Uma camada inferior deve fornecer os serviços às camadas superiores não sendo necessária a especificação de detalhes, mas somente os dados necessários. Essa abstração é necessária para permitir que uma camada *n* de uma máquina consiga se comunicar com a camada *n* de outra máquina. Os protocolos, conjunto de regras, organizam essas camadas estabelecendo sua forma de comunicação, seu formato e seu significado (TANEMBAUM, 2003).

O modelo de referência OSI (ISO7498, 2007) foi a primeira proposta de padronização para os protocolos, porém o modelo mais utilizado atualmente é o modelo TCP/IP, sendo TCP o Protocolo de Controle de Transmissão e IP o Protocolo da Internet, os dois protocolos mais importantes na Internet (KUROSE, 2006).

No modelo TCP/IP, segundo (TANEMBAUM, 2003) e (CASWELL, 2003), encontram-se as camadas:

- A camada de aplicativo é baseada nos protocolos de alto níveis, como por exemplo, os responsáveis pela troca de arquivos, pelo fornecimento de terminais e pelo correio eletrônico.
- A camada de transporte é formada pelos protocolos TCP e UDP, responsáveis por manter uma conversação entre duas máquinas na rede, o de origem e o de destino.
- A camada de rede é formada pelo protocolo IP, responsável por garantir que os pacotes enviados sejam transmitidos independentemente de seu destino.
- A camada de link de dados está ligada ao modo de conexão das redes, como Ethernet, Token Ring e ARP.
- A camada física destina-se aos equipamentos, como modens e placas de rede.

Em outras palavras, as informações em uma rede trafegam todas em pacotes, formados por dados de um tamanho e formato específico definidos pelos protocolos. Ao transmitir uma informação, o computador deve primeiro quebrá-la em pacotes seguindo as regras dos protocolos. Cada pacote desses conterá uma parte dos dados que formarão a mensagem completa. As máquinas que recebem os pacotes realizam o agrupamento e obtém a mensagem enviada. Nesses pacotes, todas as informações são guardadas, permitindo a leitura de dados, inclusive com más intenções, por alguém que os capture.

Com um crescimento exponencial das redes de computadores e conseqüentemente com o impacto provocado na sociedade, informações mais importantes passaram a trafegar, como informações de bancos, de instituições de

ensino, de lojas, de órgãos do governo, dentre outros. Devido a isso, indivíduos com diferentes intenções começaram a explorar falhas e capturar dados importantes com maus intuitos (KUROSE, 2006).

Dentre as explorações, um *hacker* chileno por exemplo, conseguiu fraudar a empresa Amazon em mais de US$ 220 mil e concretizou até o ano anterior mais de 2 mil transações em compras de produtos tecnológicos pagando com cartões de crédito ainda não emitidos (FRENCE PRESSE, 2007). Grandes valores podem ser perdidos além de existirem grupos destinados somente a invasão como em (ANSA, 2006) com membros que se interessam simplesmente achar falhas e divulgar aos demais.

Em (INFO ONLINE, 2007) foi informado um ataque aos e-mails não-confidenciais ao Pentágono, um grande departamento de defesa americano, que foi detectado pelos seus monitores. Mas ataques não ocorrem somente em grandes centros ou locais importantes, também são comuns em pequenas instituições e até mesmo em computadores domésticos, tanto no intuito de conseguir mais uma máquina para ajudar a capturar dados, quanto para resgatar informações importantes.

Devido a quantidade de dados importantes que trafegam, a segurança em redes de computadores tornou-se a maior preocupação de administradores de máquinas que se conectam à Internet, sejam estas residenciais ou empresariais. Embora existam diferentes formas de proteção, para monitorar de forma mais eficaz, impedir invasão e a conseqüente captura de informações importantes por indivíduos mal intencionados, surgiu a necessidade de ferramentas denominadas Sistemas de Detecção de Intrusos, capazes de identificar tentativas de invasão em tempo real (SNORTBRASIL, 2007).

4. SISTEMAS DE DETECÇÃO DE INTRUSOS

"A detecção de invasões é a prática de usar ferramentas automatizadas e inteligentes para detectar tentativas de invasão em tempo real" (ANÔNIMO, 2000). Existem dois tipos de sistemas básicos de invasão, os baseados em regras, com condições e premissas, e os adaptativos, com técnicas mais avançadas como inteligência artificial. Eles permitem uma abordagem preemptiva que pode monitorar a rede e agir assim que uma atividade suspeita é observada, ou evolucionária que se baseia em logs e age com segundos de atraso.

Embora muitos administradores pensem que o *firewall* seja essencial para a segurança total de uma rede e a única proteção necessária, isto pode estar errado. Seu trabalho pode impedir acessos a determinadas portas, gerar algum alerta a tentativas de acesso a algumas entradas que conheça, porém o tráfego não é analisado como nos SDI, o pacote não é verificado, então não há como definir o que pode e o que não pode entrar na rede. Contudo o uso de um *firewall* deve ser realizado, pois seu serviço seria de tipo um porteiro, um primeiro guardião (CASWELL, 2003).

Os modelos atuais de detecção de invasão são:

- por anomalia: observa atividades anormais de usuários, indicando outra pessoa na conta de um usuário conhecido do sistema.
    - estatístico: gerando perfis de usuário;
    - prognóstico de padrões: os eventos do sistema não são aleatórios, após conhecer conjuntos determina qual o próximo passo mais provável;
    - Bayesiana: modelo não supervisionado como os demais, faz cálculos para verificar qual o processo que pode ter gerado o grupo de classes. Usa probabilidade agrupando grupos de usuários com perfis semelhantes.
- por abuso: define e observa comportamentos de intrusos, chamados de assinaturas de intrusão.

- sistema especialista: através de regras “se-então” reconhece ataques e formula uma solução ao problema;
- modelamento: adquire informações de uma base de dados de situações, contendo seqüências de comportamento de ataques.

- híbridos: envolve técnicas de anomalia e de abuso.

Os métodos de detecção por abuso são os preferidos visto que possuem custo computacional reduzido e pequeno comprometimento de performance (CANSIAN, 1997).

A captura de pacotes que trafegam em uma rede e a avaliação dos mesmos permite uma análise de sua periculosidade, sendo que o mesmo pode fazer parte de um ataque. Para a captura mais simplificada dos pacotes, pode-se usar a biblioteca *libpcap*, que possibilita a obtenção de pacotes que trafegam na rede a nível de *kernel*, ou seja, diretamente da camada de “link de dados” do modelo TCP/IP. Também torna-se possível capturar pacotes destinados a outras máquinas na rede, e tudo isso de forma simplificada ao programador (TCPDUMP/LIBPCAP, 2007). Um exemplo de uso pode ser encontrado em (ANEXOS: Programa de exemplo com a biblioteca libpcap)

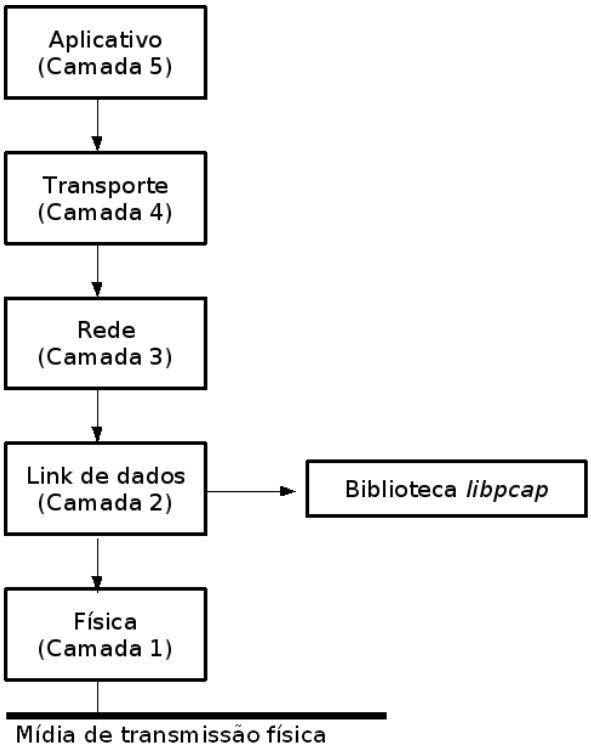

Figura 2: Camada onde a biblioteca libpcap adquire os pacotes

A biblioteca *libpcap* foi escrita dentro do projeto *TCPDump* e permite que os desenvolvedores escrevam um código que recebe os pacotes da camada de link em diferentes sistemas Unix e Windows (*winpcap*), sem a preocupação com detalhes e diferenças encontradas em diferentes sistemas operacionais. Isso permite decodificar, exibir e registrar os pacotes que trafegam na rede (CASWELL, 2003).

Alguns softwares que utilizam essa biblioteca:

- Sniff: Torna a saída dos pacotes do *tcpdump* mais fácil de ler
- *Snort*: Sistema de detecção de Intrusos
- TCP Replay: Repete o fluxo de pacotes em uma interface
- TTT: Simular ao TCPDump, mas disponibiliza gráficos e monitoramento em tempo real
- USI++: Adaptação do *libpcap* para C++
- Wireshark: Protocolo analisador de redes para Windows e Unix

- Uma lista mais detalhada pode ser encontrada em (TCPDUMP/LIBPCAP, 2007)

O *Snort*, escolhido para utilização neste trabalho, foi uma aprimoração, melhoramento, do desenvolvedor do TCPDump criado utilizando a biblioteca *libpcap*. Devido a enormes modificações, desenvolvimentos e aprimorações posteriores, passou a possuir pré-processadores para decodificar os pacotes, uma enorme flexibilidade nas configurações de regras para encontrar intrusos e plugins para melhorar seu desempenho (CASWELL, 2003).

4.1 Snort

O *Snort* é um software livre inicialmente desenvolvido por Martin Roesch, trabalha com assinaturas de tráfego malicioso para o controle das informações da rede. Essas assinaturas são partes de texto que podem ser encontradas em pacotes de uma possível invasão. Seu código fonte é otimizado em módulos e possui um constante desenvolvimento e atualização. Pode trabalhar de quatro formas diferentes:

- Farejador (Sniffer): simplesmente lê os pacotes e exibe seu conteúdo como um fluxo contínuo na tela;
- Registro de Pacotes (Packet Logger): cria um log dos pacotes no disco;
- Sistema de Detecção de Intruso (Network Intrusion Detection System): seu modo mais complexo, permite analisar o tráfego da rede através de regras predefinidas e executar ações diversas baseadas no que encontra;
- Inline: obtém os pacotes através do *iptables* ao invés do *libpcap* e faz o próprio bloquear ou liberar os pacotes.

Ao ser iniciado, o *Snort* adquire o tráfego da rede através da biblioteca *libpcap*, que habilita o modo promíscuo da interface, permitindo que todos os pacotes que passem sejam capturados e não somente os destinados àquela máquina. Logo após entra em um *loop* que permite capturar os pacotes da rede. Os pacotes são passados por uma série de rotinas de decodificação e preenchidos em uma estrutura denominada *Packet* (Tabela 1). Então são enviados para uma série de pré-processadores correspondentes, cada um verifica se o pacote é como deveria ser, seguindo após para um motor de detecção que verifica cada pacote baseado nas várias opções listadas nos arquivos de regras. Uma regra no *Snort* é formada por uma frase que possui o tipo, a fonte, o destino, conteúdo e outras informações dos pacotes que podem ser um ataque, por exemplo:

alert tcp $EXTERNAL_NET any -> $INTRENAL_NET 139

A regra descrita é responsável por alertar quando um endereço IP externo transmitir os pacotes TCP para a rede interna na porta 139 (CASWELL, 2003).

Devido aos pacotes estarem bem estruturados em uma estrutura interna, o desenvolvimento sobre o Snort permite a obtenção dos dados de forma simplificada. Dentre a estrutura Packet, encontram-se variáveis como:

Tabela 1: Variáveis da estrutura Packet

Variáveis	Descrição
pkth	Dados BPF
pkt	Ponteiro para os dados brutos do pacote
fddihdr	Cabeçalho para suporte FDDI atualmente utilizado em redes de fibras ópticas
trh	Cabeçalhos de suporte Token Ring
sllh	Cabeçalho de sockets Linux
pfh	Cabeçalho para a interface pflog do OpenBSD
eh	Cabeçalhos padrões para TCP/IP/Ethernet/ARP
wifih	Cabeçalho para rede local sem fio
eplh	Cabeçalho para EAPOL
iph, orig_iph	Dados como tamanho do datagrama, IP fonte e destino, e outros dados do protocolo IP
tcph, orig_tcph	Dados como portas, número da sequência, janela, e outros do protocolo TCP
udph, orig_udph	Dados como portas, tamanho e outros do protocolo UDP
icmph, orig_icmph	Dados como tipo, código e outros do protocolo ICMP
sp	Porta de origem (TCP/UDP)
dp	Porta de destino (TCP/UDP)
orig_sp	Porta de origem (TCP/UDP) do datagrama original
orig_dp	Porta de destino (TCP/UDP) do datagrama original

Mesmo com todas essas características, ainda encontram-se falsos positivos, alertas de ataques que na verdade não existem, e falsos negativos, ataques que não foram alertados. Para diminuir estes erros, plugins são criados e incorporados ao *Snort*, mas embora atinjam um bom desempenho, nenhum ótimo ainda foi

alcançado. Isso se torna uma preocupação ao administrador, pois como não sabe do acontecimento do ataque, não pode ter atitudes.

Seria interessante uma característica de aprendizado para o reconhecimento de padrões e de derivados que fosse capaz de se aprimorar com o decorrer do tempo. Devido a isso, o desempenho do *Snort* seria aprimorado e poderia atingir um quase ótimo em sua percepção.

Segundo (MITNICK, 2001) a inteligência é crucial para obter informações e tirar proveito delas, grandes ataques foram feitos tendo como base a engenharia social. A prevenção também deveria ser feita de forma inteligente, com um sistema capaz de ter percepções melhores do meio externo, podendo alertar e até impedir que falhas de hardware e outros softwares permitam à intrusos o acesso à informações importantes da rede.

Em (FRANCESCHI, 2003) foi abordado que o processo de gerenciar uma rede de dados depende muito da intervenção humana e define como as técnicas de inteligência artificial podem auxiliar no desenvolvimento de softwares e em como redes neurais artificiais podem atuar para demonstrar conhecimento do ambiente em que se encontra.

5. INTELIGÊNCIA ARTIFICIAL

“A palavra 'inteligência' vem do latim inter(entre) e legere(escolher)”. A possibilidade do ser humano de fazer escolhas, de notar diferenças, de aprender, é a inteligência, sendo esta a responsável por habilitar as pessoas a realizarem suas tarefas.

Vinda do latim, a palavra 'artificial' significa algo não natural, que não vem da natureza, mas é produzido por alguém. Denominados inteligência artificial técnicas criadas pela inteligência humana com a finalidade de capacitar a máquina a simular a inteligência de um homem (FERNANDES, 2005).

Inteligência artificial não é só uma disciplina, é também um objeto de investigação científica, saber como um programa consegue aprender, consegue emitir respostas a perguntas ainda desconhecidas e como ele se modifica em resposta as suas novas informações é algo que exige estudos à respeito. Programas conhecidos como tradutores, controle de tráfego aéreo, sistemas supervisionados, assistentes pessoais automatizados e outros utilizam os conceitos de inteligência artificial, empresas bem estruturadas e fortes como a NASA possuem tal tecnologia em alguns de seus produtos (DEAN, 1995).

O que mais se destaca no administrador de redes é a capacidade de aprendizado que registra, graças a seu cérebro, as informações relativas ao que já aconteceu para criar novas técnicas e aprimorar outras existentes. Esta característica de aprendizado é encontrada nas Redes Neurais que possuem a habilidade de tolerar entradas ruidosas e simular um aprendizado (RUSSEL, 2004). Elas possuem uma característica importante comum na inteligência humana, o reconhecimento de padrões, estes podendo não ser exatamente iguais aos apresentados anteriormente (CANSIAN, 1997).

Devido as características encontradas na capacidade e em sua tentativa de simulação de um cérebro humano, escolheu-se redes neurais artificiais (RNA) como o modelo mais apropriado de inteligência artificial para a tentativa de aprimorar os sistemas de detecção de intrusos neste trabalho.

5.1 Redes Neurais Artificiais

Redes neurais artificiais podem ser vistas de dois pontos diferentes, do ponto de vista computacional é um método de representação de funções usando redes de simples elementos aritméticos computacionais, ou métodos de aprendizado como representações por exemplos, sendo estas redes como circuitos que representam funções booleanas. Do ponto de vista biológico é um modelo matemático para a representação das operações do cérebro, onde os simples elementos aritméticos da computação representam os neurônios e a rede corresponde a coleção de neurônios interconectados (RUSSEL, 1995).

O cérebro humano possui cerca de 10 milhões de neurônios. Cada um possui dentritos, por onde recebem o sinal vindo dos outros neurônios. Possuem o corpo ou soma, responsável por receber e analisar as informações vindas dos outros neurônios. Para a transmissão de estímulos para outros neurônios, utilizam o axônio (Figura 3).

Sinais que chegam de olhos, pele, etc, chegam através dos dentritos aos neurônios. Se possuírem valor importante são enviados para o axônio. O sinal passa por uma sinapse, processo de ligação entre o axônio e o dentrito, antes de chegar em outro neurônio. Sua passagem é avaliada e somente segue o fluxo se for superior a um valor estipulado.

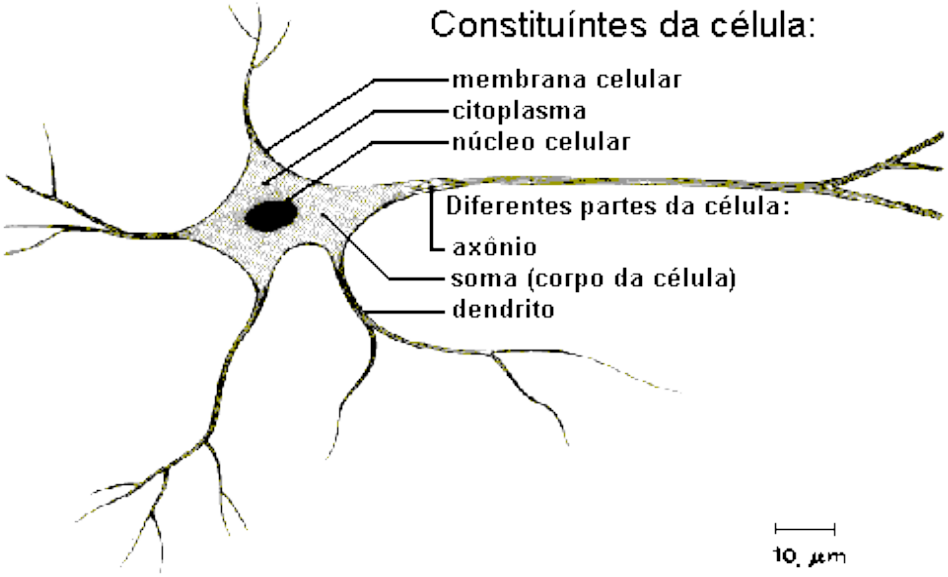

Figura 3: Neurônio (RNA, 2007)

O neurônio matemático recebe um ou mais sinais de entrada, que é avaliado e distribuído para outros neurônios ou definido como saída da rede. Os axônios e dentritos são representados por uma sinapse (LUDWIG, 2007).

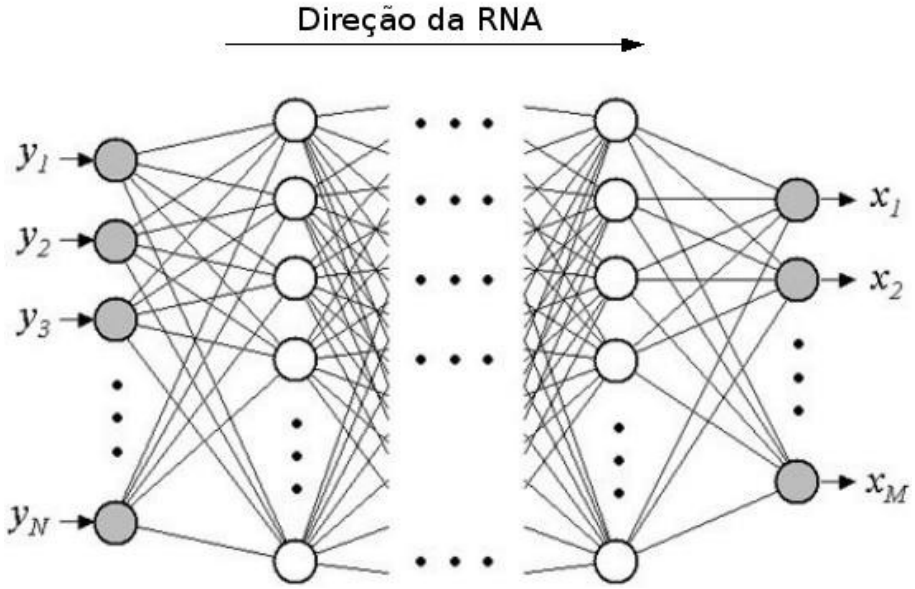

Figura 4: Rede Neural Artificial

Uma rede neural artificial (Figura 4) é composta por várias unidades de processamento, conectadas por canais de comunicação, sinapses, que estão

associadas a determinado peso. Os neurônios fazem operações apenas sobre os dados locais, que são entradas recebidas pelas suas conexões. O comportamento inteligente vem das interações entre as unidades de processamento da rede (FERNANDES, 2005).

A maior diferença entre a rede neural biológica e a matemática concentra-se nas sinapses, pois na rede biológica ocorre uma troca de informações por neurotransmissores, não sendo uma forma física, enquanto na rede matemática existe uma ligação física onde os pesos, responsáveis por guardar as informações, são armazenados (FRANCESCHI, 2003).

Baseado nos conceitos de (RUSSEL, 2004) a implementação de sistemas que possuem capacidade de aprendizado fornecida pelas redes neurais, teria um ganho da capacidade artificial de aprender e decidir o que fazer sem a necessidade de um administrador de redes, tomando decisões tão eficazes ou melhores quanto.

As redes neurais possuem três formas de aprendizado das informações:

- Supervisionado: precisa-se conhecer quais serão as entradas e as saídas correspondentes, para que um erro seja calculado e a rede possa sofrer alterações em seus pesos para diminuir esses pesos. É considerado um aprendizado com auxílio de um professor.

- Não supervisionado: precisa-se conhecer quais são as entradas. Os neurônios competem entre si e fornecem uma classificação como saída. A rede aprende de forma competitiva ou auto-supervisionada.

- Por reforço: os resultados satisfatórios são reforçados e os insatisfatórios provocam modificações nos valores das conexões.

As redes neurais artificiais podem ser diretas ou recorrentes. Em redes diretas o fluxo das informações são em uma única direção e redes recorrentes podem se realimentar de saídas de neurônios, em camadas à frente. Este procedimento permite que a RNA troque seu estado, deixando de utilizar certos neurônios e passando a utilizar outros com o passar do tempo (FRANCESCHI, 2003).

As redes neurais recorrentes apresentam a capacidade tanto de detectar quanto de reconhecer padrões que variam com o tempo. Deve-se haver um conjunto

suficiente de neurônios em sua camada interior para que possa estabilizar em um ponto desejado, que será capaz do reconhecimento desejado (FERNANDES, 2005).

Dentre as redes diretas e indiretas encontramos as diferenças através da Tabela 2 apresentada por (FRANCESCHI, 2003).

Tabela 2: Diferenças entre as redes diretas e recorrentes

Características	RNAs Diretas	RNAs Recorrentes
Quanto ao estado	Estáticas – não existe troca de estados	Dinâmicas – existe troca de estados conforme o tempo
Quando a topologia	Não possui ciclos. O fluxo dos dados possui apenas uma direção, da entrada para a saída.	Possui ciclos com realimentação. A saída de uma das camadas pode realimentar a entrada de dados (o fluxo pode ser da entrada para a saída, da camada intermediária para a entrada,da camada intermediária para intermediária, da saída para camada intermediária ou da camada de saída para a entrada).
Quanto ao número de camadas	Para solucionar um problema LINEAR: duas camadas, uma de entrada e uma de saída. NÃO LINEAR: tem que ter no mínimo uma camada intermediária.	Como existem ciclos com realimentação, pode haver ou não camada intermediária, depende da ordem do predicado.
Quanto a sua construção	Treinadas ou de forma supervisionada (como é o caso do algoritmo de treinamento Backpropagation) ou não-supervisionada (como é o caso dos mapas de Kohonen).	Construídas : exemplos de redes neurais construídas são BAM (Bidirectional Associative Memory - Kosko) e Hopfield . Treinadas : o ajuste dos pesos deve ser feito nos dois sentidos para frente e para trás. Exemplo deste tipo de rede ART (Adaptative Resonance Theory – Grossberg & Carpenter). Adaptadas através de exemplos : redes com retardos que aprendem através de exemplos de entrada

Os ataques em uma rede de computadores se modificam, alternando estados e tempos de execução, além dos novos tipos de ataque criados. Com uma rede neural capaz de identificar diferenças no modo padrão de funcionamento da rede de computadores, que seja capaz de reconhecer o tráfego normal de pacotes, seria possível identificar como ataques como modos distintos ao utilizados. Dentre as características apresentadas, as redes recorrentes merecem destaque e estudos para aplicações como a detecção de intrusos em redes.

6. UTILIZAÇÃO DE RNA NA GERÊNCIA DE REDES

Baseando-se em (FRANCESCHI, 2003) dentre os problemas encontrados, existem os bem definidos, onde o conjunto de entrada e saída é bem definido e as heurísticas podem ser utilizadas para solucionar o problema, e os mal definidos que não possuem conjuntos de entrada e saída bem definidos, são necessários exemplos para que uma rede neural consiga aprender, sendo sua saída um resultado próximo a solução e não uma saída exata.

Problemas bem definidos podem ser resolvidos por redes neurais diretas, enquanto problemas mal definidos necessitam das características apresentadas nas redes neurais recorrentes. Com os exemplos, consegue-se uma representação indireta dos problemas (Figura 5).

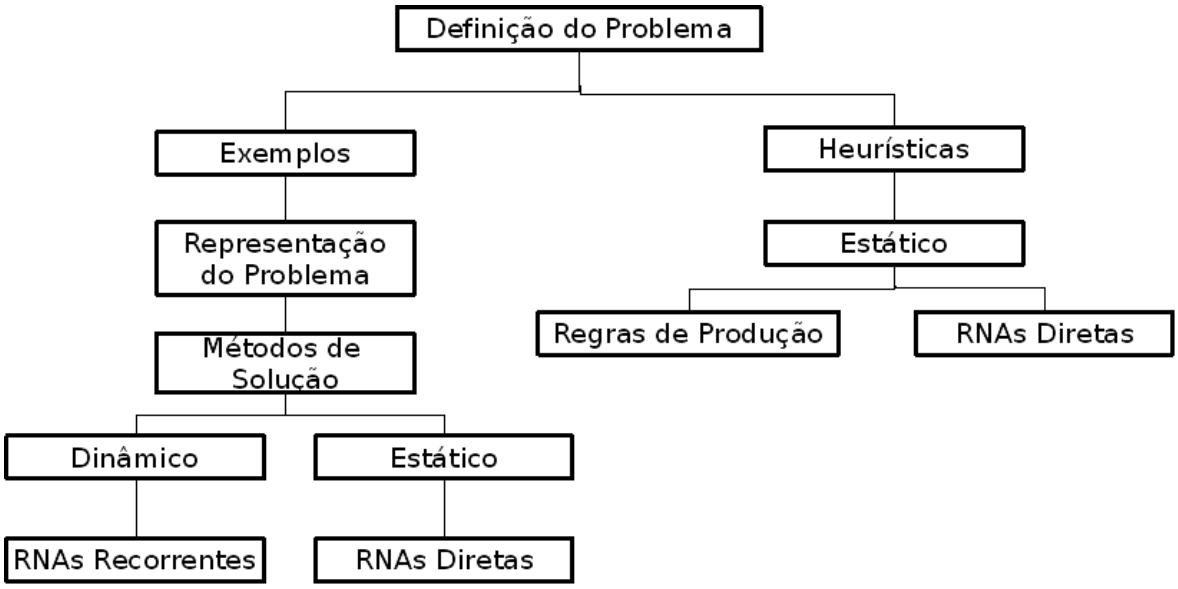

Figura 5: Métodos de Solução de Problemas (FRANCESCHI, 2003)

Uma caracterização válida para o funcionamento normal de uma rede de computadores é denominada *baseline*, porém torna-se necessário modificá-la ao ocorrer trocas ou funcionalidades novas no tráfego da rede de computadores.

No trabalho citado utilizou-se uma rede neural recorrente para detectar criar um *baseline* da taxa de utilização da rede de computadores. Porém comprovou-se que nem todas as redes neurais recorrentes podem ser utilizadas, pois uma rede

neural artificial denominada Elman não foi capaz de aprender com os exemplos. A realimentação dessa rede neural ocorre da camada intermediária para ela mesma, optou-se então por realimentar a rede da camada de saída para a camada intermediária, permitindo a criação de uma rede recorrente que aprendesse com os exemplos da rede.

Para a criação das redes neurais foi utilizado um programa denominado MatLab, onde as redes são modeladas e as chamadas de funções permitem a criação das redes, denominação de suas funções de aprendizagem e ativação. Permite-se também a modelagem de redes neurais personalizadas definindo-se as camadas, funções e demais atributos necessários. Mais informações podem ser encontradas em (MATLABNNT, 2007).

A rede neural artificial apresentada como testes em seu trabalho apresentava três camadas com a realimentação da camada de saída para a camada intermediária. Como funções de transferência das camadas foram a função sigmoidal, definida no MatLab como *sigmod*, nas primeiras camadas e a função *initnw*, na última camada. Foram adaptados atrasos de tempo ao enviar saídas. A função de treinamento foi a função *traingdx*. A quantidade de neurônios apresentados na camada oculta foi de 300 a 500, quanto maior o número de neurônios, mais rápido o aprendizado, porém não atingindo os resultados melhores. A entrada da rede neural foi através do protocolo SNMP em sua camada system (RFC1907, 2007), que apresentava o tráfego de entrada e de saída.

O fluxo de dados de uma rede de computadores modifica, mas eles costumam apresentar padrões modificados, uma espécie de rotina, ou seja, uma repetição de procedimentos comuns. Como exemplo, pode-se sempre ser aberto o mesmo *webmail*, que sempre apresenta as mensagens enviadas geralmente pelos mesmos conhecidos da pessoa, as mensagens incomuns podem ser *spams*, ou vírus, ou *trojans*, significando a tentativa de invasão à máquina.

Com tantos comportamentos comuns, pode-se obter um *baseline* da rede de computadores, para isso seria utilizada uma rede neural capaz de aprender esses padrões e reconhecê-los com pequenas modificações. Além de reconhecer também

comportamentos diferentes, não comuns, a determinados usuários ou máquinas de uma rede, utilizando o IP da estação de trabalho por exemplo.

A procura por uma rede que tenha características como essas é um árduo trabalho, na comunidade do SourceFire, grupo de desenvolvimento do Snort, foram numerosas as tentativas atuais de inclusão de RNA ao Snort (Informação obtida do *IRC* dos desenvolvedores do *Snort*).

Uma plataforma que permitisse a troca da estrutura da RNA, da modificação do treinamento e respostas rápidas às modificações sem a preocupação com desenvolvimento e adaptações complexas de códigos para essas novas tarefas, ajudaria no processo de escolha da rede neural a ser utilizada, além da inclusão de novas redes neurais em outros pontos do programa. Essa ferramenta contribuiria no desenvolvimento de novas formas de detecção de ataques, além de estimular futuras implementações com novos modelos.

Como a rede neural artificial de (FRANCESCHI, 2003) foi desenvolvida com o Matlab, em uma ferramenta proprietária e sem acesso ao código fonte, foi adotado o núcleo de um simulador de redes neurais (JAVANNS) com o código disponível e mantido pela universidade de Tübingen, na Alemanha. Com este núcleo poderiam ser desenvolvidas redes neurais graficamente e inseridas de forma simples no snort para serem treinadas e testadas. Este simulador possui a interface em Java e é portável em diferentes plataformas de sistemas operacionais.

6.1 Matlab

Matlab é uma ferramenta para facilitar e agilizar os processos de computação. Possui versões para diferentes tipos de sistemas operacionais.

Seu projeto foi iniciado em 1970 como uma interface mais simples para resolver problemas como grandes sistemas de equações. Sua idéia sempre foi prover e simplificar uma forma simples de uso para resolver programas que seriam de implementação complicada.

Foi modificada constantemente até tornar-se uma simples ferramenta que auxilia os usuários a resolver seus problemas expressando, por exemplo, as equações matemáticas que necessitam resolver.

Fornece gráficos em 2D e 3D, além de modelos e suportes a diferentes assuntos com base matemática (MATLAB, 2007).

6.2 JavaNNS

Java Neural Network Simulator (*JavaNNS*) é um simulador de redes neurais com a parte visual elaborada em Java e com o núcleo do simulador Stuttgart Neural Network Simulator (SNNS) em sua versão 4.2. Ambos foram desenvolvidos no Instituto Wilhelm-Schickard-Institute para Ciência da Computação em Tübingen, na Alemanha.

Com desenvolvimento constante desde 2006, apresenta grande suporte às redes neurais artificiais, possuindo estruturas e treinamentos variáveis. Com os cliques de um mouse e definições de alguns parâmetros, as redes neurais podem ser criadas, visualizadas e treinadas. Possui também um gráfico de erros para o acompanhamento dos erros e opções de modificação em sua interface visual, além de ser compatível em diferentes sistemas operacionais.

Pelas características citadas, esta ferramenta foi adequada ao *Snort*, permitindo a mudança facilitada do método de aprendizado e a criação facilitada da RNA. Deve-se salientar que nem todas as redes neurais serão compatíveis, foram priorizadas as redes recorrentes, apresentadas como a possível rede a detectar um *baseline* na rede de computadores. Devido a isso, topologias diferentes de RNAs nos códigos fontes podem não funcionar, o próximo capítulo demonstra maiores informações.

Um manual simplificado desenvolvido pode ser encontrado em (ANEXOS: Tutorial JavaNNS). Um manual mais detalhado pode ser encontrado em (JAVANNS, 2007).

7. DESENVOLVIMENTO

7.1 Necessidades

Para o desenvolvimento da ferramenta seria necessário que a rede neural criada no *JavaNNS* fosse compatível com o *Snort*, ou pudesse ser exportada para a compilação integrada ao programa. O *Snns*, núcleo do *JavaNNS*, apresenta um programa para a conversão do projeto da rede neural para a linguagem C, fornecido nos fontes do *Snns*, é denominado *snns2c*. Ele convertia os arquivos ".net", gerados pelo *JavaNNS* em ".c", sua sintaxe:

snns2c <rede-snns> <nome-arquivo-saída> <nome-da-função>

Onde:

- <rede-snns>: informa a rede criada no *JavaNNS*, extensão .net;
- <nome-arquivo-saída>: define nome do arquivo em linguagem C que será criado;
- <nome-da-função>: define o nome da função do arquivo a ser produzido.

Os primeiros testes de incorporação realizados, mostraram que em um estado normal, apresentariam várias etapas que consumiriam o tempo do programador com a parte de implementação do programa, ao invés da rede neural artificial.

Mesmo a rede neural sendo incluída ao *Snort*, após suas alterações e adaptações para seu funcionamento, ainda havia um problema, o treinamento na RNA somente poderia ocorrer no *JavaNNS*, pois as redes exportadas pelo *snns2c* possuíam somente o código de execução da rede neural com seus pesos.

Junto ao programa *snns2c* foi encontrada outra ferramenta denominada *netlearn*, capaz de treinar a rede neural construída no JavaNNS, para sua execução eram necessários:

- o arquivo da rede neural criada no *JavaNNS;*

- o arquivo de exemplos de entrada, denominados *patterns;*
- a definição dos parâmetros para o treinamento;
- a definição da quantidade de ciclos a serem executados.

Com os arquivos e as definições citados acima, a rede neural poderia ser treinada. Porém os conjuntos de entrada deveriam ser obtidos. Para isso, em primeiros testes, foi criada uma função que escreveu na saída padrão os valores desejados. Após a coleta das entradas, um arquivo de treino tinha que ser criado manualmente e executado no JavaNNS sobre o código da rede neural (“.net”), utilizado para exportar o código para linguagem C. Após o treinamento, obtinha-se um novo arquivo “.net” com novos pesos, que deveria ser novamente convertido e modificado para funcionar no *Snort*.

Optou-se a modificação do código do programa *snns2c* para a importação dos pesos da rede neural, do arquivo da rede localizado em “/etc/snort/rna.net” no Linux ou em “./rna.net” no Windows. Assim, quando a rede fosse treinada, bastaria substituir o arquivo referenciado, que os novos pesos seriam recarregados automaticamente.

Para o processo de aprendizagem da rede neural, optou-se pela modificação do código do programa *netlearn*, assim o treinamento poderia ser feito diretamente pelo *Snort*, que também seria o responsável por salvar os novos pesos da RNA no arquivo descrito no parágrafo anterior.

7.2 Implementação efetuada no Snort

Os dados do Snort são agrupados em estruturas internas do programa, que fornecem informações de execução e variáveis auxiliares. Para adequar-se a rede neural, a estrutura _progvars, onde são armazenadas as principais variáveis do programa, foi alterada adicionando-se uma variável denominada rna que continha a estrutura _rna_opts, apresentada na Tabela 3.

Tabela 3: Estrutura _rna_opts

Declaração da variável	Descrição
int type	tipo de execução: run, train, trainoff ou printresult
int net_loaded	verificação se a rede pode ser carregada
int train_cycles	quantidade de ciclos que devem ser feitos no treinamento
int num_patterns	número de entradas existentes no arquivo .pat
char *config_filename	nome do arquivo de configuração
char *net_filename	nome do arquivo da rede neural
char *train_filename	nome do arquivo de treinamento
fpos ...	posição do arquivo de treinamento que deverá ser escrito:
fpos_t num_pat	a quantidade de patterns
fpos_t num_input	o número de entradas
fpos_t num_output	o número de saídas
float attack	número mínimo para a detecção de ataques
float answer	armazenamento do maior número que a rede neural respondeu
FILE *train_file	stream do arquivo de treinamento
float parameters[4]	parâmetros para o treinamento

No programa, esta estrutura pode ser acessada pela variável pv.rna, definida na parte inicial de sua execução. Exemplo: pv.rna.type = 1;

Para a escolha do usuário na utilização da RNA, foi adicionado um novo parâmetro de entrada, o -Z, que recebe uma opção para a definição do tipo de execução do *Snort* com RNA:

- run: responsável por executar como detecção de intrusos. Todo pacote com resposta superior a variável *attack*, demonstrada na estrutura anterior e determinada no arquivo de configuração, seria considerado ataque.
- train: responsável por capturar os pacotes da rede e fornecê-los como entrada à rede neural para seu treinamento.
- trainoff: responsável por repetir o último treinamento efetuado com a rede neural.

- printresult: responsável por exibir a maior saída obtida da rede neural e auxiliar na escolha do valor da variável *attack*.

Para a configuração dos parâmetros de execução e treinamento da rede neural, foi criado o suporte a um arquivo de configuração localizado em "/etc/snort/rna.conf" no Linux ou "./rna.conf" no Windows, onde podem ser configurados:

- attack: valor mínimo de saída para o *Snort* considerar ataque;
- parameter_0: primeiro parâmetro para o treinamento da rede neural;
- parameter_1: segundo parâmetro para o treinamento da rede neural;
- parameter_2: terceiro parâmetro para o treinamento da rede neural;
- parameter_3: quarto parâmetro para o treinamento da rede neural;
- parameter_4: quinto parâmetro para o treinamento da rede neural;
- cycles: quantidade de ciclos para o treinamento da rede neural.

Esses parâmetros podem ser visualizados no "Painel de Controle" do *JavaNNS*, sendo utilizados de dois a cinco parâmetros (parameter_0, parameter_1, parameter_2, parameter_3, parameter_4,), dependentes da função de aprendizado escolhida, como pode ser visto, por exemplo, na Figura 6 (opções: n, dmax, forceT). Para mais informações acesse o manual em (SNNS, 2007).

A opção de disponibilizar randomização da ordem do conjunto de entradas na rede neural não foi disponibilizada, pois podem identificar ataques. Por exemplo, a procura de portas por *nmap* é seguida de tentativas de conexão nas portas abertas, nessa ordem de execução.

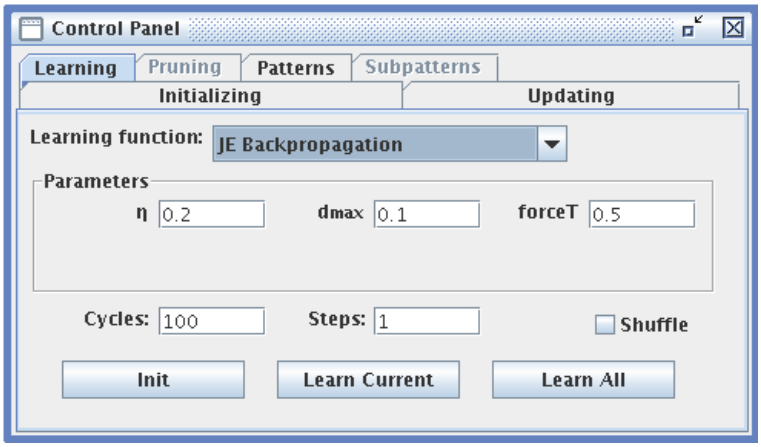

Figura 6: Painel de Controle do aplicativo JavaNNS

As variáveis que não estão presentes no arquivo de configuração são inicializadas com valores padrões.

Para o treinamento da rede neural no *Snort* foi criado um arquivo com o conjunto de entradas. Para criar o início e o fim deste arquivo em uma parte do Snort que não estava no *loop* de execução, foram adicionadas as funções:

- rna_write_pat_header: responsável por iniciar a escrita do arquivo "rna.pat";
- rna_finalize_pat_file: responsável por finalizar o arquivo de treinamento, definindo o número de conjuntos de entrada gravados.

Além dessas modificações no *Snort*, foram incluídos novos arquivos:

- kernel_snns: diretório que armazena os arquivos do *kernel* do snns, responsável pelo treinamento da rede neural, leitura dos pesos iniciais, etc.
- rna.{c,h}: implementado para o controle da execução e treinamento da rede neural. Estruturas e funções:
  - estrutura _my_ip: possui quatro inteiros internos para representação do IP, sendo: <inteiro>.<inteiro>.<inteiro>.<inteiro> ;
  - função ip_to_double: função responsável por transformar um número IP em um double, embora não seja utilizada pelo programa, foi mantida para ajuda em futuras implementações;

- função get_ip_part: função responsável por capturar cada parte do IP e estruturá-lo na estrutura _my_ip;
- função which_protocol: função responsável por devolver uma palavra que simboliza o nome do protocolo;
- função write_dados: função responsável por escrever na tela características do pacote, embora não seja utilizada pelo programa, foi mantida para ajuda em futuras implementações;
- função write_csv: função responsável por escrever na tela uma saída que poderá ser aberta como planilha eletrônica, embora não seja utilizada pelo programa, foi mantida para ajuda em futuras implementações;
- função write_snns_pattern: função responsável por escrever os conjuntos de entradas e saídas da RNA no arquivo de treinamento;
- função Test_with_rna: função responsável per invocar a RNA e detectar se o pacote corresponde a um ataque ou qual a maior resposta da rede neural, dependendo da opção de execução escolhida;
- função Rna: função principal do arquivo, que controla a chamada das outras funções, o armazenamento das entradas no vetor de entrada da rede neural.

• rna_alg.c: arquivo criado com modificações do *snns2c*, para compilação com *Snort* e para adquirir os pesos do arquivo da rede neural;

• rna_lib.{c,h}: importado do SNNS para compilação da função de treino e de definir valores;

• rna_train.{c,h}: modificação do arquivo netlearn do SNNS, responsável por treinar a rede neural do *Snort*;

• rna_functions.h: importado do SNNS para compilação da função de treino e de definir valores;

• define_values.{c,h}: modificação do arquivo snns2c do SNNS, responsável por carregar os pesos do arquivo rna.net para a rede neural do *Snort;*

- rna_templates.h: importado do SNNS para compilação da função de treino e de definir valores;

Foi implementado uma modificação do programa *snns2c* para que a rede neural seja exportada no formato adequado. Sua sintaxe:

snns2c <rede-snns>

onde <rede-snns> corresponde ao arquivo da rede neural. Será produzido o arquivo “rna_alg.c” no formato correto para incorporação no *Snort*.

Embora todos os testes e implementações tenham sido realizados no ambiente Linux, a opção de execução com RNA também foi adicionada para sistemas Windows, pois mesmo sem testes realizados, a rede neural não depende do sistema operacional, somente de suas entradas corretas. A estrutura do *Snort* fornece as informações dos pacotes para ambos os sistemas, devido a biblioteca *libpcap*.

Após as alterações especificadas no código fonte da ferramenta, pode-se incorporar ao *Snort* um suporte a diversas redes neurais artificiais criadas no *JavaNNS*, permitindo sua inclusão simplificada neste programa de detecção de intrusos.

A implementação descrita pode ser encontrada em (ANEXOS: Implementações Efetuadas)

7.3 Execução

A execução do *Snort* com RNA, ocorre de quatro formas, dependentes da opção do usuário:

- run: Carrega os pesos da RNA, a executa recebendo como entrada os pacotes que trafegam na rede de computadores e define qual pacote pode pertencer a um ataque, baseado na saída da rede neural. Quando uma detecção é realizada, é exibida a mensagem na saída padrão:

```
======== ATTACK ========
| [icmp] rna(0.934059) |
\======================/
```

onde é apresentado o tipo de protocolo do pacote capturado e a saída da RNA.

- printresult: Carrega os pesos da rede neural e a executa armazenando o maior valor de saída apresentado. Este valor é apresentado no final da execução do programa, com a mensagem:

```
The bigger ANN answer: 0.196062
```

- train: Carrega os pesos da rede neural, capturando os dados dos pacotes e formando o conjunto de entradas da rede neural, salvando no arquivo de treinamento. A cada pacote capturado apresenta a mensagem:

```
N patterns colleted...
```

Sendo `N` substituído pelo número do conjunto obtido.

Durante a finalização do programa, o arquivo de treinamento é finalizado e ocorre o treinamento da rede neural. O aprendizado baseia-se na função de treino, contida no arquivo da rede neural, e nas opções do arquivo de configuração. Ao terminar o treinamento, os novos pesos são salvos no arquivo da rede neural.

Para cada entrada da rede neural deve-se determinar um valor de saída. Considera-se que o treinamento deva ocorrer no tráfego normal da rede de computadores, assim a saída é definida como 0, para aprender o equivalente a um *baseline.*

- trainoff: Diferente do fluxo dos demais, esta opção modifica o fluxo de execução do programa para chamar diretamente finalização do programa, onde o aprendizado irá ocorrer com o último conjunto de entradas capturado.

Exceto na opção *trainoff*, o *Snort* inicia definindo valores padrões para os dados internos, conectando-se a biblioteca *libpcap* e iniciando em um *loop* onde a cada pacote recebido será chamada a função *ProcessPacket*, que se encarregará de decodificar o pacote para e estrutura interna, *Packet*, e iniciar o sistema de detecção *Preprocess* ou *Rna*.

Ao ser chamada a função *Rna* são capturadas dos pacotes e definidas como enrada da RNA as informações especificadas na Tabela 4.

Tabela 4: Entrada da rede neural

Número da entrada	Descrição
0	Protocolo utilizado
1	Primeira parte inteira do ip de origem
2	Segunda parte inteira do ip de origem
3	Terceira parte inteira do ip de origem
4	Quarta parte inteira do ip de origem
5	Primeira parte inteira do ip de destino
6	Segunda parte inteira do ip de destino
7	Terceira parte inteira do ip de destino
8	Quarta parte inteira do ip de destino
9	Tempo de vida do pacote
10	Tamanho do datagrama
11	Identificação do pacote
12	Soma interna do pacote (checksum)
13	Porta de Origem
14	Porta de Destino

Para a definição da entrada da rede neural foi realizada uma coleta das informações de um grande número de pacotes, sendo escolhidos os campos que apresentavam maiores modificações no decorrer do tempo, como pode ser visualizado na Tabela 5.

Tabela 5: Informações dos Pacotes (IP: 201.78.79.89 / Rede doméstica)

Protocolo	IP Origem	IP Destino	Porta orig	Porta dest	Tamanho do datagrama	Identificação	Tempo de vida do pacote	Soma dos dados internamente
icmp	201.78.79.89	201.78.100.38	1194	1194	17920	48694	64	7421
icmp	201.78.79.89	201.78.100.38	1194	1194	17920	47670	64	8445
icmp	201.78.79.89	201.78.100.38	1194	1194	17920	51766	64	4349
icmp	189.11.70.74	201.78.79.89	4672	4672	22528	55901	57	52743
icmp	201.78.79.89	201.78.100.38	1194	1194	17920	10806	64	45309
tcp	84.90.61.93	201.78.79.89	2537	4662	54277	24360	64	33277
tcp	201.78.79.89	211.121.19.44	4662	63593	10240	59804	64	39326
tcp	201.78.79.89	201.86.34.85	4662	61007	10240	37499	64	60090
tcp	201.78.79.89	201.86.34.85	4662	61007	13312	31612	64	62905
tcp	201.78.79.89	122.20.169.195	4662	37337	13312	47935	111	26163
tcp	201.24.43.121	201.78.79.89	50573	4662	16386	35578	64	35262
udp	221.5.157.165	201.78.79.89	33305	4672	16128	23364	24	62616
udp	87.196.197.148	201.78.79.89	62221	4672	16128	32101	36	191
udp	201.78.79.89	83.208.34.63	4672	12165	37632	0	108	12467
udp	189.175.173.152	201.78.79.89	4672	4672	35072	49342	113	46086
udp	201.78.100.38	201.78.79.89	1194	1194	10752	0	62	42742
udp	218.160.60.151	201.78.79.89	48815	4672	16128	38792	101	38812

Após o vetor de entrada ser preenchido, os métodos de execução, exceto trainoff, são executados.

Os pesos da rede neural artificial e a função de treinamento estão contidos no arquivo da rede neural, localizado em “/etc/snort/rna.net” no Linux e “./rna.net” no Windows. O arquivo de treinamento gerado pelo *Snort* se localiza em “/etc/snort/rna.pat” no Linux e ”./rna.pat” no Windows. Ambos os arquivos são compatíveis com o *JavaNNS*.

Em uma visão de alto nível da arquitetura do *Snort*, o código original seguia o comportamento da Figura 7, onde os pacotes eram capturados da rede, eram organizados e processados para a estrutura *Packet*, seguiam para o mecanismo de detecção baseado em regras e continuavam seu trajeto pelos módulos de alertas e registro.

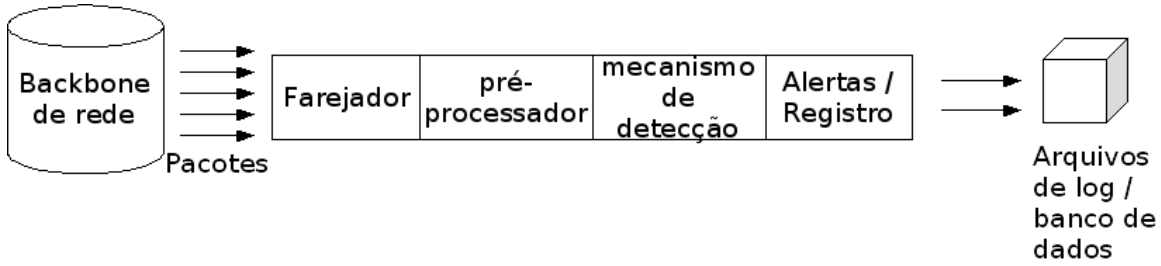

Figura 7: Arquitetura original do Snort

Na nova arquitetura, uma nova opção de detecção foi incorporada, permitindo o comportamento como da Figura 8, onde os pacotes são capturados da rede, organizados e processados para a estrutura *Packet*, seguindo para o mecanismo de detecção baseado em regras, ou para o mecanismo de detecção com RNA, e continuando seu trajeto pelos módulos de alertas e registro.

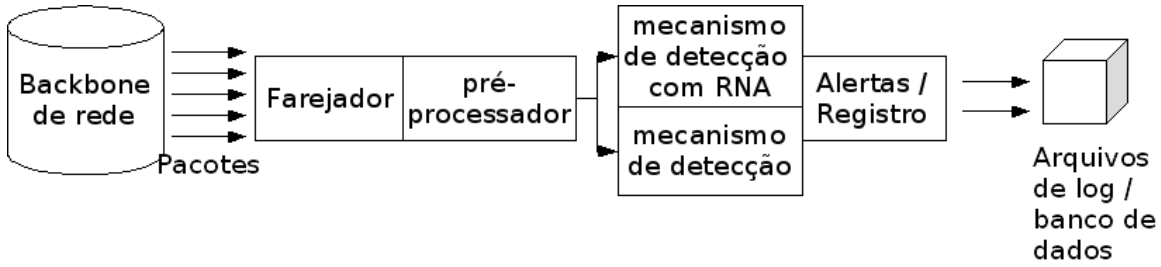

Figura 8: Nova arquitetura do Snort

8. ANÁLISE

8.1 Comparação entre o modo inicial e o modelo proposto

Nesta seção são descritas as formas de incorporar uma rede neural do programa *JavaNNS* ao *Snort*. Primeiramente são descritos os passos no modo inicial de implementação, sem a utilização da ferramenta desenvolvida neste trabalho, logo após é exibida a forma atual para a incorporação.

A primeira forma de adaptação na rede neural do *JavaNNS* ao *Snort*:

a) Criação da rede neural com o *JavaNNS*

b) Conversão do código da rede para linguagem C

c) Criação de novas rotinas no *Snort* para que a rede neural fosse chamada

d) Adaptação do código fonte da rede neural

e) Criação de uma função para escrever os dados que iriam compor o arquivo de treinamento

f) Compilação e execução para obter os conjuntos de testes corretos

g) Criação manual do cabeçalho do arquivo de treinamento, seguido dos dados obtidos no item anterior

h) Utilização do *JavaNNS* para abrir a rede neural e o arquivo de treinamento criado

i) Utilização do *JavaNNS* para treinar a RNA, que logo depois foi salva com os novos pesos

j) Conversão do código do *JavaNNS* para linguagem C

k) Adaptação do código fonte da rede neural

l) Compilação e execução com o *Snort* para verificar a RNA

m) Modificação do código fonte para demonstrar todas as saídas da rede neural

n) Criação de novas rotinas desconhecidas na rede de computadores (*nmap, ssh, etc*)

o) Compilação e obtenção das saídas da rede

p) Modificação do limite para invasão no código fonte do programa

q) Exclusão da demonstração de todas as saídas da rede

r) Inclusão de mensagens de alertas para saídas acima do limite definido

s) Execução e testes do *Snort* com a rede neural treinada

t) Modificações posteriores ao encontrar resultados ruins

A forma atual para a incorporação da rede neural ao *Snort* são:

a) Criação e inicialização dos pesos da rede neural no *JavaNNS*. O suporte do trabalho abrange as redes que ao serem transformadas em linguagem C, possuam a estrutura:

- Vetor UnitType com nome Units
- Vetor pUnit com nome Sources
- Função Rna_alg

b) Conversão do código do *JavaNNS* para linguagem C com o programa *snns2c* modificado para gerar o arquivo “rna_alg.c”.

c) Substituição do arquivo “rna_alg.c” nos fontes do *Snort*

d) Compilar o *Snort* com o novo arquivo

e) Cópia do arquivo com o código do *JavaNNS* para “/etc/snort/rna.net” no Linux, ou “./rna.net” no Windows.

f) Definir as variáveis do arquivo de configuração

g) Executar o *Snort* executando o treinamento e execução

O modelo atual demonstrou ser bem mais simplificado do que o modo antigo de implementação, fornecendo ao programador mais pesquisas e tempo para o trabalho com as redes neurais.

8.2 Utilização do modelo proposto

Como exemplo, uma rede neural baseada em conceitos apresentados neste trabalho foi criada e inserida no *Snort*, para avaliação da funcionalidade da ferramenta desenvolvida.

No *JavaNNS*, foram criadas quatro camadas de neurônios demonstrados na Figura 9:

- Camada de entrada (*Input*) contendo 15 neurônios ativados pela função Act_Identify. Quantidade de neurônios relacionada a entrada da rede neural implementada no trabalho
- Camada oculta (*Hidden*) contendo 30 neurônios ativados pela função Act_Logistic.
- Camada oculta especial (*Special Hidden*) contendo 30 neurônios ativados pela função Act_Logistic. Sendo a responsável pela recorrência
- Camada de saída (*Output*) contendo 1 neurônio ativado pela função Act_Logistic. Quantidade de neurônios relacionada a saída da rede esperada pela implementação do trabalho.

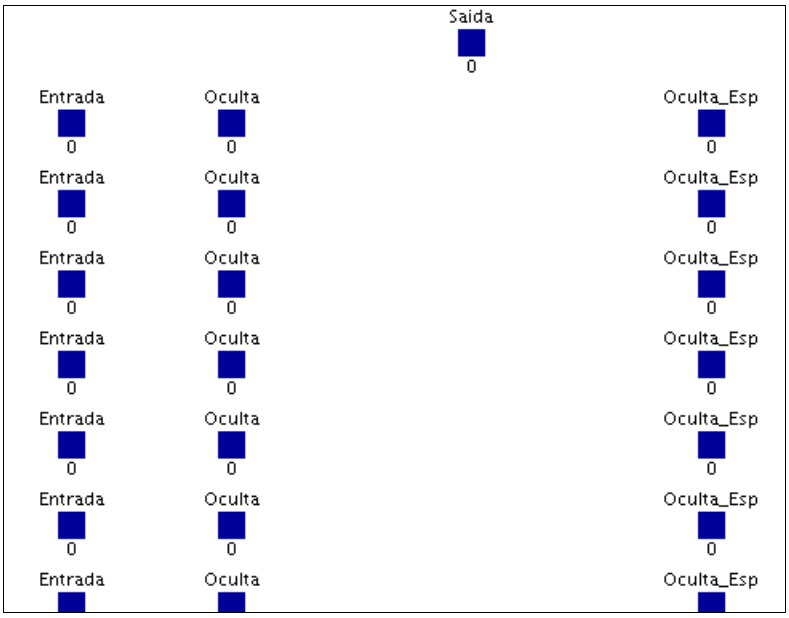

Figura 9: Camadas da rede neural artificial criada

A saída dos neurônios da camada de Entrada foi ligada como entrada da camada Oculta, como pode ser visto na Figura 10.

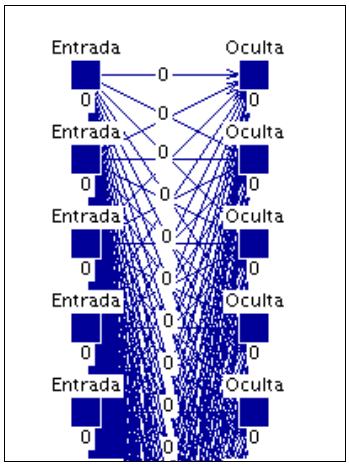

Figura 10: Ligação da camada de entrada a camada oculta

A saída dos neurônios da camada Oculta foi ligada como entrada da Camada de Saída (Figura 11).

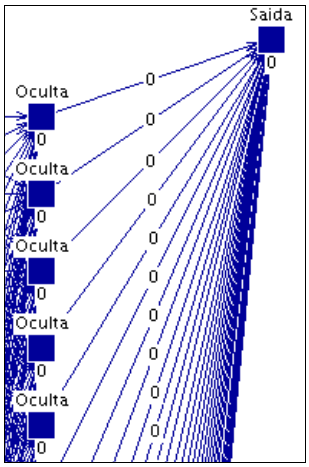

Figura 11: Ligação da camada oculta a camada de saída

A Saída foi ligada a camada Oculta Especial, Figura 12.

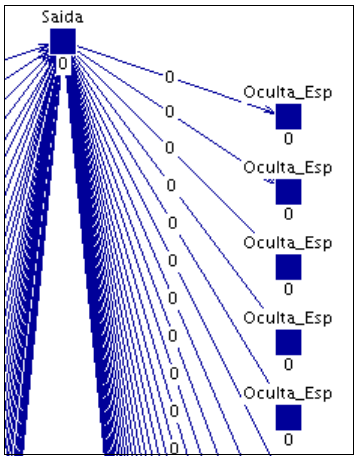

Figura 12: Ligação da camada de saída a camada oculta especial

A saída dos neurônios da camada Oculta Especial foi ligada como entrada à Camada de Saída e à camada Oculta, como pode ser visto na Figura 13.

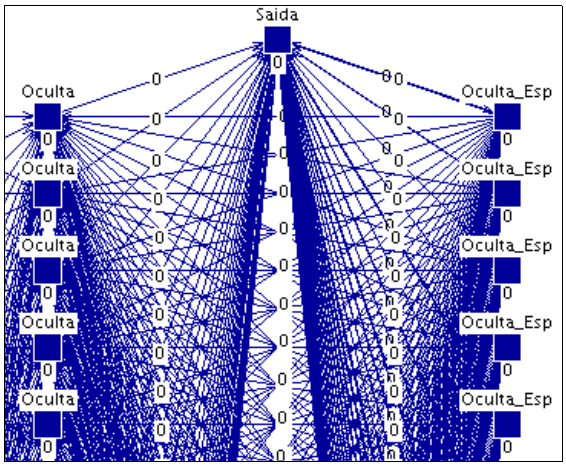

Figura 13: Ligação da camada oculta especial a camada de saída e a oculta

No “Painel de Controle” foi escolhida a função “*for Jordan or Elman Networks*” e os pesos foram inicializados. A Figura 14 demonstra o painel de controle inicializando os valores da rede neural, que não são valores 0, como na figura 12.

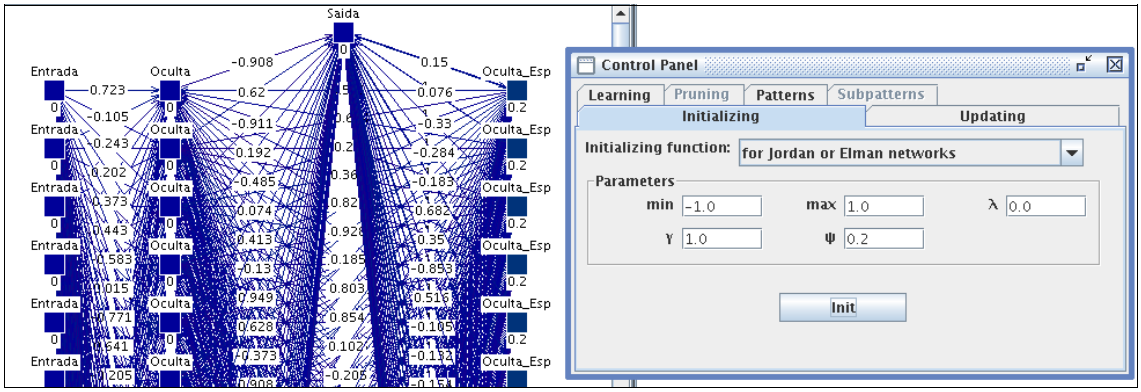

Figura 14: Inicialização dos valores da rede neural

Na aba Learning, Figura 15, foi escolhida a função JE Backpropagation, suportada pelo *Snort*, para o aprendizado.

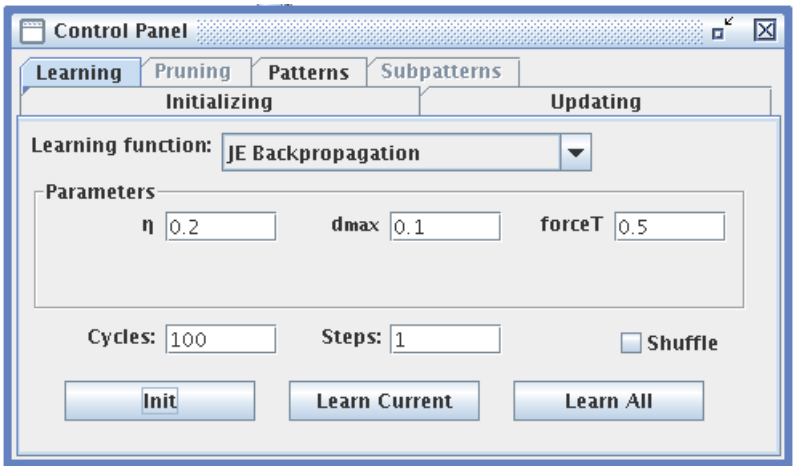

Figura 15: Escolha da função de treinamento da RNA

A rede foi salva e convertida com o programa *snns2c* modificado, produzindo o arquivo rna_alg.c, que foi colocado nos fontes do *Snort* substituindo um arquivo com o mesmo nome.

O *Snort* foi recompilado e o arquivo da rede neural do *JavaNNS* (com a extensão .net), foi copiado para "/etc/snort/rna.net"

O arquivo /etc/snort/rna.conf foi alterado, colocando os três parâmetros padrões apresentados no "Painel de Controle" do *JavaNNS*:

- parâmetro 0: 0.2
- parâmetro 1: 0.1
- parâmetro 2: 0.5

Foi feito então a coleta de dados dos pacotes em um arquivo de log, pra que pudesse ser utilizado na verificação do aprendizado da rede neural:

```
# snort -i ppp0 -L /tmp/coleta
```

A opção "-i" permite a escolha da interface a ser utilizada.

Foram executados na rede os comandos abaixo, sendo *ping* um comando de pergunta para servidores, *wget* um comando para obter conteúdo da Internet e *apt-get* a ferramenta para manipulação de pacotes em algumas distribuições Linux:

```
# ping www.terra.com.br

# ping www.google.com.br
```

```
# wget www.google.com.br

# apt-get update
```

Após estes comandos, a coleta do tráfego foi finalizada. Então foi realizado o treinamento da rede neural utilizando o *Snort*, com o tráfego armazenado no arquivo /tmp/coleta:

```
# snort -Z train -r /tmp/coleta.*
```

A opção “-Z” permite a utilização da rede neural artificial.

Após o treinamento, foi obtido o maior valor de saída da rede com o comando:

```
# snort -Z printresult -r /tmp/coleta.*
```

Foi estabelecido este valor no arquivo de configuração “/etc/snort/rna.conf”.

O programa foi executado para verificar como se comportaria com outros comandos:

```
# snort -i ppp0 -Z run
```

Executando na forma mais verbal com os primeiros pacotes coletados no arquivo /tmp/coleta:

```
# snort -Z printresult -r /tmp/coleta.* | grep RNA
```

Percebeu-se que a maioria dos números de saída apresentados na rede neural eram menores que a saída marcada como maior, a rede não teria obtido a quantidade de pacotes e de ciclos adequados para o aprendizado, ou havia sido formulada de forma errada, ou pela estrutura, ou pela função de treinamento, etc. Problemas deste tipo serão encontrados aos usuários que se dedicarem a utilizar RNAs em um sistema de detecção de intrusos.

O comportamento da ferramenta foi o esperado.

9. CONCLUSÃO

Dentre os resultados e o conteúdo estudado observa-se ser uma tarefa complicada projetar uma RNA capaz de aprender a detectar intrusos em uma rede de computadores. O caminho mais próximo na área de gerência de redes encontrado dentre os estudos foi o de (FRANCESCHI, 2003) que conseguiu projetar uma rede recorrente para o aprendizado de *baselines* em relação a taxa de tráfego de uma rede de computadores. O problema encontrado foi a utilização de uma ferramenta com código fechado que não permitiu a visualização ampla do modo de implementação da rede, por isso a escolha do uso do *JavaNNS*, ferramenta menos poderosa, mas que possui um código aberto para futuras implementações.

Embora as RNAs recorrentes apresentem características interessantes e resultados melhores do que as RNAs diretas, seu conceito é mais complexo e valores como o número de neurônios na camada intermediária ou número de ciclos de treinos tornam-se muito importantes para seu aprendizado, podendo levar o usuário a descartar uma rede neural pensando estar errada, quanto na verdade não foi treinada com ciclos adequados ou não recebeu entradas suficientes.

Uma dificuldade encontrada foi em adaptar o treinamento das redes neurais, existem várias topologias de redes neurais e várias formas de treinamento para cada tipo, havendo também modelos de treinamentos não compatíveis com certos modelos de redes neurais. Devido a isso foi necessária a priorização de um tipo de rede neural e modificação do arquivo de treino de uma forma específica, por isso nem todos os tipos de RNAs poderão ser feitos e testados no programa. Para a adaptação de todos os tipos de treinamento e de todas as redes, o modelo atual deve ser modificado, pois as estruturas de outros tipos de redes são diferentemente formuladas e organizadas.

Outra dificuldade encontrada foi na especificação da entrada da rede neural, devido a reconhecer quais os padrões dos pacotes que sofrem mais alterações em uma invasão. Como meio final foi projetada uma tabela com as informações

importantes do pacote, colocado um fluxo normal na rede e visualizados quais os dados que sofriam mais alterações com o decorrer do tempo.

Pelos resultados obtidos conclui-se que a dificuldade apresentada, anteriormente ao trabalho, foi solucionada, sendo necessário no modelo proposto poucos passos para o uso de uma rede neural recorrente na ferramenta de detecção de intrusos *Snort*. Essa adaptação simplificada facilitará a adequação da RNA ao *Snort*, disponibilizando ao usuário mais tempo para estudos e trabalhos na formulação das redes neurais artificiais.

10. TRABALHOS FUTUROS

Com base no trabalho de desenvolvimento deste trabalho a as flexibilidades que ainda podem ser implementadas, destacam-se os trabalhos futuros:

- Adaptar para que a rede neural consiga efetuar a leitura de regras de ataque do *Snort*, podendo adquirir conhecimentos maiores que de um *baseline;*
- Terminar a integração do *JavaNNS*, permitindo todos os tipos de treinamento. Além de fornecer somente o arquivo .net gerado para a leitura da rede, descartando-se a necessidade de recompilação do código do *Snort;*
- Atribuir mais redes neurais, uma cada tipo de protocolo, com resultados unificados e avaliados;
- Criar uma rede neural para tomar as decisões de defesa na rede de computadores;
- Estudos mais elaborados sobre os pacotes para definir entradas mais eficientes para a rede neural artificial.

11. BIBLIOGRAFIA

ANÔNIMO. **Segurança Máxima para Linux**. Rio de Janeiro: Editora Campus, 2000.

ANSA. **Invasores de site da Nasa são presos no Chile**. Disponível em: http://www1.folha.uol.com.br/folha/informatica/ult124u20916.shtml. Acesso em: 27 de maio de 2007.

CANSIAN, Adriano Mauro. **Detecção de Intrusos em Redes de Computadores**. Tese (Doutorado em Física Aplicada), Instituto de Física de São Carlos, São Carlos, 1997.

CASWELL, Brian, BEALE, Jay, FOSTER, James C., POSLUNS, Jeffrey. **Snort 2, Sistema de Detecção de Intruso**. Rio de Janeiro: Editora Alta Boks Ltda, 2003.

CISCO SYSTEMS. **Internetworking Technologies HandBook, Fourth Edition**. Cisco Press, (S/L), 2003.

CORRÊA, Angelita de Cássia. **Metodologia para análise comparativa de Sistemas de Detecção de Intrusão**.2005, Dissertação (Mestrado em Engenharia da Computação), Instituto de Pesquisas Tecnológicas do Estado de São Paulo, São Paulo, 2005.

DEAN, Thomas L, ALLEN, James, ALOIMONOS, John. **Artificial Inteligence**: theory and practice. Menlo Park: Addison Wesley Publishing Company, 1995.

FERNANDES, Anita Maria da Rocha. **Inteligência Artificial**: noções gerais. Florianópolis: Visual Books, 2005.

FONSECA, Caio Roncaratti. **Segurança de redes com uso de um aplicativo firewall nativo do sistema Linux**. Ribeirão Preto: Centro Universitário Barão de Mauá, 2006.

FRANCESCHI, Analúcia Schiaffino Morales de. **Aplicação de Técnicas de Inteligência Artificial no desenvolvimento de agentes para gerência de Redes**. Tese (Doutorado em Engenharia Elétrica), Universidade Federal de Santa Catarina, Florianópolis, 2003.

FRENCE PRESSE. **Polícia chilena prende hacker que fraudou a Amazon em US$ 220 mil**. Disponível em: http://www1.folha.uol.com.br/folha/informatica/ult124u21387.shtml. Acesso em: 27 de maio de 2007.

INFO ONLINE. **Pentágono confirma ataque cracker**. Disponível em: http://info.abril.com.br/aberto/infonews/092007/04092007-21.shl. Acesso em: 21 de novembro de 2007.

ISO7498. **Open System Interconnection Model**. Disponível em: http://www.sigcomm.org/standards/iso_stds/OSI_MODEL/index.html. Acesso em: 10 de dezembro de 2007.

JAVANNS. **University of Tübingen: JavaNNS**. Disponível em: http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome_e.html. Acesso em: 10 de dezembro de 2007.

KIRCH, Olaf. **Guia do Administrador de Redes Linux**. Curitiba: Conectiva Informática, 1999.

KOMAR, Amit. **Artificial Inteligence and Soft Computing - Behavioral and Cognitive Modeling of the Human Brain**. USA: CRC Press LCC, 2000.

KRÖSE, Bem, SMAGT, Pattrick van der. **An introdution to Neural Networks**. The Amsterdam: University of Amsterdam, 1996.

KUROSE, James F, ROSS, Keith W. **Redes de Computadores e a Internet**: Uma abordagem top-down. São Paulo: Pearson Addison Wesley, 2006.

LUDWIG, Oswaldo, MONTGOMERY, Eduard. **Redes Neurais, Fundamentos e Aplicações em Programas em C**. Rio de Janeiro: Editora Ciência Moderna Ltda, 2007.

MATLAB. **MATLAB**. Disponível em: http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html. Acessado em: 10 de dezembro de 2007.

MATLABNNT. **Neural Network Toolbox**. Disponível em: http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/. Acessado em: 10 de dezembro de 2007.

MENDES, Wayne Rocha. **Linux e os hackers – proteja seus sistema**: ataques e defesas. Rio de Janeiro: Editora Ciência Moderna Ltda., 1999.

MITNICK, Kevin D., SIMON, Willian L. **The Art of Deception**: Controling the Human Element of Security. USA: Wiley, 2001.

NETFILTER. **NetFilter/Iptables project home page**. Disponível em: http://www.netfilter.org. Acesso em: 21 de maio de 2007.

RNA. **Redes Neurais Artificiais**. Disponível em: http://www.icmc.usp.br/~andre/research/neural/. Acessado em: 20 de novembro de 2007.

RFC1907. **Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)**. Disponível em: http://www.faqs.org/rfcs/rfc1907.html. Acessado em: 19 de maio de 2007

RFC3577. **Introduction to the Remote Monitoring (RMON)**. Disponível em: http://www.faqs.org/rfcs/rfc3577.html. Acessado em: 19 de maio de 2007

RUSSEL, Stuart J, NORVING, Peter. **Artificial Inteligence**: a modern approach. Rio de Janeiro: Editora Prentice-Hall do Brasil, 1995.

RUSSEL, Stuart J, NORVING, Peter. **Inteligência artificial**: tradução da segunda edição. Rio de Janeiro : Elsevier Editora, 2004.

SMITH, Roderick W. **Redes Linux Avançadas**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2003.

SNNS. **University of Tübingen: SNNS**. Disponível em: http://www-ra.informatik.uni-tuebingen.de/software/snns/. Acessado em: 11 de dezembro de 2007

SNORT. **Snort Users Manual 2.6.1**. Disponível em: http://www.snort.org/docs/snort_manual/2.6.1/snort_manual.pdf. Acessado em: 10 de maio de 2007.

SNORTBRASIL. **Snort Brasil**. Disponível em: http://www.clm.com.br/snort/. Acessado em: 10 de novembro de 2007

SPYMAN. **Manual Completo do Hacker**. Rio de Janeiro: Editora Book Express, 2001.

STREBE, Matthew, PERKINS, Charles. **Firewalls**. São Paulo: Makron Books, 2002.

TANENBAUM, Andrew S. **Redes de Computadores**: Tradução da Terceira Edição. Rio de Janeiro: Editora Campus Ltda., 1997.

TANENBAUM, Andrew S. **Redes de Computadores**. Rio de Janeiro: Editora Campus, 2003.

TCPDUMP/LIBPCAP. **TCPDUMP/LIBPCAP public repository**. Disponível em: http://www.tcpdump.org. Acesso em: 30 de Novembro de 2007.

ANEXOS

Programa de exemplo com a biblioteca libpcap

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
//inclusão da biblioteca libpcap
#include <pcap.h>
#include <netinet/ip.h>

//função chamada ao término do programa
void sair(){
   printf("Saindo...\n");
   exit(0);
}

//programa que receberá os pacotes da rede
pcap_handler roda
(u_char * user, struct pcap_pkthdr * pkthdr, u_char * packet)
{
   printf("Pacote passando...\n");
   printf("IP Fonte: %s\n\n",
           inet_ntoa( ((struct ip*) (packet+14))->ip_src));
}

//função principal do programa
int main(int argc, char **argv)
{
      char *arquivo, *expr, ebuf[PCAP_ERRBUF_SIZE];
      int opt, snaplen = BUFSIZ, promisc = 1;
      pcap_t *cap; bpf_u_int32 net, mask;

      signal (SIGTERM, sair);
      signal (SIGINT, sair);
      signal (SIGHUP, sair);
      signal (SIGKILL, sair);

      //abre a interface eth0 em modo promíscuo
      if ((cap = pcap_open_live ("eth0", snaplen,
                                 promisc, 1000, ebuf)) == NULL)
      {
            fprintf (stderr, "pcap: %s\n", ebuf);
            exit (1);
      }

      if (pcap_lookupnet ("eth0", &net, &mask, ebuf) < 0)
      {
```

```
            fprintf (stderr, "pcap: %s\n", ebuf);
            exit (1);
        }

        //chama a rotina de repetição que pegará os pacotes,
        //passando-os à função roda()
        if ((pcap_loop (cap, -1, (pcap_handler) roda, NULL)) < 0)
        {
            fprintf (stderr, "pcap: %s\n");
            exit (1);
        }
}
```

Utilizando o JavaNNS

O *JavaNNS* possui uma interface em Java amigável, possuindo janelas internas como a rede neural, o gráfico de erros, o painel de controle e outros. Além das opções encontradas em seu Menu, visualize a interface do programa na ilustração abaixo.

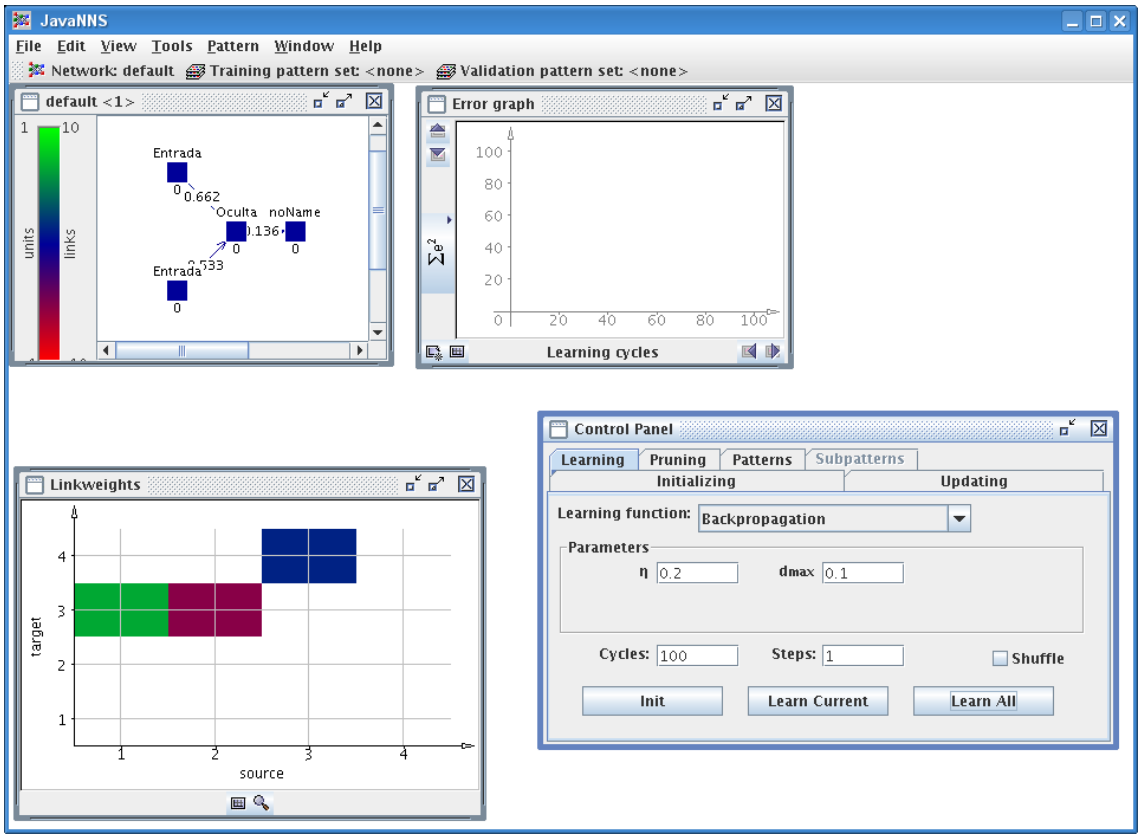

Ilustração 1: JavaNNS

Criando uma rede neural artificial

Para criar uma rede neural, escolha no menu *Tools*, a opção *Create*, depois a opção *Layers*.

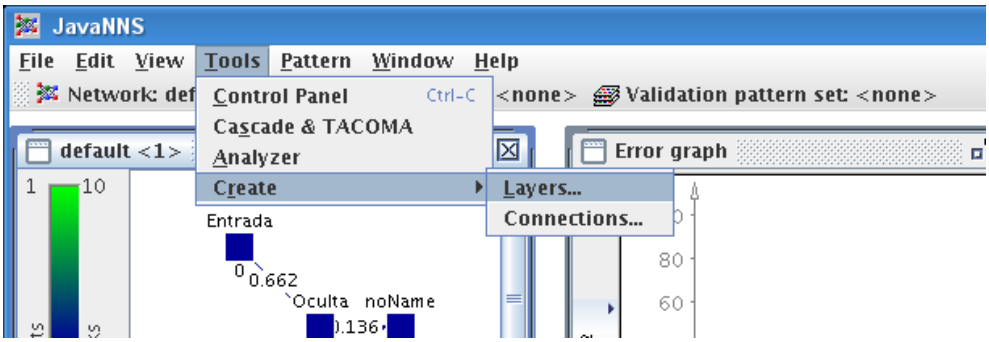

Ilustração 2: Criação das camadas

Será apresentada a seguinte janela de diálogo:

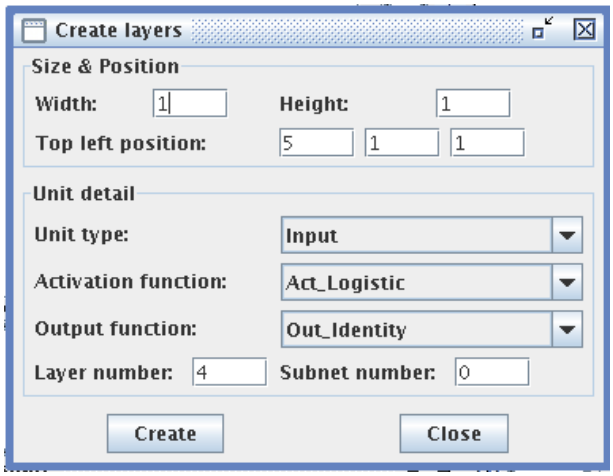

Ilustração 3: Diálogo da criação das camadas

Descrição das escolhas do diálogo:

- *Width*: o número de neurônios a serem criados na horizontal
- *Height*: o número de neurônios a serem criados na vertical
- *Top left position*: Posição inicial que devem ser criados os neurônios
- *Unit type*: Tipo de neurônio, podendo-se esconder como da camada de entrada, da camada escondida, da camada de saída, dentre outros.
- *Activation function*: Função de ativação a ser utilizada. Dentre as funções encontramos:
  - *Act_Identify*: função linear utilizada geralmente na camada de entrada
  - *Act_Logistic*: função sigmoidal utilizada geralmente nas camadas ocultas

- *Output function*: função de saída do neurônio a ser utilizada
  - *Out_Identity*: função utilizada geralmente na camada de saída
- *Layer number*: Número da camada
- *Subnet number*: Número da rede interna

No campo *Height* escreva o número 2, selecione *Input*(entrada) no campo *Unit type* e *Act_Identity* no campo *Activaction function*, após clique em *Create*. Acaba de ser criada uma camada de entrada com dois neurônios utilizando a função *Act_Identity*.

Logo após, no campo *Height* escreva o número 1, selecione *Hidden*(oculto) no campo *Unit type* e *Act_Logistic* no campo *Activaction function*, após clique em *Create*. Acaba de ser criada uma camada oculta com um neurônio.

Por último, no campo *Height* escreva o número 1, selecione *Output*(Saída) no campo *Unit type* e *Act_Logistic* no campo *Activaction function*, após clique em *Create*. Acaba de ser criada uma camada oculta com um neurônio.

As camadas da rede neural acabaram de ser criadas.

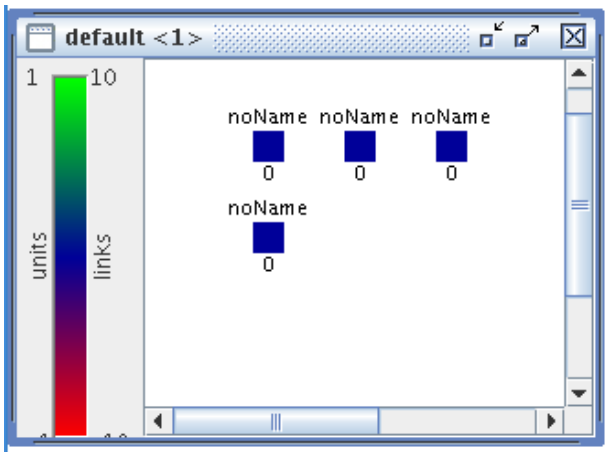

Ilustração 4: Camadas da rede neural

Agora devem ser criadas as conexões, escolha no menu *Tools*, a opção *Create*, depois a opção *Layers*.

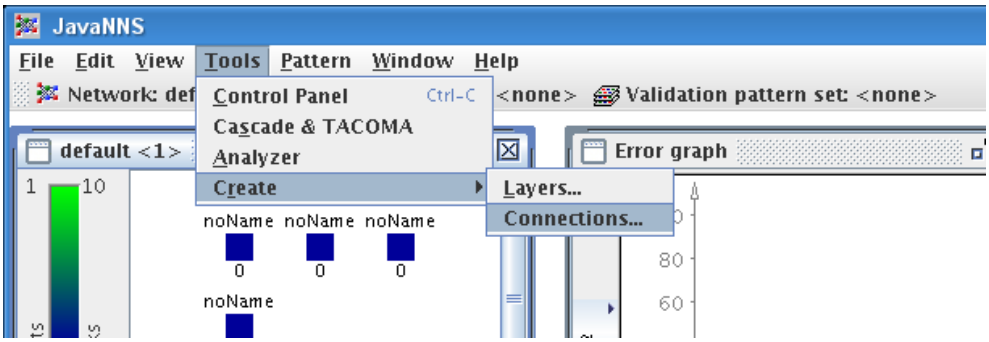

Ilustração 5: Criação das conexões das camadas

Aparecerá o diálogo:

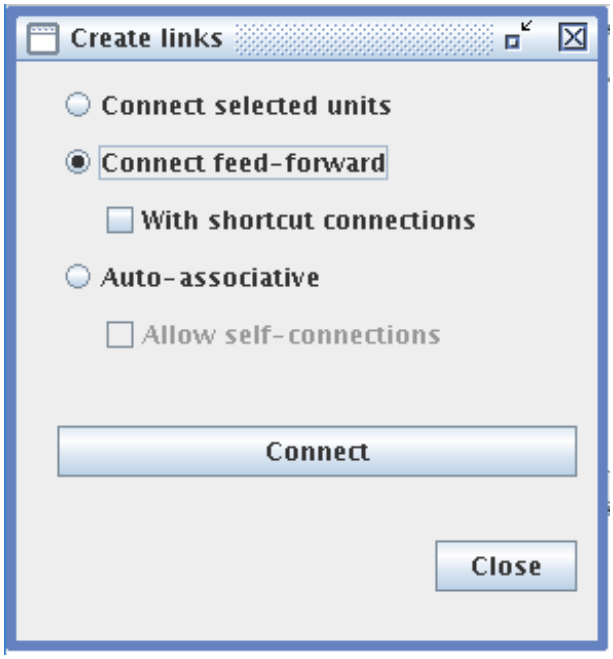

Ilustração 6: Diálogo de criação as conexões das camadas

Descrição das escolhas do diálogo:

- *Connect selected units*: permite selecionar e cria as conexões manualmente
- *Connect feed-forward*: Cria conexões feed-forward
- *Auto-associative*: Cria conexões auto associativas

Escolha a opção *Connect feed-forward* e clique em *Connect*, a rede está criada e conectada.

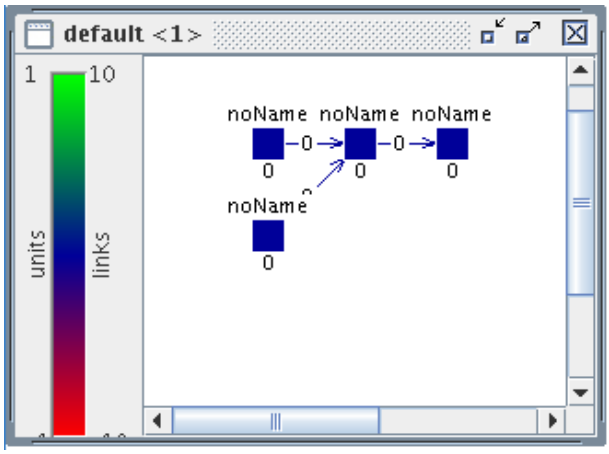

Ilustração 7: Rede neural conectada

Manipulando a rede neural criada

Para Inicializar a rede, treinar e manipular entradas, deve-se abrir o “Painel de Controle”. Escolha no menu *Tools* a opção *Control Panel*.

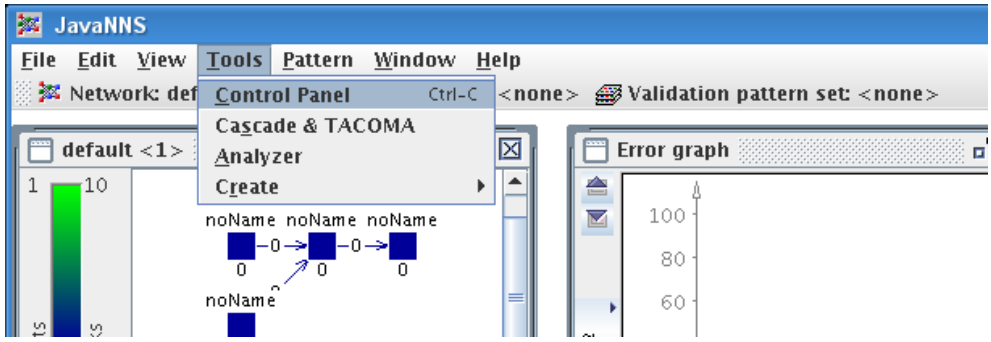

Ilustração 8: Acesso ao Painel de Controle

Será exibido o diálogo:

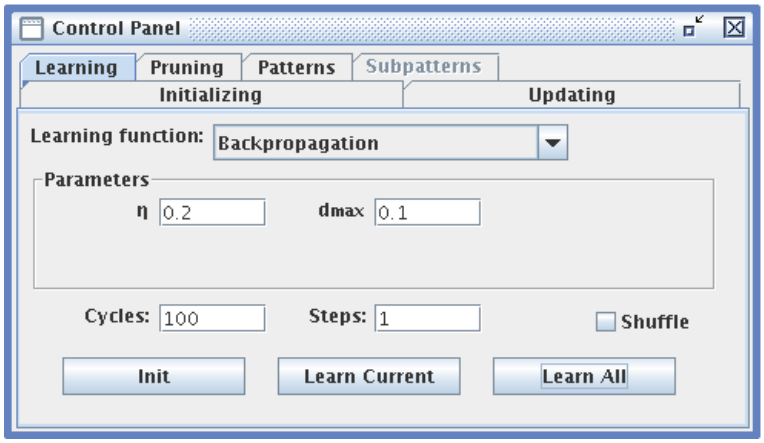

Ilustração 9: Painel de Controle

Nele a rede poderá ser iniciada e treinada nas abas:

- *Initializing*: inicia os pesos da rede neural, escolha a função de inicialização em Initializing function
- *Learning*: define as opções para a aprendizagem da rede, escolha a função em Learning function
- *Patterns*: conjuntos de entrada e saída da rede

O *Learning* (item aprendizado) só funciona se existirem *patterns* criados e carregados no programa.

Os patterns devem vir com o seguinte cabeçalho:

```
SNNS pattern definition file V1.4
generated at Sun Jan 10 00:00:00 2008


No. of patterns     : <NUM>
No. of input units  : <INPUT>
No. of output units : <OUTPUT>

<PATTERNS>
```

Onde:

- <NUM> deve ser o número de conjuntos que este arquivo possui
- <INPUT> deve ser o número de entradas da rede neural
- <OUTPUT> deve ser o número de saídas da rede neural
- <PATTERNS> deve ser o conjunto de entradas e saídas, sendo cada linha formada por <INPUT> números seguidos de <OUTPUT> números. Por exemplo, uma RNA com 2 neurônios de entrada, sendo os números 1 e 0, e uma de saída, onde deseja-se que seja 1. A linha a ser escrita deve ser: 1 0 1

O arquivo *pattern* deve ter a extensão .pat

Treinamento da rede neural artificial

Deve-se criar o arquivo de entrada, como descreve o item enterior. Neste exemplo, poderia ser:

```
SNNS pattern definition file V1.4
generated at Sun Jan 10 00:00:00 2008


No. of patterns     : 4
No. of input units  : 2
No. of output units : 1

1 0 1
1 1 1
0 1 1
0 0 0
```

O arquivo deve ser carregado no *JavaNNS*.

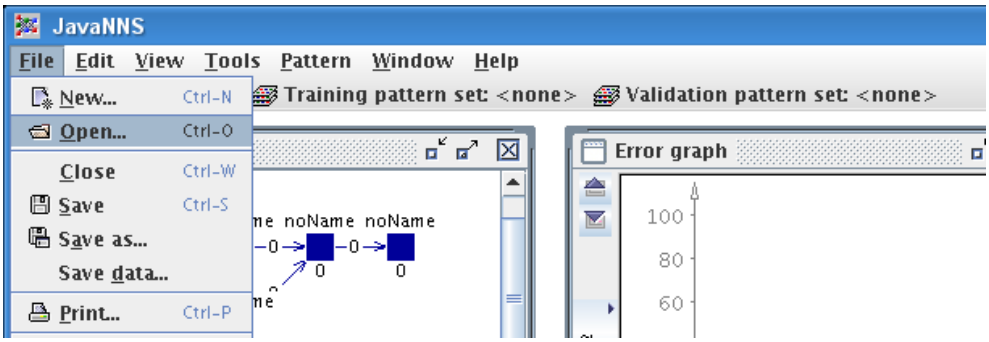

Ilustração 10: Acesso à opção abrir arquivo

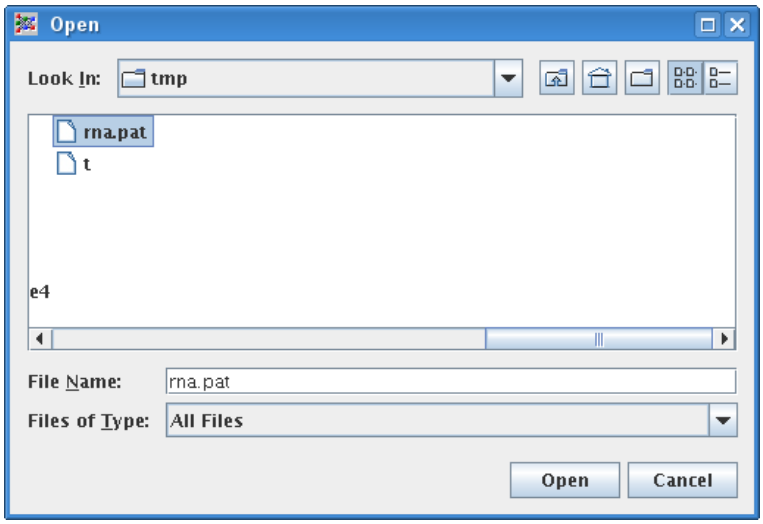

Ilustração 11: Escolha do arquivo a ser aberto

Após abrir o conjunto de entradas(*pattern*), abra o “Painel de Controle” e clique na aba *Initializing*, pressione então o botão *Init*. Com a rede inicializada, clique na aba *Learning* e clique em *Learn All*.

Implementações Efetuadas

As implementações adaptadas ao programa Snort, as alterações de códigos fontes do SNNS e as modificações de código podem ser encontradas nos quadros que seguem.

O arquivo *rna.c*, Quadro 1, foi adicionado ao programa Snort.

Quadro 1: Código fonte rna.c

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <string.h>
#include <stdlib.h>
#include <assert.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <time.h>

#include "snort.h"
#include "detect.h"
#include "plugbase.h"
#include "debug.h"
#include "util.h"
#include "mstring.h"
#include "tag.h"
#include "pcrm.h"
#include "fpcreate.h"
#include "fpdetect.h"
#include "sfthreshold.h"
#include "event_wrapper.h"
#include "event_queue.h"
#include "stream.h"
#include "rna.h"
#include "rna_alg.h"

#ifdef GIDS
#include "inline.h"
#endif /* GIDS */

// extern function definition, it's in define_values
extern int LoadNetworkValues();
```

```
// struct to ip
typedef struct _my_ip{
   int seg_1;
   int seg_2;
   int seg_3;
   int seg_4;
} my_ip;

// transform a ip to a double number
double ip_to_double(struct in_addr ip)
{
   char *point = inet_ntoa(ip);
   int ret[4]={ 0 , 0 , 0 }, i=0;
   double retorno=0.0;
   while( (*point) != '\0')
   {
      if ( (*point) == '.')
      {
         i++;
      }else{
            ret[i]*=10;
            ret[i]+=(*point)-48;
      }
      point++;
   }
   retorno = ( ( (ret[0])*1000.0+ret[1] )*1000.0 +ret[2] )*1000.0
+ret[3];
   return retorno;
}

// function to complete ip address in struct my_ip
my_ip *get_ip_part(struct in_addr ip)
{
  my_ip *ip_part = (my_ip*) malloc(sizeof(my_ip));
  unsigned long ip_comp = inet_addr(inet_ntoa(ip));
  ip_part->seg_1 = (ip_comp)&0xFF;
  ip_part->seg_2 = (ip_comp>>8)&0xFF;
  ip_part->seg_3 = (ip_comp>>16)&0xFF;
  ip_part->seg_4 = (ip_comp>>24)&0xFF;

  //DEBUG printf("%d.%d.%d.%d\n",
  //      ip_part.seg_1,ip_part.seg_2,ip_part.seg_3,ip_part.seg_4);

  return ip_part;
}

// function that answer which protocol is being used
char * which_protocol(int protocolo)
{
   switch(protocolo)
   {
      case IPPROTO_TCP:
         return "tcp";
```

```
         break;

      case IPPROTO_UDP:
         return "udp";
         break;

      case IPPROTO_ICMP:
         return "icmp";
         break;

      case ETHERNET_TYPE_IP:
         return "ip";
         break;

      default:
         return "other";
         break;
   }
}

// write some data in default output
void write_dados(Packet *p,my_ip *ip_origem,my_ip *ip_destino)
{
   printf("Protocol: %s\n",which_protocol(p->iph->ip_proto));
   printf("Source: %d.%d.%d.%d\n",
      ip_origem->seg_1,ip_origem->seg_2,
      ip_origem->seg_3,ip_origem->seg_4);
   printf("Dest: %d.%d.%d.%d\n",
      ip_destino->seg_1,ip_destino->seg_2,
      ip_destino->seg_3,ip_destino->seg_4);
   printf("Ports:\n\tSource: %d\n\tDest: %d\n\tSource(datagram):
%d\n\tDest(datagram): %d\n",
      p->sp,p->dp,p->orig_sp,p->orig_dp);
   printf("CheckSum: %d\n",p->csum_flags);
   printf("PacketFlags: %d\n",p->packet_flags);
   printf("version & header length: %d\n", p->iph->ip_verhl);
   printf("type of service: %d\n",p->iph->ip_tos);
   printf("datagram length: %d\n",p->iph->ip_len);
   printf("identification: %d\n",p->iph->ip_id);
   printf("fragment offset: %d\n",p->iph->ip_off);
   printf("time to live field: %d\n",p->iph->ip_ttl);
   printf("datagram protocol: %d\n",p->iph->ip_proto);
   printf("checksum: %d\n",p->iph->ip_csum);
   if ( !strcmp( which_protocol(p->iph->ip_proto), "tcp" ) )
      printf("TCP th_off is %d, passed len is %% lu\n",
               TCP_OFFSET(p->tcph)/*, (unsigned long)len*/);
   printf("\n");
}

// write data in default output at csv format
void write_csv(Packet *p,my_ip *ip_origem, my_ip *ip_destino)
{
   printf("\"%s\",",which_protocol(p->iph->ip_proto));
```

```
   printf("\"%d.%d.%d.%d\",",
      ip_origem->seg_1,ip_origem->seg_2,
      ip_origem->seg_3,ip_origem->seg_4);
   printf("\"%d.%d.%d.%d\",",
      ip_destino->seg_1,ip_destino->seg_2,
      ip_destino->seg_3,ip_destino->seg_4);
   printf("\"%d\",\"%d\",\"%d\",\"%d\",",
      p->sp,p->dp,p->orig_sp,p->orig_dp);
   printf("\"%d\",",p->csum_flags);
   printf("\"%d\",",p->packet_flags);
   printf("\"%d\",",p->iph->ip_verhl);
   printf("\"%d\",",p->iph->ip_tos);
   printf("\"%d\",",p->iph->ip_len);
   printf("\"%d\",",p->iph->ip_id);
   printf("\"%d\",",p->iph->ip_off);
   printf("\"%d\",",p->iph->ip_ttl);
   printf("\"%d\",",p->iph->ip_proto);
   printf("\"%d\"",p->iph->ip_csum);
}

// write a snns pattern in default output
void write_snns_pattern(float *in, int size, FILE *file)
{
   int i=1;
   fprintf(file,"%d",(int)in[0]);
   while(i<size)
   {
      fprintf(file,"\t%d",(int)in[i++]);
   }
   fprintf(file,"\t0\n");
}

// Call the ANN (RNA) and detect attacks
void Test_with_rna(Packet *p,float *in)
{
   int ret_rna;
   float *out=(float *)malloc(Rna_algREC.NoOfInput*sizeof(float));
   ret_rna = Rna_alg(in, out, 0);
   if (pv.verbose_flag) LogMessage("RNA: %f\n",*out);

   if (pv.rna.type == 4)
   {
       if (*out > pv.rna.answer)
       {
           pv.rna.answer = *out;
           LogMessage("New bigger answer: %f\n",pv.rna.answer);
       }
   }
   else if (*out >= pv.rna.attack)
   {
       // at future, left the user choice the message
      char message[100];
      strcpy(message,"Attack");
```

```
      CallAlertPlugins(p, message, NULL, NULL);
      pc.alert_pkts++;

      LogMessage("\n======== ATTACK ========\n");
      LogMessage("| [%4s] rna(%1.6f) |\n",
                 which_protocol(p->iph->ip_proto),*out);
      LogMessage("\\======================/\n");
   }
}

// control the flux, called by ProcessPacket in main program
void Rna(Packet *p)
{
   my_ip *ip_origem, *ip_destino;

   float *in = (float *)malloc(Rna_algREC.NoOfInput *
sizeof(float));
   struct in_addr ip_src,ip_dst; // They are: u_int32_t

   if (p->iph == NULL)
      return;
      ip_src = p->iph->ip_src;
      ip_dst = p->iph->ip_dst;

//     FOR HELP IN FUTURE IMPLEMENTATION
//     puts(inet_ntoa(ip_src));
//     ip_to_double(ip_src);
//     ip_to_double(ip_dst);
//     inet_netof(ip_src);
//     printf("Ips: %.0lf -> %.0lf %d\n",
//             in[1],in[2],ntohl(inet_addr(inet_ntoa(ip_src))));
//     PrintIPPkt(stdout, p->iph->ip_proto, p);
//     PrintCharData(stdout, (char*) p->data, p->dsize);
//     Print2ndHeader(stdout,p);
//     PrintEthHeader(stdout,p);
//     PrintIPHeader(stdout,p);
//     PrintTCPHeader(stdout,p);
//     PrintUDPHeader(stdout,p);
//     PrintICMPHeader(stdout,p);
//     printf("\tSIZE  %d\n", p->dsize);

        in[0]  = p->iph->ip_proto;

//DEBUG      printf("\ngetting IP src\n");

        ip_origem=get_ip_part(ip_src);
        in[1] = ip_origem->seg_1;
        in[2] = ip_origem->seg_2;
        in[3] = ip_origem->seg_3;
        in[4] = ip_origem->seg_4;

//DEBUG     printf("\ngetting IP dst\n");
        ip_destino=get_ip_part(ip_dst);
```

```
        in[5] = ip_destino->seg_1;
        in[6] = ip_destino->seg_2;
        in[7] = ip_destino->seg_3;
        in[8] = ip_destino->seg_4;

////
        in[9]  = p->iph->ip_ttl;
        in[10] = p->iph->ip_len;
        in[11] = p->iph->ip_id;
        in[12] = p->iph->ip_csum;

   if(p->sp)
   {
      in[13] = p->sp;
      in[14] = p->dp;
   }else{
      in[13] = p->orig_sp;
      in[14] = p->orig_dp;
   }

   switch (pv.rna.type)
   {
       case 1:
       case 4:
            if (!pv.rna.net_loaded) //weights and bias were loaded?
            {
                    /* LoadNetworkValues exit 0 means no error,
                                    another is the error number */
                    if (LoadNetworkValues() == 0)
                    {
                        LogMessage("Neural network loaded.\n");
                        pv.rna.net_loaded = 1;
                    }
                    else
                    {
                        FatalError(
                          "Neural Network couldn't be loaded!\n");
                    }
            }
            Test_with_rna(p,in);
            break;
       case 2:
            write_snns_pattern(in,15,pv.rna.train_file);
            pv.rna.num_patterns++;
            LogMessage("%d patterns colleted...\n",
                        pv.rna.num_patterns);
            break;
   }
}
```

O arquivo *rna.h*, Quadro 2, foi acrescentado ao programa Snort.

Quadro 2: Código fonte rna.h

```
#ifndef __RNA__
#define __RNA__

void Rna(Packet *p);

#endif /*RNA*/
```

O arquivo *rna_train.c* foi adicionado ao programa Snort. Seu código encontra-se no Quadro 3.

Quadro 3: Código fonte rna_train.c

```
#include "snort.h"
#include "rna_train.h"

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
FROM: SNNSv4.2/tools/sources/RCS/netlearn.c
Copyright (c) 1990-1995  SNNS Group, IPVR, Univ. Stuttgart, FRG

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void errChk( int err_code )
{
  if (err_code != KRERR_NO_ERROR)  {
    printf( "%s\n", krui_error( err_code ));
    exit( 1 );
  }
}

int Rna_train()
{
  int   ret_code, N, i, j, no_of_sites, no_of_links, no_of_units,
     no_of_patterns, dummy, NoOfReturnVals, no_of_input_params,
        no_of_output_params, step, set_no;
  char     *netname;
  bool     shuffle;
  float learn_parameters[5], updateParameterArray[5],
     parameterInArray[5], sum_error;
  float *return_values;
  int spIsize[5], spOsize[5], spIstep[5], spOstep[5];


  printf( "\n%s\n", krui_getVersion() );
  printf( "----  Network Learning -----\n" );
  printf( "Loading the network %s ...\n",pv.rna.net_filename );
  ret_code = krui_loadNet( pv.rna.net_filename, &netname );
  errChk( ret_code );
  krui_getNetInfo( &no_of_sites, &no_of_links, &dummy, &dummy );
  no_of_units = krui_getNoOfUnits();
```

```
 printf( "Network name: %s\n", netname );
 printf( "No. of units       : %d\n", no_of_units );
 printf( "No. of input units : %d\n",
     krui_getNoOfTTypeUnits( INPUT ) );
 printf( "No. of output units: %d\n",
     krui_getNoOfTTypeUnits( OUTPUT ) );
 printf( "No. of sites: %d\n", no_of_sites );
 printf( "No. of links: %d\n\n", no_of_links );
 printf( "Learning function: %s\n", krui_getLearnFunc() );
 printf( "Update function  : %s\n", krui_getUpdateFunc() );

 printf( "Loading the patterns %s ...\n",pv.rna.train_filename );
 ret_code=krui_loadNewPatterns( pv.rna.train_filename, &set_no );
 errChk( ret_code );
 no_of_patterns = krui_getNoOfPatterns();
 printf( "No. of patterns: %d\n", no_of_patterns );

 /*  determine the no. of parameters
     of the current learning function
 */
 (void) krui_getFuncParamInfo(
           krui_getLearnFunc(), LEARN_FUNC,
           &no_of_input_params, &no_of_output_params );

 LogMessage(
     "\nThe learning function '%s' needs %d input parameters:\n",
     krui_getLearnFunc(), no_of_input_params );

 for (i = 0; i < no_of_input_params; i++)
 {
     learn_parameters[i] = pv.rna.parameters[i];
     LogMessage( "\tParameter [%d]: %f\n",
                 i + 1 ,learn_parameters[i] );
 }

 N = (pv.rna.train_cycles);
 LogMessage("Cycles: %d\n",N);

 if (N <= 0)  FatalError( "\nInvalid no. of cycles !\n" );

 shuffle=0;

 printf( "\nBegin learning ...\n" );

 step = ((N - 1) / 20) + 1;

 for(j=0; j<5; j++) {
       spIsize[j] = 0; spIstep[j] = 0;
       spOsize[j] = 0; spOstep[j] = 0;
 }

errChk (
        krui_DefTrainSubPat(
```

```
                              spIsize, spOsize,
                              spIstep, spOstep, &dummy )
   );

    ret_code = krui_learnAllPatterns( learn_parameters,
no_of_input_params , &return_values, &NoOfReturnVals );
    errChk( ret_code );


  /*  save the network     */
  printf( "Saving the network: %s...\n",pv.rna.net_filename );
  ret_code = krui_saveNet( pv.rna.net_filename, netname );
  errChk( ret_code );

  /*  before exiting: delete network  */
  krui_deleteNet();
  printf("Done!\n");
  return( 0 );
}
```

O arquivo *rna_train.h*, Quadro 4, foi adicionado ao programa Snort.

Quadro 4: Código fonte rna_train.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*  SNNS-Kernel constants and data type definitions  */
#include "glob_typ.h"
/*  SNNS-Kernel User-Interface Function Prototypes   */
#include "kr_ui.h"

extern void errChk( err_code );
extern int Rna_train();
```

No Quadro 5, foi demonstrada somente a estrutura do arquivo *rna_alg.c*, as linhas não exibidas foram substituídas por "..." . Para gerar este arquivo deve ser feita uma rede neural no JavaNNS, após deve-se converter o arquivo com o programa modificado *snns2c*.

Quadro 5: Exemplo de código fonte rna_alg.c

```
#include <math.h>
#include "rna_alg.h"
#define Act_Logistic(sum, bias)  ( (sum+bias<10000.0) ? ( 1.0/(1.0
+ exp(-sum-bias) ) ) : 0.0 )
```

```
#define Act_Identity(sum, bias)     ( sum )
#define NULL (void *)0

  /* Forward Declaration for all unit types */
  UnitType Units[77];

  /* Sources definition section */
  pUnit Sources[] =  {
  ...
  }

  /* Weigths definition section */
  float Weights[] =  {
  ...
  }

  /* unit definition section (see also UnitType) */
  UnitType Units[77] = {
  ...
  }

  int Rna_alg(float *in, float *out, int init)
  {
  ...
  }
```

O arquivo *rna_alg.h*, Quadro 6, foi adicionado ao programa Snort.

Quadro 6: Código fonte rna_alg.h

```
extern int Rna_alg(float *in, float *out, int init);

static struct {
  int NoOfInput;    /* Number of Input Units  */
  int NoOfOutput;   /* Number of Output Units */
  int(* propFunc)(float *, float*, int);
} Rna_algREC = {15,1,Rna_alg};

typedef struct UT {
          float act;         /* Activation       */
          float Bias;        /* Bias of the Unit */
          int   NoOfSources; /* Number of predecessor units */
   struct UT   **sources; /* predecessor units */
          float *weights; /* weights from predecessor units */
        } UnitType, *pUnit;

extern UnitType Units[];
extern float Weights[];
```

O arquivo *define_values.c*, Quadro 7, foi incluído ao programa Snort.

Quadro 7: Código fonte define_values.c

```
#include "define_values.h"
#include "snort.h"

void freeUnits(pUnit_val Units_val)
{
      pUnit_val unit = Units_val;

      while(unit->number) {  /* e.g. non-empty unit */
            if (unit->members) killList(unit->members);/* make sure,
that there is */
            if (unit->sources) killList(unit->sources); /* memory to
release        */
            if (unit->weights) free(unit->weights);
            if (unit->name) free(unit->name);
            if (unit->dest) free(unit->dest);

            if(unit->FeatureWidth) {
                  int i;
                  for(i = 0; i < unit->FeatureWidth; i++) {
                        if(unit->TDNNsources[i]) free(unit-
>TDNNsources[i]);
                        if(unit->TDNNweights[i]) free(unit-
>TDNNweights[i]);
                  }
                  if(unit->TDNNsources) free(unit->TDNNsources);
                  if(unit->TDNNweights) free(unit->TDNNweights);
            }
            unit++;
      }

      free(Units_val);
}


void freeLayers(pLayer Layers)
{
      pLayer layer = Layers;

      while(layer->members) {/* e.g. non-empty layer */
            if (layer->members) free(layer->members); /* make sure,
that there is */
            if (layer->sources) free(layer->sources); /* memory to
release        */
            if (layer->name) free(layer->name);
            if (layer->readCounter) free(layer->readCounter);
            if (layer->writeCounter) free(layer->writeCounter);

            layer++;
      }

      free(Layers);
```

```
}

int checkLearnFunc(void)
{
  static char *NotSupportedLearnFuncs[] = {
    "ART1", "ART2", "ARTMAP", "BackPercolation", "Hebbian",
"RM_delta",
    "RCC", "Kohonen", NULL
  };

  char *LearnFunc = krui_getLearnFunc();     /* learning function
of the network */
  char **string   = NotSupportedLearnFuncs;  /* current function
name to test */

  while(*string) {
    if (!strcmp(*string, LearnFunc)) {          /* e.g. the same
function-name */
      return(NOT_SUPPORTED);
    }
    string++;
  }

  return(OK);
}

void checkErr(int errCode)
{
  switch(errCode)
      {
      case OK          : ;
     break;
      case ERR         : printf("unspecified Error\n");
     break;
      case CANT_ADD    :
      case MEM_ERR     : printf("not enough memory\n");
     break;
      case CANT_LOAD   : printf("can't load file\n");
     break;
      case WRONG_PARAM : printf("wrong parameters\n");
     break;
      case CANT_OPEN   : printf("can't open file\n");
     break;
      case NO_CPN      : printf("net is not a CounterPropagation
network\n");
     break;
      case NO_TDNN     : printf("net is not a Time Delay Neural
Network\n");
     break;
      case ILLEGAL_CYCLES : printf("net contains illegal
cycles\n");
     break;
      case WRONG_ACT_FUNC : ;
```

```
      break;
       case NOT_SUPPORTED  : printf("not supported network type\n");
      break;
       default             : printf("unknown error code : %d\n",
errCode);
       }
}

bool is_BPTT_net(void)
{
  return (   (0 == strcmp("BPTT", krui_getLearnFunc() ))
       || (0 == strcmp("BBPTT", krui_getLearnFunc() ))
       || (0 == strcmp("QPTT", krui_getLearnFunc() )) );
}

int checkActFunc(char *actName)
{
  int i=0;

  while (**(ACT_FUNC_NAMES + i) ) {
    if (!strcmp(ACT_FUNC_NAMES[i], actName) ) return (i);
    i++;
  }
  fprintf(stderr, "Can't find the function <%s>\n", actName);
  return(-1);
}

int matchLayer(pLayer layer, pUnit_val unit)
{
  static int is_BPTT = 0, first_time = 1;

  /* a special flag is set to avoid unneeded function calls */
  if (first_time) {
    is_BPTT = is_BPTT_net();
    first_time = 0;
  }

  /* input neurons are all treated the same way */
  if ( (unit->type == INPUT) && (layer->type == INPUT) ) return
(TRUE);

  /* unit should match the attributes of the Layer */
  if (unit->type != layer->type) return(FALSE);
  if (unit->ActFunc != layer->ActFunc) return(FALSE);

  /* BPTT-nets have no topological order */
  if (is_BPTT) return (TRUE);

  /* unit must not be a member of the source units */
  if (isMember(layer->sources, unit->number) ) return (FALSE);

  /* Neue Version von Matthias Oderdorfer */
  return ( CompareSources(unit->sources, layer->sources) );
```

```
}

int initLayer(pLayer layer, pUnit_val unit)
{
  layer->members = newList();           /* a list for member unit */
  if (!layer->members) return(MEM_ERR);

  layer->sources = newList();   /* a list for all predecessor */
  if (!layer->sources) return(MEM_ERR); /* units of all members  */

  addList(layer->members, unit->number); /* prototype unit is the
first member */
  if (copyList(layer->sources, unit->sources) ) return(MEM_ERR);

  layer->ActFunc = unit->ActFunc;
  layer->type    = unit->type;

  unit->layer = layer;

  return(OK);
}

int searchLayer(pUnit_val unit, pLayer globalLayers)
{
  pLayer layer;

  layer = globalLayers;
  while(TRUE) {
    if (layer->members == NULL) {        /* empty layer found */
      return(initLayer(layer, unit));   /* give possible Errors to
caller */
    }
    else if (matchLayer(layer, unit) ) { /* matching layer found */
      if (addList(layer->members, unit->number)) {
     return(MEM_ERR);
      }
      unit->layer = layer;
      return(mergeList(layer->sources, unit->sources));
      /* returns Status */
    }
    layer++;
  }
}

int divideNet( pUnit_val globalUnits,
               pLayer globalLayers, int *TDNN_prot )
{
  int     unitNo, sourceNo;/* number of the unit and source unit */
  pUnit_val     unit;              /* unit and prototype unit */
  FlintType dummy, weight;         /* link weights */
  int       error;                 /* error code    */
  char      *string;               /* free variable */
```

```
  int       pos;              /* free variable */

  unitNo = krui_getFirstUnit();
  unit   = globalUnits;

  while (unitNo) {
    unit->members = newList();
    if (!unit->members) return (MEM_ERR);

    unit->number  = unitNo;
    addList(unit->members, unitNo); /* the Prototype is also part
of the member list */

    /* copy the entries from SNNS to the own format */
    unit->act     = krui_getUnitActivation(unitNo);
    unit->type    = krui_getUnitTType(unitNo);
    unit->Bias    = krui_getUnitBias(unitNo);

    /* units always have a name (at least its old number) */
    string = krui_getUnitName(unitNo);
    if (NULL == string) {
      unit->name = malloc(12 * sizeof(char));
      if(! unit->name) {
     return (MEM_ERR);
      }
      sprintf(unit->name, "Old: %d", unit->number);
    }
    else {
      unit->name = malloc(MAX(1,strlen(string)+1));
      if(! unit->name) {
     return (MEM_ERR);
      }
      strcpy(unit->name, string);
    }

    unit->ActFunc = checkActFunc(krui_getUnitActFuncName(unitNo) );
    if (unit->ActFunc < 0) return(WRONG_ACT_FUNC);

    /* insert all Source units in the list */
    unit->sources = newList();
    if (!unit->sources) return (MEM_ERR);
    sourceNo = krui_getFirstPredUnit(&dummy);
    while (sourceNo) {
      /* only special-hidden-neurons may have links to itself */
      if ( (unit->type != SPECIAL_H) && !is_BPTT_net() ) {
        if (unit->number == sourceNo) return(ILLEGAL_CYCLES);
      }
      if(addList(unit->sources, sourceNo)) return (MEM_ERR);
      sourceNo = krui_getNextPredUnit(&dummy);
    }
    /* now the weights can be written in the right order */
    /* One more Element is allocated, because the array might have
size 0 */
```

```
    unit->weights = (float *)malloc(NoOf(unit->sources) *
sizeof(float) + sizeof(float));
    if (!unit->weights) return(MEM_ERR);
    sourceNo = krui_getFirstPredUnit(&weight);
    while (sourceNo) {
      pos = searchList(unit->sources, sourceNo);
      unit->weights[pos] = weight;
      sourceNo = krui_getNextPredUnit(&weight);
    }
#ifdef DEBUG
    printUnit(unit);
#endif
    error = searchLayer(unit, globalLayers);
           if (error) return(error);

    unit++;
    unitNo = krui_getNextUnit();
  }

  return(OK);
}

#ifdef DEBUG
/*function for DEBUG*/
void printLayer(pLayer layer)
{
  int i;
  printf("\nLayer %d", layer->number);
  printf("\nmembers: ");
  for (i = 0; i < NoOf(layer->members); i++) {
    printf("%d ", element(layer->members, i) );
  }
  printf("\nsources: ");
  for (i = 0; i < NoOf(layer->sources); i++) {
    printf("%d ", element(layer->sources, i) );
  }
  printf("\n");
}
#endif

signed char checkOrder(pLayer globalLayers, int x, int y)
{
  /* preference of the unit type : low value means early update */
  static char pref[12] = {0, 0, 2, 0, 1, 3, 3, 3, 3, 3, 3, 3};

  signed char order = 0;

  if (pref[globalLayers[x].type] < pref[globalLayers[y].type]) {
    order = (signed char)-1;    /* e.g. layer x before layer y */
  }
  else if (pref[globalLayers[x].type] > pref[globalLayers[y].type])
{
    order = 1;             /* e.g. layer x after layer y */
```

```
  }

     /* BPTT-Networks may contain any cycles so they must not be
checked */
     if (is_BPTT_net() ) {
          return(order);
     }

  if (haveIntersection(globalLayers[x].sources,
globalLayers[y].members) ) {
    globalLayers[y].SuccDelay = globalLayers[x].delay; /* Side-
Effect */
    if (order == -1) {
      if (SPECIAL_H != globalLayers[y].type) return
(ILLEGAL_CYCLES);
    }
    else {
      order = 1;
    }
  }
  if (haveIntersection
        (globalLayers[x].members, globalLayers[y].sources)
     ) {
    globalLayers[x].SuccDelay = globalLayers[y].delay; /* Side-
Effect */
    if (order == 1) {
      if (SPECIAL_H != globalLayers[x].type) return
(ILLEGAL_CYCLES);
    }
    else {
      order = (signed char)-1;
    }
  }

  return(order);
}


int sortNet(pLayer globalLayers, int NoOfLayers, int *order)
{
  char **matrix;      /* precedence matrix        */
  char  *mask;             /* already chosen layers    */
  int    i, j, x, y, ord, isSource = TRUE;
  char   precedence;

  /* reserve memory for all the arrays and matrices */
  matrix = (char **)malloc(NoOfLayers * sizeof(char *) );
  if (!matrix) return (MEM_ERR);

  for (i = 0; i < NoOfLayers; i++) {
    matrix[i] = (char *)calloc(NoOfLayers, sizeof(char) );
    if (!matrix[i]) return (MEM_ERR);
  }
```

```
  mask  = (char *)calloc(NoOfLayers, sizeof(char) );
  if (!mask) {
          free(matrix);
          return (MEM_ERR);
     }

  /** build the precedence matrix of the Layer-Graph */
  for (y = 0; y < NoOfLayers; y++) {
    for (x = y + 1; x < NoOfLayers; x++) {
      precedence = checkOrder(globalLayers, x, y);
      if (precedence == ILLEGAL_CYCLES) return(ILLEGAL_CYCLES);

      matrix[x][y] = precedence;
      matrix[y][x] = -precedence;

    } /* for x */

  } /* for y */

#ifdef DEBUG
  printf("\nPrecedence Matrix is:\n");
  for (y = 0; y < NoOfLayers; y++) {
    for (x = 0; x < NoOfLayers; x++) {
      printf("%3d ", matrix[y][x]);
    }
    printf("\n");
  }
#endif

  /** put the Layers in the right order **/
  for (ord = 0; ord < NoOfLayers; ord++) {

    for (i = 0; i < NoOfLayers; i++) {
      if (mask[i]) continue;    /* Layer already chosen */

      isSource = TRUE;
      for (j = 0; j < NoOfLayers; j++) {
        /* exists a layer wich must be updated before ? */
        if (matrix[i][j] == 1) {
          isSource = FALSE;
          break;
        }
      }

      if (isSource) {
        order[ord] = i;    /* the number of the Layer becomes ord */
        mask[i] = 1;       /* must not test this Layer again      */

        for (j = 0; j < NoOfLayers; j++) {
          matrix[j][i] = 0;/* clear depencies for other Layers */
        }
```

```
        break;              /* find next Layer */
      }

    } /* for i */

    if (!isSource) {
      return(ILLEGAL_CYCLES);
    }

  } /* for ord */

  for (i = 0; i < NoOfLayers; i++) {
    free(matrix[i]);
  }
  free(matrix);
  free(mask);

#ifdef DEBUG
  printf("\nLayers sorted in following order :\n");
  for (i = 0; i < NoOfLayers; i++) {
    printf(" %d", order[i]);
  }
  printf("\n");
#endif

  return(OK);
}

int NameLayers(pLayer globalLayers, int NoOfLayers)
{
#define NAME_LENGTH 11

  int  nr;
  int  hcounter = 0, ocounter = 0, scounter =0;
  pLayer layer = NULL;

  for(nr = 0; nr < NoOfLayers; nr++) {
    layer = (globalLayers + nr);
    layer->name = (char *)malloc(NAME_LENGTH * sizeof(char) );
    if (!(layer->name)) return(MEM_ERR);

    switch( (globalLayers + nr)->type) {
    case INPUT  : sprintf(layer->name, "Input");
      break;

    case OUTPUT : sprintf(layer->name, "Output%d", ++ocounter);
      break;

    case HIDDEN : sprintf(layer->name, "Hidden%d", ++hcounter);
      break;

    default     : sprintf(layer->name, "Special%d", ++scounter);
    }
```

```
  }

  return(OK);
}

void LoadAllWeights(pUnit_val Units_val, int NoOfUnits)
 {
   int nr, i ;
   unsigned int pos=0;
   for(nr = 0; nr < NoOfUnits; nr++)
   {
      for(i = 0; i < NoOf((Units_val + nr)->sources); i++)
      {
         Weights[pos] = (Units_val + nr)->weights[i];
#ifdef DEBUG
         printf("loading Weights[%d]=%f\n",pos,Weights[pos]);
#endif
         pos++;
      }
   }

 }

void writeUnitNew(pUnit_val unit)
 {
   Units[unit->number].Bias=unit->Bias;
#ifdef DEBUG
   printf("Units[%d].Bias=%f\n",(unit->number),Units[unit-
>number].Bias);
#endif
 }


void LoadAllUnits(pUnit_val Units_val, int NoOfUnits)
{
   int nr;
   int count_links;

   count_links = 0; /* no links yet */
   for(nr = 0; nr < NoOfUnits; nr++)
   {
      writeUnitNew(Units_val + nr);
      count_links += NoOf((Units_val+nr)->sources);
   }
}

int LoadValues(pLayer globalLayers, pUnit_val globalUnits,
             int NoOfLayers, int NoOfUnits,
             int *order, char *ProcName)
{
  pLayer layer;
  pUnit_val  unit;
```

```
  int    nr, maxSource = 0, maxFeature = 0;
  pList  OutList, FunctionList;
  time_t timer;

  /* calculate time for the date in the header of the output file
*/
  time(&timer);

  /* prepare Output Stream */
  /* maxSource is needed for writing the Unit-array */
  for(unit = globalUnits; unit < globalUnits + NoOfUnits; unit++) {
    maxSource  = MAX(maxSource, NoOf(unit->sources) );
    /* Calculating the Maximum Feature-Width is harmless for non-
TDNNs, dont need.. but will be left */
    maxFeature = MAX(maxFeature, unit->FeatureWidth);
  }

  /* Net-Output may consist of several (output) Layers     */
  /* so there must be an extra merge-list : OutList        */
  /* FunctionList will contain a set of all used functions */
  OutList = newList();      if (!OutList) return (MEM_ERR);
  FunctionList = newList(); if (!FunctionList) return (MEM_ERR);

  for(nr = 0; nr < NoOfLayers; nr ++) {
    layer = globalLayers + nr;
    if( layer->type == OUTPUT) {
      if (mergeList(OutList, layer->members) ) return (MEM_ERR);
    }
    if (addList(FunctionList, layer->ActFunc) ) return(MEM_ERR);
  }

  /* find Names for the Layers */
  if (NameLayers(globalLayers, NoOfLayers)) return(MEM_ERR);

   LoadAllWeights(globalUnits, NoOfUnits);

   LoadAllUnits(globalUnits, NoOfUnits);

   killList(OutList);
   killList(FunctionList);
   return(OK);
}

/*
main function
*/
int LoadNetworkValues()
{
   krui_err err;                /* error code of SNNS - krui */
   char     *netname;                /* internal name of the SNNS-
network */
   char     *NetFileName = {pv.rna.net_filename}; /* input file */
   pUnit_val    Units_val;                /* all Units and unit
```

```
variable */
   pLayer   Layers;                  /* all Layers and layer
variable */
   int      NoOfUnits, NoOfLayers;        /* Number of units and
layers */
   int      *order;                  /* array with the order of the
sorted layers */
   int      nr;                      /* help variables */
   int      error;                   /* error code */
   char     *ProcName = {"Rna_alg"};      /* function name in the
output */
   int      *TDNN_prot;                      /* Array with the
numbers of the prototype units */

#ifdef DEBUG
   printf("Importing values from %s...\n",NetFileName);
#endif

   printf("Loading Artificial Network...\n");

   /* load Net */
   err = krui_loadNet(NetFileName, &netname);
   if (err) {
      fprintf(stderr, "%s\n", krui_error(err) );
      return(CANT_LOAD);
   }

   error = checkLearnFunc();
   if (error) {
      checkErr(error);
      return(error);
   }

   NoOfUnits = krui_getNoOfUnits();

   Units_val  =  (pUnit_val)calloc((NoOfUnits + 1),
sizeof(tUnit) );  /* because of sentinels */
   if (! Units_val) {
            checkErr(MEM_ERR);
            return(MEM_ERR);
   }

   Layers = (pLayer)calloc((NoOfUnits + 1), sizeof(tLayer) ); /*
because of sentinels */
   if (! Layers) {
            free(Units_val);
            checkErr(MEM_ERR);
            return(MEM_ERR);
   }

   /* TDNN_prot needs one more Element, because the enumeration of
the units starts with one */
   TDNN_prot = (int *)malloc((NoOfUnits+1) * sizeof(int) );
```

```
   if (! TDNN_prot) {
                  free(Units_val);
                  free(Layers);
                  checkErr(MEM_ERR);
                  return(MEM_ERR);
   }

   for (nr = 0; nr <= NoOfUnits; nr++) {
      Layers[nr].number  = nr;
   }

#ifdef DEBUG
   printf("Dividing net into Layers...\n");
#endif

   /* part Net into groups */
   error =  divideNet(Units_val, Layers, TDNN_prot);
   if (error) {
            checkErr(error);
            FREE_ALL;
            return(error);
   }

   /* count the Non-empty Layers */
   for (nr = 0; Layers[nr].members != NULL; nr++);
   NoOfLayers = nr;
   order = (int *)malloc(NoOfLayers * sizeof(int) );


   /* count the real number of units (e.g. the prototype units in
TDNN) */
   /* unused units have the number 0 the total Number can't exeed
the   */
   /* Number of Units given by the SNNS-Interface-Function
*/
   for(nr = 0; (Units_val[nr].number != 0) && (nr < NoOfUnits); nr+
+) {
      Units_val[nr].index = nr;
   }
   NoOfUnits = nr;

#ifdef DEBUG
   for(nr = 0; nr < NoOfLayers; printLayer(Layers + nr++) );
#endif

   /* topological sort of the layers */
#ifdef DEBUG
   printf("Sorting Layers...\n");
#endif

   error = sortNet(Layers, NoOfLayers, order);
   if (error) {
      checkErr(error);
```

```
      FREE_ALL;
      return(error);
   }

#ifdef DEBUG
   printf("Loading Values...\n");
#endif

   /* Load values */
   error = LoadValues(Layers, Units_val, NoOfLayers, NoOfUnits,
order, ProcName);
   if (error) {
      checkErr(error);
      FREE_ALL;
      return(error);
   }

   FREE_ALL;
   return(0);
}
```

O arquivo *define_values.h*, Quadro 8, foi adicionado ao programa Snort.

Quadro 8: Código fonte define_values.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <time.h>
#include <memory.h>
#include "glob_typ.h"
#include "kr_ui.h"
#include "rna_functions.h"
#include "rna_templates.h"
#include "rna_lib.h"

#include "kr_typ.h"
#include "kernel.h"

#include "rna_alg.h"
// gcc -Wall -c define_values.c -O4  -I./kernel_snns
// gcc define_values.o rna_lib.o ./kernel_snns/libkernel.a
./kernel_snns/libfunc.a -lm -ll -Wall -o define_values
/* Macros for calculating the minimum or maximum of two values */
#define MAX(a, b) ( (a > b) ? (a) : (b) )
#define MIN(a, b) ( (a < b) ? (a) : (b) )

/* Macro for releasing memory of units and layers */
#define FREE_ALL freeUnits(Units_val); freeLayers(Layers);
free(TDNN_prot);
```

```
/* Status (Error) Codes : OK = 0 (NO Error), ERR = 1, ...  */
typedef enum { OK, ERR, CANT_ADD, CANT_LOAD, MEM_ERR,
               WRONG_PARAM, WRONG_ACT_FUNC, CANT_OPEN,
               ILLEGAL_CYCLES, NO_CPN, NO_TDNN, NOT_SUPPORTED}
Status;

/* Recordtype for Layers */
typedef struct {
  int   number;            /* of the Layer (not used yet) */
  pList members;      /* Numbers of all member-units */
  pList sources;      /* numbers of all sources of the member-units
*/
  int   type;                /* INPUT , OUTPUT ...                */
  int   ActFunc;      /* No in the ActivationFunctionList */
  char  *name;             /* Name of the Layer */
  /* Special entries for TDNN */
  int   TotalDelay;        /* Total Delay Number of the Layer */
  int   delay;             /* Delay of the receptive Field  */
  int   SuccDelay;         /* Delay of the following Layer */
  char  *readCounter;      /* Name of the DelayPointer      */
  char  *writeCounter;          /* Name of the WriteDelayPointer */
} tLayer, *pLayer;

/* Recordtype for Units */
typedef struct {
  int    number;      /* of the Unit                          */
  pList  members;          /* Units with the same Prototype (TDNN)*/
  pList  sources;          /* numbers of the source-Units         */
  float  *weights;           /* Link-Weights to the Source-Units
*/
  int    ActFunc;          /* No in the ActivationFunctionList    */
  int    type;             /* INPUT , OUTPUT ...                  */
  char   *name;            /* Name of the unit, given by the user */
  float  Bias;             /* Bias of the unit                    */
  pLayer layer;            /* Pointer to the layer wich contains the
unit */
  /* Special entries for BPTT and Elman/Jordan */
  float  act;                   /* Initial Activation of the Unit
*/
  /* Special entries for CPN */
  float *dest;      /* weights from Hidden to Output written in the
hidden units */
  int   NoOfDest;   /* Numbers of the weights to the Output */
  /* Special entries for TDNN */
  int    index;            /* Index in the global Array           */
  int   FeatureWidth;      /* Number of Prototype Source units    */
  int   DelayLength;       /* Delay Length of the receptive Field */
  int   **TDNNsources;          /* special Format for TDNNs
*/
  float **TDNNweights;          /* special Format for TDNNs
*/
} tUnit, *pUnit_val;
```

```
void freeUnits(pUnit_val Units_val);
void freeLayers(pLayer Layers);
int checkLearnFunc(void);
void checkErr(int errCode);
int checkActFunc(char *actName);
int matchLayer(pLayer layer, pUnit_val unit);
int initLayer(pLayer layer, pUnit_val unit);
int searchLayer(pUnit_val unit, pLayer globalLayers);
int divideNet(pUnit_val globalUnits, pLayer globalLayers, int
*TDNN_prot);
void printLayer(pLayer layer);
int sortNet(pLayer globalLayers, int NoOfLayers, int *order);
int NameLayers(pLayer globalLayers, int NoOfLayers);
void LoadAllWeights(pUnit_val Units_val, int NoOfUnits);
void writeUnitNew(pUnit_val unit);
void LoadAllUnits(pUnit_val Units_val, int NoOfUnits);
int LoadNetworkValues();
```

No código fonte principal, *snort.c*, foram adicionadas inclusões dos cabeçalhos *rna.h*, *rna_alg.h* e as novas funções implementadas (Quadro 9).

Quadro 9: Funções implementadas em snort.c

```
#include "rna.h"
#include "rna_alg.h"

/* RNA locale functions */
static void rna_write_pat_header();
static void rna_finalize_pat_file();
static void rna_load_config();

// write the pat header in pv.rna.train_file stream
static void rna_write_pat_header()
{
    time_t t;
    time(&t);
    fprintf(pv.rna.train_file,
        "SNNS pattern definition file V1.4\n");
    fprintf(pv.rna.train_file,"generated at %s\n\n", ctime( &t));
    fprintf(pv.rna.train_file,"No. of patterns     : ");
    fgetpos(pv.rna.train_file,&(pv.rna.num_pat));
    fprintf(pv.rna.train_file,"                                \n");
    fprintf(pv.rna.train_file,"No. of input units  : %d\n",
        Rna_algREC.NoOfInput);
    fprintf(pv.rna.train_file,"No. of output units : %d\n",
        Rna_algREC.NoOfOutput);
    fprintf(pv.rna.train_file,"\n");
}

// finalize the pat file
```

```
static void rna_finalize_pat_file()
{
    fsetpos(pv.rna.train_file,&(pv.rna.num_pat));
    fprintf(pv.rna.train_file,"%d",pv.rna.num_patterns);
    fclose(pv.rna.train_file);
}

// used by rna_load_config
void read_float_rna(FILE *arq, float *where, char opt[50])
{
    fscanf(arq,"%f",where);
    opt[0]='\0';
}

// used by rna_load_config
void read_int_rna(FILE *arq, int *where, char opt[50])
{
    fscanf(arq,"%d",where);
    opt[0]='\0';
}

// reads the ann config file
static void rna_load_config(FILE *arq)
{
    char car;
    char opt[50];
    if (arq==NULL)
       ErrorMessage("Couldn't open the ANN config file!\n");

    opt[0]='\0';
    do
    {
        car = fgetc(arq);
        if (car == '=')
        {
            if (strncasecmp(opt,"attack",6)==0){
                read_float_rna(arq,&(pv.rna.attack),opt);

            }else if (strncasecmp(opt,"parameter_0",11)==0){
                read_float_rna(arq,&(pv.rna.parameters[0]),opt);

            }else if (strncasecmp(opt,"parameter_1",11)==0){
                read_float_rna(arq,&(pv.rna.parameters[1]),opt);

            }else if (strncasecmp(opt,"parameter_2",11)==0){
                read_float_rna(arq,&(pv.rna.parameters[2]),opt);

            }else if (strncasecmp(opt,"parameter_3",11)==0){
                read_float_rna(arq,&(pv.rna.parameters[3]),opt);

            }else if (strncasecmp(opt,"parameter_4",11)==0){
                read_float_rna(arq,&(pv.rna.parameters[4]),opt);
```

```
            }else if (strncasecmp(opt,"cycles",11)==0){
                read_int_rna(arq,&(pv.rna.train_cycles),opt);

            }

        }else if (car == '\n')
        {
            opt[0]='\0';
        }
        else{
            opt[strlen(opt)+1]='\0';
            opt[strlen(opt)]=car;
        }
    }while(!feof(arq));
    fclose(arq);

    if (pv.verbose_flag){
        if (pv.rna.type==1 || pv.rna.type==4)
        {
            LogMessage("RNA Attack: %f\n\n",pv.rna.attack);
        }else if (pv.rna.type==2 || pv.rna.type==3)
        {
            LogMessage(
               "Reading Parameter[0]=%f\n",pv.rna.parameters[0] );
            LogMessage(
               "Reading Parameter[1]=%f\n",pv.rna.parameters[1] );
            LogMessage(
               "Reading Parameter[2]=%f\n",pv.rna.parameters[2] );
            LogMessage(
               "Reading Parameter[3]=%f\n",pv.rna.parameters[3] );
            LogMessage(
               "Reading Parameter[4]=%f\n",pv.rna.parameters[4] );
            LogMessage("\n");
        }
    }
}
```

O Quadro 10 apresenta o código das funções que foram modificadas no código fonte principal *snort.c*.

Quadro 10: Funções modificadas em snort.c

```
int SnortMain(int argc, char *argv[])
{
#ifndef WIN32
    #if defined(LINUX) || defined(FREEBSD) || defined(OPENBSD) ||
defined(SOLARIS)
        sigset_t set;

        sigemptyset(&set);
        sigprocmask(SIG_SETMASK, &set, NULL);
```

```
    #else
        sigsetmask(0);
    #endif
#endif  /* !WIN32 */

    /*    malloc_options = "AX";*/

    /* Make this prog behave nicely when signals come along.
     * Windows doesn't like all of these signals, and will
     * set errno for some.  Ignore/reset this error so it
     * doesn't interfere with later checks of errno value.
     */
    signal(SIGTERM, SigTermHandler);    if(errno!=0) errno=0;
    signal(SIGINT, SigIntHandler);      if(errno!=0) errno=0;
    signal(SIGQUIT, SigQuitHandler);    if(errno!=0) errno=0;
    signal(SIGHUP, SigHupHandler);      if(errno!=0) errno=0;
    signal(SIGUSR1, SigUsr1Handler);    if(errno!=0) errno=0;

    /*
     * set a global ptr to the program name so other functions can
tell what
     * the program name is
     */
    progname = argv[0];
    progargs = argv;

#ifdef WIN32
    if (!init_winsock())
        FatalError("Could not Initialize Winsock!\n");
#endif

    memset(&pv, 0, sizeof(PV));

    /*
     * setup some lookup data structs
     */
    InitNetmasks();
    InitProtoNames();

    /*
    **  This intializes the detection engine for later
configuration
    */
    /* TODO: only do this when we know we are going into IDS mode
*/
    fpInitDetectionEngine();

    /*for rna*/
    pv.rna.type = 0;
    pv.rna.train_cycles=1000;
    pv.rna.net_loaded = 0;
    pv.rna.attack=-9999.0;
    pv.rna.answer=-999.0;
```

```
    pv.rna.parameters[0]=0.2;
    pv.rna.parameters[1]=0.1;
    pv.rna.parameters[2]=0.5;
    pv.rna.parameters[3]=0.0;
    pv.rna.parameters[4]=0.0;
    pv.rna.net_filename = NULL;
    pv.rna.train_filename = NULL;
    pv.rna.train_file = NULL;

    /* initialize the packet counter to loop forever */
    pv.pkt_cnt = -1;

    /* set the alert filename to NULL */
    pv.alert_filename = NULL;

    /* set the default alert mode */
    pv.alert_mode = ALERT_FULL;

    /* set the default assurance mode (used with stream 4) */
    pv.assurance_mode = ASSURE_ALL;

    pv.use_utc = 0;

    pv.log_mode = 0;

    /*
     * provide (limited) status messages by default
     */
    pv.quiet_flag = 0;

    InitDecoderFlags();

    /* turn on checksum verification by default */
    pv.checksums_mode = DO_IP_CHECKSUMS | DO_TCP_CHECKSUMS |
                        DO_UDP_CHECKSUMS | DO_ICMP_CHECKSUMS;

#if defined(WIN32) && defined(ENABLE_WIN32_SERVICE)
    /* initialize flags which control the Win32 service */
    pv.terminate_service_flag = 0;
    pv.pause_service_flag = 0;
#endif  /* WIN32 && ENABLE_WIN32_SERVICE */

    /* chew up the command line */
    ParseCmdLine(argc, argv);

    /* If we are running non-root, install a dummy handler instead.
*/
    if (userid != 0)
        signal(SIGHUP, SigCantHupHandler);

    /* determine what run mode we are going to be in */
    if (pv.rna.type)
    {
```

```
        runMode = MODE_IDS;
        LogMessage("Running in IDS mode with Artificial Neural
Network\n");
    }
    else if(pv.config_file)
    {
        runMode = MODE_IDS;
        if(!pv.quiet_flag)
            LogMessage("Running in IDS mode\n");
    }
    else if(pv.log_mode || pv.log_dir)
    {
        runMode = MODE_PACKET_LOG;
        if(!pv.quiet_flag)
            LogMessage("Running in packet logging mode\n");
    }
    else if(pv.verbose_flag)
    {
        runMode = MODE_PACKET_DUMP;
        if(!pv.quiet_flag)
            LogMessage("Running in packet dump mode\n");
    }
    else if((pv.config_file = ConfigFileSearch()))
    {
        runMode = MODE_IDS;
        if(!pv.quiet_flag)
            LogMessage("Running in IDS mode with inferred config
file: %s\n",
                    pv.config_file);
    }
    else
    {
        /* unable to determine a run mode */
        DisplayBanner();
        ShowUsage(progname);
        PrintError("\n\nUh, you need to tell me to do something...\
n\n");
        exit(1);
    }

    /* set the default logging dir if not set yet */
    /* XXX should probably be done after reading config files */
    if(!pv.log_dir)
    {
        if(!(pv.log_dir = strdup(DEFAULT_LOG_DIR)))
            FatalError("Out of memory setting default log dir\n");
    }

    /*
    **  Validate the log directory for logging packets
    */
    if(runMode == MODE_PACKET_LOG)
    {
```

```
        CheckLogDir();

        if(!pv.quiet_flag)
        {
            LogMessage("Log directory = %s\n", pv.log_dir);
        }
    }

    /* if we are in packet log mode, make sure we have a logging
mode set */
    if(runMode == MODE_PACKET_LOG && !pv.log_mode)
    {
        pv.log_mode = LOG_ASCII;
    }

    /*
     * if we're not reading packets from a file, open the network
interface
     * for reading.. (interfaces are being initalized before the
config file
     * is read, so some plugins would be able to start up properly.
     */
#ifdef GIDS
#ifdef IPFW
    /* Check to see if we got a Divert port or not */
    if(!pv.divert_port)
    {
        pv.divert_port = 8000;
    }

#endif /* IPFW */

    if (InlineMode())
    {
        InitInline();
    }
    else
#endif /* GIDS */

    if(!pv.readmode_flag)
    {
        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Opening interface: %s\
n",
                    PRINT_INTERFACE(pv.interface)););
        /* open up our libpcap packet capture interface */
        OpenPcap();
    }
    else
    {
        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Opening file: %s\n",
                    pv.readfile););

        /* open the packet file for readback */
```

```
        OpenPcap();
    }

    /* extract the config directory from the config filename */
    if(pv.config_file)
    {
        /* is there a directory seperator in the filename */
        if(strrchr(pv.config_file,'/'))
        {
            char *tmp;
            /* lazy way, we waste a few bytes of memory here */
            if(!(pv.config_dir = strdup(pv.config_file)))
                FatalError("Out of memory extracting config
dir\n");

            tmp = strrchr(pv.config_dir,'/');
            *(++tmp) = '\0';
        }
        else
        {
#ifdef WIN32
        /* is there a directory seperator in the filename */
        if(strrchr(pv.config_file,'\\'))
        {
            char *tmp;
            /* lazy way, we waste a few bytes of memory here */
            if(!(pv.config_dir = strdup(pv.config_file)))
                FatalError("Out of memory extracting config
dir\n");

            tmp = strrchr(pv.config_dir,'\\');
            *(++tmp) = '\0';
        }
        else
#endif
            if(!(pv.config_dir = strdup("./")))
                FatalError("Out of memory extracting config
dir\n");
        }
        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Config file = %s,
config dir = "
                    "%s\n", pv.config_file, pv.config_dir););
    }

    /* XXX do this after reading the config file? */
    if(pv.use_utc == 1)
    {
        thiszone = 0;
    }
    else
    {
        /* set the timezone (ripped from tcpdump) */
        thiszone = gmt2local(0);
```

```
    }

    if(!pv.quiet_flag)
    {
        LogMessage("\n        ---== Initializing Snort ==---\n");
    }

    if(runMode == MODE_IDS && pv.rules_order_flag)
    {
        if(!pv.quiet_flag)
        {
            LogMessage("Rule application order changed to Pass-
>Alert->Log\n");
        }
    }
    /*
     * if daemon mode requested, fork daemon first, otherwise on
linux
     * interface will be reset.
     */
    if(pv.daemon_flag)
    {
        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Entering daemon
mode\n"););
        GoDaemon();
    }

    InitOutputPlugins();

    /* create the PID file */
    /* TODO should be part of the GoDaemon process */
    if((runMode == MODE_IDS) || pv.log_mode || pv.daemon_flag
            || *pv.pidfile_suffix)
    {

        /* ... then create a PID file if not reading from a file */
        if (!pv.readmode_flag && (pv.daemon_flag ||
*pv.pidfile_suffix))
        {
#ifndef WIN32
#ifdef GIDS
            if (InlineMode())
            {
                CreatePidFile("inline");
            }
            else
            {
#else
            CreatePidFile(pv.interface);
#endif /* GIDS */
#ifdef GIDS
            }
#endif /* GIDS */
```

```
#else
            CreatePidFile("WIN32");
#endif
        }
    }


    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Setting Packet
Processor\n"););

    /* set the packet processor (ethernet, slip, t/r, etc ) */
    SetPktProcessor();

    /* if we're using the rules system, it gets initialized here */
    if(runMode == MODE_IDS)
    {
        /* initialize all the plugin modules */
        InitPreprocessors();
        InitPlugIns();
        InitTag();

#ifdef DEBUG
        DumpPreprocessors();
        DumpPlugIns();
        DumpOutputPlugins();
#endif
        /* setup the default rule action anchor points */
        CreateDefaultRules();

        if(pv.rules_order_flag)
        {
#ifdef GIDS
            OrderRuleLists("activation dynamic pass drop sdrop
reject alert log");
#else
            OrderRuleLists("pass activation dynamic alert log");
#endif /* GIDS */
        }

        if(!(pv.quiet_flag && !pv.daemon_flag))
            LogMessage("Parsing Rules file %s\n", pv.config_file);

        if (!pv.rna.type) ParseRulesFile(pv.config_file, 0);

        CheckLogDir();

        OtnXMatchDataInitialize();

        FlowBitsVerify();

        asn1_init_mem(512);

        /*
```

```
        **  Handles Fatal Errors itself.
        */
        SnortEventqInit();

#ifdef GIDS
#ifndef IPFW
        if (InlineMode())
        {
            InitInlinePostConfig();
        }
#endif /* IPFW */
#endif /* GIDS */

        if(!(pv.quiet_flag && !pv.daemon_flag))
        {
            print_thresholding();
            printRuleOrder();
            LogMessage("Log directory = %s\n", pv.log_dir);
        }
    }


#ifndef WIN32
    /* Drop the Chrooted Settings */
    if(pv.chroot_dir)
        SetChroot(pv.chroot_dir, &pv.log_dir);

    /* Drop privileges if requested, when initialization is done */
    SetUidGid();

#endif /*WIN32*/

    /*
     * if we are in IDS mode and either an alert option was
specified on the
     * command line or we do not have any alert plugins active, set
them up
     * now
     */
    if(runMode == MODE_IDS &&
       (pv.alert_cmd_override || !pv.alert_plugin_active))
    {
        ProcessAlertCommandLine();
    }

    /*
     * if we are in IDS mode or packet log mode and either a log
option was
     * specified on the command line or we do not have any log
plugins active,
     * set them up now
     */
    if((runMode == MODE_IDS || runMode == MODE_PACKET_LOG) &&
```

```
            (pv.log_cmd_override || !pv.log_plugin_active))
    {
        ProcessLogCommandLine();
    }

    /*
    **  Create Fast Packet Classification Arrays
    **  from RTN list.  These arrays will be used to
    **  classify incoming packets through protocol.
    */
    fpCreateFastPacketDetection();

    if(!pv.quiet_flag)
    {
        mpsePrintSummary();
    }

    if(!pv.quiet_flag)
    {
        LogMessage("\n        --== Initialization Complete
==--\n");
    }

    /* Tell 'em who wrote it, and what "it" is */
    if(!pv.quiet_flag)
        DisplayBanner();

    if(pv.test_mode_flag)
    {
        LogMessage("\nSnort sucessfully loaded all rules and
checked all rule "
                "chains!\n");
        CleanExit(0);
    }

    if(pv.daemon_flag)
    {
        LogMessage("Snort initialization completed successfully
(pid=%u)\n",getpid());
    }

#ifdef GIDS
    if (InlineMode())
    {
#ifndef IPFW
        IpqLoop();
#else
        IpfwLoop();
#endif
    }
    else
    {
#endif /* GIDS */
```

```
    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Entering pcap loop\n"););

    InterfaceThread(NULL);

#ifdef GIDS
    }
#endif /* GIDS */

    return 0;
}



void ProcessPacket(char *user, struct pcap_pkthdr * pkthdr, u_char
* pkt)
{
    Packet p;

    /* reset the packet flags for each packet */
    p.packet_flags = 0;

    pc.total++;

    /*
    ** Save off the time of each and every packet
    */
    packet_time_update(pkthdr->ts.tv_sec);


    /* reset the thresholding subsystem checks for this packet */
    sfthreshold_reset();

    SnortEventqReset();

#if defined(WIN32) && defined(ENABLE_WIN32_SERVICE)
    if( pv.terminate_service_flag || pv.pause_service_flag )
    {
        ClearDumpBuf();  /* cleanup and return without processing
*/
        return;
    }
#endif  /* WIN32 && ENABLE_WIN32_SERVICE */

    /* call the packet decoder */
    (*grinder) (&p, pkthdr, pkt);

    /* print the packet to the screen */
    if(pv.verbose_flag)
    {
        if(p.iph != NULL)
            PrintIPPkt(stdout, p.iph->ip_proto, &p);
        else if(p.ah != NULL)
```

```
            PrintArpHeader(stdout, &p);
        else if(p.eplh != NULL)
        {
            PrintEapolPkt(stdout, &p);
        }
        else if(p.wifih && pv.showwifimgmt_flag)
        {
            PrintWifiPkt(stdout, &p);
        }
    }

    //choose run mode
    switch(runMode)
    {
        case MODE_PACKET_LOG:
            CallLogPlugins(&p, NULL, NULL, NULL);
            break;
        case MODE_IDS:
            /* allow the user to throw away TTLs that won't apply
to the
               detection engine as a whole. */
            if(pv.min_ttl && p.iph != NULL && (p.iph->ip_ttl <
pv.min_ttl))
            {
                DEBUG_WRAP(DebugMessage(DEBUG_DECODE,
                            "MinTTL reached in main detection loop\
n"););
                return;
            }

            /* start calling the detection processes */
            //jeiks - RNA will enter around here.. hehe
            if (pv.rna.type) Rna(&p);
            else          Preprocess(&p);
            break;
        default:
            break;
    }
    ClearDumpBuf();

}


int ShowUsage(char *progname)
{
    fprintf(stdout, "USAGE: %s [-options] <filter options>\n",
progname);
#if defined(WIN32) && defined(ENABLE_WIN32_SERVICE)
    fprintf(stdout, "       %s %s %s [-options] <filter
options>\n", progname

, SERVICE_CMDLINE_PARAM
```

```
, SERVICE_INSTALL_CMDLINE_PARAM);
    fprintf(stdout, "       %s %s %s\n", progname
                                       , SERVICE_CMDLINE_PARAM
                                       ,
SERVICE_UNINSTALL_CMDLINE_PARAM);
    fprintf(stdout, "       %s %s %s\n", progname
                                       , SERVICE_CMDLINE_PARAM
                                       ,
SERVICE_SHOW_CMDLINE_PARAM);
#endif

#ifdef WIN32
    #define FPUTS_WIN32(msg) fputs(msg,stdout)
    #define FPUTS_UNIX(msg)  NULL
    #define FPUTS_BOTH(msg)  fputs(msg,stdout)
#else
    #define FPUTS_WIN32(msg)
    #define FPUTS_UNIX(msg)  fputs(msg,stdout)
    #define FPUTS_BOTH(msg)  fputs(msg,stdout)
#endif

    FPUTS_BOTH ("Options:\n");
    FPUTS_BOTH ("        -A         Set alert mode: fast, full,
console, or none "
                                  " (alert file alerts only)\n");
    FPUTS_UNIX ("                   \"unsock\" enables UNIX socket
logging (experimental).\n");
    FPUTS_BOTH ("        -b         Log packets in tcpdump format
(much faster!)\n");
    FPUTS_BOTH ("        -c <rules> Use Rules File <rules>\n");
    FPUTS_BOTH ("        -C         Print out payloads with
character data only (no hex)\n");
    FPUTS_BOTH ("        -d         Dump the Application Layer\n");
    FPUTS_UNIX ("        -D         Run Snort in background
(daemon) mode\n");
    FPUTS_BOTH ("        -e         Display the second layer header
info\n");
    FPUTS_WIN32("        -E         Log alert messages to NT
Eventlog. (Win32 only)\n");
    FPUTS_BOTH ("        -f         Turn off fflush() calls after
binary log writes\n");
    FPUTS_BOTH ("        -F <bpf>   Read BPF filters from file
<bpf>\n");
    FPUTS_UNIX ("        -g <gname> Run snort gid as <gname> group
(or gid) after initialization\n");
    FPUTS_BOTH ("        -h <hn>    Home network = <hn>\n");
    FPUTS_BOTH ("        -i <if>    Listen on interface <if>\n");
    FPUTS_BOTH ("        -I         Add Interface name to alert
output\n");
#ifdef GIDS
#ifdef IPFW
    FPUTS_BOTH ("        -J <port>  ipfw divert socket <port> to
listen on vice libpcap (FreeBSD only)\n");
```

```
#endif
#endif
    FPUTS_BOTH ("        -k <mode>  Checksum mode
(all,noip,notcp,noudp,noicmp,none)\n");
    FPUTS_BOTH ("        -l <ld>    Log to directory <ld>\n");
    FPUTS_BOTH ("        -L <file>  Log to this tcpdump file\n");
    FPUTS_UNIX ("        -m <umask> Set umask = <umask>\n");
    FPUTS_BOTH ("        -n <cnt>   Exit after receiving <cnt>
packets\n");
    FPUTS_BOTH ("        -N         Turn off logging (alerts still
work)\n");
    FPUTS_BOTH ("        -o         Change the rule testing order
to Pass|Alert|Log\n");
    FPUTS_BOTH ("        -O         Obfuscate the logged IP
addresses\n");
    FPUTS_BOTH ("        -p         Disable promiscuous mode
sniffing\n");
    fprintf(stdout, "        -P <snap>  Set explicit snaplen of
packet (default: %d)\n",
                                    SNAPLEN);
    FPUTS_BOTH ("        -q         Quiet. Don't show banner and
status report\n");
#ifdef GIDS
#ifndef IPFW
    FPUTS_BOTH ("        -Q         Use ip_queue for input vice
libpcap (iptables only)\n");
#endif
#endif
    FPUTS_BOTH ("        -r <tf>    Read and process tcpdump file
<tf>\n");
    FPUTS_BOTH ("        -R <id>    Include 'id' in
snort_intf<id>.pid file name\n");
    FPUTS_BOTH ("        -s         Log alert messages to
syslog\n");
    FPUTS_BOTH ("        -S <n=v>   Set rules file variable n equal
to value v\n");
    FPUTS_UNIX ("        -t <dir>   Chroots process to <dir> after
initialization\n");
    FPUTS_BOTH ("        -T         Test and report on the current
Snort configuration\n");
    FPUTS_UNIX ("        -u <uname> Run snort uid as <uname> user
(or uid) after initialization\n");
    FPUTS_BOTH ("        -U         Use UTC for timestamps\n");
    FPUTS_BOTH ("        -v         Be verbose\n");
    FPUTS_BOTH ("        -V         Show version number\n");
    FPUTS_WIN32("        -W         Lists available interfaces.
(Win32 only)\n");
#ifdef DLT_IEEE802_11
    FPUTS_BOTH ("        -w         Dump 802.11 management and
control frames\n");
#endif
    FPUTS_BOTH ("        -X         Dump the raw packet data
starting at the link layer\n");
```

```
    FPUTS_BOTH ("        -y         Include year in timestamp in
the alert and log files\n");
    FPUTS_BOTH ("        -z         Set assurance mode, match on
established sesions (for TCP)\n");
    FPUTS_BOTH ("        -Z <opt>   Use Artificial Neural Network,
<opt> can be:\n\t\t\t"
                                       "'run' to execute detection
with ANN,\n\t\t\t"
                                       "'train' to train ANN\n\t\t\
t"
                                       "'trainoff' to train again
with the last pattern\n\t\t\t"
                                       "'printresult' to print the
ANN bigger answer\n");
    FPUTS_BOTH ("        -?         Show this information\n");
    FPUTS_BOTH ("<Filter Options> are standard BPF options, as seen
in TCPDump\n");

#undef FPUTS_WIN32
#undef FPUTS_UNIX
#undef FPUTS_BOTH

    return 0;
}


int ParseCmdLine(int argc, char *argv[])
{
    int ch;                         /* storage var for getopt info
*/
    int read_bpf = 0;
    char bpf_file[STD_BUF];
    char *eq_n;
    char *eq_p;
    char errorbuf[PCAP_ERRBUF_SIZE];
    int umaskchange = 1;
    int defumask = 0;
#ifdef WIN32
    char *devicet;
    int adaplen;
#endif
    char *valid_options;

    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Parsing command
line...\n"););
    /* generally speaking, Snort works best when it's in
promiscuous mode */
    pv.promisc_flag = 1;

    /* just to be sane.. */
    username = NULL;
    groupname = NULL;
    pv.pidfile_suffix[0] = 0;
```

```
    /*
    **  Set this so we know whether to return 1 on invalid input.
    **  Snort uses '?' for help and getopt uses '?' for telling us
there
    **  was an invalid option, so we can't use that to tell invalid
input.
    **  Instead, we check optopt and it will tell us.
    */
    optopt = 0;

#ifndef WIN32
#ifdef GIDS
#ifndef IPFW
    valid_options = "?
A:bB:c:CdDefF:g:h:i:Ik:l:L:m:n:NoOpP:qQr:R:sS:t:Tu:UvVwXyz:Z:";
#else
    valid_options = "?
A:bB:c:CdDefF:g:h:i:IJ:k:l:L:m:n:NoOpP:qr:R:sS:t:Tu:UvVwXyz:Z:";
#endif /* IPFW */
#else
    /* Unix does not support an argument to -s <wink marty!> OR -E,
-W */
    valid_options = "?
A:bB:c:CdDefF:g:h:i:Ik:l:L:m:n:NoOpP:qr:R:sS:t:Tu:UvVwXyz:Z:";
#endif /* GIDS */
#else
    /* Win32 does not support:  -D, -g, -m, -t, -u */
    /* Win32 no longer supports an argument to -s, either! */
    valid_options = "?
A:bB:c:CdeEfF:h:i:Ik:l:L:n:NoOpP:qr:R:sS:TUvVwWXyz:Z:";
#endif

    /* loop through each command line var and process it */
    while((ch = getopt(argc, argv, valid_options)) != -1)
    {
        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Processing cmd line
switch: %c\n", ch););
        switch(ch)
        {
            case 'A':                /* alert mode */
                if(!strcasecmp(optarg, "none"))
                {
                    pv.alert_mode = ALERT_NONE;
                }
                else if(!strcasecmp(optarg, "full"))
                {
                    pv.alert_mode = ALERT_FULL;
                }
                else if(!strcasecmp(optarg, "fast"))
                {
                    pv.alert_mode = ALERT_FAST;
                }
```

```
                else if(!strcasecmp(optarg, "console"))
                {
                    pv.alert_mode = ALERT_STDOUT;
                }
                else if(!strcasecmp(optarg, "cmg"))
                {
                    pv.alert_mode = ALERT_CMG;
                    /* turn off logging */
                    pv.log_mode = LOG_NONE;
                    pv.log_cmd_override = 1;
                    /* turn on layer2 headers */
                    pv.show2hdr_flag = 1;
                    /* turn on data dump */
                    pv.data_flag = 1;
                }
                else if(!strcasecmp(optarg, "unsock"))
                {
                    pv.alert_mode = ALERT_UNSOCK;
                }
                else
                {
                    FatalError("Unknown command line alert option:
%s\n", optarg);
                }

                /* command line alert machanism has been specified,
override
                 * the config file options
                 */
                pv.alert_cmd_override = 1;
                break;

            case 'b':                /* log packets in binary
format for
                                      * post-processing */
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Tcpdump
logging mode "
                            "active\n"););
                pv.log_mode = LOG_PCAP;
                pv.log_cmd_override = 1;
                break;

            case 'B': /* obfuscate with a substitution mask */
                pv.obfuscation_flag = 1;
                GenObfuscationMask(optarg);
                break;

            case 'c':                /* use configuration file x */
                if(!(pv.config_file = strdup(optarg)))
                    FatalError("Out of memory processing command
line\n");
                break;
```

```
            case 'C':  /* dump the application layer as text only
*/
                pv.char_data_flag = 1;
                break;

            case 'd':                /* dump the application layer
data */
                pv.data_flag = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Data Flag
active\n"););
                break;

            case 'D':                /* daemon mode */
#ifdef WIN32
                FatalError("Setting the Daemon mode is not
supported in the "
                           "WIN32 port of snort!  Use 'snort
/SERVICE ...' "
                           "instead\n");
#endif
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Daemon mode
flag set\n"););
                pv.daemon_flag = 1;
                flow_set_daemon();
                pv.quiet_flag = 1;
                break;

            case 'e':                /* show second level header
info */
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Show 2nd level
active\n"););
                pv.show2hdr_flag = 1;
                break;

#ifdef WIN32
            case 'E':                /* log alerts to Event Log */
                pv.alert_mode = ALERT_SYSLOG;
                pv.syslog_remote_flag = 0;
                pv.alert_cmd_override = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Logging alerts
to Event "
                            "Log\n"););
                break;
#endif
            case 'f':
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Pcap
linebuffering "
                           "activated\n"););
                pv.line_buffer_flag = 1;
                break;

            case 'F':                /* read BPF filter in from a
file */
```

```
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Tcpdump
logging mode "
                           "active\n"););
                strlcpy(bpf_file, optarg, STD_BUF);
                read_bpf = 1;
                break;

            case 'g':                /* setgid handler */
#ifdef WIN32
                FatalError("Setting the group id is not supported
in the WIN32 port of snort!\n");
#else
                if(groupname != NULL)
                    free(groupname);
                if((groupname = calloc(strlen(optarg) + 1, 1)) ==
NULL)
                    FatalPrintError("malloc");

                bcopy(optarg, groupname, strlen(optarg));

                if((groupid = atoi(groupname)) == 0)
                {
                    gr = getgrnam(groupname);
                    if(gr == NULL)
                        FatalError("Group \"%s\" unknown\n",
groupname);

                    groupid = gr->gr_gid;
                }
#endif
                break;

            case 'h':                /* set home network to x, this
will help
                                     * determine what to set
logging diectories
                                     * to */
                GenHomenet(optarg);
                break;

            case 'i':
                if(pv.interface)
                {
                    FatalError("Cannot specify more than one
network "
                               "interface on the command line.\n");
                }
#ifdef WIN32
                /* first, try to handle the "-i1" case, where an
interface
                 * is specified by number.  If this fails, then
fall-through
                 * to the case outside the ifdef/endif, where an
```

```
interface
                 * can be specified by its fully qualified name,
like as is
                 * shown by running 'snort -W', ie.
                 * "\Device\Packet_{12345678-90AB-
CDEF-1234567890AB}"
                 */
                devicet = NULL;
                adaplen = atoi(optarg);
                if( adaplen > 0 )
                {
                    devicet = pcap_lookupdev(errorbuf);
                    if ( devicet == NULL )
                    {
                        perror(errorbuf);
                        exit(1);
                    }

                    pv.interface = GetAdapterFromList(devicet,
adaplen);
                    if ( pv.interface == NULL )
                    {
                        LogMessage("Invalid interface '%d'.\n",
atoi(optarg));
                        exit(1);
                    }


                    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Interface
= %s\n",
                                PRINT_INTERFACE(pv.interface)));
                }
                else
#endif  /* WIN32 */
                /* this code handles the case in which the user
specifies
                   the entire name of the interface and it is
compiled
                   regardless of which OS you have */
                {
                    pv.interface = (char *)malloc(strlen(optarg) +
1);
                    /* XXX OOM check */
                    strlcpy(pv.interface, optarg, strlen(optarg)
+1);
                    DEBUG_WRAP(DebugMessage(DEBUG_INIT,
                        "Interface = %s\n",
                        PRINT_INTERFACE(pv.interface)););
                }
                break;

            case 'I':       /* add interface name to alert string
*/
```

```
                pv.alert_interface_flag = 1;
                break;

#ifdef GIDS
#ifdef IPFW
            case 'J':
                LogMessage("Reading from ipfw divert socket\n");
                pv.inline_flag = 1;
                pv.divert_port = atoi(optarg);
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Divert port
set to: %d\n", pv.divert_port););
                LogMessage("IPFW Divert port set to: %d\n",
pv.divert_port);
                pv.promisc_flag = 0;
                pv.interface = NULL;
                break;
#endif
#endif


            case 'k':  /* set checksum mode */
                if(!strcasecmp(optarg, "all"))
                {
                    pv.checksums_mode = DO_IP_CHECKSUMS |
DO_TCP_CHECKSUMS |
                                        DO_UDP_CHECKSUMS |
DO_ICMP_CHECKSUMS;
                }
                else if(!strcasecmp(optarg, "noip"))
                {
                    pv.checksums_mode ^= DO_IP_CHECKSUMS;
                }
                else if(!strcasecmp(optarg, "notcp"))
                {
                    pv.checksums_mode ^= DO_TCP_CHECKSUMS;
                }
                else if(!strcasecmp(optarg, "noudp"))
                {
                    pv.checksums_mode ^= DO_UDP_CHECKSUMS;
                }
                else if(!strcasecmp(optarg, "noicmp"))
                {
                    pv.checksums_mode ^= DO_ICMP_CHECKSUMS;
                }
                if(!strcasecmp(optarg, "none"))
                {
                    pv.checksums_mode = 0;
                }
                break;

            case 'l':                /* use log dir <X> */
                if(!(pv.log_dir = strdup(optarg)))
                {
```

```
                    FatalError("Out of memory processing command
line\n");
                }

                if(access(pv.log_dir, 2) != 0)
                {
                    FatalError("log directory '%s' does not
exist\n",
                            pv.log_dir);
                }
                break;

            case 'L':  /* set BinLogFile name */
                /* implies tcpdump format logging */
                if (strlen(optarg) < 256)
                {
                    pv.log_mode = LOG_PCAP;
                    pv.binLogFile = strdup(optarg);
                    pv.log_cmd_override = 1;
                }
                else
                {
                    FatalError("ParseCmdLine, log file: %s, > than
256 characters\n",
                               optarg);
                }
                break;

            case 'm': /* set the umask for the output files */
#ifdef WIN32
                FatalError("Setting the umask is not supported in
the "
                           "WIN32 port of snort!\n");
#endif
                {
                    char *p;
                    long val = 0;

                    umaskchange = 0;

                    val = strtol(optarg, &p, 8);
                    if (*p != '\0' || val < 0 || (val &
~FILEACCESSBITS))
                    {
                        FatalError("bad umask %s\n", optarg);
                    }
                    else
                    {
                        defumask = val;
                    }
                }
                break;
```

```
            case 'n':                /* grab x packets and exit */
                pv.pkt_cnt = atoi(optarg);
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Exiting after
%d packets\n", pv.pkt_cnt););
                break;

            case 'N':                /* no logging mode */
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Logging
deactivated\n"););
                pv.log_mode = LOG_NONE;
                pv.log_cmd_override = 1;
                break;

            case 'o': /* change the rules processing order to
                       * passlist first */
                pv.rules_order_flag = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Rule
application order changed to Pass->Alert->Log\n"););
                break;

            case 'O':  /* obfuscate the logged IP addresses for
                        * privacy */
                pv.obfuscation_flag = 1;
                break;

            case 'p':  /* disable explicit promiscuous mode */
                pv.promisc_flag = 0;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Promiscuous
mode disabled!\n"););
                break;

            case 'P':  /* explicitly define snaplength of packets
*/
                pv.pkt_snaplen = atoi(optarg);
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Snaplength of
Packets set to: %d\n", pv.pkt_snaplen););
                break;

            case 'q':  /* no stdout output mode */
                pv.quiet_flag = 1;
                break;
#ifdef GIDS
#ifndef IPFW
            case 'Q':
                LogMessage("Reading from iptables\n");
                pv.inline_flag = 1;
                pv.promisc_flag = 0;
                pv.interface = NULL;
                break;
#endif
#endif /* GIDS */

            case 'r':  /* read packets from a TCPdump file instead
```

```
                        * of the net */
                strlcpy(pv.readfile, optarg, STD_BUF);
                pv.readmode_flag = 1;
                if(argc == 3)
                {
                    LogMessage("No run mode specified, defaulting
to verbose mode\n");
                    pv.verbose_flag = 1;
                    pv.data_flag = 1;
                }
                break;

            case 'R': /* augment pid file name CPW*/
                if (strlen(optarg) < MAX_PIDFILE_SUFFIX &&
strlen(optarg) > 0)
                {
                    if (!strstr(optarg, "..") && !(strstr(optarg,
"/")))
                    {
                        snprintf(pv.pidfile_suffix,
MAX_PIDFILE_SUFFIX, "%s",
                                optarg);
                    }
                    else
                    {
                        FatalError("ERROR: illegal pidfile suffix:
%s\n",
                                optarg);
                    }
                }
                else
                {
                    FatalError("ERROR: pidfile suffix length
problem: %d\n",
                            strlen(optarg) );
                }
                break;

            case 's':  /* log alerts to syslog */
                pv.alert_mode = ALERT_SYSLOG;
#ifndef WIN32
                /* command line alerting option has been specified,
                 * override the alert options in the config file
                 */
                pv.alert_cmd_override = 1;
#else
                pv.alert_cmd_override = 0;
                pv.syslog_remote_flag = 1;
#endif
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Logging alerts
to "
                            "syslog\n"););
                break;
```

```
            case 'S':  /* set a rules file variable */
                if((eq_p = strchr(optarg, '=')) != NULL)
                {
                    struct VarEntry *p;
                    int namesize = eq_p-optarg;
                    eq_n = calloc(namesize+2, sizeof(char));
                    strlcpy(eq_n, optarg, namesize+1);
                    p = VarDefine(eq_n, eq_p + 1);
                    p->flags |= VAR_STATIC;
                    free(eq_n);
                }
                else
                {
                    FatalError("Format for command line variable
definitions "
                               "is:\n -S var=value\n");
                }
                break;

            case 't':  /* chroot to the user specified directory */
#ifdef WIN32
                FatalError("Setting the chroot directory is not
supported in "
                           "the WIN32 port of snort!\n");
#endif  /* WIN32 */
                if(!(pv.chroot_dir = strdup(optarg)))
                    FatalError("Out of memory processing command
line\n");
                break;

            case 'T': /* test mode, verify that the rules load
properly */
                pv.test_mode_flag = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Snort starting
in test mode...\n"););
                break;

            case 'u':  /* setuid */
#ifdef WIN32
                FatalError("Setting the user id is not "
                           "supported in the WIN32 port of
snort!\n");
#else
                if((username = calloc(strlen(optarg) + 1, 1)) ==
NULL)
                    FatalPrintError("malloc");

                bcopy(optarg, username, strlen(optarg));

                if((userid = atoi(username)) == 0)
                {
                    pw = getpwnam(username);
```

```
                    if(pw == NULL)
                        FatalError("User \"%s\" unknown\n",
username);

                    userid = pw->pw_uid;
                }
                else
                {
                    pw = getpwuid(userid);
                    if(pw == NULL)
                        FatalError(
                                "Can not obtain username for uid:
%lu\n",
                                (u_long) userid);
                }

                if(groupname == NULL)
                {
                    char name[256];

                    snprintf(name, 255, "%lu", (u_long) pw-
>pw_gid);

                    if((groupname = calloc(strlen(name) + 1, 1)) ==
NULL)
                    {
                        FatalPrintError("malloc");
                    }
                    groupid = pw->pw_gid;
                }
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "UserID: %lu
GroupID: %lu\n",
                    (unsigned long) userid, (unsigned long)
groupid););
#endif  /* !WIN32 */
                break;

            case 'U': /* use UTC */
                pv.use_utc = 1;
                break;

            case 'v': /* be verbose */
                pv.verbose_flag = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Verbose Flag
active\n"););
                break;

            case 'V': /* prog ver already gets printed out, so we
                       * just exit */
                DisplayBanner();
                exit(0);

#ifdef WIN32
```

```
            case 'W':
                if ((pv.interface = pcap_lookupdev(errorbuf)) ==
NULL)
                    perror(errorbuf);

                DisplayBanner();
                PrintDeviceList(pv.interface);
                exit(0);
                break;
#endif  /* WIN32 */

#ifdef DLT_IEEE802_11
            case 'w':                /* show 802.11 all frames info
*/
                pv.showwifimgmt_flag = 1;
                break;
#endif

            case 'X':  /* display verbose packet bytecode dumps */
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Verbose packet
bytecode dumps enabled\n"););
                pv.verbose_bytedump_flag = 1;
                break;

            case 'y':  /* Add year to timestamp in alert and log
files */
                pv.include_year = 1;
                DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Enabled year
in timestamp\n"););
                break;

            case 'z': /* set assurance mode (used with stream 4) */
                pv.assurance_mode = ASSURE_EST;
                break;

            //jeiks
            case 'Z':
#ifndef WIN32
                pv.rna.config_filename=strdup("/etc/snort/rna.conf"
);
#else
                pv.rna.config_filename=strdup("./rna.conf");
#endif

                if (!strcasecmp(optarg, "run")){
                   pv.rna.type=1;
#ifndef WIN32
                   pv.rna.net_filename=strdup("/etc/snort/rna.net")
;
#else
                   pv.rna.net_filename=strdup("./rna.net");
#endif
                }
```

```
                else if (!strcasecmp(optarg, "train")){
                   pv.rna.type=2;
#ifndef WIN32
                   pv.rna.net_filename=strdup("/etc/snort/rna.net")
;
                   pv.rna.train_filename=strdup("/etc/snort/rna.pat
");
#else
                   pv.rna.net_filename=strdup("./rna.net");
                   pv.rna.train_filename=strdup("./rna.pat");
#endif
                }else if (!strcasecmp(optarg, "trainoff")){
                   pv.rna.type=3;
#ifndef WIN32
                   pv.rna.net_filename=strdup("/etc/snort/rna.net")
;
                   pv.rna.train_filename=strdup("/etc/snort/rna.pat
");
#else
                   pv.rna.net_filename=strdup("./rna.net");
                   pv.rna.train_filename=strdup("./rna.pat");
#endif
                }else if (!strcasecmp(optarg, "printresult")){
                   pv.rna.type=4;
#ifndef WIN32
                   pv.rna.net_filename=strdup("/etc/snort/rna.net")
;
#else
                   pv.rna.net_filename=strdup("./rna.net");
#endif
                }else{
                   FatalError("Unknown Command, try [run] or
[train]\n");
                }
                break;

            case '?':  /* show help and exit with 1 */
                DisplayBanner();
                ShowUsage(progname);

                if(optopt)
                    exit(1);

                exit(0);
        }
    }

    /* TODO relocate all of this to later in startup process */

    /* if the umask arg happened, set umask */
    if (umaskchange)
    {
        umask(077);           /* set default to be sane */
```

```
    }
    else
    {
        umask(defumask);
    }

    /* if we're reading in BPF filters from a file */
    if(read_bpf)
    {
        /* suck 'em in */
        pv.pcap_cmd = read_infile(bpf_file);
    }
    else
    {
        /* set the BPF rules string (thanks Mike!) */
        pv.pcap_cmd = copy_argv(&argv[optind]);
    }

    if((pv.interface == NULL) && !pv.readmode_flag)
    {
#ifdef GIDS
        if (!InlineMode())
        {
#endif /* GIDS */
        pv.interface = pcap_lookupdev(errorbuf);

        if(pv.interface == NULL)
            FatalError( "Failed to lookup for interface: %s."
                    " Please specify one with -i switch\n",
errorbuf);
#ifdef GIDS
        }
#endif /* GIDS */
    }

    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "pcap_cmd is %s\n",
                pv.pcap_cmd !=NULL ? pv.pcap_cmd : "NULL"););
    return 0;
}


void *InterfaceThread(void *arg)
{
    static int intnum = 0;
    int myint;
    struct timeval starttime;
    struct timeval endtime;
    struct timeval difftime;
    struct timezone tz;

    myint = intnum;
    intnum++;
```

```
    bzero((char *) &tz, sizeof(tz));
    gettimeofday(&starttime, &tz);

    if (pv.rna.type)
rna_load_config( fopen(pv.rna.config_filename,"r") );
    /* open the rna pattern file */
    switch(pv.rna.type)
    {
        case 2:
            if ( (pv.rna.train_file =
fopen(pv.rna.train_filename,"w"))==NULL )
                ErrorMessage("Couldn't open the pattern file for
write!\n");
            else   rna_write_pat_header();
            break;
        case 3:
            CleanExit(0);
            break;
    }

    /* Read all packets on the device.  Continue until cnt packets
read */
    if(pcap_loop(pd, pv.pkt_cnt, (pcap_handler) ProcessPacket,
NULL) < 0)
    {
        if(pv.daemon_flag)
            syslog(LOG_CONS | LOG_DAEMON, "pcap_loop: %s",
pcap_geterr(pd));
        else
            ErrorMessage("pcap_loop: %s\n", pcap_geterr(pd));

        CleanExit(1);
    }

    gettimeofday(&endtime, &tz);

    TIMERSUB(&endtime, &starttime, &difftime);

    printf("Run time for packet processing was %lu.%lu seconds\n",
            (unsigned long)difftime.tv_sec, (unsigned
long)difftime.tv_usec);

    CleanExit(0);

    return NULL;                /* avoid warnings */
}


void CleanExit(int exit_val)
{
    PluginSignalFuncNode *idx = NULL;

    /* This function can be called more than once.  For example,
```

```
     * once from the SIGINT signal handler, and once recursively
     * as a result of calling pcap_close() below.  We only need
     * to perform the cleanup once, however.  So the static
     * variable already_exiting will act as a flag to prevent
     * double-freeing any memory.  Not guaranteed to be
     * thread-safe, but it will prevent the simple cases.
     */
    static int already_exiting = 0;
    if( already_exiting != 0 )
    {
        return;
    }
    already_exiting = 1;
    /* Print Statistics */
    if(!pv.test_mode_flag)
    {
        fpShowEventStats();
        DropStats(0);
    }

    /* Exit plugins */
    idx = PluginCleanExitList;
    //if(idx)
    //    LogMessage("WARNING: Deprecated Plugin API still in
use\n");

#ifdef GIDS
#ifndef IPFW
    if (InlineMode())
    {

        if (ipqh)
        {
            ipq_destroy_handle(ipqh);
        }

    }
#endif /* IPFW (may need cleanup code here) */
#endif /* GIDS */

    while(idx)
    {
        idx->func(SIGQUIT, idx->arg);
        idx = idx->next;
    }

    /* free allocated memory */

    /* close pcap */
    if (pd && !InlineMode())
        pcap_close(pd);
```

```
    /* remove pid file */
    if(pv.pid_filename)
        unlink(pv.pid_filename);

    /*if the train file is open...*/
    if(pv.rna.type==2 || pv.rna.type==3)
    {
        if (pv.rna.type==2) rna_finalize_pat_file();
        if (pv.rna.num_patterns) Rna_train();
        else FatalError("No Patterns to train ANN\n");
    }else if(pv.rna.type == 4)
    {
        LogMessage("The bigger ANN answer: %f\n\n",pv.rna.answer);
    }

    LogMessage("Snort exiting\n");
    /* exit */
    exit(exit_val);
}

static void Restart()
{
    PluginSignalFuncNode *idx = NULL;

    /* Print statistics */
    if(!pv.test_mode_flag)
    {
        fpShowEventStats();
        DropStats(0);
    }

    /* Exit plugins */
    /* legacy exit code */
    idx = PluginRestartList;
    //if(idx)
    //    LogMessage("WARNING: Deprecated Plugin API still in
use\n");

    while(idx)
    {
        idx->func(SIGHUP, idx->arg);
        idx = idx->next;
    }

    /* free allocated memory */

    /* close pcap */
    if(pd)
        pcap_close(pd);

    /* remove pid file */

    if(pv.pid_filename)
```

```
        unlink(pv.pid_filename);

    /*if the train file is open...*/
    if(pv.rna.type==2 || pv.rna.type==3)
    {
        if (pv.rna.type==2) rna_finalize_pat_file();
        Rna_train();
    }else if(pv.rna.type == 4)
    {
        LogMessage("The bigger ANN answer: %f\n\n",pv.rna.answer);
    }

    LogMessage("Restarting Snort\n");

    /* re-exec Snort */
#ifdef PARANOID
    execv(progname, progargs);
#else
    execvp(progname, progargs);
#endif

    /* only get here if we failed to restart */
    LogMessage("Restarting %s failed: %s\n", progname,
strerror(errno));
    exit(1);
}
```

O programa *snns2c* foi alterado a criação do arquivo *rna_alg.c* adaptado para inclusão no programa Snort. O Quadro 11 apresenta as funções alteradas do programa citado.

Quadro 11: Código fonte snns2c.c

```
void writeLayer(pLayer layer, FILE *fOutFile)
{
     int i;

  fprintf(fOutFile, "\n  pUnit %s[%d] = ",
          layer->name, NoOf(layer->members) );

  /* write the Members of the layer */
     fprintf(fOutFile, "{");
  for(i = 0; i < NoOf(layer->members); i++) {
           /* write the members as pointer to the member units */
    fprintf(fOutFile, "Units + %d", element(layer->members,i) );
    if (i < NoOf(layer->members) - 1) fprintf(fOutFile, ", ");
  }

  fprintf(fOutFile, "}; /* members */\n");
}
```

```
void writeUnitNew(pUnit unit, FILE *fOutFile, int count_links)
 {
  int i;
  static int first_time = 1, is_Bptt = 0; /* to avoid unneeded
procedure calls */

     /* initialisation of is_Bptt */
     if (first_time) {
          is_Bptt = is_BPTT_net();
          first_time = 0;
     }

  /* write Number and Name of the unit for identification,
   * if a user is reading the code
   */
  fprintf(fOutFile, "    { /* unit %d (%s) */\n", unit->number,
unit->name);

  /* write Activation, Bias and number of sources */
  if (is_Bptt) {
    fprintf(fOutFile, "      {%f, 0.0}, %f, %d,\n",
         unit->act, unit->Bias, NoOf(unit->sources));
   }
   else {
     fprintf(fOutFile, "      0.0, %f, %d,\n",unit->Bias,
NoOf(unit->sources));
   }

   /* write the Sources of the unit */
   fprintf(fOutFile, "       &Sources[%d] , \n", count_links);
   /* write the weights of the units */
   fprintf(fOutFile, "       &Weights[%d] , \n", count_links);

   fprintf(fOutFile, "      }");
 }

void writeAllUnitsOld(pUnit Units, int NoOfUnits, FILE *fOutFile)
{
  int nr;

  fprintf(fOutFile, "  /* unit definition section (see also
UnitType) */\n");

  /* Writing declaration of the Unit-Array */
  /* the 0-Element is left free because the comiler was easier
   * to implement this way. The only exeptions are the TDNNs
   * because here the Units are rewritten completely */
  if (is_TDNN_net()) {
    fprintf(fOutFile,
         "  UnitType Units[%d] = \n  {\n", NoOfUnits
         );
  }
```

```
  else if (is_CPN_net() ){
    fflush(fOutFile);
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n %s,\n",
          NoOfUnits + 1, CpnDummyUnit
          );
    fflush(fOutFile);
  }
  else if (is_BPTT_net() ) {
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n    %s,\n",
          NoOfUnits + 1,
          "{ {0.0, 0.0}, 0.0, 0, {NULL /* NO SOURCES */}, {0.0 /* NO
MEMBERS*/} }"
          );
  }
  else {
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n    %s,\n",
          NoOfUnits + 1,
          "{ 0.0, 0.0, 0, {NULL /* NO SOURCES */}, {0.0 /* NO
MEMBERS*/} }"
          );
  }

  /* Because of the special requirements of each network-type
   * the Unit types are slightly modified for each update-function.
   * so they need a special output-template */
  if (is_TDNN_net() ) {
    for(nr = 0; nr < NoOfUnits; nr++) {
     writeTdnnUnit(Units + nr, Units, fOutFile);
      if (nr < NoOfUnits -1) fprintf(fOutFile, ",\n");
      else                   fprintf(fOutFile, "\n");
    }
  }
  else if(is_CPN_net() ) {
    for(nr = 0; nr < NoOfUnits; nr++) {
      if (nr != 0) {
     fprintf(fOutFile, ",\n");
      }
      writeCpnUnit(Units + nr, fOutFile);
    }
    fprintf(fOutFile, "\n");
  }
  /* Here also BPTT-units are included, because they are very
similar to
     the other types */
  else {
    for(nr = 0; nr < NoOfUnits; nr++) {
      writeUnit(Units + nr, fOutFile);
      if (nr < NoOfUnits -1) fprintf(fOutFile, ",\n");
      else                   fprintf(fOutFile, "\n");
    }
```

```
  }
  fprintf(fOutFile, "\n  };\n\n");
}

void writeAllUnits(pUnit Units, int NoOfUnits, FILE *fOutFile)
{
  int nr;
  int count_links;  /* to give number to links */

  fprintf(fOutFile, "  /* unit definition section (see also
UnitType) */\n");

  /* Writing declaration of the Unit-Array */
  /* the 0-Element is left free because the comiler was easier
   * to implement this way. The only exeptions are the TDNNs
   * because here the Units are rewritten completely */
  if (is_TDNN_net()) {
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n", NoOfUnits
          );
  }
  else if (is_CPN_net() ){
    fflush(fOutFile);
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n %s,\n",
          NoOfUnits + 1, CpnDummyUnit
          );
    fflush(fOutFile);
  }
  else if (is_BPTT_net() ) {
    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n    %s,\n",
          NoOfUnits + 1,
          "{ {0.0, 0.0}, 0.0, 0, {NULL /* NO SOURCES */}, {0.0 /* NO
MEMBERS*/} }"
          );
  }
  else {

    fprintf(fOutFile,
          "  UnitType Units[%d] = \n  {\n    %s,\n",
          NoOfUnits + 1,
          "{ 0.0, 0.0, 0, NULL , NULL }"
          );
  }

  /* Because of the special requirements of each network-type
   * the Unit types are slightly modified for each update-function.
   * so they need a special output-template */
  if (is_TDNN_net() ) {
    for(nr = 0; nr < NoOfUnits; nr++) {
     writeTdnnUnit(Units + nr, Units, fOutFile);
      if (nr < NoOfUnits -1) fprintf(fOutFile, ",\n");
```

```
      else                    fprintf(fOutFile, "\n");
    }
  }
  else if(is_CPN_net() ) {
    for(nr = 0; nr < NoOfUnits; nr++) {
      if (nr != 0) {
     fprintf(fOutFile, ",\n");
      }
      writeCpnUnit(Units + nr, fOutFile);
    }
    fprintf(fOutFile, "\n");
  }
  /* Here also BPTT-units are included, because they are very
similar to the other types */
   else {
    count_links = 0; /* no links yet */

    for(nr = 0; nr < NoOfUnits; nr++) {
      writeUnitNew(Units + nr, fOutFile, count_links);
      count_links += NoOf((Units+nr)->sources);
   /* old  writeUnit(Units + nr, fOutFile); */
      if (nr < NoOfUnits -1) fprintf(fOutFile, ",\n");
      else                   fprintf(fOutFile, "\n");
    }
  }
  fprintf(fOutFile, "\n  };\n\n");
}

 void writeForwardDeclarationAllUnits(pUnit Units, int NoOfUnits,
FILE *fOutFile)
 {
   int nr;

   fprintf(fOutFile, "  /* Forward Declaration for all unit types
*/\n");

   /* Writing declaration of the Unit-Array */
   /* the 0-Element is left free because the comiler was easier
    * to implement this way. The only exeptions are the TDNNs
    * because here the Units are rewritten completely */
   if (is_TDNN_net()) {
     fprintf(fOutFile,
         "  UnitType Units[%d];\n", NoOfUnits
         );
   }
   else if (is_CPN_net() ){
     fflush(fOutFile);
     fprintf(fOutFile,
         "  UnitType Units[%d];\n",
         NoOfUnits + 1
         );
     fflush(fOutFile);
   }
```

```
   else if (is_BPTT_net() ) {
     fprintf(fOutFile,
          "  UnitType Units[%d];\n",
          NoOfUnits + 1
          );
   }
   else {
     fprintf(fOutFile,
          "  UnitType Units[%d];\n",
          NoOfUnits + 1
          );
   }
 }

 void writeAllSources(pUnit Units, int NoOfUnits, FILE *fOutFile)
 {
   int nr, i ;

   fprintf(fOutFile, "  /* Sources definition section */\n");
   fprintf(fOutFile, "  pUnit Sources[] =  {\n");

   for(nr = 0; nr < NoOfUnits; nr++)
     {
       if(writeSource(Units + nr, fOutFile))
         fprintf(fOutFile, "\n");
     }
   fprintf(fOutFile, "\n  };\n\n");
 }

 void writeAllWeights(pUnit Units, int NoOfUnits, FILE *fOutFile)
 {
   int nr, i ;

   fprintf(fOutFile, "  /* Weigths definition section */\n");
   fprintf(fOutFile, "  float Weights[] =  {\n");

   for(nr = 0; nr < NoOfUnits; nr++)
     {
      if( writeWeigths(Units + nr, fOutFile))
         fprintf(fOutFile, "\n");
   }
   fprintf(fOutFile, "\n  };\n\n");
 }

int writeTdnnNet(pLayer globalLayers, pUnit globalUnits,
                 int NoOfLayers, int NoOfUnits,
                 int *order, char *OutFile, char *ProcName)
{
  pLayer layer, source;
  pUnit  unit;
  int    nr, layerNo, unitNo, sourceNo, pos, maxSource = 0,
  maxFeature = 0, maxDelay = 0, maxTotalDelay = 0;
  FILE   *fOutFile;//, *fHeaderFile;
```

```
  pList  OutList, FunctionList;
  time_t   timer;
  char   HeaderFile[50];

  /* calculate time for the date in the header of the output file*/
  time(&timer);

  /* Preparations for the Header-File */
  strcpy(HeaderFile, OutFile);
  HeaderFile[strlen(HeaderFile) - 1] = 'h';
  fHeaderFile = fopen(HeaderFile, "w");
  if (!fHeaderFile) return(CANT_OPEN);

  /* prepare Output Stream */
  fOutFile = fopen(OutFile, "w");
  if (!fOutFile) return(CANT_OPEN);

  /* maxSource is needed for writing the Unit-array */
  for(unit = globalUnits; unit < globalUnits + NoOfUnits; unit++) {
    maxSource  = MAX(maxSource, NoOf(unit->sources) );
  /* Calculating the Maximum Feature-Width is harmless for non-
TDNNs */
    maxFeature = MAX(maxFeature, unit->FeatureWidth);
  }

  /* TDNN needs the Maximum Receptive Field */
  for(layer = globalLayers; layer < globalLayers + NoOfLayers;
layer++) {
    maxDelay      = MAX(maxDelay, layer->delay);
    maxTotalDelay = MAX(maxTotalDelay, layer->TotalDelay);
  }

  /* Net-Output may consist of several (output) Layers     */
  /* so there must be an extra merge-list : OutList        */
  /* FunctionList will contain a set of all used functions */
  OutList = newList();
  if (!OutList) return (MEM_ERR);

  FunctionList = newList();
  if (!FunctionList) return (MEM_ERR);

  for(nr = 0; nr < NoOfLayers; nr ++)
   {
    layer = globalLayers + nr;
    if( layer->type == OUTPUT)
     {
      if (mergeList(OutList, layer->members) ) return (MEM_ERR);
     }
    else
    if (addList(FunctionList, layer->ActFunc) ) return(MEM_ERR);
   }

  /** find Names for the Layers **/
```

```
  if (NameLayers(globalLayers, NoOfLayers+1)) return(MEM_ERR);


  /** write Header-File **/
  fprintf(fHeaderFile, TdnnHeaderFileTemplate,
       HeaderFile, ctime(&timer), ProcName,
       ProcName, NoOf(globalLayers[order[0]].members),
       NoOf(OutList), globalLayers[order[0]].TotalDelay,
       ProcName);
  fclose(fHeaderFile);

  /** write the Programm Header and Act-Functions  **/
  fprintf(fOutFile, ProgHeader, OutFile, ctime(&timer) );
  for(nr = 0; nr < NoOf(FunctionList); nr ++) {
    fprintf(fOutFile, "%s\n", ACT_FUNCTIONS[element(FunctionList,
nr)]);
  }
  /* to avoid unneeded include-files */
  fprintf(fOutFile, "#define NULL (void *)0\n");

  /* write Procedure-Header */
  fprintf(fOutFile, ProcHeader, ProcName);

  /* The Delays of the TDNN are organized as a ring-buffer
     e.g they needs a variable for the current number */
  for (layerNo = 0; layerNo < NoOfLayers; layerNo++) {
    layer = (globalLayers + layerNo);

    /* one variable for the current read Position */
    layer->readCounter = malloc( (strlen(layer->name) + 11) *
sizeof (char) );
    if (NULL == layer->readCounter) return(MEM_ERR);
    sprintf(layer->readCounter, "%sReadCounter", layer->name);

    /* and one for the current write position */
    layer->writeCounter = malloc( (strlen(layer->name) + 12) *
sizeof (char) );
    if (NULL == layer->writeCounter) return(MEM_ERR);
    sprintf(layer->writeCounter, "%sWriteCounter", layer->name);

    /* write buffer-variables and their initialisation */
    fprintf(fOutFile, "  static int %s = %d, %s = %d; \n",
         layer->readCounter,  layer->TotalDelay - layer->SuccDelay,
         layer->writeCounter, layer->TotalDelay - 1);
  }
  /* a counter for the pattern is also needed */
  fprintf(fOutFile, "  static int Pattern_counter = 0;\n");
     /* a universal variable for units */
     fprintf(fOutFile, "  pUnit unit;\n");

  /* write the Units, their weights and biases */
  writeAllUnits(globalUnits, NoOfUnits, fOutFile);
```

```
  /** write  Layers e.g. Member of the layers **/
  fprintf(fOutFile,
          "\n  /* layer definition section (names & member units)
*/\n");
  for (nr = 0; nr < NoOfLayers; nr++) {
    writeLayer( (globalLayers + nr), fOutFile);
  }
  /* the Output list may be treated as a layer */
  fprintf(fOutFile, "\n  static int Output[%d] = ",
NoOf(OutList) );
  writeList(OutList, fOutFile);
  fprintf(fOutFile, ";\n\n");

  /** last not least the Update-Function **/
  layer = globalLayers + order[0];     /* first in order e.g.
Input-Layer */

  fprintf(fOutFile, TdnnFirstTemplate);
  fprintf(fOutFile, TdnnInputTemplate, NoOf(layer->members) );

  for (nr = 1; nr < NoOfLayers; nr++) {
    layerNo = order[nr];   /* update Layers in the right order */
    layer   = globalLayers + layerNo; /* current Layer
*/

      unitNo = element(layer->members, 0); /* Number of the first
member unit */
      unit = (globalUnits + unitNo);
      sourceNo = element(unit->sources, 0); /* Number of the first
source unit */
      unit = searchUnit(sourceNo, globalUnits, &pos);
      source = unit->layer;     /* layer of the first source unit
*/

      fprintf(fOutFile, TdnnTemplate,
           NoOf(layer->members),
           layer->name,
           source->readCounter, layer->delay, source->TotalDelay,
           source->readCounter, source->readCounter, layer->delay,
           source->readCounter,
           source->readCounter, source->TotalDelay,
           source->readCounter,
           source->readCounter, layer->delay, source->TotalDelay,
           source->readCounter, source->TotalDelay,
           layer->writeCounter,  ACT_FUNC_NAMES[layer->ActFunc]);
  }

  fprintf(fOutFile, TdnnOutputTemplate, NoOf(OutList) );
  /* Updating the counters of the ring-buffers */
  for (layer = globalLayers; layer < globalLayers + NoOfLayers;
layer++) {
    fprintf(fOutFile, "  %s = (++%s) %% %d;\n",
         layer->readCounter, layer->readCounter, layer-
```

```
>TotalDelay);
    fprintf(fOutFile, "  %s = (++%s) %% %d;\n",
         layer->writeCounter, layer->writeCounter, layer-
>TotalDelay);
  }

  /** the procedure should also have an end **/
  fprintf(fOutFile, TdnnExitTemplate,
globalLayers[order[0]].TotalDelay);
  fprintf(fOutFile, "}\n");

  /** that's all folks, or in German: "Ende gut, alles gut" **/
     killList(OutList);
     killList(FunctionList);
  return(OK);
}

int writeNet(pLayer globalLayers, pUnit globalUnits,
             int NoOfLayers, int NoOfUnits,
             int *order, char *OutFile, char *ProcName)
{
  pLayer layer;
  pUnit  unit;
  int    nr, layerNo, maxSource = 0, maxFeature = 0;
  FILE   *fOutFile, *fHeaderFile;
  pList  OutList, FunctionList;
  time_t timer;
  char   HeaderFile[50];

  /* calculate time for the date in the header of the output file*/
  time(&timer);

  /* Preparations for the Header-File */
  strcpy(HeaderFile, OutFile);
  HeaderFile[strlen(HeaderFile) - 1] = 'h';

  /* prepare Output Stream */
  fOutFile = fopen(OutFile, "w");
  if (!fOutFile) return(CANT_OPEN);

  /* maxSource is needed for writing the Unit-array */
  for(unit = globalUnits; unit < globalUnits + NoOfUnits; unit++) {
    maxSource  = MAX(maxSource, NoOf(unit->sources) );
    maxFeature = MAX(maxFeature, unit->FeatureWidth);
  }

  /* Net-Output may consist of several (output) Layers     */
  /* so there must be an extra merge-list : OutList        */
  /* FunctionList will contain a set of all used functions */
  OutList = newList();
  if (!OutList) return (MEM_ERR);
  FunctionList = newList();
  if (!FunctionList) return (MEM_ERR);
```

```
  for(nr = 0; nr < NoOfLayers; nr ++) {
    layer = globalLayers + nr;
    if( layer->type == OUTPUT) {
      if (mergeList(OutList, layer->members) ) return (MEM_ERR);
    }
    if (addList(FunctionList, layer->ActFunc) ) return(MEM_ERR);
  }

  /** find Names for the Layers **/
  if (NameLayers(globalLayers, NoOfLayers)) return(MEM_ERR);


  fprintf(fOutFile, "#include <math.h>\n#include \"rna_alg.h\"\n",
OutFile, ctime(&timer) );
  for(nr = 0; nr < NoOf(FunctionList); nr ++) {
    fprintf(fOutFile, "%s\n", ACT_FUNCTIONS[element(FunctionList,
nr)]);
  }
  /* to avoid unneeded include-files */
  fprintf(fOutFile, "#define NULL (void *)0\n");

  /* write the Units, their weights and biases */
   writeForwardDeclarationAllUnits(globalUnits, NoOfUnits,
fOutFile);

   if( ! is_CPN_net() && !is_BPTT_net())
    {

     /* write the Units, their weights and biases */
     writeAllSources(globalUnits, NoOfUnits, fOutFile);

     /* write the Units, their weights and biases */
     writeAllWeights(globalUnits, NoOfUnits, fOutFile);
    }
   if( !is_BPTT_net() )
   {
    /* write the Units, their weights and biases */
    writeAllUnits(globalUnits, NoOfUnits, fOutFile);
   }
  else
  {
    /* write the Units, their weights and biases */
    writeAllUnitsOld(globalUnits, NoOfUnits, fOutFile);
   }

   /* write Procedure-Header */
   fprintf(fOutFile, ProcHeader, ProcName);

  /* DLVQ needs the biggest Scalar Product */
  if(is_DLVQ_net() ) {
    fprintf(fOutFile, "  float maxSum = -1.0;\n");
    fprintf(fOutFile, "  pUnit unit;\n\n");  /* variable needed for
update */
```

```
  }
  /* CounterPropagation needs a winner Unit */
  else if(is_CPN_net() ) {
   fprintf(fOutFile, "  float maxSum = -1.0e30;\n"); /* biggest
Scalar Product        */
   fprintf(fOutFile, "  pUnit winner, unit;\n\n");   /* winner unit
and variable unit */
  }
  else {
    fprintf(fOutFile, "  pUnit unit;\n\n");  /* variable needed for
update */
  }


  /** write  Layers e.g. Member of the layers **/
  fprintf(fOutFile,
          "\n  /* layer definition section (names & member units)
*/\n");
  if (is_DLVQ_net() || is_CPN_net() ) {
    /* DLVQ and CPN don't need the output layer */
    for (nr = 0; nr < NoOfLayers; nr++) {
      if (globalLayers[nr].type != OUTPUT) {
                    writeLayer( (globalLayers + nr), fOutFile);
      }
    }
    fprintf(fOutFile, "\n");
  }
  else {     /* not (DLVQ or CPN) */
    for (nr = 0; nr < NoOfLayers; nr++) {
      writeLayer( (globalLayers + nr), fOutFile);
    }
    /* the Output list may be treated as a layer */
    fprintf(fOutFile, "\n  static int Output[%d] = ", NoOf(OutList)
);
    writeList(OutList, fOutFile);
    fprintf(fOutFile, ";\n\n");
  }

  /* BPTT-nets may be inialised by a flag */
  if (is_BPTT_net() ) {
    fprintf(fOutFile, BpttFirstTemplate, NoOfUnits);
  }

  /** last not least the Update-Function **/
  layer = globalLayers + order[0];     /* first in order e.g.
Input-Layer */

  if (is_BPTT_net() ) {
    fprintf(fOutFile, BpttInputTemplate, NoOf(layer->members) );
  }
  else {
    fprintf(fOutFile, InputTemplate, NoOf(layer->members) );
  }
```

```
  for (nr = 1; nr < NoOfLayers; nr++) {
    layerNo = order[nr];   /* update Layers in the right order */
    layer   = globalLayers + layerNo; /* current Layer
*/

    if (is_DLVQ_net() ) {
      if (layer->type == HIDDEN) {
     fprintf(fOutFile, DlvqTemplate, NoOf(layer->members),
          layer->name);
      }
      /* Output Layer not needed (see DlvqOutputTemplate) */
    }
    else if (is_CPN_net() ) {
      if (layer->type == HIDDEN) {
                     fprintf(fOutFile, CpnTemplate, NoOf(layer-
>members),
                                              layer->name);
      }
      /* Output Layer not needed (see CpnOutputTemplate) */
    }
          else if (is_BPTT_net() ) {
      fprintf(fOutFile, BpttTemplate, NoOf(layer->members),
              layer->name,  ACT_FUNC_NAMES[layer->ActFunc]);
          }
    else if (layer->ActFunc >= ActRbfNumber) {
      fprintf(fOutFile, RbfTemplate, NoOf(layer->members),
              layer->name,  ACT_FUNC_NAMES[layer->ActFunc]);
    }
    else {
      fprintf(fOutFile, NormalTemplate, NoOf(layer->members),
              layer->name,  ACT_FUNC_NAMES[layer->ActFunc]);
    }
  }

  if (is_DLVQ_net() ) {
    fprintf(fOutFile, DlvqOutputTemplate);
  }
  else if (is_CPN_net() ) {
    fprintf(fOutFile, CpnOutputTemplate,
NoOf(globalLayers[2].members));
  }
  else if (is_BPTT_net() ) {
    fprintf(fOutFile, BpttOutputTemplate, NoOf(OutList) );
    fprintf(fOutFile, BpttExitTemplate);
  }
  else {
    fprintf(fOutFile, OutputTemplate, NoOf(OutList) );
  }

  /** the procedure should also have an end **/
  fprintf(fOutFile, "  return(OK);\n");
  fprintf(fOutFile, "}\n");
```

```
  /** that's all folks, or in German: "Ende gut, alles gut" **/
     killList(OutList);
     killList(FunctionList);
  return(OK);
}

int main(int argc, char **argv)
{
  krui_err err;           /* error code of SNNS - krui */
  char     *netname;      /* internal name of the SNNS-network */
  char     NetFileName[200];      /* input file */
  char     CFileName[200];        /* output file */
  pUnit    Units, unit = NULL;    /* all Units and unit variable */
  pLayer   Layers, layer;  /* all Layers and layer variable */
  int      NoOfUnits, NoOfLayers; /* Number of units and layers */
  int      *order; /* array with the order of the sorted layers */
  int      nr, pos;       /* help variables */
  int      error;         /* error code */
  pList    HelpList;      /* needed for exchange */
  char     ProcName[50];    /* function name in the output */

  /* Array with the numbers of the prototype units */
  int      *TDNN_prot;

  /* check Params */
  if (argc < 2) {
    fprintf(stderr, "usage : %s <netfile>\n", argv[0]);
    return(WRONG_PARAM);
  }
  strcpy(NetFileName, argv[1]);
  strcpy(CFileName, "rna_alg.c");
  strcpy(ProcName, "Rna_alg");
  toAlphaNum(ProcName);       /* Function Name must not contain
special chars */
  /* Write a Message (what to do) on screen */
  printf(HeadingTemplate, NetFileName, CFileName, ProcName);
  printf("loading net... \n");

  /* load Net */
  err = krui_loadNet(NetFileName, &netname);
  if (err) {
    fprintf(stderr, "%s\n", krui_error(err) );
    return(CANT_LOAD);
  }

  error = checkLearnFunc();
  if (error) {
    checkErr(error);
    return(error);
  }

  NoOfUnits = krui_getNoOfUnits();
```

```
  Units  =  (pUnit)calloc((NoOfUnits + 1), sizeof(tUnit) );  /*
because of sentinels */
     if (! Units) {
          checkErr(MEM_ERR);
          return(MEM_ERR);
     }

  Layers = (pLayer)calloc((NoOfUnits + 1), sizeof(tLayer) ); /*
because of sentinels */
     if (! Layers) {
          free(Units);
          checkErr(MEM_ERR);
          return(MEM_ERR);
     }

  /* TDNN_prot needs one more Element, because the enumeration
     of the units starts with one */
  TDNN_prot = (int *)malloc((NoOfUnits+1) * sizeof(int) );
  if (! TDNN_prot) {
          free(Units);
          free(Layers);
          checkErr(MEM_ERR);
          return(MEM_ERR);
     }

  for (nr = 0; nr <= NoOfUnits; nr++) {
    Layers[nr].number  = nr;
  }

  printf("dividing net into layers ...\n");

  /* part Net into groups */
  error =  divideNet(Units, Layers, TDNN_prot);
     if (error) {
          checkErr(error);
          FREE_ALL;
          return(error);
     }

  /* count the Non-empty Layers */
  for (nr = 0; Layers[nr].members != NULL; nr++);
  NoOfLayers = nr;
  order = (int *)malloc(NoOfLayers * sizeof(int) );


  /* count the real number of units (e.g. the prototype units in
TDNN) unused units have the number 0 the total Number can't exeed
the Number of Units given by the SNNS-Interface-Function */
  for(nr = 0; (Units[nr].number != 0) && (nr < NoOfUnits); nr++) {
    Units[nr].index = nr;
  }
  NoOfUnits = nr;
```

```
#ifdef debug
  for(nr = 0; nr < NoOfLayers; printLayer(Layers + nr++) );
#endif

  /* the TDNN_units must have a special format */
  if ( is_TDNN_net() ) {
    prepareTDNNunits(Units, TDNN_prot);

    /* some values are passed to the layers for TDNNs */
    for(layer = Layers; layer < Layers + NoOfLayers; layer++) {
      unit = searchUnit(element(layer->members, 0), Units, &pos);
      layer->delay      = unit->DelayLength;
      layer->TotalDelay = NoOf(unit->members);
      layer->SuccDelay  = 0;
      /* only initalisation (see function checkOrder )*/
    }
  }

  /* topological sort of the layers */
  printf("sorting layers ...\n");
  error = sortNet(Layers, NoOfLayers, order);
  if (error) {
    checkErr(error);
    FREE_ALL;
    return(error);
  }

  if ( is_TDNN_net() ) {
    /* Update the entries in the member list of the layer */
    /* now the snns2c-indices are valid                   */

    for(layer = Layers; layer < Layers + NoOfLayers; layer++) {
      HelpList = newList();
      for(nr = 0; nr < NoOf(layer->members); nr++) {
     searchUnit(element(layer->members, nr), Units, &pos);
     addList(HelpList, pos);
      }
      killList(layer->members);
      layer->members = HelpList;
    }
  }
  else if( is_CPN_net() ) {
    /* copy the weights from hidden to output
       units into hidden units */
    error = prepareCpnUnits(Units, Layers);
    if (error) {
      checkErr(error);
      FREE_ALL;
      return(error);
    }
  }
```

```
  printf("writing net ...\n");

  /* write the net as a C-source */
  if (is_TDNN_net() ) {
    error = writeTdnnNet(Layers, Units, NoOfLayers, NoOfUnits,
order, CFileName, ProcName);
    if (error) {
      checkErr(error);
      FREE_ALL;
      return(error);
    }
  }
  else {
    error = writeNet(Layers, Units, NoOfLayers, NoOfUnits, order,
CFileName, ProcName);
    if (error) {
      checkErr(error);
      FREE_ALL;
      return(error);
    }
  }

  FREE_ALL;
  return(0);
}
```

Todo o código fonte demonstrado pode ser encontrado em: http://jacsonrcsilva.googlepages.com/snort-rna