UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MECATRÔNICA

EDER SANTANA FREIRE

UMA ARQUITETURA PARALELA BASEADA NA CODIFICAÇÃO DE HUFFMAN PARA OTIMIZAÇÃO DE MEMÓRIA EM HARDWARE ESPECIALIZADO PARA DETECÇÃO DE INTRUSÃO EM REDES

Salvador
2014

EDER SANTANA FREIRE

UMA ARQUITETURA PARALELA BASEADA NA CODIFICAÇÃO DE HUFFMAN PARA OTIMIZAÇÃO DE MEMÓRIA EM HARDWARE ESPECIALIZADO PARA DETECÇÃO DE INTRUSÃO EM REDES

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Universidade Federal da Bahia, como requisito parcial à obtenção do título de Mestre em Mecatrônica.

Linha de pesquisa: Sistemas Computacionais

Orientador: Prof. Dr. Leizer Schitman
Co-orientador: Prof. Dr. Angelo Amâncio Duarte

Salvador
2014

F866 Freire, Eder Santana.

Uma arquitetura paralela baseada na codificação de Huffman para otimização de memória em hardware especializado para detecção de intrusão em redes. – Salvador, 2014.

117f. : il. color.

Orientador: Prof. Dr. Leizer Schitman.

Co-orientador: Prof. Dr. Angelo Amâncio Duarte.

Dissertação (mestrado) – Universidade Federal da Bahia. Escola Politécnica, 2014.

1. Sistemas Computacionais. 2. Segurança da Informação. 3. Arquitetura paralela. 4. Detecção de intrusão em redes. 5. Field-Programmable Gate Arrays. 6. Otimização de memória. 7. Codificação de Huffman. I. Schnitman, Leizer. II. Duarte, Ângelo Amâncio. III. Universidade Federal da Bahia. IV. Título.

CDD: 004.2

EDER SANTANA FREIRE

UMA ARQUITETURA PARALELA BASEADA NA CODIFICAÇÃO DE HUFFMAN PARA OTIMIZAÇÃO DE MEMÓRIA EM HARDWARE ESPECIALIZADO PARA DETECÇÃO DE INTRUSÃO EM REDES

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Universidade Federal da Bahia, como requisito parcial à obtenção do título de Mestre em Mecatrônica.

Linha de pesquisa: Sistemas Computacionais

Aprovada em 13 de Março de 2014.

Banca Examinadora

Angelo Amâncio Duarte – Co-orientador _____
Doutor em Ciência da Computação pela Universidade Autónoma de Barcelona, UAB, Espanha
Universidade Federal da Bahia

Wagner Luiz Alves de Oliveira _____
Doutor em Engenharia Elétrica pela Universidade Estadual de Campinas, Unicamp, Brasil
Universidade Federal da Bahia

Norian Marranghello _____
Livre-Docente em Sistemas Digitais pela Universidade Estadual Paulista, Unesp, Brasil
Universidade Estadual Paulista

Dedico este trabalho aos meus pais, por todo esforço empreendido, por cada incentivo dado, e pelo apoio irrestrito em todos os momentos da minha vida.

AGRADECIMENTOS

Agradeço em princípio a Deus, criador de todas as coisas, por tudo que Ele tem me proporcionado.

Aos meus pais, Eurides e Levi; e aos meus irmãos, Jéssica e Levi Júnior, pelo amor e carinho incondicionais.

A Rafaela, pelo seu amor, compreensão e apoio, e por todos os momentos presente ao meu lado.

Aos professores Angelo e Leizer, orientadores deste trabalho, pelo conhecimento compartilhado e por terem apostado no projeto mesmo quando ele ainda não passava de uma simples ideia.

Aos professores Wagner e Norian, membros da Banca Examinadora, pelas sábias considerações feitas e importantes contribuições para o enriquecimento desse trabalho.

A todos os colegas da Superintendência de Tecnologia da Informação da UFBA, em especial a Cleidson, Luis Marcos, Saulo e Vagner, pela amizade e suporte demonstrados, inclusive, com a presença na ocasião da defesa desse trabalho; e também a Luiz Cláudio e Claudete, pelo encorajamento contínuo e por todo o investimento feito em minha capacitação durante o período em que atuei profissionalmente nessa instituição.

Aos demais docentes do Programa de Pós-Graduação em Mecatrônica da UFBA, por todos os conhecimentos e valores sabiamente transmitidos.

FREIRE, Eder Santana. Uma arquitetura paralela baseada na codificação de Huffman para otimização de memória em hardware especializado para detecção de intrusão em redes. 117 f. il. 2014. Dissertação (Mestrado) – Programa de Pós-Graduação em Mecatrônica, Universidade Federal da Bahia, Salvador, 2014.

RESUMO

O projeto de *hardware* especializado para detecção de intrusão em redes de computadores tem sido objeto de intensa pesquisa ao longo da última década, devido ao seu desempenho consideravelmente maior, comparado às implementações em *software*. Nesse contexto, um dos fatores limitantes é a quantidade finita de recursos de memória embarcada, em contraste com o crescente número de padrões de ameaças a serem analisados. Este trabalho propõe uma arquitetura baseada no algoritmo de Huffman para codificação, armazenamento e decodificação paralela de tais padrões, a fim de reduzir o consumo de memória embarcada em projetos de *hardware* destinado à detecção de intrusão em redes. Experimentos foram realizados através de simulação e síntese em FPGA de conjuntos de regras atuais do sistema de detecção de intrusão Snort, e os resultados indicaram uma economia de até 73% dos recursos de memória embarcada do *chip*. Adicionalmente, a utilização de uma estrutura paralelizada apresentou ganhos de desempenho significantes durante o processo de decodificação das regras.

Palavras-chave: Sistemas Computacionais. Segurança da Informação. Arquitetura paralela. Detecção de intrusão em redes. Field-Programmable Gate Arrays. Otimização de memória. Codificação de Huffman.

FREIRE, Eder Santana. A parallel architecture based on the Huffman encoding for memory optimization on specialized hardware for network intrusion detection. 117 pp. ill. 2014. Master Dissertation – Programa de Pós-Graduação em Mecatrônica, Universidade Federal da Bahia, Salvador, 2014.

ABSTRACT

The design of specialized hardware for Network Intrusion Detection has been subject of intense research over the last decade due to its considerably higher performance compared to software implementations. In this context, one of the limiting factors is the finite amount of embedded memory resources in contrast to the increasing number of threat patterns to be analyzed. This work proposes an architecture based on the Huffman algorithm for encoding, storage, and parallel decoding of such patterns in order to reduce the embedded memory usage in hardware designs intended for network intrusion detection. Experiments were carried out with simulation and FPGA synthesis of current rule subsets of the Snort intrusion detection system, and the results indicated a saving of up to 73% of the on-chip embedded memory resources. Moreover, the use of a parallelized structure showed significant performance gains during the rules decoding process.

Keywords: Computing Systems. Information Security. Parallel architecture. Network intrusion detection. Field-Programmable Gate Arrays. Memory optimization. Huffman encoding.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASCII *American Standard Code for Information Interchange* (em português: Código Padrão Americano para o Intercâmbio de Informação)

ASIC *Application-Specific Integrated Circuit* (em português: Circuito Integrado de Aplicação Específica)

CAM *Content-Addressable Memory* (em português: Memória Endereçável por Conteúdo)

CLBs *Configuration Logical Blocks* (em português: Blocos Lógicos de Configuração)

DDoS *Distributed Denial of Service* (em português: Negação de Serviço Distribuída)

DNS *Domain Name System* (em português: Sistema de Nomes de Domínios)

DoS *Denial of Service* (em português: Negação de Serviço)

DPI *Deep Packet Inspection* (em português: Inspeção Profunda de Pacotes)

EUA Estados Unidos da América

FPGA *Field-Programmable Gate* Array (em português: Arranjo de Portas Programável em Campo)

FTP *File Transfer Protocol* (em português: Protocolo de Transferência de Arquivos)

Gbps Gigabits por segundo

GRC *Governance, Risk and Compliance* (em português: Governança, Risco e Conformidade)

HDL *Hardware Description Language* (em português: Linguagem de Descrição de *Hardware*)

HIDS *Host-based Intrusion Detection System* (em português: Sistema de Detecção de Intrusão baseado em *Host*)

HTTP *Hypertext Transfer Protocol* (em português: Protocolo de Transferência de Hipertexto)

ICMP *Internet Control Message Protocol* (em português: Protocolo de Mensagens de Controle da Internet)

IDS *Intrusion Detection System* (em português: Sistema de Detecção de Intrusão)

IOB *Input/Output Block* (em português: Bloco de Entrada/Saída)

IP *Internet Protocol* (em português: Protocolo de Internet)

IPS *Intrusion Prevention System* (em português: Sistema de Prevenção de Intrusão)

kB *Kilobyte* (em português: quilobyte)

MAC *Media Access Control* (em português: Controle de Acesso ao Meio)

Malware *Malicious Software* (em português: *software* malicioso)

MHz Mega-Hertz

ms Milissegundo

NIDS *Network-based Intrusion Detection System* (em português: Sistema de Detecção de Intrusão baseado em Rede)

NIST *National Institute of Standards and Technology* (em português: Instituto Nacional de Padrões e Tecnologia)

NSM *Network Security Monitoring* (em português: Monitoramento de Segurança de Redes)

P2P *Peer to Peer* (em português: Ponto a Ponto)

PDU *Protocol Data Unit* (em português: Unidade de Dados de Protocolo)

RAM *Random Access Memory* (em português: Memória de Acesso Aleatório)

RFC *Request for Comments* (em português: Pedidos de Comentários)

ROM *Read-Only Memory* (em português: Memória de Apenas Leitura)

SIEM *Security Information and Event Management* (em português: Gerenciamento e Correlação de Eventos de Segurança)

SMTP *Simple Mail Transfer Protocol* (em português: Protocolo Simples de Transferência de Correio)

SNMP *Simple Network Management Protocol* (em português: Protocolo Simples de Gerenciamento de Rede)

SSH *Secure Shell* (em português: Terminal Seguro)

TCP *Transmission Control Protocol* (em português: Protocolo de Controle de Transmissão)

UDP *User Datagram Protocol* (em português: Protocolo de Datagrama do Usuário)

VHDL *VHSIC Hardware Description Language* (em português: Linguagem de Descrição de *Hardware* VHSIC)

VHSIC *Very High Speed Integrated Circuit* (em português: Circuito Integrado de Velocidade Muito Alta)

VPN *Virtual Private Network* (em português: Rede Privada Virtual)

SUMÁRIO

.

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

À medida que os sistemas e as redes computacionais se desenvolvem e se tornam cada vez mais populares – a exemplo da própria Internet, cresce a demanda por soluções e serviços que tragam benefícios à sociedade contemporânea. Da mesma forma, cresce a quantidade de ameaças à segurança das informações que trafegam nas redes computacionais, resultando em prejuízos de grande vulto às organizações e, principalmente, aos usuários comuns dessas tecnologias. Entre os principais prejuízos figuram a violação à privacidade, o roubo de informações, o dano à imagem e as perdas financeiras.

De acordo com o estudo anual sobre crimes virtuais com foco nos usuários divulgado pela Symantec Corporation (2013), no ano analisado, as perdas financeiras diretamente relacionadas a crimes virtuais no Brasil atingiram a marca de 8 bilhões de dólares, o que representa cerca de 7% do prejuízo de mais de US$ 113 bilhões calculado em todo o mundo no mesmo período (Figura 1) – valor suficiente para cobrir os custos dos Jogos Olímpicos de Londres em 2012 por quase dez vezes. Ainda de acordo com esse estudo, cerca de 60% dos brasileiros que fazem uso da Internet (isto é, 22 milhões de pessoas) foram vítimas de algum crime virtual num período de 12 meses. O número mundial de vítimas, por sua vez, atingiu os 378 milhões, o que representa uma média de 12 vítimas por segundo. Além disso, o prejuízo médio global por vítima aumentou de US$ 197 em 2012 para US$ 298 em 2013, o que representa um crescimento de mais de 50%.

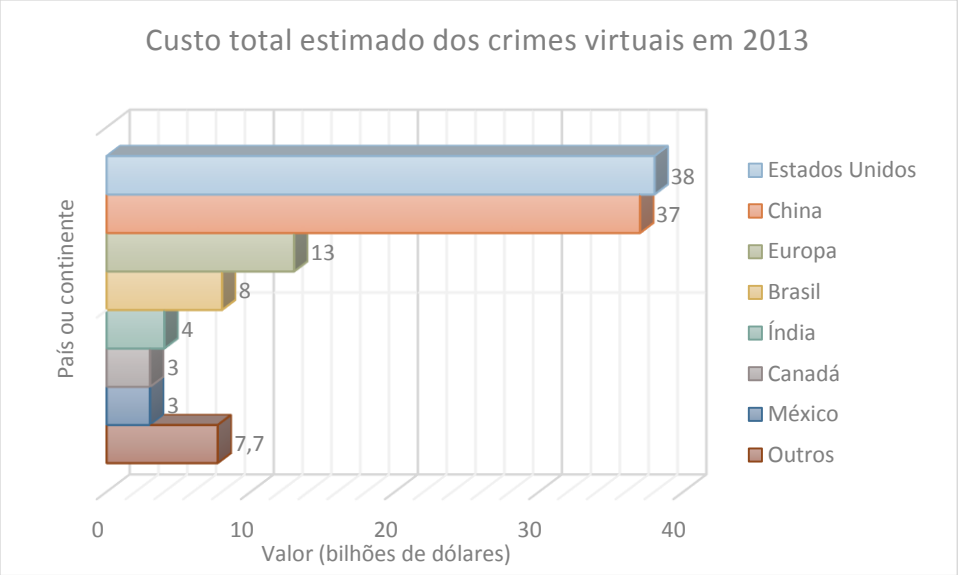


Figura 1 – Custo total estimado dos crimes virtuais em 2013.

Fonte: Symantec Corporation (2013).

Um outro estudo, patrocinado pela Hewlett-Packard Enterprise Security Products e conduzido pelo Ponemon Institute (2013), visando estimar o impacto financeiro dos crimes virtuais nas organizações dos Estados Unidos da América, demonstrou que o prejuízo médio anual por organização chegou a US$ 11,56 milhões em 2013, o que representa um aumento de 78% desde que a primeira edição do estudo foi publicada, em 2010. Esse estudo também revelou que o tempo necessário para solucionar um ataque virtual aumentou em quase 130% durante o mesmo período, com o custo médio para solução de um único ataque ultrapassando o patamar de US$ 1 milhão. A Figura 2 apresenta a evolução do prejuízo médio causado pelos crimes virtuais nas organizações estadunidenses, ao longo dos últimos quatro anos.

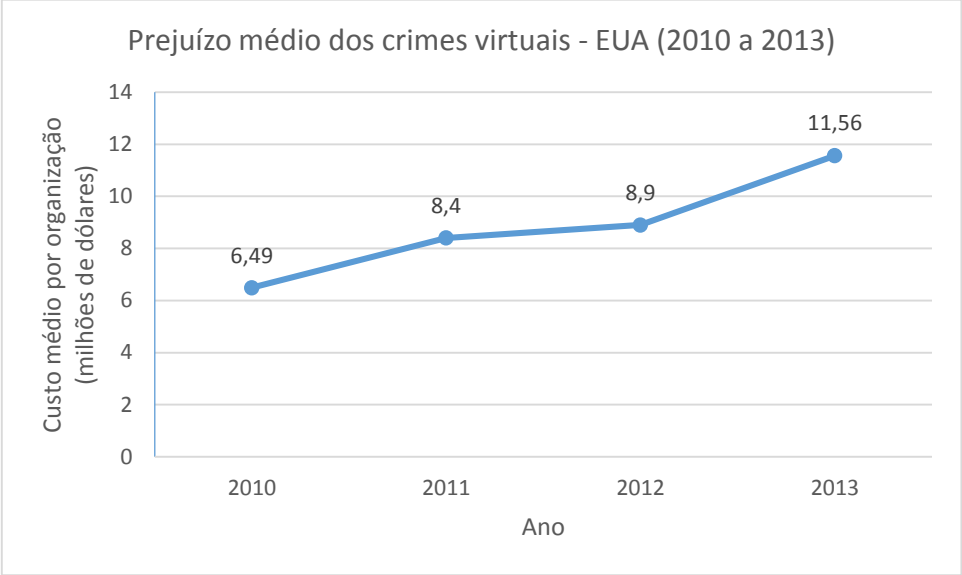


Figura 2 – Evolução do prejuízo médio anual dos crimes virtuais (organizações estadunidenses).

Fonte: Ponemon Institute (2013).

Ainda de acordo com o Ponemon Institute (2013), as três atividades internas relacionadas aos crimes virtuais que mais demandaram custos para as organizações nesse ano foram: detecção (21% dos custos), contenção (17%) e recuperação de incidentes de segurança (28%). Entre as atividades maliciosas que trouxeram maior prejuízo, figuram os ataques de negação de serviço (em inglês: *Denial of Service* – DoS), a disseminação de códigos maliciosos (*malwares*) e os ataques baseados na *web*. Combinadas, essas atividades foram responsáveis por mais de 55% de todos os custos relacionados aos crimes virtuais no período mencionado.

Em um ataque de negação de serviço, os serviços disponibilizados em uma rede de computadores são intencionalmente sobrecarregados de tal forma que se tornam indisponíveis, constituindo a principal forma de ataque direcionado. Ataques de negação de serviço distribuídos (em inglês: *Distributed Denial of Service* – DDoS) são uma evolução dos ataques DoS tradicionais, e podem causar danos muito mais significativos. Tais ataques são executados remotamente por um atacante que, após comprometer diversos computadores e

recrutá-los em um exército “zumbi”, os utiliza para direcionar ataques massivos para alvos específicos (PENG, LECKIE e RAMAMOHANARAO, 2007).

De uma forma geral, um computador é transformado em “zumbi” a partir da sua infecção por uma espécie de código malicioso conhecido como *worm*, que se espalha rapidamente pelas redes vulneráveis. Como exemplo, pode-se citar o *worm* Sapphire, também conhecido como Slammer, que em 2003 se espalhou por 75 mil computadores em apenas 10 minutos, sendo considerado o *worm* mais rápido criado até então, dobrando o número de infecções em apenas 8,5 segundos (MOORE *et al.*, 2003). Outro exemplo de destaque é o CodeRed, um *worm* que em 2001 infectou 359 mil computadores em 14 horas de operação (Figura 3), e desencadeou ataques DoS para diversos endereços IP específicos, tais como o da Casa Branca dos Estados Unidos da América (MOORE e SHANNON, 2013).

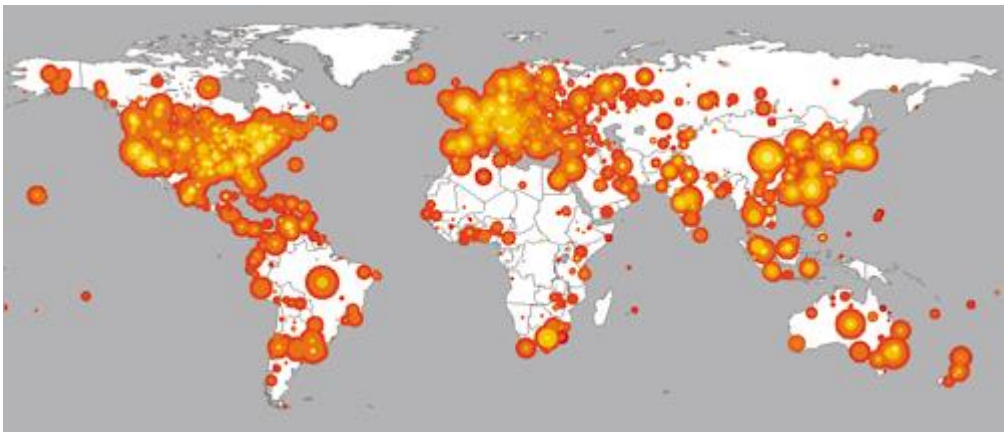


Figura 3 – Mapa de infecção do *worm* CodeRed.
Fonte: Moore (2013).

Em seu relatório sobre o custo dos crimes virtuais, o Ponemon Institute (2013) aponta que a mitigação desses tipos de ataque requer o uso de tecnologias que agreguem maior segurança às operações das organizações, tais como: Gerenciamento e Correlação de Eventos de Segurança (em inglês: *Security Information and Event Management* – SIEM); soluções de teste de

segurança de aplicações; soluções corporativas de Governança, Risco e Conformidade (em inglês: *Governance, Risk and Compliance* – GRC); e Sistemas de Detecção e Prevenção de Intrusão.

Diante do atual cenário apresentado, no qual novas ameaças emergem constantemente, cresce a necessidade de se conceber mecanismos de segurança capazes de garantir a segurança e operação continuada dos sistemas e redes computacionais. Esta situação leva a uma demanda por sistemas automatizados para detecção de atividades maliciosas tanto em sistemas individuais como em redes de computadores.

Sistemas de Detecção de Intrusão baseados em Rede (em inglês: *Network-based Intrusion Detection Systems* – NIDS) têm por finalidade detectar e prevenir diversas ameaças de segurança ao tráfego das redes de computadores. O seu funcionamento baseia-se na comparação dos pacotes que trafegam pela rede com padrões de ataque previamente conhecidos ou comportamentos anômalos, permitindo a detecção e a contenção de tráfego prejudicial e, portanto, indesejado.

A literatura mostra que não é possível e nem necessário manter a estrita segurança em todos os pontos de uma rede de computadores. Isto é, desde que os componentes ou áreas pertinentes de uma rede estejam devidamente protegidos, toda a sua infraestrutura pode ser considerada segura (CHEN, CHEN, e SUMMERVILLE, 2011). Dessa forma, os NIDS normalmente são implantados próximos ao perímetro da rede, conectados a um equipamento de *firewall* ou roteador de borda, de modo que possam monitorar todo o seu tráfego de entrada e/ou saída, conforme exemplificado pela Figura 4.

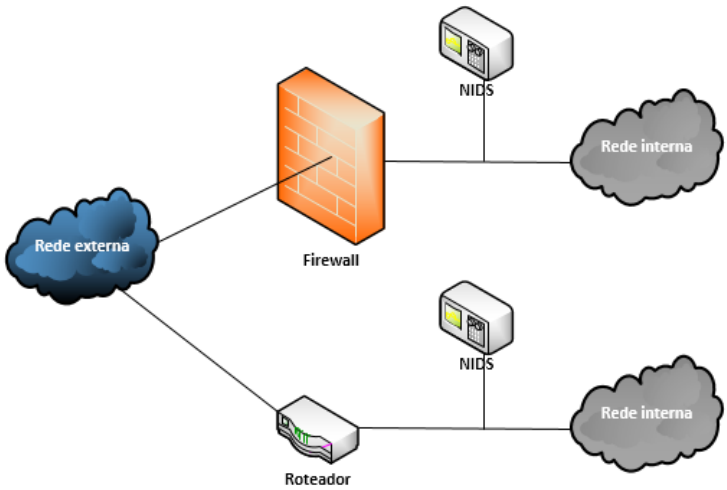


Figura 4 – Esquema simplificado de funcionamento de um NIDS.

Conforme descrito por Kong *et al.* (2009), soluções NIDS implementadas em *software* se tornaram bastante populares devido à sua facilidade de uso, configuração e atualização. Exemplos desse tipo de solução são os *softwares* de código aberto Snort (2014) e Suricata (2014), que têm como principal objetivo examinar todo o tráfego de uma rede, registrando os eventos de potenciais ataques ou intrusões. Estas ferramentas são alimentadas por regras de detecção que possuem sintaxe própria, as quais são normalmente armazenadas em arquivos de texto codificados no padrão ASCII.

Um dos problemas encontrados pelas soluções NIDS baseadas apenas em *software* é o crescente número de ameaças às redes de computadores e, consequentemente, da quantidade de regras de detecção necessárias. Tomando como exemplo o caso do Snort, o número de regras de detecção passou de aproximadamente 1.700 no ano 2003 para mais de 20.000 em Setembro de 2013, representando um crescimento superior a 1.000% em um período de dez anos (SEN, 2006; SNORT, 2014). Diante desse acentuado crescimento, as abordagens baseadas puramente em *software* passaram a encontrar gargalos de processamento nos últimos anos, visto que, com o aumento das taxas de transferência nas novas tecnologias de rede, tornou-se

necessário detectar um número cada vez maior de comportamentos suspeitos em um período de tempo menor.

Conforme apontado por Chen, Chen e Summerville (2011), a implementação de soluções de segurança em *hardware* tem se tornado uma tendência, à medida que a evolução do poder computacional das arquiteturas baseadas em processadores de propósito geral não vem acompanhando a demanda cada vez maior pelo processamento e detecção de padrões de ameaça, o que torna insuficiente o desempenho de soluções NIDS puramente baseadas em *software*.

A implementação de NIDS diretamente em *hardware* surge, portanto, como uma solução para essa questão de déficit de processamento, tendo sido objeto de intensa pesquisa desde o início da última década, devido à sua capacidade de comparação e detecção de padrões de ameaças consideravelmente superior em relação às implementações em *software* e, principalmente, devido à possibilidade de utilização de processamento paralelo nativo. Nesse contexto, destacam-se os dispositivos reconfiguráveis tais como os *Field-Programmable Gate Arrays* – FPGAs (em português: Arranjos de Portas Programáveis em Campo), utilizados pela maioria dos trabalhos publicados nessa área, em razão da sua rapidez e flexibilidade de projeto, implementação e simulação de arquiteturas de lógica digital (CHO, NAVAB e MANGIONE-SMITH, 2002; CHEN, CHEN e SUMMERVILLE, 2011).

Após a análise dos trabalhos publicados a respeito desse tema de pesquisa, foi possível perceber uma tendência de projeto de sistemas otimizados em *hardware* voltados para a verificação e detecção de padrões de ameaças que atendam a altas taxas de transferência (acima de 1 Gbps), tais como aqueles propostos por Loinig, Wolkerstorfer e Szekely (2007) e Yusuf *et al.* (2005). No entanto, em poucas das publicações analisadas durante a pesquisa percebeu-se o interesse específico de se conceber métodos para otimizar o consumo de memória embarcada dos dispositivos de *hardware* diante do crescente número de ameaças às redes computacionais e, consequentemente, das regras de detecção de intrusão.

É importante ressaltar que, embora seja possível a utilização de memórias externas – e de menor custo – na implementação de um *hardware* dedicado para detecção de intrusão em redes, há diversos fatores a favor do uso de memórias embarcadas. Tal como descrito por Zhang (2009), alguns desses fatores são: tempo de acesso reduzido, menor consumo de energia, maior integração e aumento da confiabilidade do dispositivo.

1.2 OBJETIVOS

O presente trabalho tem por objetivo principal apresentar uma proposta de arquitetura computacional paralela, integrando software e *hardware*, baseada no algoritmo de Huffman (1952) para codificação, armazenamento e decodificação de regras de detecção de comportamentos suspeitos em redes de computadores. Esta arquitetura foi concebida de modo que pudesse ser utilizada em abordagens de NIDS implementados em *hardware*, visando a otimização dos recursos de memória embarcada. Adicionalmente, é proposto um método para armazenamento dos bits das regras codificadas em um arranjo contíguo, de modo a reduzir o desperdício causado por espaços não utilizados nas posições de memória.

Entre os objetivos específicos desse trabalho, figuram:

a) A pesquisa abrangente sobre a atividade de detecção de intrusão em redes computacionais, suas principais funções, tecnologias, métodos de detecção e escopos de atuação;

b) A investigação a respeito da implementação de NIDS através de *hardware*, das principais abordagens apresentadas, e do uso de técnicas anteriormente propostas para a otimização dos recursos de memória embarcada;

c) A proposição, a implementação e a verificação de uma arquitetura paralela baseada no algoritmo de Huffman, visando a otimização dos recursos de memória embarcada em projetos de NIDS implementados em *hardware*.

d) A análise e avaliação da eficácia e eficiência da arquitetura proposta, através da execução do *software* e da simulação e síntese do *hardware* implementados.

1.3 ESTRUTURA DO TRABALHO

O restante desse texto está organizado conforme descrito a seguir. O capítulo 2 introduz os fundamentos teóricos necessários para a compreensão da proposta e os trabalhos relacionados ao aqui descrito. No capítulo 3 é apresentada a arquitetura proposta para otimização dos recursos de memória por meio da codificação, do rearranjo e da decodificação de regras de detecção de intrusão. No capítulo 4 são demonstrados os experimentos realizados e a análise dos resultados obtidos. Por fim, no capítulo 5 são apresentadas as considerações finais e possibilidades de trabalhos futuros.

2 FUNDAMENTOS TEÓRICOS

2.1 SISTEMAS DE DETECÇÃO E PREVENÇÃO DE INTRUSÃO

De acordo com o Guia para Sistemas de Detecção e Prevenção de Intrusão do Instituto Nacional de Padrões e Tecnologia dos Estados Unidos da América (em inglês: *National Institute of Standards and Technology* – NIST) (SCARFONE e MELL, 2007), Detecção de Intrusão é o processo de monitoramento dos eventos ocorridos em um sistema ou rede de computadores, assim como a sua análise em busca de indícios de possíveis incidentes de segurança. Tais incidentes podem ser a violação ou a ameaça iminente de violação das políticas de segurança da informação, políticas de uso aceitável, ou práticas e padrões de segurança da informação adotadas por determinada organização.

Incidentes de segurança podem ter muitas causas diferentes, tais como infecção por *malwares* (por exemplo: vírus, *worms* e *spywares*), o acesso não autorizado a sistemas por atacantes a partir da Internet, e usuários de sistemas que fazem mau uso dos seus privilégios de acesso ou tentam obter privilégios adicionais para os quais não estão autorizados. Embora muitos incidentes de segurança sejam maliciosos por natureza, boa parte deles ocorre de forma não intencional – como, por exemplo, a tentativa acidental de acesso não autorizado a determinado sistema, por parte de um usuário que tenha digitado um endereço incorretamente.

Sistemas de Detecção de Intrusão (em inglês*: Intrusion Detection Systems* – IDS) são ferramentas implementadas em *software* ou *hardware*, destinadas a automatizar o processo de detecção de intrusão de ameaças. A sua principal função é monitorar a rede ou um *host* específico, a fim de alertar os seus

administradores a respeito de comportamentos suspeitos identificados, sendo por isso chamados de sistemas passivos.

Sistemas de Prevenção de Intrusão (em inglês: *Intrusion Prevention Systems* – IPS), por sua vez, são um tipo específico de IDS que, além de possuírem capacidade de detecção, podem responder automaticamente aos possíveis incidentes de intrusão, por meio do bloqueio do seu tráfego, por exemplo, e por isso são chamados de sistemas reativos. No entanto, devido à exigência da manutenção de taxas de transferência cada vez maiores nas redes computacionais, é comum que a funcionalidade de prevenção das ferramentas IPS seja desabilitada em ambientes críticos, visando a obtenção de ganhos de desempenho, de modo que funcionem como IDS convencionais.

Devido às similaridades entre ambas as abordagens mencionadas e para fins de simplificação, o termo IDS, mais abrangente, será o adotado ao longo do restante desse trabalho para se referir a ambos os casos, a menos que seja explicitamente mencionado de outra forma.

Os sistemas IDS possuem como foco principal a identificação de possíveis incidentes de segurança como, por exemplo, a detecção de um sistema comprometido por um atacante devido à exploração de uma vulnerabilidade existente. Uma vez que o incidente de segurança tenha sido detectado, ele poderá ser reportado pelo IDS aos administradores do sistema, de modo que ações de resposta ao incidente possam ser rapidamente iniciadas para minimizar os danos causados. Estes sistemas também são capazes de registrar informações que podem ser úteis para o tratamento de incidentes, e podem ser configurados para reconhecer automaticamente eventuais violações às políticas de segurança ou de uso aceitável da organização. Além disso, alguns IDS também podem monitorar transferências de arquivos e identificar aquelas que podem ser suspeitas, tais como a cópia de uma extensa base de dados de um servidor para o computador pessoal de um usuário, por exemplo.

Adicionalmente, os sistemas IDS podem ser utilizados para outros propósitos, tais como: a identificação de problemas com políticas de segurança;

a documentação de ameaças existentes; e a dissuasão de indivíduos que possuam a intenção de violar as políticas de segurança. Desta forma, tais sistemas se tornaram um recurso valioso necessário para a segurança da infraestrutura de rede de praticamente todas as organizações.

2.1.1 Principais funções das tecnologias IDS

Existem diversos tipos de tecnologias IDS, que por sua vez se diferenciam principalmente pelos tipos de eventos que elas podem reconhecer e os métodos que elas utilizam para identificar os incidentes. Além das atividades de monitoramento e análise de eventos para identificação de atividades suspeitas, todos os tipos de tecnologias IDS normalmente executam as seguintes funções:

- Registro das informações relacionadas aos eventos observados – a informação geralmente é gravada localmente, e pode também ser enviada para sistemas separados, tais como servidores centralizados de registro de *logs*, soluções para Gerenciamento e Correlação de Eventos de Segurança (em inglês: *Security Information and Event Management* – SIEM), e sistemas de gerenciamento corporativo;
- Notificação aos administradores de segurança a respeito dos eventos observados – essa notificação, conhecida como alerta, ocorre por meio de diversos métodos diferentes, incluindo os seguintes: *e-mails*, páginas eletrônicas, mensagens na interface de usuário do IDS, *traps* do Protocolo Simples de Gerenciamento de Rede (em inglês: *Simple Network Management Protocol* – SNMP), mensagens no padrão Syslog, além de *scripts* e programas definidos pelo usuário. Uma mensagem de notificação normalmente inclui apenas informações básicas sobre um

evento, de forma que os administradores precisam acessar a interface do IDS para obter informações adicionais; e

- Geração de relatórios – essa funcionalidade resume os eventos monitorados ou fornecem detalhes sobre eventos específicos de interesse da equipe de segurança.

De acordo com o NIST (SCARFONE e MELL, 2007), alguns IDS também são capazes de modificar automaticamente o seu perfil de segurança quando uma nova ameaça é detectada. Por exemplo, um IDS pode ser capaz de coletar informações mais detalhadas a respeito de uma sessão de rede específica, após uma atividade maliciosa ter sido detectada nessa sessão. O sistema pode também modificar as configurações para quando certos alertas forem disparados, de qual prioridade deve ser atribuída a alertas subsequentes após uma ameaça em particular ter sido detectada.

2.1.2 Métodos de detecção de intrusão

Os IDS utilizam diversos métodos de detecção com o propósito de identificar potenciais incidentes de segurança, e a maioria desses sistemas utiliza mais de um método, de forma separada ou integrada, para proporcionar um processo de detecção mais amplo e preciso. A seguir são descritos os três principais métodos de detecção de intrusão: detecção baseada em assinatura, detecção baseada em anomalia e análise de protocolo por estado.

2.1.2.1 Detecção baseada em assinatura

Uma assinatura pode ser compreendida como um padrão que corresponde a uma ameaça conhecida. Detecção baseada em assinatura, por sua vez é o processo de comparação das assinaturas com eventos observados para identificar possíveis incidentes de segurança. O NIST (SCARFONE e MELL, 2007) cita alguns exemplos de assinaturas comuns, tais como:

- Uma tentativa de conexão remota a um servidor através do protocolo SSH (*Secure Shell*, em português: Terminal Seguro) com o usuário *root*, o que pode ser caracterizado como uma violação à política de segurança de uma organização;
- Uma mensagem de correio eletrônico com o assunto “Nossas fotos!”, e com um arquivo em anexo de nome “nossasfotos.jpg.exe”, que são características de uma forma conhecida de *malware*; e
- Um registro de *log* de um sistema operacional com o valor do código de *status* igual a 645, o que indica que o serviço de auditoria foi desabilitado nesse sistema.

O método de detecção de intrusão baseada em assinatura é bastante eficaz para detecção de ameaças previamente conhecidas, mas ineficaz para detecção de ameaças ainda desconhecidas (chamadas de *zero-day*), ou ameaças disfarçadas através do uso de técnicas de camuflagem. Por exemplo, se um atacante modificar o nome do arquivo malicioso mencionado no exemplo anterior para “suasfotos.jpg.exe”, uma assinatura em busca do padrão “nossasfotos.jpg.exe” não iria detectá-lo.

Conforme descrito por Foster (2005), a atividade de detecção de intrusão por assinatura envolve a análise e a comparação do tráfego de rede com uma série de bytes ou sequências de pacotes conhecidos por serem maliciosos. Uma vantagem importante desse método de detecção se deve ao fato de as

assinaturas serem fáceis de desenvolver e compreender, uma vez que se conheça qual comportamento de rede se pretende identificar. Por exemplo, é possível utilizar uma assinatura que busca por sequências de caracteres (*strings*) específicas dentro do *payload* (isto é, da carga útil) de um pacote para detectar ataques que tentam explorar uma vulnerabilidade específica de estouro de *buffer* (FOSTER, 2005). Os eventos gerados por um IDS baseado em assinatura podem informar a causa do alerta. Além disso, a comparação de assinaturas pode ser realizada com desempenho razoável em sistemas computacionais modernos, de modo que a quantidade de processamento necessário para desempenhar essas comparações é aceitável, desde que sejam utilizados conjuntos de regras restritos. Por exemplo, se os sistemas a serem protegidos se comunicam apenas através dos serviços de DNS (*Domain Name System*, em português: Sistema de Nomes de Domínios), ICMP (*Internet Control Message Protocol*, em português: Protocolo de Mensagens de Controle da Internet) e SMTP (*Simple Mail Transfer Protocol*, em português: Protocolo Simples de Transferência de Correio), todas as demais assinaturas podem ser removidas do conjunto de regras do IDS.

Os mecanismos de detecção baseados em assinatura também possuem desvantagens. Visto que esses detectam apenas ataques conhecidos, uma assinatura precisa ser criada para cada nova ameaça, de forma que ataques desconhecidos não podem ser detectados. Mecanismos baseados em assinatura também são propensos a gerar falsos positivos, uma vez que eles são comumente baseados na comparação de expressões regulares e sequências de caracteres. Conforme destaca Foster (2005), a detecção de intrusão baseada em assinatura resume-se a uma “corrida armamentista” entre os atacantes e os desenvolvedores de assinaturas de intrusão, onde o delta (isto é, a diferença entre os seus poderes) é a velocidade com que novas assinaturas podem ser desenvolvidas e aplicadas ao mecanismo de detecção.

2.1.2.2 Detecção baseada em anomalia

Detecção baseada em anomalia, por sua vez, é o processo de comparação dos eventos observados em um sistema ou rede com uma base de atividades previamente definidas como normais, visando a identificação de desvios significativos (SCARFONE e MELL, 2007).

Um IDS que utiliza detecção baseada em anomalia possui perfis que representam o comportamento normal de objetos tais como usuários, *hosts*, conexões de rede ou aplicações. Estes perfis são criados através do monitoramento das características de atividade normal ao longo de um período de tempo. Por exemplo, um perfil para uma determinada rede pode ser definido de modo que o tráfego *web* legítimo compreende uma média de 30% de toda a largura de banda da rede durante o horário de trabalho em um dia útil. O IDS, em seguida, utiliza métodos estatísticos para comparar as características da atividade em curso com os limiares estabelecidos para o perfil, sendo capaz de detectar quando a atividade *web* compreende significativamente mais largura de banda do que o esperado, alertando os administradores da rede a respeito da anomalia. Perfis podem ser criados para muitos atributos comportamentais, tais como o número de e-mails enviados por um usuário, o número de tentativas de *login* sem sucesso em um sistema, e o nível de uso de recursos como processador e memória de um servidor em um determinado período de tempo.

De acordo com Foster (2005), a técnica de detecção de anomalia se baseia no conceito de uma linha de base para o comportamento da rede. Essa linha de base é uma descrição do comportamento aceitável da rede, que é aprendido ou especificado pelos seus administradores, ou ambos. Em um mecanismo de detecção de anomalias, os eventos são causados por qualquer comportamento que não se enquadre no modelo de comportamento previamente definido ou aceito.

A principal vantagem da detecção baseada em anomalia sobre os mecanismos baseados em assinatura é o fato de ser possível detectar uma ameaça recente para a qual ainda não existe uma assinatura, caso ela não se enquadre nos padrões normais de tráfego (SCARFONE e MELL, 2007). Como exemplo, pode-se citar o processo de detecção de *worms* automatizados que, após se instalarem nos sistemas hospedeiros, geralmente iniciam a varredura por outros sistemas vulneráveis a uma taxa acelerada ou anormal, inundando a rede com tráfego malicioso, acionando uma regra de comportamento anormal. Além disso, uma vez que um protocolo tenha sido construído e o seu comportamento tenha sido definido, o mecanismo pode ser dimensionado mais rapidamente e facilmente do que o modelo baseado em assinatura, pois não há necessidade de criação de uma nova assinatura para cada ataque ou variante em potencial que venha a surgir (FOSTER, 2005).

Entre as desvantagens dos mecanismos de detecção baseados em anomalia, está a dificuldade de se definir os perfis de comportamento normal, visto que cada perfil de comportamento precisa ser definido, implementado e testado quanto à sua precisão. O processo de definição das regras é agravado pelas diferenças de implementação dos vários protocolos pelos fornecedores, e a análise de protocolos personalizados em uso na rede requer um grande esforço adicional (SCARFONE e MELL, 2007). Adicionalmente, o conhecimento detalhado a respeito do comportamento normal da rede precisa ser construído antes de ser transferido para a memória do mecanismo de detecção de intrusão, a fim de que esta ocorra corretamente.

Outro inconveniente da detecção baseada em anomalia é o fato de que atividades maliciosas que estejam dentro dos padrões aceitáveis de uso da rede não são detectadas. Conforme exemplificado por Foster (2005), um ataque de passagem de diretório em um servidor vulnerável, cujo tráfego esteja em conformidade com o padrão aceitável de uso, facilmente passará despercebido, uma vez que não acionará sinalizador algum de inconformidade, carga de processamento ou consumo de banda no mecanismo de detecção de anomalia. Conforme destacado por Karthikeyan e Indra (2010), outra característica

negativa desse tipo de detecção é o fato de ele possibilitar a geração de uma grande quantidade de falsos positivos, devido à ausência de total previsibilidade dos comportamentos dos usuários e das redes de computadores, sobretudo em ambientes mais dinâmicos e heterogêneos.

2.1.2.3 Análise de protocolo por estado

Análise de protocolo por estado é o processo de comparação dos eventos observados nas redes de computadores com perfis predeterminados de definições geralmente aceitas de atividades de protocolo benignas para cada estado do protocolo, a fim de se identificar prováveis desvios (SCARFONE e MELL, 2007). Ao contrário da detecção baseada em anomalia, que utiliza perfis específicos criados para um *host* ou uma rede, esse tipo de análise se baseia em perfis universais desenvolvidos pelos fornecedores que especificam como determinados protocolos devem ou não ser utilizados. O termo “por estado”, nesse caso, significa que o IDS é capaz de compreender e rastrear o estado dos protocolos das camadas de Rede, Transporte e Aplicação que possuem noção de estado.

Como exemplo desse tipo de análise, pode-se citar o uso do protocolo FTP (*File Transfer Protocol*, em português: Protocolo de Transferência de Arquivos), cuja sessão sempre é iniciada no estado não autenticado, no qual apenas alguns comandos podem ser executados, tais como a inserção de credenciais de acesso ou a visualização do arquivo de ajuda. Isto é, uma eventual tentativa de execução de comandos adicionais do protocolo FTP (como para listagem de um diretório) enquanto a sessão permanecer nesse estado, pode indicar um comportamento suspeito. Ao monitorar as requisições com suas respostas correspondentes, e considerando que cada requisição deve possuir uma

resposta previsível, o IDS é capaz de sinalizar para análise todas as respostas que não se enquadram nos resultados esperados (FREDERICK, 2010).

Entre as vantagens do uso da análise de protocolo por estado, Scarfone e Mell (2007) destacam: a capacidade de identificar sequências de comandos inesperados; a incorporação de características de estado do protocolo ao processo de análise convencional; e a verificação de razoabilidade de determinados comandos – como, por exemplo, se o comprimento de determinado argumento está dentro dos limites mínimo e máximo. Como desvantagens dessa técnica de detecção, podem ser citadas: a geração de alto consumo de recursos, devido à complexidade da análise e o *overhead* (isto é, a sobrecarga de dados) envolvido na execução do rastreamento dos estados; a incapacidade de detectar ataques que não violem as características de comportamento geralmente aceitável do protocolo; e a possibilidade de conflito entre os perfis de protocolo usados pelo IDS e a implementação efetiva desses protocolos na rede.

2.1.3 Tipos de sistemas de detecção e prevenção de intrusão

Embora existam diversas formas diferentes de se implementar um sistema IDS, esses sistemas podem ser classificados em categorias distintas, com base nos tipos de eventos que são monitorados e nos escopos em que atuam. O NIST (SCARFONE e MELL, 2007) classifica em quatro as principais categorias de Sistemas de Detecção e Prevenção de Intrusão, descritas a seguir:

1. Detecção e prevenção baseada em *host*: responsável pelo monitoramento das características e dos eventos ocorridos em um único *host*, em busca de atividades suspeitas. Fazem parte dessa categoria os Sistemas de Detecção de Intrusão baseados em *Host* (em inglês: *Host*-

based Intrusion Detection Systems – HIDS). São exemplos dos tipos de características que um HIDS pode monitorar: tráfego de rede (apenas para o *host* específico), utilização de recursos, *logs* de sistemas, processos em execução, atividade das aplicações, acesso e modificação de arquivos, e mudanças nas configurações do sistema e das aplicações. Sistemas HIDS são mais comumente implantados em *hosts* críticos, tais como servidores acessíveis através da Internet e aqueles que contêm informações sensíveis.

2. Detecção e prevenção baseada em rede: responsável por monitorar o tráfego de segmentos de rede ou de dispositivos em particular, e por analisar as atividades da rede e dos protocolos de aplicação de modo a identificar atividades suspeitas. Fazem parte dessa categoria os Sistemas de Detecção de Intrusão baseados em Rede (em inglês: *Network-based Intrusion Detection Systems* – NIDS). Tais sistemas são mais comumente implantados na fronteira entre redes distintas, como próximos a *firewalls* ou a roteadores borda, servidores de VPN (*Virtual Private Network*, em português: Rede Privada Virtual), de acesso remoto e redes sem fio.
3. Detecção e prevenção *wireless*: responsável por monitorar o tráfego das redes sem fio, e analisá-lo de modo a identificar atividades suspeitas envolvendo protocolos próprios desse tipo de rede. Esse tipo de ferramenta não é capaz de identificar atividades suspeitas nas aplicações ou em protocolos de rede de camadas mais altas (por exemplo, protocolos da camada de Transporte TCP e UDP), sendo utilizados no tráfego sem fio. É mais comumente utilizado para monitoramento do tráfego sem fio considerado legítimo dentro de organizações, mas também pode ser implantado em locais onde o acesso não autorizado à rede sem fio pode estar ocorrendo.
4. Análise do comportamento da rede: responsável por examinar o tráfego de rede com o intuito de identificar ameaças que geram fluxos de tráfego anormais, tais como ataques de negação de serviço distribuído (DDoS),

alguns tipos de *malwares* (por exemplo, *worms* e *backdoors*), e violações à política de segurança (por exemplo, uma estação de trabalho sendo utilizada como servidor P2P clandestino na rede). Esse tipo de sistema é mais frequentemente implantado para monitorar tráfegos em redes internas de uma organização, e também são por vezes utilizados onde possam monitorar tráfegos entre redes internas da organização e redes externas (como a Internet, por exemplo).

2.2 VISÃO GERAL DE UMA REDE TCP/IP

O objetivo desta subseção é trazer uma visão geral a respeito da pilha de protocolos TCP/IP, na qual a grande maioria das redes de computadores está baseada – inclusive a própria Internet, e cuja descrição é requisito para compreensão do funcionamento básico de um sistema de detecção e prevenção de intrusão baseado em redes.

A pilha TCP/IP constitui um conjunto de protocolos de comunicação utilizado nas redes de computadores, composta por cinco camadas que trabalham de forma integrada para transmitir dados entre *hosts*, de modo que cada camada é responsável por executar um grupo de tarefas específicas. O seu nome é derivado dos dois protocolos primeiramente definidos para esse padrão, considerados os mais importantes:

- TCP – *Transmission Control Protocol* (em português: Protocolo de Controle de Transmissão); e
- IP – *Internet Protocol* (em português: Protocolo de Internet).

Quando um conjunto de dados é transmitido através de uma rede TCP/IP, esses dados partem da camada de nível mais alto e atravessam as camadas

intermediárias até encontrarem a camada de nível mais baixo da pilha. Nesse processo, cada camada adiciona mais informações à camada imediatamente superior da pilha, isto é, os dados produzidos por uma camada são encapsulados em um *container* maior, pertencente à camada imediatamente inferior. A camada de nível mais baixo, então, é responsável por enviar todo esse conjunto de dados através de meios físicos até que ele chegue ao seu destino. Quando isso ocorre, os dados percorrem o caminho inverso da pilha, partindo da camada de nível mais baixo até a camada de nível mais alto.

É importante destacar que, em algumas publicações, como a própria RFC (*Request for Comments*, em português: Pedidos de Comentários) 1122 – que padronizou as comunicações na Internet (IETF, 1989), é considerada a existência de apenas quatro camadas no modelo TCP/IP. No entanto, visando a melhor compreensão da pilha de protocolos, nesse trabalho foi adotado o modelo de cinco camadas proposto por Kurose e Ross (2012). Nessa publicação, os seus autores descrevem detalhadamente cada uma destas camadas (Figura 5), que são brevemente apresentadas a seguir.

Figura 5 – Camadas da pilha de protocolos TCP/IP.

2.2.1 Camada de Aplicação

Esta camada permite que as aplicações transmitam dados entre um *host* servidor de aplicação e um *host* cliente. Como exemplo de protocolo dessa camada, pode-se citar o HTTP (*Hypertext Transfer Protocol*, em português: Protocolo de Transferência de Hipertexto), que transmite dados entre um servidor *web* e um aplicativo navegador, responsável pela visualização das páginas eletrônicas da *web*. Outros protocolos comuns na camada de Aplicação são o FTP, o SSH e o SMTP. Os dados gerados pela camada de Aplicação são transmitidos para a camada de Transporte, para posterior processamento.

2.2.2 Camada de Transporte

A camada de Transporte é responsável por empacotar os dados para que possam ser transmitidos entre os *hosts*. Esta camada provê serviços orientados e não orientados à conexão para transportar serviços da camada de Aplicação através da rede, e também pode (opcionalmente) garantir a confiabilidade das comunicações. A Unidade de Dados de Protocolo (em inglês: *Protocol Data Unit* – PDU) dessa camada é conhecida como segmento, e os protocolos mais comumente utilizados são o TCP (*Transmission Control Protocol*, em português: Protocolo de Controle de Transmissão) e o UDP (*User Datagram Protocol*, em português: Protocolo de Datagrama do Usuário).

De acordo com o descrito por Kurose e Ross (2012), de um modo geral, a estrutura de um segmento da camada de Transporte é dividida em dois tipos de dados:

1. Cabeçalho, contendo informações de controle de comprimento e formato pré-determinados; e
2. Carga útil (*payload*), que contém dados da camada de Aplicação, com comprimento e formato variáveis.

Conforme ilustrado na Figura 6, cada segmento TCP ou UDP possui um campo de número de porta de origem (*Source port #*) e outro de porta de destino (*Dest port #*), cada um contendo 16 bits. Um desses números está associado com uma aplicação em um sistema servidor, enquanto o outro número está associado à aplicação cliente correspondente, na outra extremidade da conexão. Os sistemas clientes normalmente selecionam qualquer número de porta disponível para uso da aplicação, enquanto sistemas servidores geralmente utilizam um número de porta estático, dedicado a cada aplicação. Detalhes a respeito dos demais campos dos segmentos TCP e UDP podem ser consultados na obra de Kurose e Ross (2012).

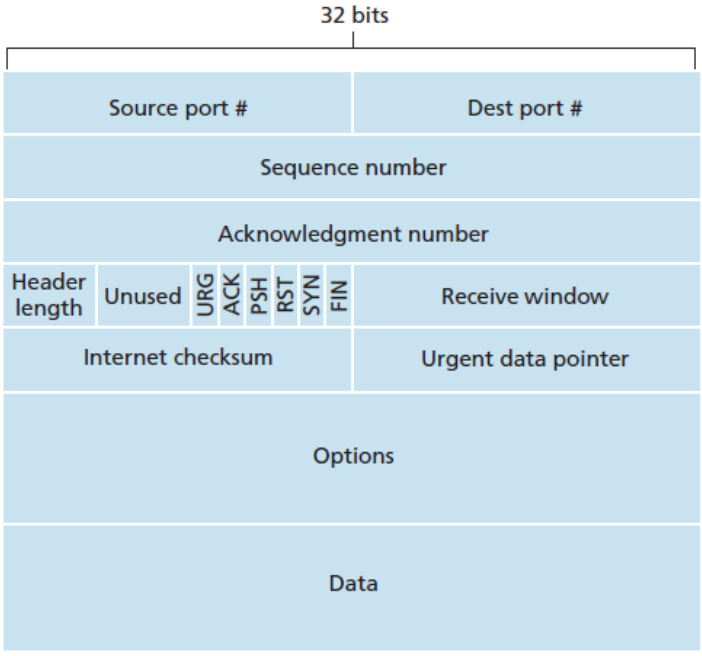


Figura 6 – Estrutura de um segmento TCP.
Fonte: Kurose e Ross (2012).

2.2.3 Camada de Rede

A camada de Rede, também conhecida como camada IP, é responsável por lidar com o endereçamento e roteamento dos dados recebidos pela camada de Transporte. Estes dados, após terem sido encapsulados pela camada de Rede, são conhecidos como pacotes ou datagramas.

De forma análoga à camada de Transporte, cada pacote da camada IP possui um cabeçalho, que é composto por vários campos que especificam as características do protocolo de transporte em uso. Opcionalmente, os pacotes também podem conter uma carga útil (*payload*), onde ficam encapsulados os dados das camadas de Transporte e Aplicação. Entre os diversos campos do cabeçalho IP, destacam-se os seguintes para a atividade de detecção de intrusão em redes:

- *Version* – campo de 4 bits que indica qual versão do protocolo IP está sendo utilizada. Normalmente esse valor corresponde a ‘4’, em referência à versão 4 do protocolo (IPv4). A esse campo também pode ser atribuído o valor ‘6’, referente à versão 6 do protocolo (IPv6), em fase de implantação, e que ainda corresponde a uma parcela muito pequena do tráfego total da Internet. Tomando como exemplo os acessos ao provedor de serviços Google Inc., o percentual de utilização do IPv6 em Janeiro de 2014 foi de apenas 2,45% (GOOGLE, 2014).
- *Protocol*: campo de 8 bits que indica, através de um valor numérico, o tipo de tráfego contido na carga útil (*payload*) do pacote IP. Por exemplo: ‘1’ = ICMP; ‘6’ = TCP; e ‘17’ = UDP.
- *Source Address* e *Destination Address*: campos de 32 bits (IPv4) ou 128 bits (IPv6), contendo os endereços IP do remetente e do destinatário final do pacote. São exemplos de endereços IP: 192.188.11.21 (IPv4) e 2801:86:a:1::21 (IPv6).

A Figura 7 apresenta a estrutura de um pacote IPv4. Mais detalhes a respeito dessa estrutura, assim como informações adicionais sobre a versão 6 do protocolo, podem ser consultados na obra de Kurose e Ross (2012).

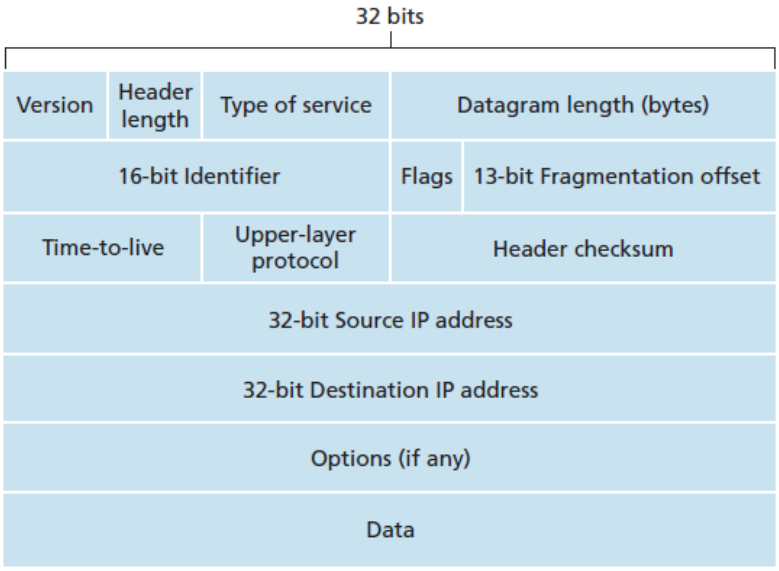


Figura 7 – Estrutura de um pacote IPv4.

Fonte: Kurose e Ross (2012).

2.2.4 Camadas de Enlace e Física

A camada de Enlace define os protocolos usados para descrever a topologia da rede local, estabelecendo um enlace de comunicação entre sistemas diretamente conectados e as interfaces necessárias para efetuar a transmissão dos pacotes da camada de Rede para os *hosts* vizinhos. Esta camada inclui vários protocolos de enlace, sendo o Ethernet o protocolo mais amplamente utilizado, e a sua unidade de dados de protocolo é conhecida como quadro.

O protocolo Ethernet baseia-se no conceito de Controle de Acesso ao Meio (em inglês: *Media Access Control – MAC*), que é um valor único de seis bytes (tal como 01-B8-FA-C8-D5-ED) que está permanentemente associado a um dispositivo físico da rede. Cada quadro contém dois endereços MAC, que indicam o endereço do dispositivo que encaminhou o quadro e o endereço MAC do próximo dispositivo para o qual o quadro está sendo enviado. À medida que um quadro passa pelos equipamentos intermediários da rede (tais como roteadores e *firewalls*) em seu percurso entre o *host* de origem e o *host* de destino final, os endereços MAC são atualizados a fim de que, ao final, coincidam com os dispositivos físicos de origem e de destino.

A camada Física, por sua vez, é responsável pela transmissão dos bits da camada de Enlace através de um meio físico (cabos de cobre, fibras ópticas, sinais de rádio, etc.). Também fazem parte dessa camada os componentes físicos da rede, tais como, roteadores, comutadores, pontes, etc.

2.3 ABORDAGENS PARA DETECÇÃO DE INTRUSÃO BASEADA EM REDE

Para que um Sistema de Detecção de Intrusão baseado em Rede opere corretamente, é necessário que ele monitore e analise o tráfego de rede nas camadas de Rede, Transporte ou Aplicação, a fim de identificar atividades suspeitas. A quantidade de camadas analisadas, bem como a complexidade envolvida em cada análise, dependerá da abordagem a ser utilizada no processo de detecção de intrusão.

De forma análoga à forma como está dividida a estrutura de um segmento ou pacote de rede, existem duas abordagens principais para realização da atividade de detecção de intrusão em redes, quanto ao conteúdo a ser analisado:

1. Classificação de pacotes, que possui foco na análise dos cabeçalhos dos pacotes e segmentos de rede; e
2. Inspeção profunda de pacotes (em inglês: *Deep Packet Inspection* – DPI), que é dedicada ao processamento da carga útil dos pacotes e segmentos de rede, e à sua comparação com padrões de assinaturas conhecidas.

A classificação de pacotes consiste no processo de mapeamento dos pacotes e segmentos, e a sua comparação com um conjunto finito de fluxos ou categorias, previamente definidos em regras, utilizando informações disponíveis nos seus cabeçalhos. Essas informações incluem os endereços IP de origem e destino, os números de porta de origem e destino, e o número do protocolo utilizado, e são extraídas dos campos presentes nos cabeçalhos das camadas de Transporte e Rede da pilha de protocolos TCP/IP. Caso ocorra correspondência entre as informações do pacote e mais de uma regra, caberá ao classificador de pacotes selecionar aquela regra que possuir maior prioridade, de modo que esta seja considerada no processo de detecção.

A inspeção profunda de pacotes constitui um processo de maior complexidade, no qual a carga útil dos pacotes deve ser analisada em busca de um padrão (ou conjunto de padrões) predefinidos. De uma forma geral, esses padrões são representados por *strings* (ou cadeias de caracteres) fixas ou por expressões regulares. Conforme descrito por Abuhmed, Mohaisen e Nyang (2008), alguns dos algoritmos utilizados para comparação de *strings* mais conhecidos são: Knuth-Morris-Pratt (KNUTH, 1997), Boyer-Moore (BOYER e MOORE, 1977), Aho-Corasick (AHO e CORASICK, 1975), Wu-Manber (WU e MANBER, 1994) e Commentz-Walter (COMMENTZ-WALTER, 1979).

De acordo com Song e Lockwood (2005), ambas as abordagens descritas podem ser combinadas para compor uma solução completa para detecção de intrusão em redes. Enquanto o processo de classificação de pacotes possui a capacidade de identificar ameaças e ataques primários através da análise dos cabeçalhos das camadas de Transporte e de Rede, ele também pode sinalizar

possíveis tráfegos suspeitos, a fim de que as informações de carga útil sejam analisadas em detalhes através do processo de inspeção profunda de pacotes.

2.4 O *SOFTWARE* SNORT

Lançado originalmente em 1998 por Martin Roesch (ROESCH, 1999), o Snort é um *software* de código aberto voltado para detecção e prevenção de intrusão baseada em rede, capaz de realizar análise de tráfego em tempo real e registro de pacotes em redes IP. Inicialmente pensado como uma tecnologia “leve” para detecção de intrusão, o Snort evoluiu ao longo dos anos para uma tecnologia de IDS/IPS madura e rica em recursos, tornando-se padrão *de facto* entre as soluções voltadas para detecção e prevenção de intrusão em redes. Com mais de 4 milhões de *downloads* e cerca de 400 mil usuários registrados, essa é a tecnologia de detecção e prevenção de intrusão mais amplamente difundida no mundo (SNORT, 2014). A Figura 8 apresenta a interface de linha de comando do Snort, na qual é possível observar o processo de detecção de um tráfego potencialmente malicioso em andamento.

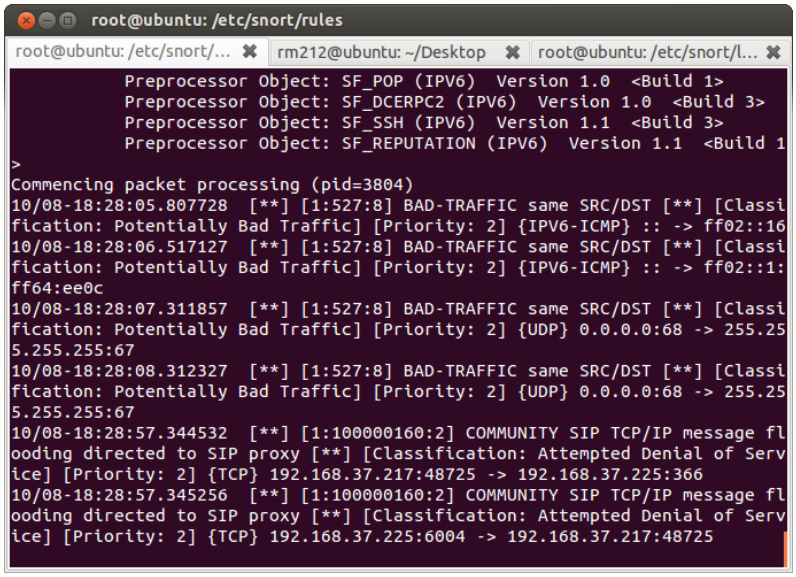


Figura 8 – Snort em execução a partir de um terminal Linux.

2.4.1 Principais funções do Snort

O Snort é capaz de executar análise de protocolo e busca/casamento de conteúdos, podendo ser utilizado para detectar uma grande variedade de ataques e sondas, tais como estouro de *buffer*, varredura de portas, ataques CGI, sondas SMB, tentativas de *fingerprinting* do sistema operacional, dentre muitos outros (SNORT, 2014). Detalhes a respeito de cada uma dessas ameaças podem ser consultados na obra de Edwards (2008). O *software* Snort utiliza uma linguagem de regras flexível para descrever o tráfego que ele deve coletar ou liberar, assim como um mecanismo de detecção que utiliza uma arquitetura de *plug-in* modular.

Essencialmente, o Snort possui três modos de operação, conforme descrito a seguir:

- Modo s*niffer* de pacotes, no qual o *software* faz a captura e leitura de todos os pacotes trafegando na rede e os exibe no console, de forma similar à ferramenta de monitoramento de pacotes de rede Tcpdump;
- Modo registrador de pacotes, no qual o *software* registra todo o tráfego de pacotes em disco, sendo um recurso útil para depuração do tráfego de rede, entre outros; e
- Modo de detecção e prevenção de intrusão, no qual o *software* monitora o tráfego de rede, o analisa e o compara com um conjunto de regras definido pelo usuário. A ferramenta então executa ações específicas com base no tipo de tráfego identificado.

O Snort possui também a capacidade de gerar alertas em tempo real, incorporando mecanismos de alerta para diversos formatos de armazenamento de *logs* distintos, conforme especificado pelos administradores da rede.

2.4.2 Arquitetura do Snort

Conforme descrito por Sen (2006) e exibido na Figura 9, a arquitetura do Snort é dividida em seis componentes principais, apresentados a seguir:

1. Decodificador de pacotes: responsável por transformar os pacotes capturados em estruturas de dados, e identificar os protocolos em uso a partir do nível de enlace. Posteriormente, o decodificador analisa o próximo nível e decodifica as informações presentes nas camadas de Rede e Transporte, a fim de obter informações sobre os endereços de porta. Esse componente também é capaz de detectar e enviar alertas a respeito de cabeçalhos de pacotes mal formados.

2. Pré-processadores: atuam como filtros, os quais identificam objetos que precisam ser verificados posteriormente pelo módulo do mecanismo de detecção. Exemplos desses objetos são: uma tentativa de conexão suspeita a alguma porta TCP/UDP, ou uma grande quantidade de pacotes UDP recebidos durante um ataque de varredura de portas.
3. Conjunto de regras: arquivo contendo a base de regras de detecção de intrusão, escritas em uma sintaxe compreensível pelo mecanismo de detecção de intrusão.
4. *Plug-ins* de detecção: correspondem a módulos referenciados a partir da sua definição no conjunto de regras, destinados a identificar padrões sempre que uma regra é avaliada.
5. Mecanismo de detecção: executa a comparação dos pacotes de rede com as regras de detecção previamente carregadas na memória, fazendo uso dos *plug-ins* de detecção.
6. *Plug-ins* de saída: permitem ao administrador do Snort especificar a saída para registros de *log* e para alertas. Esses módulos são executados quando os subsistemas de geração de alertas ou registros de *log* são acionados.

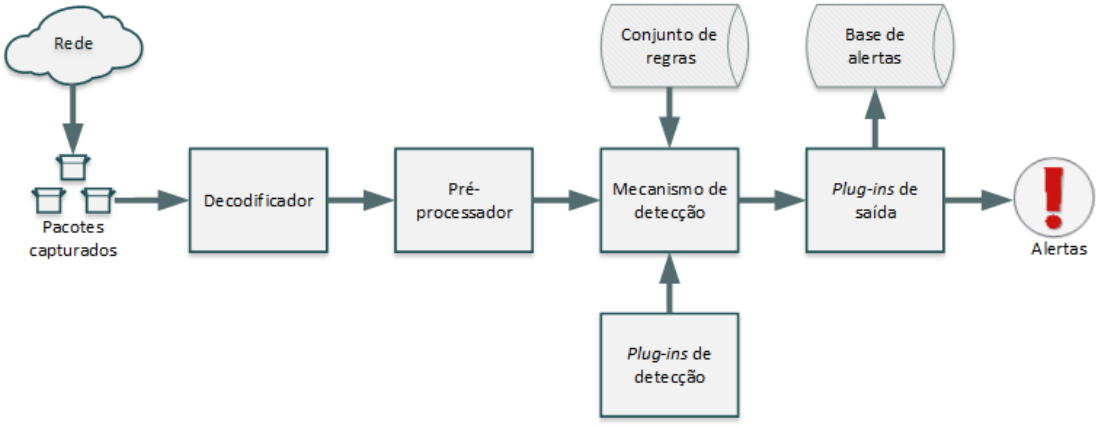


Figura 9 – Arquitetura simplificada do Snort.

2.4.3 O Mecanismo de detecção de intrusão do Snort

O Snort é capaz de realizar detecção de intrusão em redes por meio dos três principais métodos anteriormente mencionados: detecção baseada em assinatura; detecção baseada em anomalia; e análise de protocolo (SEN, 2006). O mecanismo de detecção é o componente responsável por implementar todos esses métodos de detecção e identificação de tráfego malicioso. Devido ao fato de possuir um conjunto de regras poderoso, flexível e de fácil compreensão, o processo de construção de regras no Snort é relativamente simples. Essas regras possuem uma sintaxe própria para detectar ameaças e anomalias na rede, e são tipicamente armazenadas em arquivos de texto codificados no padrão ASCII, onde cada caractere de texto ocupa um byte de espaço de armazenamento. Cada regra de detecção do Snort precisa ocupar uma linha distinta do arquivo de texto, de modo que o seu mecanismo de detecção possa interpretá-la corretamente.

Conforme descrito por Sen (2006) e ilustrado na Figura 10, uma regra de detecção de intrusão do Snort está dividida em duas partes:

a) Informações de cabeçalho, necessárias para a realização da tarefa de classificação de pacotes. Estas constituem um conjunto de campos estáticos, tais como: ação a ser tomada (normalmente um alerta), protocolo de transporte em uso (TCP ou UDP), endereço(s) IP e porta(s) de origem, direção do tráfego, e endereço(s) IP e porta(s) de destino; e

b) Opções da regra, contendo campos não obrigatórios, com definições variáveis. Entre essas definições estão os padrões de assinatura conhecidos, precedidos pela palavra-chave *content*. As definições de padrões de assinatura são usadas durante a operação de inspeção profunda de pacotes, podendo ser representadas por uma ou mais *strings* ou expressões regulares. Outra definição comum nas opções de regra é a mensagem de alerta a ser transmitida para o administrador do

sistema, precedida pela palavra-chave *msg*. O Snort conta com mais de 50 opções disponíveis para satisfazer diferentes requisitos na descrição de possíveis regras de detecção de intrusão.

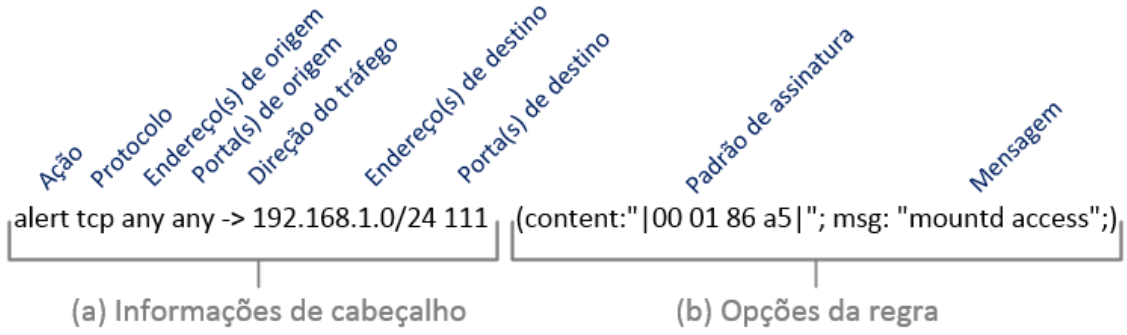


Figura 10 – Sintaxe típica de uma regra do Snort.

O Quadro 1 exemplifica uma regra do Snort usada para detectar a atividade do *malware* SubSeven em uma rede:

```
alert tcp $EXTERNAL_NET 27374 -> $HOME_NET any (msg:"BACKDOOR subseven
22"; flags: A+; content: "|0d0a5b52504c5d3030320d0a|";
reference:arachnids,485; reference:url,www.hackfix.org/subseven/;
sid:103; classtype:misc-activity; rev:4;)
```

Quadro 1 – Regra do Snort para detecção do *malware* SubSeven.

A partir da análise da regra descrita no Quadro 1, é possível extrair as seguintes informações:

1. Tipo de ação: *alert* – gera um alerta através do método selecionado e, então, registra o pacote. Outros tipos de ação configuráveis são: *pass, activate* e *dynamic*.
2. Protocolo em uso: *TCP*. Outros tipos de protocolos configuráveis são: UDP, ICMP e IP.
3. Endereços de origem e destino: previamente definidos pelo *software* através das variáveis *$EXTERNAL_NET* e *$INTERNAL_NET*,

respectivamente. Outros tipos aceitos são: endereços únicos (como 10.0.0.1), sub-redes inteiras (como 192.168.1.0/24), negação de um endereço ou sub-rede (como ![192.168.1.0/24]), ou qualquer endereço possível (definido pela palavra-chave *any*).

4. Portas de origem e destino: 27374 e *any* (que corresponde a qualquer porta), respectivamente.
5. Mensagem de alerta: ”*BACKDOOR subseven 22*”.
6. Flags: indica as *flags* usadas pelo protocolo em uso. Nesse caso, *A* indica que a *flag* ACK do protocolo TCP deve estar ativa, enquanto o sinal + indica que deve haver correspondência desse bit, além de outros.
7. Padrão de assinatura: *"|0d0a5b52504c5d3030320d0a|"*. Este campo, precedido pela palavra-chave *content*, diz respeito a uma porção da carga útil contida no pacote. Um padrão de assinatura pode ser formado por texto e dados binários. Estes últimos geralmente são delimitados pelo caractere *pipe* (|), e são representados como *bytecodes* (isto é, dados binários representados em formato hexadecimal).
8. Referência: documento onde é possível obter mais informações a respeito do alerta gerado e da ameaça identificada.
9. SID: *103* – número que identifica a regra de detecção de forma unívoca entre as demais.
10. Tipo de classe: indica em que classe a ameaça detectada melhor se encaixa. Nesse caso, *misc-activity* indica que a ameaça executa atividades variadas.
11. Número de revisão da regra: *4*. As revisões, juntamente com os números de identificação das regras, possibilitam que as assinaturas e suas descrições sejam refinadas e substituídas com informações atualizadas.

2.4.4 Integração do Snort com outras ferramentas

Devido ao fato de o Snort não possuir uma interface gráfica do usuário nativa, diversos projetos de *software* paralelos emergiram, com o intuito de aperfeiçoar e facilitar as tarefas de administração, geração de relatórios e análise de *logs* do Snort. Entre esses projetos figura o Sguil (2014), uma coleção de componentes de *software* gratuitos desenvolvida para facilitar o monitoramento e a análise de incidentes de segurança, cujo componente principal é uma interface gráfica do usuário intuitiva, que fornece acesso a eventos em tempo real, dados de sessões e captura de pacotes brutos.

O Sguil facilita a prática da atividade conhecida como Monitoramento da Segurança de Rede (em inglês: *Network Security Monitoring* – NSM) e a análise dirigida a eventos. Uma vez que essa ferramenta é desenvolvida em linguagem Tcl/Tk, ela pode ser executada em qualquer sistema operacional com suporte a essa linguagem, incluindo Linux, BSD, Solaris, MacOS e Windows (SGUIL, 2014).

A interface gráfica do Sguil, apresentada na Figura 11, permite aos administradores de rede a rápida verificação das principais informações relacionadas a um eventual incidente de intrusão, tais como: data e hora de ocorrência do incidente (a); endereço IP de origem (b); porta de origem (c); endereço IP de destino (d); porta de destino (e); mensagem de alerta (f); conteúdo da regra que possibilitou a detecção (g); e informações detalhadas extraídas das camadas de Rede e Transporte do pacote inspecionado, bem como a sua carga útil (h).

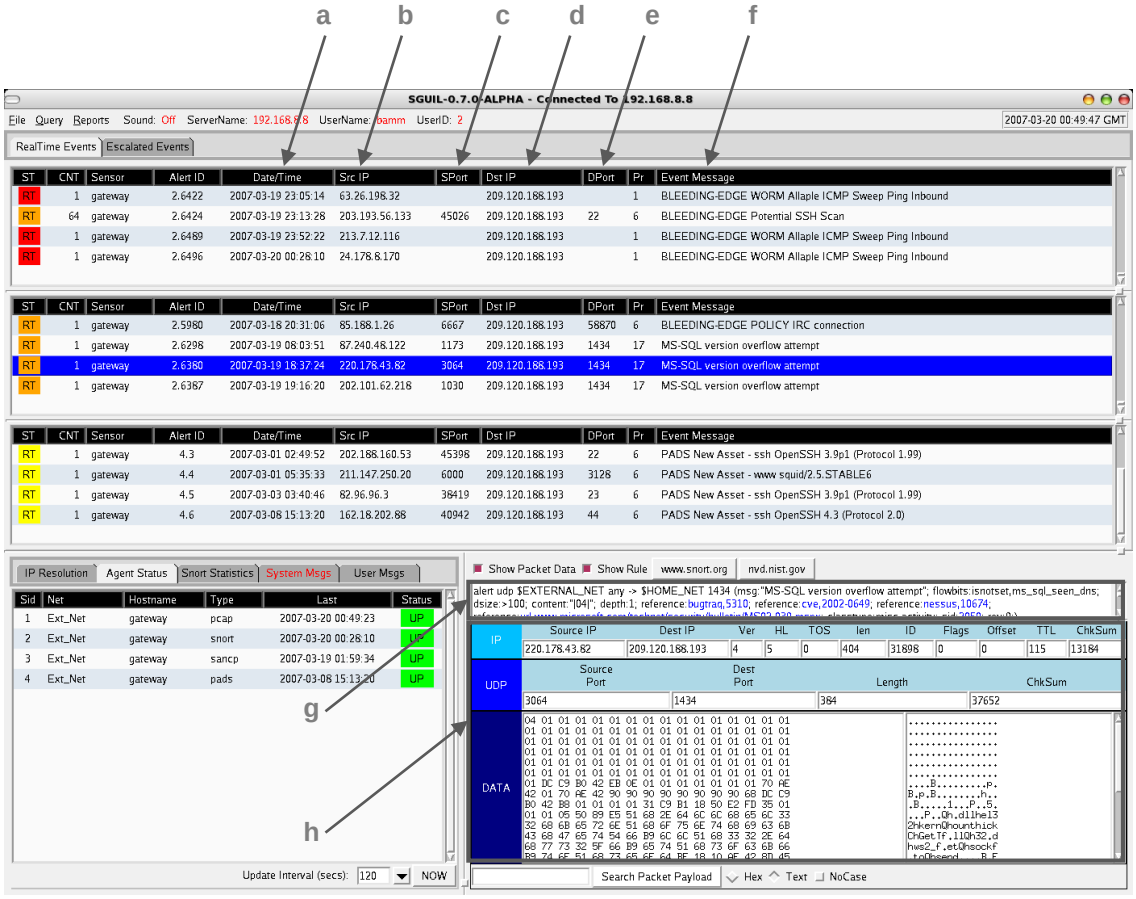


Figura 11 – Análise de alertas do Snort através da ferramenta Sguil.

Fonte: Adaptado de Sguil (2014).

2.5 *FIELD-PROGRAMMABLE GATE ARRAYS* – FPGAs

Um *Field-Programmable Gate Array* – FPGA (em português: Arranjo de Portas Programável em Campo) é um dispositivo semicondutor que contém componentes de lógica reconfigurável (denominados “blocos lógicos”) e interconexões reconfiguráveis. Os blocos lógicos podem ser configurados para executar a função de uma ou mais portas lógicas básicas (como AND e XOR) ou funções combinacionais mais complexas (tais como decodificadores ou funções aritméticas binárias). Na maioria dos dispositivos FPGA, os blocos lógicos incluem elementos de memória, que podem ser desde *flip-flops* simples até blocos de memória mais complexos. Atualmente, os principais fabricantes de FPGAs são Xilinx e Altera (FPGA DEVELOPER, 2011).

Conforme descrito por Barr (1999), uma hierarquia de interconexões reconfiguráveis do dispositivo FPGA permite que os seus blocos lógicos sejam interconectados, à medida que o projetista do sistema precisar. Blocos lógicos e interconexões podem ser configurados pelo projetista, após o FPGA ter sido fabricado, para implementar qualquer função lógica. Esta funcionalidade explica o termo "*field-programmable*", o qual pode ser livremente traduzido para "programável em campo". A configuração dos dispositivos FPGA se dá através do uso de Linguagens de Descrição de *Hardware* (em inglês: *Hardware Description Languages* – HDLs). As principais HDLs atualmente em uso são Verilog e VHDL (BROWN, 2013).

A arquitetura de um FPGA varia de acordo com o seu fabricante/família. Conforme descrito por Zeidman (2006) e ilustrado pela Figura 12, em geral, um dispositivo FPGA possui os seguintes componentes básicos, com algumas variações de nomenclatura:

- *Configuration Logical Blocks* – CLBs (em português: Blocos Lógicos de Configuração): Circuitos idênticos, construídos pela reunião de *flip-flops*

com a utilização de lógica combinacional. Utilizando os CLBs, o projetista pode construir elementos funcionais lógicos.

- *Input/Output Blocks* – IOBs (em português: Blocos de Entrada/Saída): Atuam na interface e controle de entrada e saída de sinais do circuito FPGA, podendo ou não ser bidirecionais.
- *Switch Matrix* (em português: Rede de Interconexão): Trilhas utilizadas para realizar a conexão entre os CLBs e os IOBs. Geralmente, a sua configuração é estabelecida por programação interna das células de memória estática, que determinam funções lógicas (CLBs) e conexões internas implementadas no FPGA entre os CLBs e os IOBs. O processo de escolha das interconexões é conhecido como roteamento.

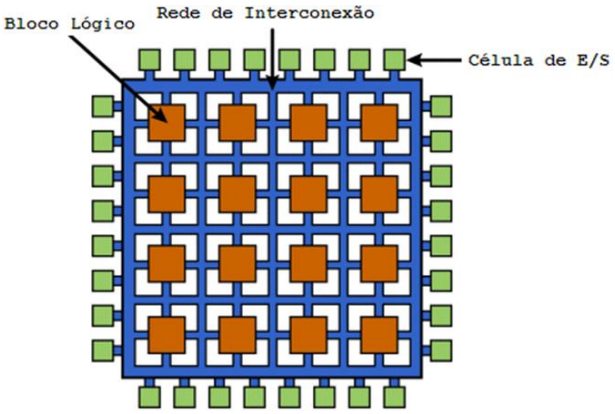


Figura 12 – Visão conceitual de uma arquitetura FPGA.
Fonte: Adaptado de Zeidman (2006).

Os dispositivos FPGA são uma alternativa aos circuitos integrados convencionais, conhecidos como *Application-Specific Integrated Circuits* – ASICs (em português: Circuitos Integrados de Aplicação Específica), que não permitem que modificações em sua estrutura interna sejam feitas depois de fabricados, tornando o seu projeto menos flexível e mais demorado.

De acordo com a Xilinx Inc. (2014), os FPGAs são normalmente mais lentos do que os ASICs, uma vez que não são otimizados para uma funcionalidade

específica. No entanto, suas vantagens incluem um *time-to-market* (isto é, o prazo compreendido entre o momento em que se deu a ideia e a efetiva chegada do produto ao mercado) mais curto; a capacidade de serem reconfigurados no local de sua aplicação (a fim de corrigir erros de projeto ou alterar a funcionalidade pretendida); e menores custos não recorrentes de engenharia. Os fabricantes de FPGAs também comercializam versões menos flexíveis de seus dispositivos a um custo menor, com porções do circuito que não possam ser modificadas após o projeto estar pronto. Nessa abordagem, os projetos são originalmente desenvolvidos em FPGAs convencionais, para então serem migrados para uma versão definitiva e não flexível de dispositivo semicondutor (semelhante a um ASIC em sua especificidade).

Como possíveis aplicações dos dispositivos FPGA, podem ser citadas:

- Implementação de lógica aleatória: em substituição a *chips* totalmente dedicados;
- *Hardware* reconfigurável: na definição de blocos de *hardware* que podem ou necessitam ter seu comportamento alterado durante a execução de uma aplicação, num processo de readequação da arquitetura (visando aceleração do processamento e flexibilidade);
- Resolução de problemas específicos: ao permitir que um componente de *hardware* possa se tornar temporariamente dedicado à solução de um determinado problema;
- Prototipagem: ao permitir que uma combinação de matrizes e portas emule um circuito que ainda não foi fabricado, proporcionando uma análise maior, melhor e mais detalhada, se comparada apenas ao processo de simulação.

Apesar de o escopo de aplicação dos dispositivos FPGA ter sido anteriormente limitado devido a fatores como menor desempenho e área útil do circuito, em comparação com os ASICs, avanços recentes na densidade e velocidade dos dispositivos FPGA permitiram que esses dispositivos se tornassem componentes primários em muitos sistemas embarcados (CHEN,

CHEN e SUMMERVILLE, 2011). Esse fator torna a escolha da tecnologia FPGA adequada para a prototipagem de sistemas NIDS diretamente em *hardware*.

2.6 TRABALHOS RELACIONADOS

De acordo com a revisão de literatura realizada, embora existam vários estudos recentes envolvendo NIDS implementados em *hardware*, foram encontradas poucas pesquisas atuais claramente voltadas especificamente para otimizar o armazenamento das regras de detecção de intrusão em memórias embarcadas *on-chip*. Nesta subseção, quatro abordagens clássicas de FPGAs aplicados na implementação de NIDS são brevemente descritas. Em seguida, são apresentadas outras quatro abordagens, voltadas especificamente para a otimização dos recursos de memória embarcada em projetos de NIDS implementados em *hardware*, que é o objetivo desse trabalho.

2.6.1 FPGAs aplicados para detecção de intrusão em redes

A aplicação de FPGAs na construção de NIDS foi inicialmente proposta por Hutchings, Franklin e Carver (2002), que implementaram autômatos finitos não determinísticos em *hardware* para comparar expressões regulares, a partir do conjunto de regras do *software* Snort. Nessa proposta, apenas a comparação de padrões foi realizada em *hardware*, uma vez que as etapas de comunicação em rede foram realizadas por um computador convencional. Apesar dessa limitação, testes mostraram que, em determinados casos, essa implementação obteve um

desempenho mais de 600 vezes superior, na época, se comparada à solução baseada apenas em *software*.

Em seguida, Cho, Navab e Mangione-Smith (2002) propuseram um NIDS autônomo baseado em FPGA alimentado por regras do Snort e do Hogwash, outra ferramenta NIDS popular à época. Nessa abordagem, as regras de detecção de intrusão eram pré-processadas e, então, convertidas diretamente para a linguagem VHDL, de modo que fossem sintetizadas diretamente em elementos lógicos. Essa implementação foi sintetizada em um dispositivo FPGA Altera EP20K, consumindo cerca de 17.000 elementos lógicos para armazenar 105 regras. A sua arquitetura possuía mecanismos de paralelismo que otimizava o processo de comparação de padrões, tornando-a capaz de filtrar tráfego de rede a uma taxa máxima de 2,88 Gbps.

Posteriormente, Clark e Schimmel (2003) propuseram uma abordagem similar à proposta por Hutchings, Franklin e Carver (2002), porém focada em expressões regulares complexas. Nela, foi desenvolvido um coprocessador para casamento de padrões baseado em autômatos finitos não determinísticos com suporte às regras de detecção do Snort. Com essa abordagem, todo o banco de dados de regras do Snort, na época composto por 1.500 regras e 17.000 caracteres, foi codificado em um dispositivo FPGA Xilinx Virtex-1000, capaz de realizar o casamento de padrões a uma taxa de transferência próxima a 1 Gbps, considerando a frequência de 100 MHz do dispositivo.

No trabalho apresentado por Baker e Prasanna (2004), por sua vez, foi apresentada uma metodologia baseada no particionamento em grafos de grandes conjuntos de padrões de assinaturas. Ao integrar a criação de grafos baseados em conjuntos de regras com o particionamento através de corte mínimo, essa metodologia viabilizou comparações eficientes de bytes múltiplos e correspondências parciais para um NIDS implementado em FPGA. Por meio do pré-processamento dos conjuntos de regras, essa metodologia permitiu o desenvolvimento de projetos de NIDS em *hardware* com uma economia da ordem de 8 vezes em área do circuito, se comparada com as abordagens até

então existentes, sendo por esse motivo o seu trabalho constantemente referenciado no meio acadêmico.

2.6.2 Abordagens específicas para a otimização de memória embarcada

No trabalho apresentado por Yi *et al.* (2007), foi proposta uma implementação *hash* de árvore *bottom-up*, visando aumento de desempenho e redução do consumo de memória para armazenar 2.770 padrões de assinaturas presentes nas regras do Snort. Com esse método, foi possível reduzir o espaço ocupado pelas assinaturas, de cerca de 512 kB para 350 kB, o que representa uma economia de aproximadamente 31,64%. Vale destacar, no entanto, que essa abordagem tratou especificamente dos campos de padrões de assinatura, que constituem uma pequena parte de cada regra do Snort.

Na pesquisa desenvolvida por Chen, Summerville e Chen (2009), os seus autores propuseram um método para decomposição de *strings* de texto em dois passos, visando eliminar redundâncias nos conjuntos de caracteres presentes nos padrões de assinatura do Snort. Embora tenha atingido uma economia de cerca de 77,24% ao comprimir tais padrões, essa abordagem também não tratou as demais informações presentes no conjunto de regras, tais como as informações de cabeçalho e as demais opções presentes nas regras.

A abordagem proposta por Nikitakis e Papaefstathiou (2008), por sua vez, decompôs regras de classificação de pacotes de múltiplos campos em regras de campo único, combinando-as através de filtros de Bloom em multi-nível. Embora o método proposto tenha possibilitado o armazenamento de 4.000 regras voltadas para classificação de pacotes (cada qual ocupando poucas dezenas de bytes) em apenas 178 kB de memória, os seus autores não informaram o percentual de compressão obtido. Ademais, visto que o seu foco era apenas a

atividade de classificação de pacotes, esse trabalho também deixou a maior parte do conteúdo das regras de detecção sem tratamento.

De forma similar, no método apresentado por Guinde, Ziavras e Rojas-Cessa (2010), as regras de classificação de pacotes foram primeiramente agrupadas, com base no seu número de campos importantes e, então, esses campos foram comparados dois de cada vez. Posteriormente, os resultados dessa comparação foram combinados para identificar correspondências mais longas. Com essa abordagem, foi possível armazenar mais de 10.000 regras em aproximadamente 256 kB de memória. Novamente, essa abordagem teve foco apenas na operação de classificação de pacotes, e os seus autores não mencionaram a economia exata de memória obtida.

3 ARQUITETURA PROPOSTA

De acordo com o já descrito no capítulo de introdução, esse trabalho propõe a utilização do algoritmo de Huffman para otimização dos recursos de memória embarcada em projetos de NIDS implementados em *hardware*, uma vez que tal aplicação não foi encontrada em nenhuma outra publicação com esse fim específico. Conforme conceitualmente apresentado na Figura 13, a arquitetura proposta é composta por três componentes principais, que são detalhados nas subseções a seguir:

1. Codificação, através de *software*, do conjunto de regras de detecção de intrusão em um arranjo de bits, utilizando o algoritmo de Huffman;
2. Rearranjo, por meio de *software,* dos bits previamente codificados para otimização do espaço de memória consumido; e
3. Decodificação paralela e reconstrução das regras de detecção, através de *hardware*, para execução das funções principais do sistema de detecção de intrusão em redes.

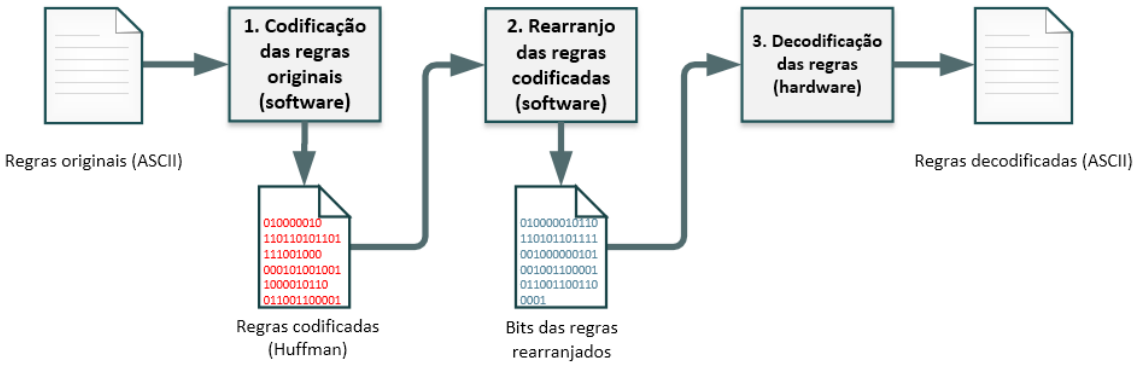


Figura 13 – Composição da arquitetura proposta.

3.1 CODIFICAÇÃO DO CONJUNTO DE REGRAS

3.1.1 Método de compressão utilizado

A codificação de Huffman (1952) é um método de compressão sem perda de dados que se baseia nas probabilidades de ocorrência dos símbolos presentes em determinado conjunto de dados, de modo que os símbolos mais frequentes sejam representados por um número menor de bits, em uma ideia similar àquela adotada pelo código Morse. Embora tenha sido inicialmente proposto há mais de seis décadas, esse método de compressão ainda hoje encontra aplicação em padrões de compressão amplamente utilizados, tais como JPEG, MPEG-2 e MPEG-4 (ISO/IEC, 2010; ISO/IEC, 2012).

A compressão de dados através da codificação de Huffman é bastante eficaz, normalmente atingindo economias de espaço entre 20% e 90%, de acordo com as características dos dados que são comprimidos. Conforme destacam Cormen *et al.* (2009), o método utilizado por Huffman se enquadra na definição de algoritmo guloso, isto é, um algoritmo que sempre faz a escolha que parece ser a melhor no momento. Em outras palavras, uma escolha ótima é feita pelo algoritmo guloso no contexto local, na esperança de que essa escolha leve a uma solução global ótima. Considerando os dados de entrada como uma sequência de caracteres, o algoritmo de Huffman utiliza uma tabela que registra quantas vezes cada caractere aparece nessa sequência (isto é, a sua frequência) para construir uma forma ótima de representar cada caractere como uma cadeia binária.

De forma simplificada, o método de Huffman analisa todo o conjunto de dados a ser codificado e calcula as probabilidades de ocorrência de cada símbolo desse conjunto. Estas probabilidades são ordenadas, agrupadas e

somadas aos pares para formar uma árvore binária cujas folhas representam os símbolos, e cujas arestas correspondem aos valores binários ‘0’ ou ‘1’. Este processo, detalhado na obra de Cormen *et al.* (2009), está representado no pseudocódigo do Quadro 2, o qual é brevemente explicado nos parágrafos a seguir.

```
HUFFMAN (C)
1       n = |C|
2       Q = C
3       para i = 1 até n – 1
4       aloca um novo nó z
5               z.esquerda = x = EXTRAI_MIN(Q)
6               z.direita = y = EXTRAI_MIN(Q)
7               z.freq = x.freq + y.freq
8       INSERE(Q , z)
9       retorna EXTRAI_MIN(Q) //retorna a raiz da árvore
```

Quadro 2 – Pseudocódigo do algoritmo de Huffman.
Fonte: Cormen *et al.* (2009).

No pseudocódigo descrito no Quadro 2, considera-se que C é um conjunto de caracteres, e que cada caractere c ∈ *C* é um objeto com um atributo c.*freq*, que possui a sua frequência de ocorrência nesse conjunto. O algoritmo de Huffman constrói a árvore *T* correspondente ao código ótimo em uma disposição *bottom-up*. Ele inicia com um conjunto de |*C*| folhas, e executa uma sequência de |*C*| – 1 operações de agrupamento para criar a árvore final. O algoritmo utiliza uma fila de prioridade mínima *Q*, cuja chave é o atributo *freq*, para identificar os dois objetos menos frequentes, para que sejam agrupados. O agrupamento desses dois objetos resulta em um novo objeto, cujo atributo de frequência é a soma das suas frequências originais.

De modo a exemplificar o processo de codificação do método de Huffman, suponha-se a existência de um arquivo de texto de entrada, contendo 100 caracteres, cujo alfabeto varia de *A* até *F*. A primeira etapa do algoritmo consiste

na identificação da frequência de ocorrência de cada caractere do conjunto de dados. A Figura 14 apresenta o resultado dessa etapa, na qual é possível constatar que o caractere *A*, mais frequente, aparece 45 vezes, enquanto o caractere *F*, menos frequente, aparece apenas 5 vezes no conjunto de dados.

Caractere	A	B	C	D	E	F
Frequência	45	13	12	16	9	5

Figura 14 – Frequência de caracteres em um dado arquivo de texto.

A partir das informações obtidas sobre as frequências de cada caractere do conjunto de dados, e do algoritmo descrito no Quadro 2, é possível construir uma árvore binária com base no método proposto por Huffman, cujo resultado pode ser visto na Figura 15.

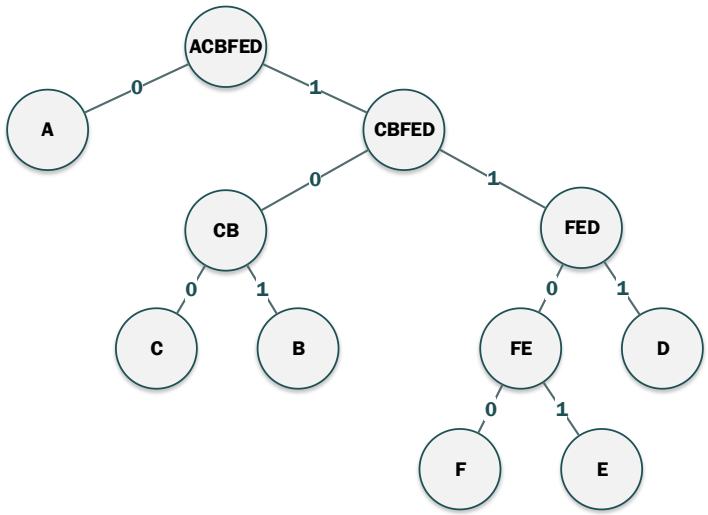


Figura 15 – Exemplo de uma árvore binária de Huffman.

Como pode ser observado na Tabela 1, quando a árvore binária resultante do algoritmo de Huffman é percorrida da raiz até cada uma das suas folhas, é possível formar um código de comprimento variável associado a cada folha, considerando os valores binários presentes nas arestas percorridas. Esses

códigos são então atribuídos aos símbolos representados pelas folhas, formando um dicionário de símbolos de Huffman. Uma vez que os códigos gerados por esse algoritmo possuem comprimento variável, uma informação do dicionário igualmente útil para o processo de decodificação é o comprimento (em bits) de cada código gerado.

É possível concluir, a partir do dicionário de símbolos de Huffman formado, que os caracteres mais frequentes no conjunto de dados original do exemplo de fato passaram a ser representados por uma quantidade menor de bits, enquanto os caracteres menos frequentes ocupam um número maior de bits. O processo detalhado de codificação de dados através do método de Huffman pode ser consultado na obra de Cormen *et al.* (2009).

Tabela 1 – Exemplo de um dicionário de símbolos de Huffman

Símbolo	**Código de Huffman**	**Comprimento do código**
A	0	1
B	101	3
C	100	3
D	111	3
E	1101	4
F	1100	4

3.1.2 Modelo de referência para codificação e decodificação de Huffman

Para validar a proposta de utilização do algoritmo de Huffman para compressão de regras de detecção de intrusão e comprovar a sua eficácia, inicialmente foi desenvolvido, no ambiente computacional Matlab (versão R2012a), um modelo contendo um conjunto de *scripts* para codificação e decodificação de cadeias de caracteres, originalmente codificadas no padrão ASCII. Um segundo objetivo desse modelo foi o de servir como referência para a implementação do componente de decodificação em *hardware*, a ser detalhado nas seções subsequentes.

Embora o Matlab possua funções implementadas internamente para geração do dicionário de símbolos (*huffmandict*), codificação (*huffmanenco*) e decodificação (*huffmandeco*) de dados utilizando o método de Huffman, inicialmente se optou pela implementação manual de tais funções, em nível detalhado, de modo que todo o processo fosse melhor compreendido, com o intuito de que a posterior implementação da lógica digital para decodificação das regras fosse simplificada. De uma forma resumida, esse modelo executa as seguintes operações:

a) Decomposição de uma cadeia de caracteres de entrada em uma árvore binária, conforme propõe o algoritmo de Huffman;

b) Formação de uma versão gráfica da árvore binária construída, para fins de verificação visual;

c) Criação do dicionário de símbolos de Huffman e seus respectivos códigos, gerados a partir da árvore binária construída;

d) Codificação da cadeia de entrada, com base no dicionário de símbolos;

e) Decodificação da cadeia de caracteres codificada no passo *d*, e comparação do seu valor com a cadeia de caracteres original;

f) Repetição dos procedimentos descritos nos passos *c*, *d* e *e*, desta vez com as respectivas funções internas do Matlab, e comparação dos seus resultados com aqueles obtidos pelo modelo; e

g) Exibição dos resultados obtidos pelo modelo no console do Matlab.

Na Figura 16 é apresentada a árvore de Huffman gerada graficamente a partir do passo *b* do modelo implementado em Matlab, quando utilizada como entrada a mesma cadeia de caracteres citada no exemplo teórico descrito na subseção 3.1.1. Nessa figura, é possível constatar que os resultados obtidos pelo modelo desenvolvido em Matlab são idênticos àqueles exibidos no exemplo da Figura 15. Adicionalmente, o Quadro 3 apresenta as saídas do modelo no console do Matlab, quando utilizada como entrada a cadeia de caracteres mencionada.

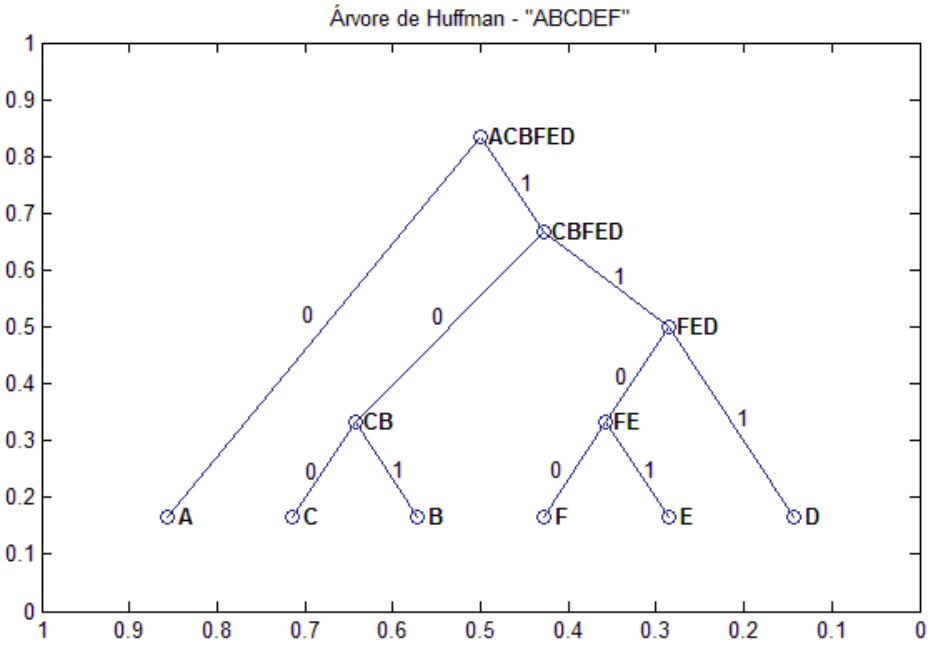


Figura 16 – Árvore binária gerada pelo modelo implementado em Matlab.

```
- Cadeia de entrada: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABB
BBBBBBBBBBCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEEEEEEEEFFFFF

- Número de caracteres da entrada: 100

- Símbolos únicos identificados: A B C D E F

- Ocorrências de cada símbolo:    45    13    12    16     9     5

- Probabilidades normalizadas de cada símbolo:
     A -->   0.45
     B -->   0.13
     C -->   0.12
     D -->   0.16
     E -->   0.09
     F -->   0.05

- Árvore de símbolos ordenada: 'ACBFED'    'A'    'CBFED'    'CB'
'FED'    'FE'    'D'    'C'    'B'    'F'    'E'

- Dicionário de símbolos de Huffman gerado:
    'A'    '0'
    'D'    '111'
    'B'    '101'
    'C'    '100'
    'E'    '1101'
    'F'    '1100'

- Saída codificada com o método de Huffman:
00000000000000000000000000000000000000000000010110110110110110110110
11011011011011011001001001001001001001001001001001001111111111111111
11111111111111111111111111111111111101110111011101110111011101110111011101
11001100110011001100

- Tamanho original da entrada (bits - ASCII):  800

- Tamanho da saída codificada (bits - Huffman): 224

- Taxa de compactação:   0.2800

- Saída após a decodificação: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAABBBBBBBBBBBBCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEEEEEEEEFFFFF

Saída decodificada = entrada? Sim
```

Quadro 3 – Saída do modelo implementado em Matlab, considerando o exemplo da seção 3.1.1.

3.1.3 Implementação em *software* do processo de codificação

Para codificação das regras de detecção de intrusão, foram feitas adaptações no modelo de referência original, de modo que este passasse a suportar a leitura e a codificação automatizada de arquivos contendo centenas ou milhares de regras, embora o conceito aplicado seja o mesmo descrito na subseção 3.1.2. Visando acelerar o processo de codificação das regras, foram utilizadas as funções otimizadas da biblioteca do Matlab, *huffmandict* e *huffmanenco*, para geração do dicionário de símbolos e codificação das regras, respectivamente. Dessa forma, o conjunto de *scripts* desenvolvido no ambiente do Matlab passou a executar as seguintes operações, conforme ilustrado no fluxograma da Figura 17:

a) Varredura inicial do arquivo de texto contendo as regras de detecção de intrusão originais (isto é, conforme disponibilizadas pelos seus autores), para contabilização da incidência de cada caractere ASCII único presente no arquivo, de modo a calcular a sua probabilidade de ocorrência;

b) Construção do dicionário de símbolos de Huffman, através do algoritmo de Huffman, com base nas probabilidades calculadas para cada caractere ASCII. Este dicionário é então armazenado em um arquivo de memória cujos dados são dispostos de forma similar à descrita na Tabela 1, para posterior decodificação pela arquitetura de *hardware*. O ponteiro de leitura do arquivo contendo as regras originais também é reiniciado, a fim de viabilizar uma segunda análise das regras para codificação;

c) Análise individual de cada regra do arquivo de regras original e posterior codificação através do método de Huffman, por meio do dicionário de símbolos previamente gerado. O caractere de “avanço de linha” (em inglês: *line feed*, código ASCII 00001010) encontrado ao final de cada

regra também é codificado, uma vez que será utilizado para detectar os limites entre as regras durante o processo de decodificação; e

d) Cada regra codificada é então armazenada em um arquivo binário (uma regra codificada por linha) que pode ser usado para carregar a memória da arquitetura de *hardware*. Este processo se repete até que não existam mais regras no arquivo para serem analisadas.

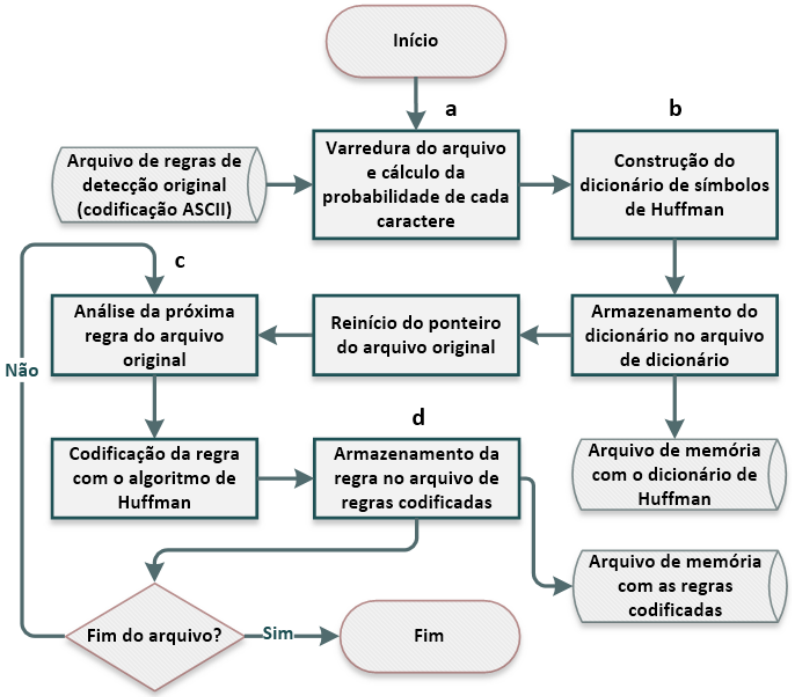


Figura 17 – Fluxograma do processo de codificação das regras de detecção.

O Quadro 4 apresenta o sumário de um processo de codificação exibido pelo modelo no console do Matlab, referente a um arquivo contendo 1.000 regras de categorias diversas do Snort.

```
***** Sumário do processo de codificação *****

Dados referentes ao dicionário de Huffman gerado:
Número de símbolos no dicionário: 97
Comprimento do menor símbolo Huffman: 4
Comprimento do maior símbolo Huffman: 18

Dados referentes às regras codificadas:
Número de regras no arquivo: 1000
Comprimento da menor regra original (bits): 1904
Comprimento da maior regra original (bits): 9384
Comprimento da menor regra codificada (bits): 1320
Comprimento da maior regra codificada (bits): 7376

Dados referentes à memória necessária:
- Para armazenar as regras em ASCII (bits): 9384*1000 = 9384000
- Para armazenar as regras codificadas (bits): 7376*1000 = 7376000

Taxa de compressão de memória obtida: 0.7860
```

Quadro 4 – Sumário do processo de codificação de um arquivo com 1.000 regras do Snort.

3.2 Rearranjo dos bits codificados em um espaço de armazenamento contíguo

De acordo com a pesquisa realizada, a técnica de busca linear é considerada a melhor escolha em projetos de NIDS que utilizam os blocos de memória embarcada em dispositivos FPGA para armazenar as regras de detecção de intrusão (CHEN, CHEN e SUMMERVILLE, 2011). Por meio dessa técnica, as posições de memória são percorridas sequencialmente em busca de padrões de detecção, que são comparados com o pacote de rede sob análise. No projeto de NIDS baseados em *hardware* que empregam a técnica de busca linear, cada regra de detecção geralmente ocupa uma posição de memória distinta. Esse arranjo possibilita uma comparação mais rápida dos pacotes que ingressam na rede com as regras de detecção armazenadas no sistema, uma vez que o *hardware* requer apenas alguns ciclos de *clock* para carregar cada

regra a partir da memória e realizar a comparação (LOINIG, WOLKERSTORFER e SZEKELY, 2007).

No entanto, tal estratégia de projeto causa desperdício de recursos de armazenamento, visto que o comprimento de palavra de memória é determinado com base no comprimento em bits da maior regra do conjunto. Dessa forma, a ferramenta de síntese do *hardware* completa cada posição de memória com bits de preenchimento inválidos (tipicamente zeros) sempre que ela não for completamente preenchida por uma regra, conforme pode ser visto na Figura 18 (a).

Após codificados através do método de Huffman, os bits das regras de detecção devem ser decodificados sequencialmente, para que o processo não resulte em ambiguidades durante a reconstrução dos caracteres ASCII originais. Para tirar proveito de tal requisito e visando reduzir o desperdício evidenciado, esse trabalho propõe o rearranjo dos bits codificados para um espaço de armazenamento contíguo, de modo que todas as posições de memória sejam utilizadas de forma ótima. Isto é, sempre que determinada posição de memória contiver espaço não aproveitado por uma determinada regra, esse espaço será alocado para armazenar os bits iniciais da regra seguinte, e assim por diante, conforme mostra a Figura 18 (b). Seguindo a lógica desse novo arranjo, bits inválidos somente serão admitidos na última posição de memória, onde não há mais bits válidos para serem armazenados Figura 18 (c).

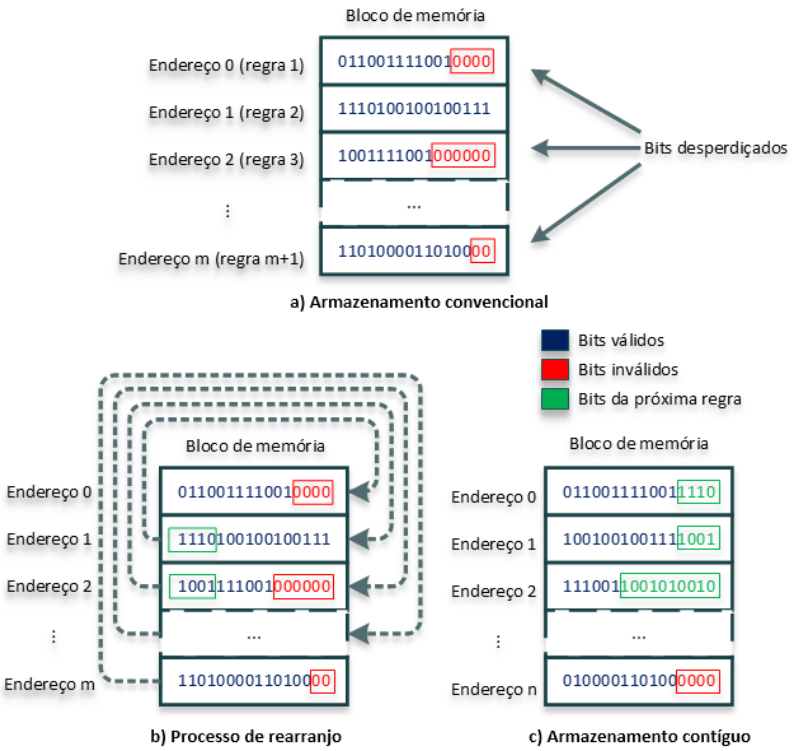


Figura 18 – Armazenamento em memória convencional *versus* contíguo.

Como consequência desse processo de rearranjo, o número total de bits ocupados com a utilização do armazenamento contíguo tende a ser consideravelmente menor do que o número de bits ocupados com a utilização do armazenamento convencional, embora ambas as abordagens possuam exatamente o mesmo número de bits válidos.

O rearranjo proposto foi implementado através de um conjunto de *scripts* no ambiente Matlab, seguindo os passos descritos no fluxograma da Figura 19, que são resumidos da seguinte forma:

a) Leitura sequencial de cada regra codificada (R) armazenada no arquivo contendo o arranjo convencional, e cálculo do seu comprimento em bits (L).

b) Para cada regra analisada, concatenação dos seus bits a um arranjo contendo os bits das regras anteriores ($A = [A\ R]$) e incremento do comprimento total de bits concatenados ($T = T + L$);

c) Após a análise das regras codificadas e a concatenação de todos os seus bits, cálculo do número de posições de memória necessárias para armazenar todas as regras ($P = teto(T / M)$), onde M é o comprimento predefinido para as posições de memória e *teto()* é uma função de aproximação para o número inteiro igual ou imediatamente superior ao seu argumento. O número de bits válidos na última posição de memória também é registrado, dada a possibilidade de existência de bits inválidos em tal caso; e

d) Leitura sequencial do arranjo de bits em blocos de comprimento fixo igual a M, e armazenamento desses blocos em um novo arquivo binário (um bloco por linha). O número de posições de memória necessárias também é decrementado ($P = P - 1$), até que os bits do arranjo sejam esgotados.

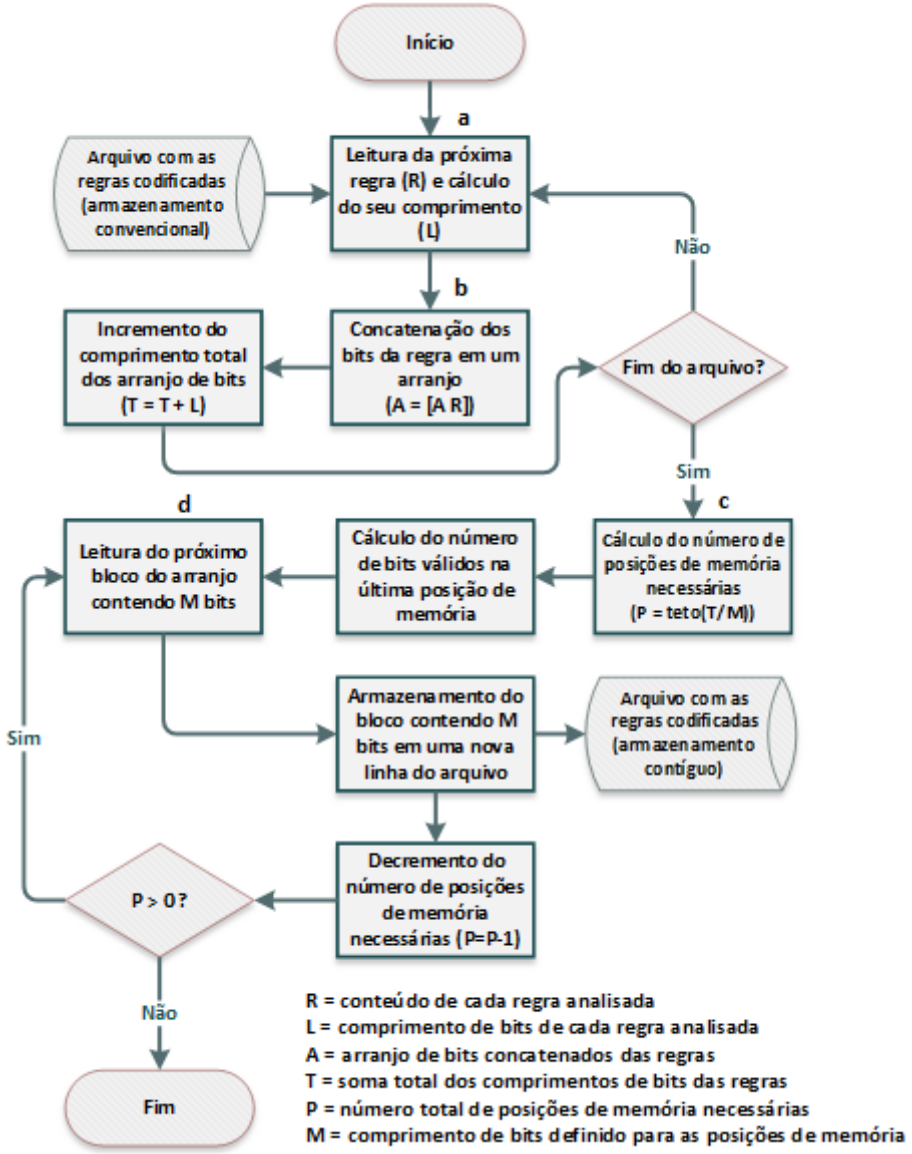


Figura 19 – Processo de rearranjo contíguo do espaço de memória.

A implementação dos passos descritos nesta subseção resulta em um arquivo binário contíguo, a ser carregado em uma memória ROM no *hardware*, de dimensões M x P, ou seja, contendo M bits de largura de palavra por P posições de profundidade de memória.

O Quadro 5 apresenta o sumário do processo de rearranjo de bits no console do Matlab, referente às 1.000 regras do Snort codificadas com o algoritmo de Huffman, conforme exibido no Quadro 4. Nesse caso, o processo

de codificação das 1.000 regras, combinado com o rearranjo contíguo dos bits, resultou em uma economia total de memória de 70,48%, quando comparado com o espaço originalmente ocupado pelas mesmas regras armazenadas no formato ASCII.

```
Rearranjo de bits concluído com sucesso!

Sumário do processo de rearranjo:

Número de blocos de memória utilizados: 1
Comprimento da palavra de memória (bits em cada posição): 256
N° de posições (profundidade) da memória: 10822
N° de bits necessários para endereçar a memória: 14
Comprimento da última palavra da memória (bits): 146
N° de bits inválidos (zeros) na última posição de memória: 110

Espaço em memória necessário para armazenar:
- As regras originais - padrão ASCII (bits): 9384000
- As regras codificadas – Arranjo convencional (bits): 7376000
- As regras codificadas – Arranjo contíguo (bits): 256*10822=2770432

Taxas de compressão obtidas:
- Após o rearranjo - Arranjo Contíguo / Arranjo Convencional: 0.3756
- Total – Codificação Huffman (Arranjo contíguo) / ASCII: 0.2952

Economia total de memória obtida: 70.48%
```

Quadro 5 – Sumário do processo para rearranjo dos bits codificados de 1.000 regras do Snort.

3.3 DECODIFICAÇÃO E RECONSTRUÇÃO DAS REGRAS DE DETECÇÃO

3.3.1 Arquitetura básica para decodificação de códigos de Huffman

A fim de que um NIDS implementado em *hardware* possa analisar o conjunto de regras codificadas pelo método proposto, é necessário que haja um processo prévio de decodificação de cada regra. Para a realização dessa etapa, um conjunto de módulos para decodificação de códigos Huffman foi projetado em lógica digital, utilizando o ambiente de desenvolvimento Altera Quartus II (versão 12.1 SP1 Web Edition), em conjunto com a linguagem de descrição de *hardware* Verilog. O diagrama de blocos da arquitetura resultante é apresentado na Figura 20.

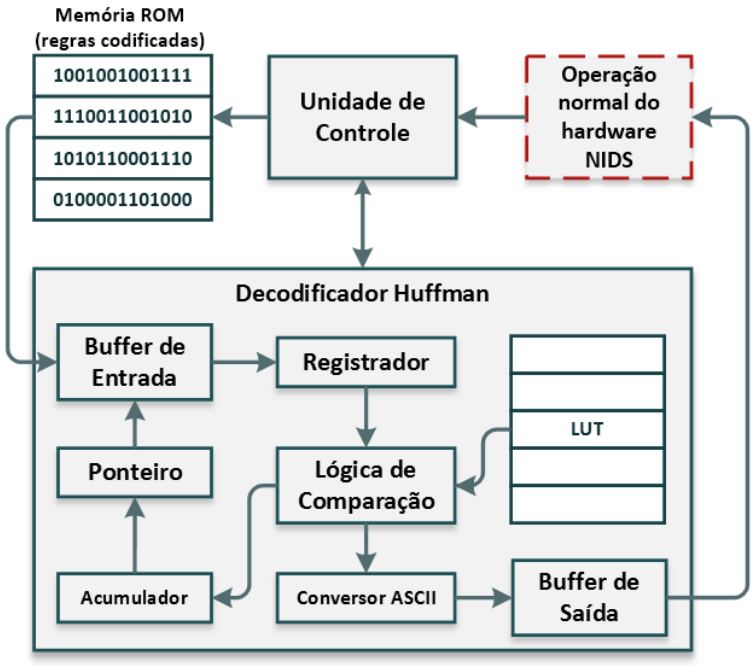


Figura 20 – Arquitetura de *hardware* para decodificação das regras de detecção de intrusão.

A arquitetura básica implementada em *hardware* é composta por três módulos principais, descritos a seguir:

1. Uma unidade de controle, responsável pelo acionamento dos demais módulos e arbitragem dos sinais digitais trocados entre eles;
2. Uma memória ROM contendo as regras codificadas, cujos bits foram reorganizados conforme o método descrito na subseção 3.2; e
3. Um conjunto decodificador de códigos de Huffman baseado em *Lookup Table*, que foi projetado de forma similar à proposta por Mansour (2007).

Os módulos da arquitetura proposta atuam em conjunto para satisfazer às seguintes especificações:

a) Quando o módulo de detecção de intrusão necessita comparar um determinado pacote de rede com o conjunto de regras de detecção, ele requisita a decodificação destas regras à unidade de controle, que passa a transmitir palavras de bits sequenciais de comprimento fixo, que estão armazenados na memória ROM, para o *buffer* de entrada do decodificador;

b) Os bits inseridos no *buffer* de entrada do decodificador são sequencialmente deslocados e armazenados em um registrador temporário para comparação;

c) A cada ciclo de *clock,* a lógica de comparação verifica se os bits atualmente armazenados no registrador temporário correspondem a algum código de Huffman conhecido. Esta comparação é feita por meio de consulta a uma *Lookup Table* – LUT (em português: Tabela de Consulta) que contém todo o dicionário de símbolos de Huffman previamente gerado, conforme descrito na subseção 3.1.

d) Caso a sequência de bits armazenada no registrador temporário corresponda a algum registro da LUT, o módulo conversor é acionado e o símbolo de Huffman é então convertido para o caractere ASCII correspondente, que por sua vez é armazenado no *buffer* de saída;

e) Se o atual conteúdo do registrador temporário não corresponder a nenhum registro da LUT, a lógica de comparação incrementa o ponteiro do *buffer* de entrada, de modo que o próximo bit seja deslocado para o registrador temporário, para uma nova consulta à LUT no próximo ciclo de *clock*;

f) Por fim, o processo de decodificação de uma regra é concluído quando um caractere de “avanço de linha” (código ASCII 00001010) é decodificado e detectado. Nesse momento, a regra armazenada no *buffer* de saída é liberada para o módulo NIDS principal, para que seja comparada com o pacote de rede sob análise, e uma nova sequência de bits passa a ser decodificada para comparação posterior.

De modo a garantir que cada consulta à LUT do decodificador seja realizada em apenas um ciclo de *clock*, essa tabela foi implementada por meio do uso de uma *Content-Addressable Memory* – CAM (em português: Memória Endereçável por Conteúdo). As CAMs constituem uma classe especial de *hardware* de armazenamento, cuja principal característica é viabilizar aplicações de busca de altíssima velocidade.

De forma geral, uma CAM é composta por matrizes de memória convencional, geralmente do tipo *Random Access Memory* – RAM (em português: Memória de Acesso Aleatório), com adição de circuitos de comparação paralelos que possibilitam que uma operação de busca em cada matriz de memória seja concluída em um único ciclo de *clock*. Isto é, diferentemente de uma RAM, em que as posições de memória são acessadas de forma sequencial, cada matriz de uma memória CAM pode ser pesquisada em paralelo com o uso de circuitos que comparam o valor de entrada com cada registro armazenado.

Conforme descrito por Chen, Chen e Summerville (2011), a tecnologia CAM é ideal para operações de busca em memória sem endereçamento, uma vez que ela é capaz de comparar simultaneamente os dados de entrada com uma lista inteira de padrões armazenados na memória. Essa característica única

proporciona às abordagens de busca baseadas em CAM um desempenho superior para comparação rápida de padrões. Tal desempenho resulta em uma redução de ordem de grandeza no tempo de busca, se comparado com outros métodos de busca em memória, tais como busca binária ou busca baseada em árvores (PENG e AZGOMI, 2001). O método de busca binária, por exemplo, possui complexidade de tempo logarítmico igual a $O(\log_2 n)$, considerando uma tabela de busca ordenada, de tamanho *n*. A abordagem baseada em CAM, por outro lado, possui complexidade de tempo constante O(1), por empregar circuitos de comparação paralela para concluir a busca em um único ciclo de *clock* (PAGIAMTZIS, 2007).

A visão conceitual da memória CAM implementada é apresentada na Figura 21. A entrada desse componente é uma palavra de busca contendo *n* bits, que é armazenada em um registrador e difundida através do barramento de busca para todas as posições da memória. Cada posição de memória possui como saída um bit de correspondência, que indica se a palavra armazenada nessa posição coincide com a palavra de busca (valor ‘1’, quando há correspondência) ou se é diferente (valor ‘0’, quando não há correspondência). Os bits de correspondência são então enviados para um circuito codificador que identifica a localização da posição de memória onde está armazenada a palavra correspondente.

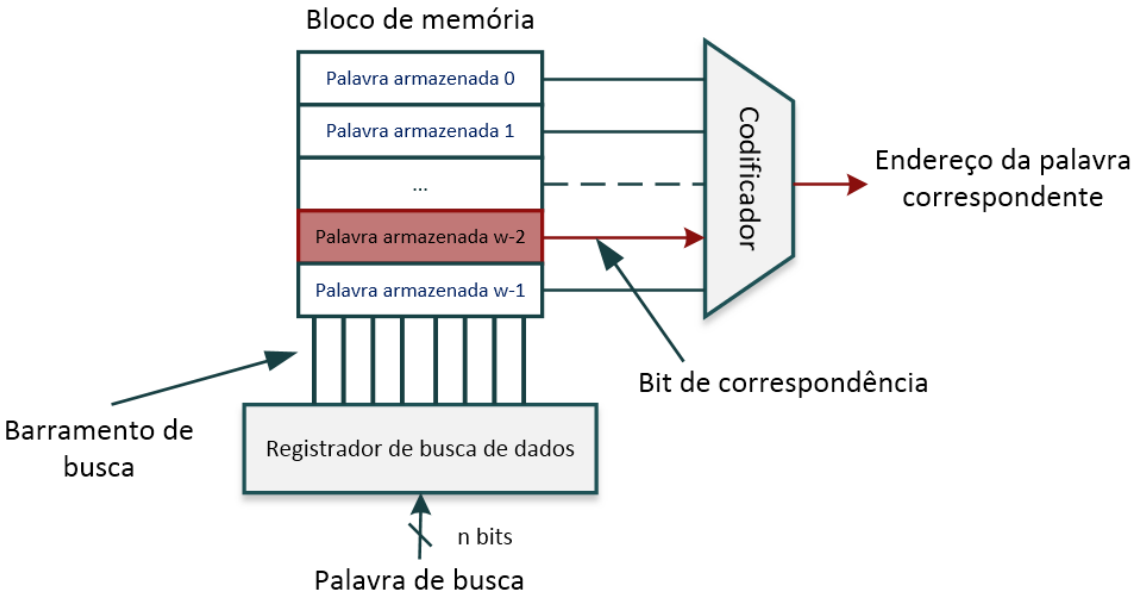


Figura 21 – Visão conceitual de uma CAM contendo *w* posições de memória.

Um codificador de prioridade também pode ser implementado em aplicações de CAM onde mais de uma posição de memória pode coincidir com a palavra de busca. Este codificador de prioridade seleciona a posição de memória correspondente que tiver maior prioridade para mapear o resultado da correspondência. Uma ordem de prioridade pode ser atribuída de modo que as palavras armazenadas nos primeiros endereços da memória possuam maior prioridade, por exemplo (PAGIAMTZIS e SHEIKHOLESLAMI, 2006).

Na Figura 22, é possível observar o bloco do módulo decodificador de Huffman implementado no Quartus II. Entre as entradas desse módulo, estão: os sinais de estímulo *clk (clock)*, *rst (reset)* e *ce (chip enable)*; o sinal *new_code*, que indica que um novo bloco de bits foi enviado da memória ROM para decodificação; o barramento *data_in*, onde se encontram os bits codificados; o sinal *done_reading*, que indica que todos os blocos da memória ROM foram enviados para decodificação; e o sinal *new_decode*, que indica que o processo de decodificação das regras deve ser reiniciado (por conta da análise de um novo pacote de rede, por exemplo). Entre as saídas do módulo decodificador, estão: o sinal *rule_out*, responsável por indicar que uma nova regra foi decodificada; o barramento *data_out*, contendo os bits da regra decodificados; o

sinal *next_code*, responsável por solicitar um novo bloco de bits codificados à memória ROM; e o sinal *done_decoding*, que indica que todos os blocos da memória ROM foram decodificados.

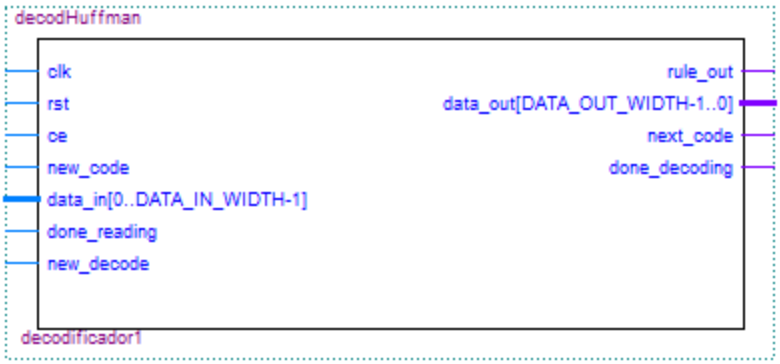


Figura 22 – Bloco do módulo decodificador de códigos Huffman, implementado no Quartus II.

3.3.2 Otimização da arquitetura de decodificação

Embora faça uso de memórias CAM para acelerar o processo de decodificação das regras codificadas com o método de Huffman, a arquitetura de *hardware* proposta na subseção 3.3.1 ainda possui limitações de desempenho. Isto se deve ao fato de os códigos gerados pelo algoritmo de Huffman possuírem comprimento variável, impedindo que o conjunto decodificador tenha conhecimento prévio sobre quantos bits, exatamente, devem ser carregados do *buffer* de entrada para o registrador de comparação, a fim de que cada caractere ASCII pudesse ser decodificado necessariamente em apenas um ciclo de *clock*. Esta característica dos códigos de Huffman limita o processo de decodificação descrito na subseção 3.3.1 à análise serial dos bits de entrada, isto é, não permite que um único módulo decodificador decodifique mais de um caractere ASCII ao mesmo tempo.

Visando reduzir as limitações de desempenho verificadas, o passo b) do processo de decodificação descrito na seção 3.3.1 foi modificado de modo que, no primeiro ciclo de *clock* da decodificação de cada novo caractere ASCII (isto é, no início de cada regra de detecção, ou logo após a decodificação de um caractere da regra atual), sejam deslocados para comparação, de uma única vez, os próximos *MIN_HUFFMAN_LENGTH* bits a partir do *buffer* de entrada. O parâmetro *MIN_HUFFMAN_LENGTH*, nesse caso, representa o comprimento do menor símbolo presente no dicionário de Huffman. Dessa forma, é possível garantir que uma parte considerável do comprimento de cada símbolo de Huffman seja analisada logo no primeiro ciclo de *clock* da sua decodificação, ao contrário da abordagem totalmente serial, em que apenas um bit por ciclo de *clock* é deslocado para análise. Com essa otimização, a probabilidade de conclusão de uma decodificação bem-sucedida nos primeiros ciclos de *clock* também aumenta, uma vez que, conforme mencionado anteriormente, no método de Huffman são atribuídos códigos menores para os caracteres mais frequentes no conjunto de dados.

Ainda com o objetivo de contornar a necessidade de decodificação parcialmente serial das regras de detecção de intrusão, esse trabalho propõe o agrupamento da arquitetura de *hardware* descrita na subseção 3.1.1 em uma estrutura paralelizada, com múltiplas instâncias de decodificação, conforme ilustrado na Figura 23.

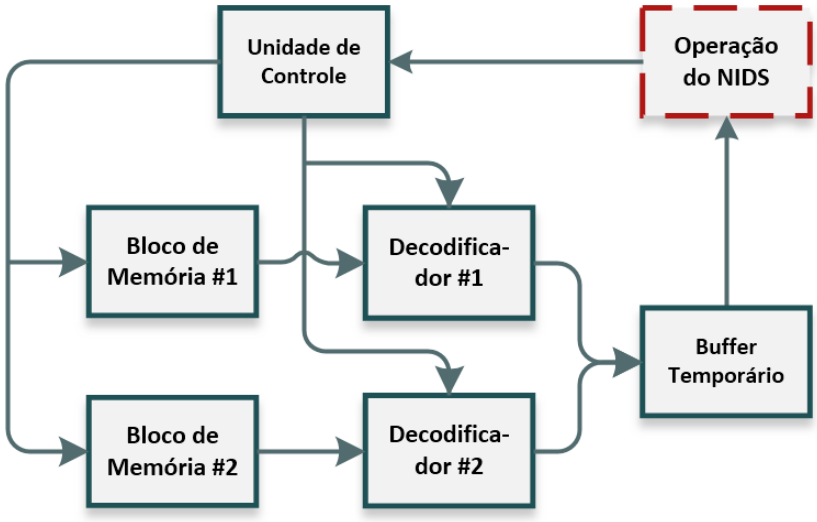


Figura 23 – Exemplo de uma estrutura paralela contendo dois decodificadores de Huffman.

Nessa estrutura, os conjuntos de regras codificadas são divididos e armazenados em blocos de memória independentes, que por sua vez são analisados e decodificados de forma paralela por decodificadores de Huffman dedicados – isto é, cada decodificador é responsável por analisar o bloco de memória ao qual está conectado. Os resultados do processo de decodificação são então armazenados em um *buffer* temporário, para que possam ser utilizados pelo módulo de detecção de intrusão. Caso a regra decodificada coincida com o pacote de rede que está sob análise do NIDS, este deve executar as operações para as quais está programado (por exemplo, registrar um alerta em memória) e acionar a unidade de controle para que os decodificadores reiniciem o processo de decodificação das regras, de modo que o processo de comparação do seu conteúdo com o próximo pacote recomece.

Visando simplificar a operação da arquitetura paralela descrita, a divisão das regras codificadas por blocos de memória foi feita de modo que cada bloco armazenasse a mesma quantidade de regras. Entretanto, uma vez que as regras de detecção de intrusão possuem comprimento variável, essa divisão simplificada pode resultar na sobrecarga de um ou mais blocos de memória, em benefício dos demais. Essa diferença de carga entre os blocos de memória pode reduzir os benefícios do paralelismo proposto, uma vez que parte dos módulos

decodificadores pode se tornar ociosa enquanto outros ainda permanecem em operação. Como possível solução para essa questão, pode-se considerar a posterior implementação de uma rotina de balanceamento dos conjuntos de regras entre os blocos de memória, de modo a considerar o comprimento total desses conjuntos, e não apenas a quantidade de regras presentes em cada bloco.

Para fins de avaliação do desempenho e do consumo de recursos obtidos pela arquitetura de *hardware* proposta para decodificação das regras de detecção de intrusão, três implementações distintas foram concebidas nesse trabalho:

- Uma implementação serial, contendo um único conjunto decodificador de regras de detecção de intrusão;
- Uma implementação utilizando a estrutura paralela descrita nessa subseção, contendo dois conjuntos decodificadores; e
- Uma implementação utilizando a estrutura paralela descrita nessa subseção, contendo quatro conjuntos decodificadores.

O esquemático completo da arquitetura paralela contendo dois conjuntos decodificadores de Huffman, desenvolvida no ambiente de desenvolvimento Quartus II, pode ser visto no Apêndice A. Os resultados de desempenho e consumo de recursos obtidos por cada uma destas implementações são descritos na seção de experimentos e análise dos resultados, a seguir.

4 EXPERIMENTOS E ANÁLISE DOS RESULTADOS

4.1 CODIFICAÇÃO E REARRANJO DAS REGRAS DE DETECÇÃO DE INTRUSÃO

Para avaliação da eficácia da aplicação da codificação de Huffman, no contexto de projetos de sistemas de detecção de intrusão em redes baseados em *hardware*, foram utilizadas 10.041 regras de detecção de intrusão do *software* Snort, pertencentes a dez subconjuntos distintos, disponibilizados em 24 de Setembro de 2013 (SNORT, 2014). Esses subconjuntos de regras foram codificados e rearranjados de acordo com os procedimentos descritos nas subseções 3.1 e 3.2, e cujos resultados podem ser observados na Tabela 2, que apresenta as seguintes informações:

- Categoria do subconjunto de regras codificado, conforme critérios definidos pela comunidade do Snort;
- Número total de regras de detecção de intrusão contidas no subconjunto;
- Espaço em memória exigido para armazenar as regras de detecção, quando utilizado o padrão de codificação original (ASCII); e
- Espaço em memória necessário para armazenamento das regras de detecção codificadas com o método de Huffman, quando associadas aos arranjos de bits convencional (isto é, uma regra por posição de memória) e contíguo. À direita das colunas relacionadas a cada tipo de arranjo, também é possível observar o percentual de economia de espaço de memória obtido com a sua utilização, em relação ao padrão de codificação original (ASCII).

Tabela 2 – Resultado da codificação de subconjuntos de regras do Snort

Subconjunto de regras do Snort		**Consumo de memória x codificações (Kilobits)**				
			Codificação de Huffman – arranjos			
Categoria	**Nº de regras**	**Codificação ASCII**	**Armazena-mento convencional**	**Economia**	**Armazena-mento contíguo**	**Economia**
Blacklist	2.001	10.341	7.310	29,32%	3.851	62,76%
Browser-Plugins	1.788	22.572	16.174	28,34%	7.205	68,08%
Malware-CNC	1.670	15.671	10.655	32,01%	4.549	70,97%
Server-Webapp	1.410	9.385	7.252	22,73%	3.161	66,31%
File-Identify	1.033	6.297	4.378	30,48%	2.175	65,46%
Malware-Backdoor	696	5.484	3.946	28,05%	1.497	72,71%
Pua-Adware	622	5.857	3.841	34,42%	1.785	69,53%
Exploit-Kit	493	4.327	2.846	34,22%	1.557	64,01%
Policy-Spam	246	1.246	818	34,34%	332	73,37%
Sql	82	384	266	30,58%	163	57,51%

O gráfico com os percentuais de economia de espaço de memória obtidos, referentes a cada subconjunto de regras codificado, é apresentado na Figura 24.

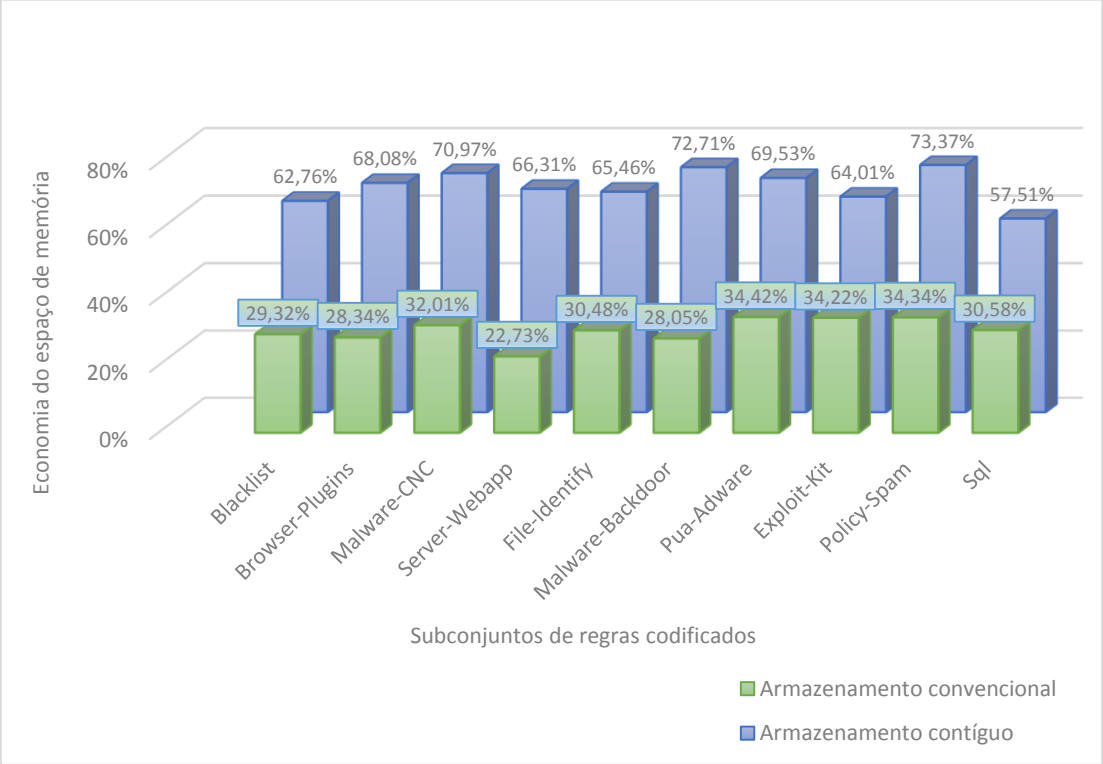


Figura 24 – Percentuais de economia de memória das categorias de regras analisadas.

A partir da observação da Tabela 2 e da Figura 24 é possível constatar que, entre os subconjuntos de regras analisados, o emprego da codificação de Huffman reduziu o consumo de memória entre 22,73% e 34,42% em relação ao padrão de codificação original (ASCII), quando mantido o arranjo de armazenamento de bits convencional. Nesse caso, a categoria de regra mais favorecida pela redução do consumo de memória foi a *Pua-Adware*, devido à maior quantidade de caracteres redundantes no seu arquivo de regras original e, consequentemente, ao maior rendimento do método de compressão de Huffman.

Quando combinada a codificação de Huffman com o rearranjo dos bits de forma contígua para melhor aproveitamento dos recursos de memória embarcada, a redução do espaço necessário para armazenamento das regras de detecção foi ainda mais significativa, variando entre 57,51% e 73,37%. Nesse segundo caso, a maior redução dos requisitos de espaço de memória foi

observada na categoria *Policy-Spam*, devido à grande variação de comprimento entre a maior regra e o comprimento médio das demais regras desse grupo, compensada pelo processo de rearranjo dos seus bits.

4.2 SIMULAÇÃO DAS ARQUITETURAS PARA DECODIFICAÇÃO DAS REGRAS

Para avaliação das arquiteturas de *hardware* implementadas para decodificação das regras de detecção de intrusão proposta na subseção 3.3, foi desenvolvido um ambiente de simulação e verificação, baseado em estímulo e resposta, através da ferramenta de simulação ModelSim (versão PE Student Edition 10.2c), fornecida pela Mentor Graphics. O ambiente de simulação desenvolvido abrangeu as três versões da arquitetura implementadas, isto é, aquela contendo apenas um conjunto decodificador e as duas implementações contendo decodificadores paralelos.

Uma vez que a especificação em lógica digital de um módulo que executa as funções de um NIDS completamente funcional se encontra fora do escopo desse trabalho, em seu lugar foi implementado um módulo mais simples, que realiza a comparação de cadeias de bits entre os dados armazenados em um *buffer* de cabeçalhos de pacotes (representando os cabeçalhos das camadas de Transporte e Rede de pacotes de rede fictícios interceptados pelo NIDS) e as regras de detecção de intrusão. Tendo em vista que o objetivo principal desse trabalho é propor uma arquitetura baseada no algoritmo de Huffman para otimização dos recursos de memória embarcada em projetos de NIDS implementados em *hardware*, e não os detalhes de operação de um módulo de detecção de intrusão propriamente dito, os experimentos relacionados à

avaliação da eficácia dessa proposta puderam ser conduzidos sem qualquer prejuízo.

A Figura 25 exemplifica o ambiente de testes (em inglês: *testbench*) desenvolvido para simulação e verificação da arquitetura implementada contendo dois decodificadores paralelos. Nesse ambiente, cada bloco de memória do projeto sob teste (em inglês: *Design Under Test* – DUT) foi carregado com um subconjunto de 500 regras do Snort, pertencentes às categorias listadas na Tabela 2 e codificadas com o método de Huffman (totalizando 1.000 regras e 490.894 caracteres ASCII codificados). Adicionalmente, o *buffer* de cabeçalhos de pacotes foi carregado com cadeias aleatórias de bits, geradas através do *software* Matlab, com o objetivo de representar cabeçalhos de pacotes de rede comuns (isto é, não detectáveis pelo NIDS). Entre essas cadeias aleatórias, foi inserido um segundo conjunto de cadeias de bits, em posições aleatórias do *buffer*, correspondentes a algumas das regras originais que foram codificadas e armazenadas nos blocos de memória. O objetivo desse segundo conjunto de entradas no *buffer* de cabeçalhos é coincidir com as regras de detecção após elas terem sido decodificadas, representando assim cabeçalhos de pacotes de rede suspeitos.

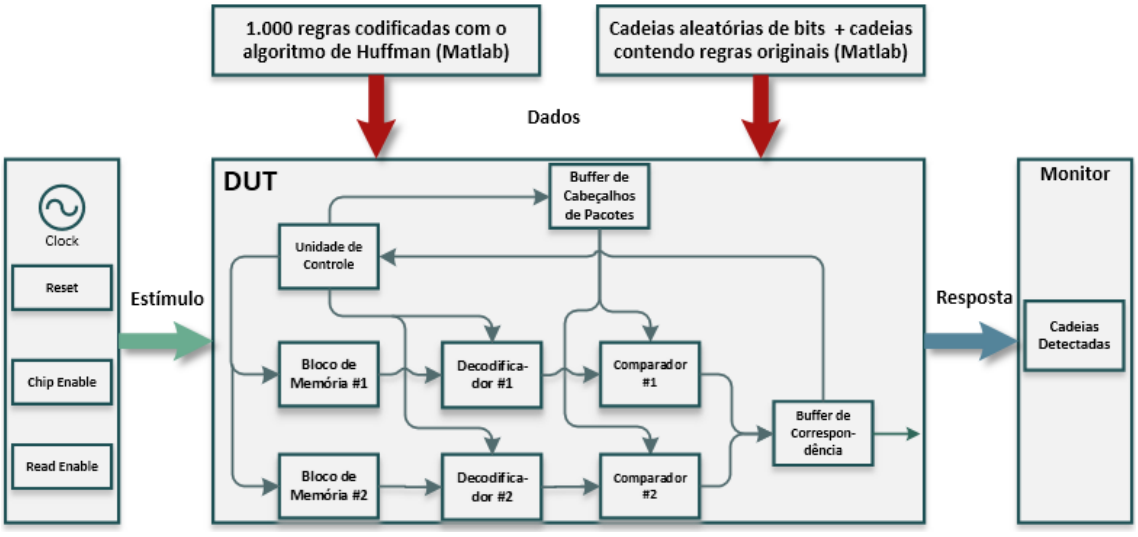


Figura 25 – Ambiente de simulação e verificação da arquitetura proposta.

A simulação do projeto compreendeu a comparação de cada entrada armazenada no *buffer* de cabeçalhos com as 1.000 regras de detecção de intrusão codificadas e armazenadas nos blocos de memória. Para viabilizar esse processo, uma sequência de estímulos de entrada (pulsos de *clock* e sinais de *reset*, *chip enable* e *read enable*) foi aplicada ao DUT, de modo que as regras fossem decodificadas pelos módulos decodificadores e, posteriormente, analisadas pelos módulos de comparação. Na ocorrência de uma correspondência (isto é, quando o conteúdo de uma regra decodificada coincidir com a entrada do *buffer* de cabeçalhos sob análise), o conteúdo dessa entrada é armazenado em um *buffer* de correspondência, para que seja enviado como resposta ao ModelSim. A ferramenta de simulação, por sua vez, exibe em seu console todas as entradas do *buffer* que correspondem a alguma regra de detecção, de modo que essas entradas podem ser comparadas com aquelas geradas (e previamente conhecidas) pelo modelo no Matlab, atestando assim o correto funcionamento da arquitetura implementada. No Apêndice B, é apresentada uma tela da ferramenta ModelSim com a simulação das formas de onda, na qual é possível acompanhar visualmente os estímulos inseridos no DUT, bem como as respostas obtidas.

Para a mensuração da eficiência de decodificação das três versões da arquitetura proposta, foi considerado o quantitativo total de ciclos de *clock* necessários para que todas as 1.000 regras armazenadas nos blocos de memória fossem decodificadas e comparadas com uma única entrada do *buffer* de cadeias de bits, representando a análise de um pacote de rede. Para o cálculo da latência introduzida pelo processo de decodificação e comparação das regras, foi utilizada a frequência nominal padrão do dispositivo FPGA Altera Cyclone IV E, de 50 MHz (Altera, 2014).

De modo a avaliar os ganhos de desempenho obtidos pela implementação das arquiteturas paralelas, foram consideradas duas importantes leis utilizadas para estimativa do desempenho teórico de arquiteturas computacionais paralelas: as leis de Amdahl (1967) e de Gustafson (1988), que serão brevemente descritas a seguir.

A lei de Amdahl afirma que o ganho de desempenho obtido através da paralelização de uma arquitetura computacional está limitado à sua carga computacional que é intrinsecamente serial. Em outras palavras, o aumento do paralelismo em um ambiente computacional por um número *N* (isto é, a multiplicação de núcleos de processamento por um fator *N*) nunca resultará em um ganho de desempenho igual a *N* (Gillespie, 2009). De forma simplificada, a lei de Amdahl pode ser traduzida na seguinte equação:

$$Ganho\ (N) = 1/(S + \ (1 - S)/N), \tag{1}$$

onde *N* representa o número de núcleos de processamento, e *S* representa o percentual da carga computacional executada de forma serial (expressa como um número decimal entre 0 e 1).

A lei de Gustafson, por sua vez, constitui um contraponto à lei de Amdahl. Enquanto esta assume que o problema computacional possui tamanho fixo, aquela considera que, à medida que o tamanho do problema aumenta, o ganho de desempenho obtido com a sua paralelização também aumenta. Isto é, o tamanho global de um problema deve aumentar proporcionalmente ao número de núcleos de processamento (*N*), enquanto o tamanho da porção serial do problema tende a se manter constante à medida que *N* aumenta (Gillespie, 2009). De forma simplificada, a lei de Gustafson é representada pela seguinte equação:

$$Ganho\ (N) = S + N(1 - S), \tag{2}$$

onde *N* representa o número de núcleos de processamento, e *S* representa o percentual da carga computacional executada de forma serial (expressa como um número decimal entre 0 e 1).

A Tabela 3 apresenta os resultados da simulação obtidos pela arquitetura serial contendo apenas um decodificador, e pelas suas duas variações contendo as estruturas paralelas. Para cada versão do projeto simulada, essa tabela apresenta as seguintes informações:

- O número de ciclos de *clock* necessários para decodificar e comparar todas as 1.000 regras do subconjunto;
- A taxa média de decodificação, medida em caracteres ASCII decodificados por ciclo de *clock*;
- A latência total, em milissegundos, gerada pelo processo de decodificação e comparação das 1.000 regras de detecção de intrusão com cada entrada do *buffer*; e
- A aceleração real obtida pela implementação das arquiteturas paralelas, quando comparada com o desempenho da arquitetura serial, bem como os respectivos valores teóricos estimados pelas leis de Amdahl e de Gustafson.

Tabela 3 – Eficiência de decodificação das três versões do projeto implementadas

Versão do projeto	**Ciclos de *clock* necessários**	**Taxa média de decodi-ficação**	**Latência total (ms)**	**Aceleração obtida**		
				Real	**Amdahl**	**Gustafson**
Serial	1.337.188	0,37	26,74	–	–	–
Paralelo (2x)	711.108	0,69	14,22	1,88	1,79	1,88
Paralelo (4x)	400.354	1,23	8,01	3,34	2,62	3,47

Conforme exibido na Tabela 3, os resultados da simulação indicaram que o projeto serial contendo um único decodificador foi capaz de decodificar todo o subconjunto de 1.000 regras a uma taxa média de 0,37 caracteres ASCII por ciclo de *clock*, introduzindo uma latência total de 26,74 ms durante o processo. De modo similar, as estruturas paralelas contendo dois e quatro decodificadores

obtiveram taxas médias de decodificação de 0,69 e 1,23 caracteres ASCII por ciclo de *clock*, representando latências totais de 14,22 ms e 8,01 ms, respectivamente. Ademais, esses indicadores de desempenho ainda podem ser melhorados em um fator teórico de até 9,5 vezes no mesmo dispositivo FPGA, visto que a sua frequência de operação pode atingir 472,5 MHz, quando alimentado com um oscilador externo (Altera, 2013). Cabe destacar, no entanto, que o alcance deste fator máximo de melhoria dependerá dos caminhos críticos do circuito após este ter sido sintetizado.

No que diz respeito à melhoria de desempenho obtida com o emprego das arquiteturas paralelas, foi possível observar que o desempenho da arquitetura contendo dois decodificadores em paralelo foi 1,88 vezes superior em relação à arquitetura serial contendo apenas um decodificador. De modo análogo, quando utilizada a implementação com quatro decodificadores, esse desempenho foi 3,34 vezes superior em relação à arquitetura serial. Embora tenham sido superiores aos ganhos teóricos estimados pela Lei de Amdahl (de 1,79 e 2,62, respectivamente), os ganhos de desempenho reais obtidos se aproximaram bastante dos ganhos teóricos estimados pela Lei de Gustafson (de 1,88 e 3,47, respectivamente). Tal resultado mostra que, para as arquiteturas implementadas nesse trabalho, a carga computacional a ser executada de forma serial para a solução do problema computacional em questão (isto é, a decodificação de regras de detecção de intrusão codificadas com o algoritmo de Huffman) tende a ser constante.

4.3 Síntese em *hardware* das arquiteturas para decodificação das regras

A arquitetura de *hardware* proposta foi sintetizada com a ferramenta Altera Quartus II para o dispositivo FPGA Altera Cyclone IV E (EP4CE115F29C7) anteriormente mencionado e, de forma análoga à etapa de simulação, a etapa de síntese considerou as três versões de projeto implementadas. Visando a manutenção da consistência de toda a análise realizada ao longo desse trabalho, nesta etapa foi utilizado o mesmo subconjunto contendo 1.000 regras do Snort, utilizado na etapa de simulação.

A Tabela 4 apresenta os resultados da síntese em *hardware* obtidos pela arquitetura serial contendo apenas um decodificador, e pelas suas duas variações contendo estruturas paralelas. Para cada versão de projeto, essa tabela apresenta as seguintes informações:

- A área consumida do dispositivo FPGA, considerando o total de elementos lógicos utilizados, e o seu respectivo percentual de consumo de área; e
- O fator de aumento do consumo de área pelas arquiteturas paralelas, em relação à arquitetura serial.

Tabela 4 – Consumo de área das três arquiteturas sintetizadas

Versão do projeto	**Área do FPGA consumida**		**Fator de aumento de área**
	Elementos lógicos	**Percentual**	
Serial	28.033	24,49%	–
Paralelo (2x)	56.236	49,12%	2,01
Paralelo (4x)	102.041	89,16%	3,64

Conforme pode ser visto na Tabela 4, o relatório de síntese apresentou um consumo de 28.033 elementos lógicos (24,49% da capacidade do dispositivo) para a arquitetura contendo um único decodificador; 56.236 elementos lógicos (49,12%) para a arquitetura contendo dois decodificadores paralelos; e 102.041 elementos lógicos (89,16%) para a arquitetura contendo quatro decodificadores em paralelo. No que diz respeito ao fator aumento do consumo de área, é possível constatar que o uso de uma estrutura paralela com dois decodificadores dobrou o consumo de elementos lógicos, em relação ao consumo apresentado pela arquitetura serial. Da mesma forma, o emprego de quatro decodificadores paralelos aumentou o consumo de elementos lógicos em uma proporção de 3,64 vezes em relação à arquitetura serial. A Figura 26 apresenta o gráfico que ilustra o comparativo do percentual de consumo de área de circuito do dispositivo FPGA, entre as três versões da arquitetura implementada.

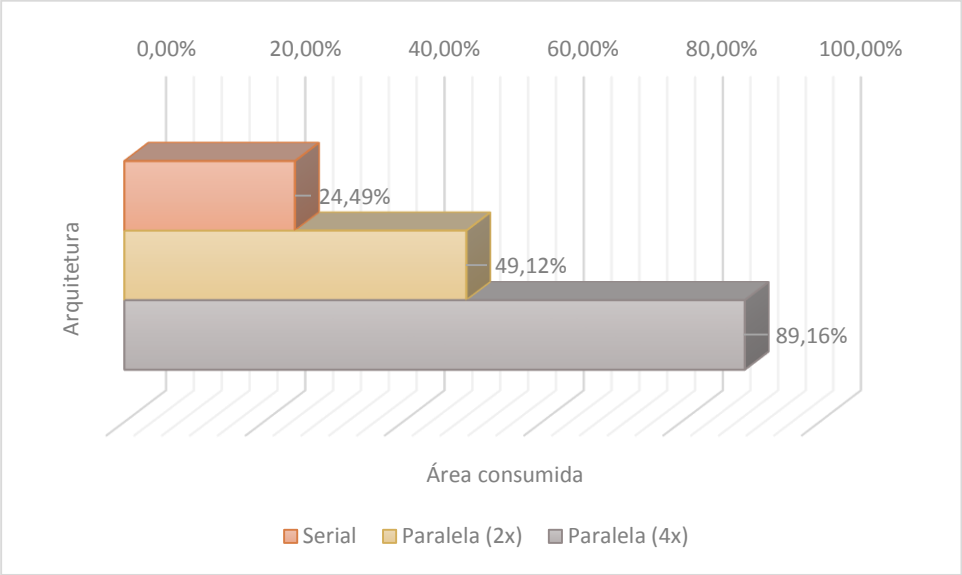
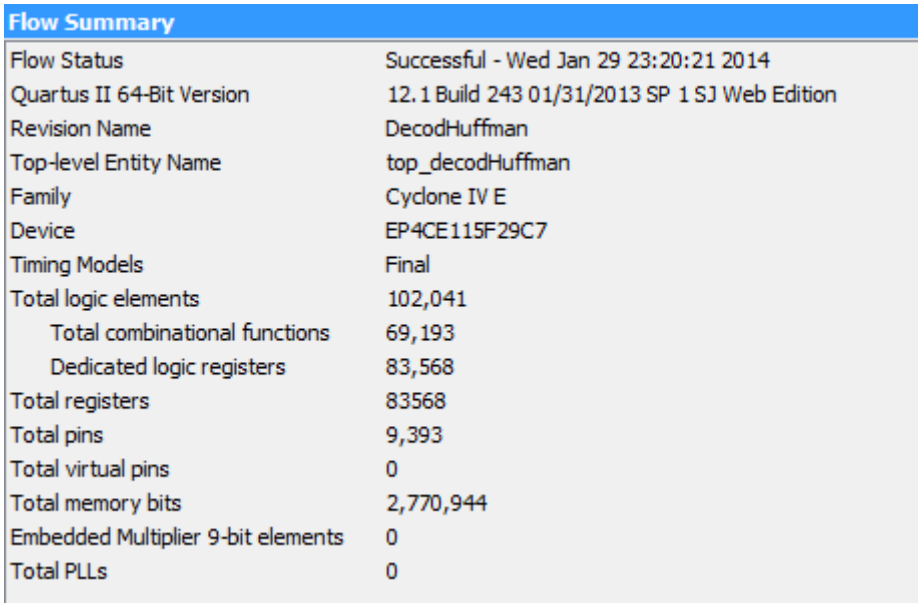


Figura 26 – Comparativo de área de lógica consumida entre as três versões de arquitetura.

Adicionalmente, o relatório de síntese (Figura 27) indicou um consumo de 2.770.944 bits da memória embarcada do dispositivo FPGA, para armazenar as 1.000 regras de detecção de intrusão codificadas com o método de Huffman. Isto representa uma economia global de 70,47% no consumo de recursos de

memória embarcada, quando comparado com o consumo original de 9.384.000 bits para esse determinado subconjunto de regras, quando utilizada a abordagem de armazenamento convencional (padrão ASCII) baseada em busca linear, na qual cada regra ocupa uma posição de memória distinta. Esse resultado confirma a economia de recursos de memória embarcada do *hardware* estimada na etapa de codificação via *software*, e comprova que o método proposto nesse trabalho é aplicável na prática.



Flow Summary	
Flow Status	Successful - Wed Jan 29 23:20:21 2014
Quartus II 64-Bit Version	12.1 Build 243 01/31/2013 SP 1 SJ Web Edition
Revision Name	DecodHuffman
Top-level Entity Name	top_decodHuffman
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	102,041
Total combinational functions	69,193
Dedicated logic registers	83,568
Total registers	83568
Total pins	9,393
Total virtual pins	0
Total memory bits	2,770,944
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Figura 27 – Relatório de síntese da arquitetura paralela contendo quatro decodificadores, no Quartus II.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esse trabalho propôs uma arquitetura paralela, baseada no algoritmo de Huffman, integrando *software* e *hardware* para codificação, armazenamento e decodificação de regras de detecção de comportamentos suspeitos e possíveis ameaças às redes de computadores. O seu objetivo principal era a otimização dos recursos de memória embarcada em projetos de *hardware* dedicado para esse fim.

A arquitetura proposta foi implementada em três componentes principais, sendo dois no nível de *software,* responsáveis pelos processos de codificação das regras e rearranjo dos bits para otimização do método de compressão, respectivamente; e um no nível de *hardware*, responsável pelo processo de decodificação das regras de modo que elas sejam compreendidas pelo módulo responsável pela detecção de intrusão.

Os experimentos conduzidos para a avaliação da eficácia da arquitetura proposta utilizaram mais de 10.000 regras de detecção de intrusão, pertencentes a dez subconjuntos distintos de regras atuais do sistema de detecção de intrusão Snort. O uso da codificação de Huffman para compressão das regras de detecção do Snort apresentou resultados expressivos, representando uma redução no consumo dos recursos memória de até 34,43%. Quando combinadas as abordagens de codificação de Huffman e de rearranjo dos bits em um espaço de armazenamento contíguo, esses experimentos resultaram em uma redução ainda mais significativa no consumo dos recursos da memória embarcada do dispositivo FPGA, atingindo uma economia de até 73,37%.

A Tabela 5 apresenta um breve resumo comparativo entre os resultados obtidos por este trabalho e aqueles alcançados pelas demais propostas descritas na subseção 2.6.2. Nessa tabela, são apresentados: os autores dos trabalhos; as suas respectivas propostas; a quantidade e o escopo das regras utilizadas durante os experimentos; a abordagem para detecção de intrusão em redes

permitida, com base no escopo das regras utilizadas; e o percentual máximo de economia dos recursos de memória obtido.

Tabela 5 – Comparativo entre os resultados obtidos e as propostas descritas na subseção 2.6.2

Trabalho	**Proposta**	**Quantidade e escopo das regras**	**Abordagem para NIDS**	**Percentual máximo de economia**
Yi *et al.* (2007)	*Hash* de árvore *bottom-up*	2.770 (padrões de assinatura)	Inspeção profunda de pacotes	31,64%
Chen, Summerville e Chen (2009)	Decomposição de *strings* em dois passos	25.802 (padrões de assinatura)	Inspeção profunda de pacotes	77,24%
Nikitakis e Papaefstathiou (2008)	Filtros de Bloom em múltiplos níveis	4.000 (regras de classificação)	Classificação de pacotes	–
Guinde, Ziavras e Rojas-Cessa (2010)	Agrupamento de campos	10.000 (regras de classificação)	Classificação de pacotes	–
Esse trabalho	Codificação de Huffman e rearranjo de bits	10.041 (regras de classificação e padrões de assinatura)	Classificação e inspeção profunda de pacotes	73,37%

A partir das informações apresentadas na Tabela 5, é possível constatar que o método de otimização proposto nesse trabalho obteve um percentual de economia dos recursos de memória consideravelmente superior ao descrito na proposta apresentada por Yi *et al.* (2007) (de 31,64%), e ligeiramente inferior àquele descrito no trabalho publicado por Chen, Summerville e Chen (2009) (de 77,24%), enquanto os demais autores não mencionaram o percentual exato de economia alcançado pelas suas propostas. Entretanto, é importante salientar que, entre os métodos descritos, o proposto por esse trabalho foi o único que abrangeu a compressão de todo o escopo das regras de detecção de intrusão (isto é, tanto regras de classificação de pacotes quanto padrões de assinatura), o que torna possível a adoção conjunta de ambas as principais abordagens para

detecção de intrusão no projeto de NIDS implementados em *hardware*: a classificação e a inspeção profunda de pacotes.

No que tange à adoção de paralelismo, cabe destacar que ganhos de desempenho efetivos de até 3,34 vezes foram alcançados na etapa de decodificação das regras em *hardware*, através da utilização de estruturas paralelas contendo dois e quatro decodificadores, quando comparados com o desempenho da arquitetura contendo um único decodificador. Em relação à área, para a arquitetura com quatro decodificadores, o relatório da síntese em *hardware* indicou um consumo próximo ao limite de um dispositivo considerado de baixo custo pelo seu fabricante, ao mesmo tempo que foi garantida a economia dos recursos de memória.

Adicionalmente, embora tenha sido avaliada com regras de detecção de intrusão do *software* Snort, a arquitetura proposta pode ser facilmente adaptada para projetos de NIDS implementados em *hardware* baseados em outras soluções de detecção de intrusão disponíveis no mercado. Isto é possível pois o método de compressão de Huffman, utilizado nesse trabalho, não leva em conta a semântica das regras de detecção a serem codificadas, e sim a quantidade de caracteres ASCII redundantes presentes no conjunto de regras em questão.

Cabe ainda lembrar que a utilização de memórias embarcadas no *chip*, em projetos de *hardware* dedicado para detecção de intrusão, traz diversas vantagens sobre o uso de memórias mais baratas e externas ao dispositivo, conforme já mencionado na sessão de introdução. Nesse sentido, é importante destacar que, por se tratar de um recurso limitado e bastante valioso em aplicações embarcadas, a otimização do espaço de memória embarcada do *chip* é essencial para a redução do custo de produção e para o aumento da competitividade no projeto de sistemas digitais, no contexto industrial.

Os resultados relatados nesse trabalho foram parcialmente apresentados e publicados nos anais do Simpósio Brasileiro de Engenharia de Sistemas Computacionais – SBESC 2013, realizado em Novembro de 2013, na cidade de Niterói – RJ, e cuja publicação (FREIRE *et al.*, 2013), disponível no Apêndice C,

foi uma das três mais bem avaliadas da Trilha de Sistemas Críticos deste congresso. Tal reconhecimento indica que a arquitetura proposta tem potencial para ser aperfeiçoada e utilizada na prática, de modo que os resultados obtidos até o momento poderão servir como base para a realização de um projeto posterior, que envolve a prototipagem de um NIDS funcional em FPGA, com baixo consumo dos recursos de memória, a ser empregado para análise do tráfego de rede da Universidade Federal da Bahia. Uma vez que as atividades de inspeção de pacotes e detecção de intrusão em redes demandam taxas de transferência cada vez mais altas, é necessário dedicar atenção especial ao processo de decodificação do conjunto de regras codificado, de modo que o protótipo do NIDS implementado em *hardware* seja capaz de alcançar taxas de transferência reais aceitáveis.

Entre as possíveis soluções para a melhoria do desempenho da arquitetura proposta, são sugeridas: a otimização do módulo decodificador de códigos de Huffman; a implementação de memórias *cache* para armazenamento das últimas regras decodificadas; e o aperfeiçoamento da arquitetura superescalar, de modo que integre múltiplas instâncias de decodificação paralelas. Tais abordagens são consideradas, e poderão ser objeto de discussão em um trabalho posterior.

REFERÊNCIAS

ABUHMED, T.; MOHAISEN, A.; NYANG, D. A Survey on Deep Packet Inspection for Intrusion Detection Systems. **Magazine of Korea Telecommunication Society**, 24, n. 11, fev. 2008. 25-36.

AHO, A. V.; CORASICK, M. J. Efficient string matching: An aid to bibliographic search. **Communications of the ACM**, 18, n. 6, 1975. 333–340.

ALTERA CORPORATION. Cyclone IV Device Handbook, Volume 1, May 2013. Disponível em: <http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf>. Acesso em: Janeiro de 2014.

ALTERA CORPORATION. Cyclone IV FPGA Family: Lowest Cost, Lowest Power, Integrated Transceivers. Disponível em: <http://www.altera.com/devices/fpga/cyclone-iv/cyiv-index.jsp>. Acesso em: Janeiro de 2014.

AMDAHL, G. M. **Validity of the single-processor approach to achieving large scale computing**. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). Reston, Va.: AFIPS Press. 1967. p. 483-485.

BAKER, Z. K.; PRASANNA, V. K. **A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs**. The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM ’04). 2004.

BARR, M. Programmable Logic: What's it to Ya? **Embedded Systems Programming**, June 1999. 75-84.

BOYER, R. S.; MOORE, J. S. A fast string searching algorithm. **Communications of the ACM**, 20, n. 10, 1977. 76–172.

BROWN, F. Introduction to Digital Circuits. **Weber State University**, 2013. Disponível em: <http://faculty.weber.edu/fonbrown/EE2700/text.html>. Acesso em: Fevereiro de 2014.

CHEN, H.; CHEN, Y.; SUMMERVILLE, D. H. A Survey on the Application of FPGAs for Network Infrastructure Security. **the IEEE Communications Surveys and Tutorial**, 13, n. 4, 2011.

CHEN, H.; SUMMERVILLE, D. H.; CHEN, Y. **Two-stage decomposition of SNORT rules towards efficient hardware implementation**. 7th International

Workshop on Design of Reliable Communication Networks, DRCN 2009. 2009. p. 359-366.

CHO, Y. H.; NAVAB, S.; MANGIONE-SMITH, W. H. **“Specialized Hardware for Deep Network Packet Filtering”**. In Proceedings of International Conference on Field Programmable Logic and Applications (FPL). 2002.

CLARK, C. R.; SCHIMMEL, D. E. **Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns**. in Proc. 11th ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL). 2003. p. 956.

COMMENTZ-WALTER, B. **A string matching algorithm fast on the average**. Proceedings of ICALP. 1979. p. 118-132.

CORMEN, T. H. et al. **Introduction to Algorithms**. 3rd edition. ed. MIT Press & McGraw-Hill, 2009.

EDWARDS, J. The Essential Guide to Intrusion Detection and Prevention Systems. **IT Security**, 24 Abril 2008. Disponível em: <http://www.itsecurity.com/features/essential-guide-idps-042408/>. Acesso em: Janeiro de 2014.

FOSTER, J. C. IDS: Signature versus anomaly detection. **Search Security**, May 2005. Disponível em: <http://searchsecurity.techtarget.com/tip/IDS-Signature-versus-anomaly-detection>. Acesso em: Janeiro de 2014.

FPGA DEVELOPER. List and comparison of FPGA companies, 2011. Disponível em: <http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html>. Acesso em: Janeiro de 2014.

FREDERICK, K. K. Network Intrusion Detection Signatures, Part Five. **Symantec**, 03 November 2010. Disponível em: <http://www.symantec.com/connect/articles/network-intrusion-detection-signatures-part-five>. Acesso em: Janeiro de 2014.

FREIRE, E. S.; DUARTE, A. A.; OLIVEIRA, W. L. A. D. **Evaluation of the Huffman Encoding for memory optimization on hardware network intrusion detection**. Simpósio Brasileiro de Engenharia de Sistemas Computacionais – SBESC 2013. Niterói-RJ: 2013.

GILLESPIE, M. Intel Corporation. **Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications**, 2009. Disponível em: <http://software.intel.com/en-us/articles/amdahls-law-gustafsons-trend-and-the-performance-limits-of-parallel-applications>. Acesso em: Janeiro de 2014.

GOOGLE. Google IPv6. **IPv6 Statistics**, 23 Janeiro 2014. Disponível em: <http://www.google.com/intl/en/ipv6/statistics.html>. Acesso em: Janeiro de 2014.

GUINDE, N.; ZIAVRAS, S. G.; ROJAS-CESSA, R. **Efficient packet classification on FPGAs also targeting at manageable memory consumption**. 4th International Conference on Signal Processing and Communication Systems (ICSPCS). 2010. p. 1-10.

GUSTAFSON, J. Reevaluating Amdahl's Law. **Communications of the ACM**, 31, n. 5, 1988. Disponível em: <http://www.johngustafson.net/pubs/pub13/amdahl.pdf>. Acesso em: Janeiro de 2014.

HUFFMAN, D. A. A Method for the Construction of Minimum Redundancy Codes. **Proc. IRE**, 40, n. 9, September 1952. 1098-1101.

HUTCHINGS, B. L.; FRANKLIN, R.; CARVER, D. **Assisting Network Intrusion Detection with Reconfigurable Hardware**. Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02). 2002. p. 111-120.

INTERNET ENGINEERING TASK FORCE. **Request for Comments: 1122. Requirements for Internet Hosts -- Communication Layers**. 1989.

ISO/IEC. **14496-1:2010. Information technology — Coding of audio-visual objects — Part 01: Systems.** International Organization for Standardization, International Electrotechnical Commission. 2010.

ISO/IEC. **15444-12:2012. Information technology — JPEG 2000 image coding system — Part 12: ISO base media file format.** International Organization for Standardization, International Electrotechnical Commission. 2012.

KARTHIKEYAN, A. R.; INDRA, A. Intrusion Detection Tools and Techniques –A Survey. **International Journal of Computer Theory and Engineering**, 2, n. 6, December 2010.

KNUTH, D. **The Art of Computer Programming:** Semi-numerical Algorithms. 3rd. ed. Addison-Wesley, v. 2, 1997.

KONG, C. et al. **A common on-board hardware architecture for intrusion detection system**. 1st International Conference on Multimedia Information Networking and Security, MINES. 2009.

KUROSE, J.; ROSS, K. **Computer Networking:** A Top-Down Approach. 6th ed. ed. Pearson Addison-Wesley Longman Publishing Co., Inc., 2012.

LOINIG, J.; WOLKERSTORFER, J.; SZEKELY, A. **Packet filtering in gigabit networks using fpgas**. in Austrochip 2007 Proc. 15th Austrian Workshop on Microelectronics. 2007. p. 53-60.

MANSOUR, M. F. **Efficient Huffman Decoding with Table Lookup**. Acoustics, Speech and Signal Processing, ICASSP 2007. 2007. p. 53-56.

MOORE, D. et al. The Spread of the Sapphire/Slammer Worm. **CAIDA:** The Cooperative Association for Internet Data Analysis, 2003. Disponível em: <http://www.caida.org/publications/papers/2003/sapphire/sapphire.html>. Acesso em: Janeiro de 2014.

MOORE, D.; SHANNON, C. The Spread of the Code-Red Worm (CRv2). **CAIDA:** The Cooperative Association for Internet Data Analysis, 2013. Disponível em: <http://www.caida.org/research/security/code-red/coderedv2_analysis.xml>. Acesso em: Janeiro de 2014.

NIKITAKIS, A.; PAPAEFSTATHIOU, I. **A memory-efficient FPGA-based classification engine"**. Field-Programmable Custom Computing Machines, FCCM '08. 2008. p. 53-62.

PAGIAMTZIS, K. Content-Addressable Memory Introduction, 25 jun. 2007. Disponível em: <https://www.pagiamtzis.com/cam/camintro/>. Acesso em: Janeiro de 2014.

PAGIAMTZIS, K.; SHEIKHOLESLAMI, A. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. **IEEE Journal of Solid-State Circuits**, 41, n. 3, Mar 2006. 712–727.

PENG, M.; AZGOMI, S. **Content-Addressable memory (CAM) and its network applications**. International IC Taipei Conference Proceedings. Taipei: 2001.

PENG, T.; LECKIE, C.; RAMAMOHANARAO, K. Survey of network-based defense mechanisms countering the dos and ddos problems. **ACM Comput. Surv.**, 39, n. 1, 2007. 3.

PONEMON INSTITUTE. 2013 Fourth Annual Cost of Cyber Crime Study: United States, 2013. Disponível em: <http://www.hpenterprisesecurity.com/ponemon-2013-cost-of-cyber-crime-study-reports>. Acesso em: Janeiro de 2014.

ROESCH, M. **Snort - Lightweight Intrusion Detection for Networks**. LISA '99 Proceedings of the 13th USENIX conference on System administration. 1999. p. 229-238.

SCARFONE, K.; MELL, P. **Guide to Intrusion Detection and Prevention Systems (IDPS):** Recommendations of the National Institute of Standards and Technology. Special Publication 800-94. Gaithersburg, MD: National Institute of Standards and Technology, 2007.

SEN, S. **Performance Characterization and Improvements of SNORT as an IDS**. 2006.

SGUIL. **Sguil:** The Analyst Console for Network Security Monitoring. Disponível em: <http://sguil.sourceforge.net>. Acesso em: Janeiro de 2014.

SNORT. **Snort IDS**, 2014. Disponível em: <http://www.snort.org>. Acesso em: Janeiro de 2014.

SONG, H.; LOCKWOOD, J. W. **Efficient packet classification for network intrusion detection using fpga**. in Proc. 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. Monterey, California, USA: ACM. 2005. p. 238-245.

SURICATA. **Suricata IDS**, 2014. Disponível em: <http://suricata-ids.org>. Acesso em: Janeiro de 2014.

SYMANTEC CORPORATION. 2013 Norton Report. Disponível em: <http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=norton-report-2013>. Acesso em: Janeiro de 2014.

WU, S.; MANBER, U. **A fast algorithm for multi-pattern searching**. 1994.

XILINX INC. FPGA vs. ASIC. Disponível em: <http://www.xilinx.com/fpga/asic.htm>. Acesso em: Janeiro de 2014.

YI, S. et al. **Memory-Efficient Content Filtering Hardware for High-Speed Intrusion Detection Systems**. Proceedings of the 2007 ACM Symposium on Applied Computing. Seoul, Korea: 2007. p. 264-269.

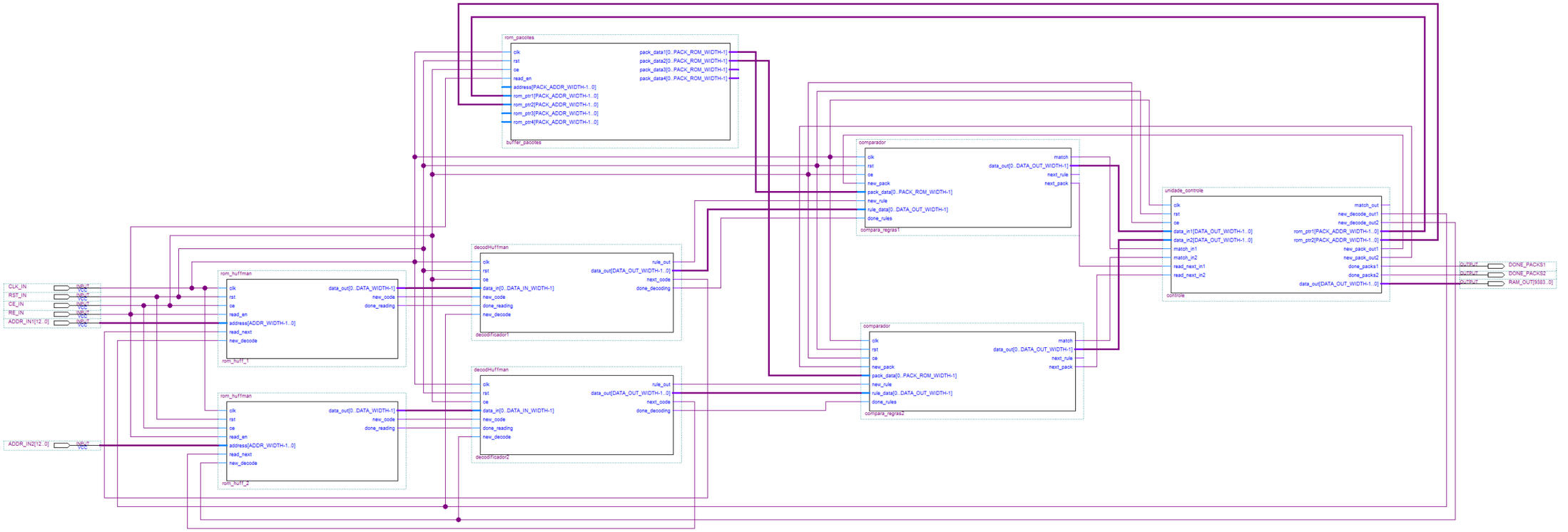
YUSUF, S. et al. **A combined hardware-software architecture for network flow analysis**. in IEEE International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05). Las Vegas, USA: 2005.

ZEIDMAN, B. All about FPGAs, 22 March 2006. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1274496>. Acesso em: Janeiro de 2014.

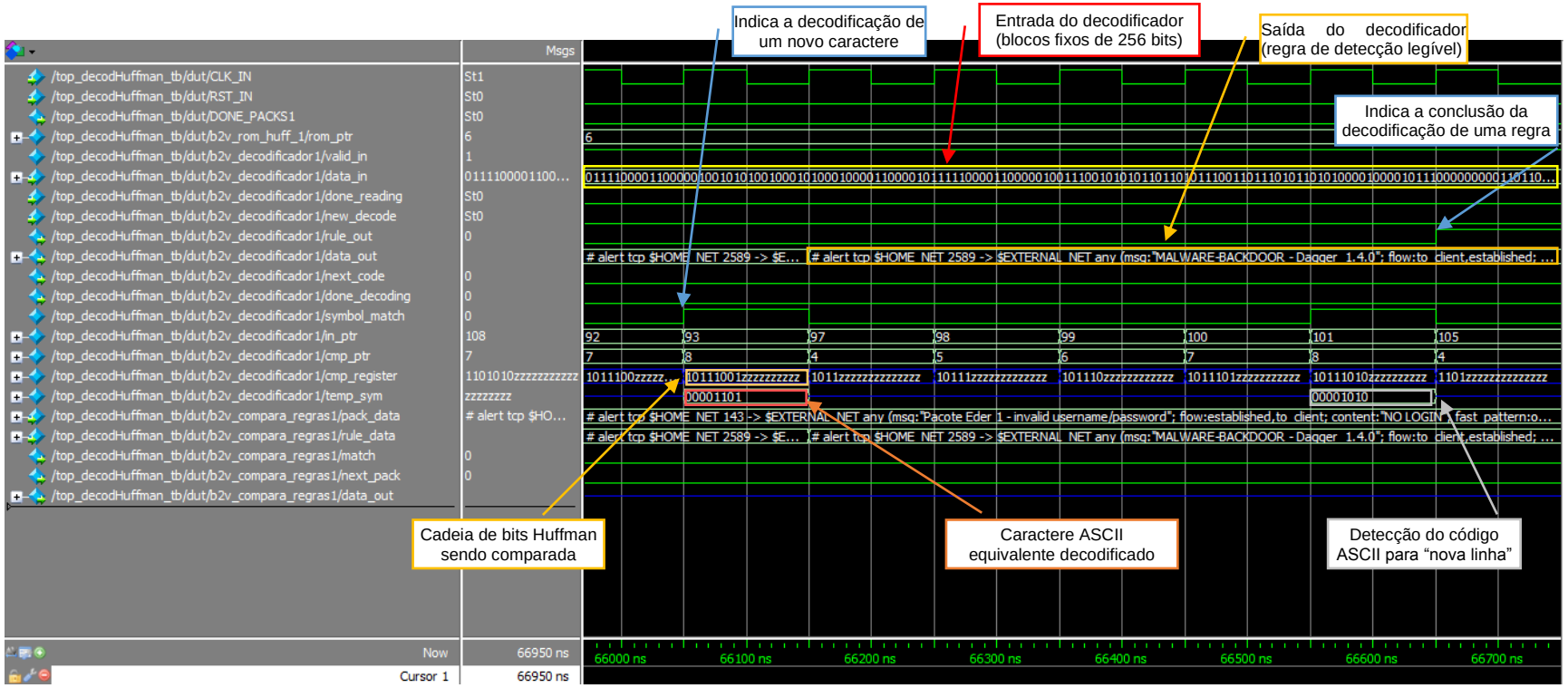
ZHANG, K. **Embedded Memories for Nano-Scale VLSIs. Integrated Circuits and Systems Series**. Springer, 2009.

APÊNDICES

APÊNDICE A – Arquivo esquemático da arquitetura paralela contendo dois módulos decodificadores de Huffman



APÊNDICE B – Simulação da arquitetura de decodificação na ferramenta ModelSim



APÊNDICE C – Artigo publicado no SBESC 2013

Evaluation of the Huffman Encoding for memory optimization on hardware network intrusion detection

Eder Freire, Angelo Duarte, Wagner Oliveira
Federal University of Bahia
Polytechnic School, 40210-630 – Salvador – BA – Brasil
eder@ufba.br, angeloduarte@ecomp.uefs.br, wlao@ufba.br

***Abstract*—The design of specialized hardware for Network Intrusion Detection has been subject of intense research over the last decade due to its considerably higher performance compared to software implementations. In this context, one of the limiting factors is the finite amount of memory resources versus the increasing number of threat patterns to be analyzed. This paper proposes an architecture based on the Huffman algorithm for encoding, storage and decoding of these patterns in order to optimize such resources. We have made tests with simulation and synthesis in FPGA of rule subsets of the Snort software, and analysis indicate a saving of up to 73 percent of the embedded memory resources of the chip.**

Keywords—network intrusion detection; FPGA; memory optimization; Huffman encoding

I. INTRODUCTION

Network Intrusion Detection Systems (NIDS) are designed to detect and prevent several security threats to the traffic of computer networks. Their operation relies on the comparison of packets traveling through the network with previously known threat patterns, allowing the detection and containment of harmful and therefore undesirable traffic.

According to [1], the structure of a network packet is divided in two types of data: packet header, containing fixed-length and fixed-format control information; and payload, containing user data with variable length and format. Similarly, there are two main approaches of network intrusion detection: packet classification, which focuses on the analysis of packet headers; and deep packet inspection, which is dedicated to the processing of the packet payloads and their match with patterns of known signatures. As described by [2], both approaches can be combined to compose a complete NIDS appliance.

Intrusion detection solutions implemented purely in software, such as Snort [3] and Suricata [4], are quite popular due to their ease of use, configuration and update. These tools use rules that have a proper syntax to detect network threats and anomalies, and are typically stored in ASCII text files (one rule per line), where each character occupies eight bits of memory storage. As described by [5] and illustrated in Fig. 1, a Snort intrusion detection rule can be split in two parts:

1) Header information, responsible for the packet classification task. They have only one set of static fields, as follows: action to be taken, network protocol, source address and ports, traffic direction, and destination address and ports.

2) Rule options, containing non-mandatory fields, with variable definitions. Among these settings are the patterns of known signatures, preceded by the *content* keyword. These signature patterns definitions are used during the deep packet inspection operation, and can be represented by one or more regular expressions.

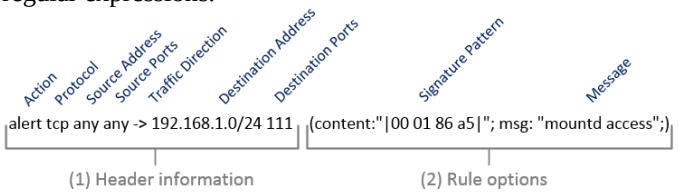


Fig. 1. Typical syntax of a Snort rule.

One of the problems encountered by the software-based NIDS solutions is the rapid growth in the number of threats in computer networks and, consequently, of the amount of detection rules. In the Snort example, the number of rules has risen from about 1,700 in 2003 to over 20,000 in May 2013, representing an increase of 1000% over a period of 10 years [3, 5]. Given this strong growth, the approaches based purely on software began to find processing bottlenecks in recent years, since with the increase of transfer rates in the new network technologies, it has become necessary to detect a much larger number of suspicious behaviors in a shorter period of time.

The implementation of NIDS directly in hardware arises as a solution to this processing deficit issue, and has been subject of intense research since the beginning of the last decade, due to its considerably higher ability of comparison and detection of threat patterns, compared to software implementations, and mainly, due to the possibility of using native parallel processing. In this context, stand out the reconfigurable devices such as FPGAs (Field-Programmable Gate Arrays), used by most studies in this area due to their speed and flexibility of design, implementation and simulation of digital logic architectures [6].

After the analysis of other published papers on this subject, we have noticed a design trend of hardware-optimized systems oriented to verification and detection of threat patterns which meet high data transfer rates (over 1 Gbps), such as those proposed by [7] and [8]. However, in a few selected publications we perceived the specific interest to devise methods in order to optimize the hardware memory consumption before the growing number of network threats and intrusion detection rules.

This paper aims to present a computing architecture proposal integrating software and hardware based on the Huffman algorithm [9] for encoding, storage and decoding of detection rules of suspicious behaviors in computer networks, which can be used in NIDS approaches implemented in hardware, in order to optimize the resources of embedded memory. In addition, we propose a method for storage of the encoded rules bits in a contiguous arrangement, in order to reduce the wastage caused by unused space in memory positions.

The remainder of this paper is organized as follows. Section II presents related works to the proposed here. Section III presents the proposed computing architecture for the optimization of memory resources through encoding, rearrangement and decoding of intrusion detection rule sets. Section IV presents the experiments performed and the analysis of results obtained. Finally, Section V presents the concluding remarks and the possibilities of future works.

II. RELATED WORK

According to the literature review performed, the few studies found involving clear proposals for optimization of memory resources in NIDS implemented in hardware are very punctual, focused on specific technologies and detection algorithms. Some of these proposals are briefly described as follows.

The work presented by [10] proposed a hash implementation of a bottom-up tree, aiming the increase of performance and reduction of memory consumption for storing patterns of 2,770 signatures present in the Snort rule database. By using this method, it was possible to reduce the space occupied by the signatures of about 512KB to 350KB, which represents a saving of about 32%. It is worth mentioning, however, that this approach specifically addressed only the fields of signatures patterns, which represent a small part of each Snort rule.

In the work described in [11], its authors proposed a method to decompose text strings in two steps in order to eliminate redundancies in sets of characters present in the Snort signature patterns. Although it achieved a saving of 77% when compressing such patterns, this approach also did not address all of the other remaining information contained in the rule set.

The approach proposed by [12], in turn, decomposed multiple-field packet classification rules into single-field rules, combining them through multi-level Bloom filters. Although the proposed method has made possible the storage of 4,000 packet classification rules (where each rule generally takes only a few tens of bytes) in just 178KB of memory, its authors did not report the percentage of compression achieved. Moreover, since it focused only on the packet classification information, this work also left most of the detection rules content without treatment.

III. PROPOSED ARCHITECTURE

The architecture proposed in this work is subdivided into three main components, which are detailed in the following subsections:

1) Encoding, by means of software, of the detection rules set into an array of bits through the Huffman algorithm.

2) Rearrangement, via software, of the encoded bits for optimization of the memory space used.

3) Decoding and reconstruction of the rules, through hardware, for execution of the main NIDS functions.

A. *Encoding of the rule set*

The Huffman encoding [9] is a lossless method of data compression that is based on the probabilities of occurrence of the symbols found in a given data set, so that the most frequent symbols are represented by fewer bits. Although it was initially proposed more than six decades ago, this compression method even today still finds application in compression standards widely used, such as JPEG, MPEG-2 and ISO [13].

In a simplified manner, the Huffman method analyzes all the data set to be encoded and calculates the occurrence probability of each symbol of the data. These probabilities are sorted, grouped and summed in pairs to form a binary tree whose leaves represent the symbols, and whose edges correspond to the binary values '0' or '1'. Fig. 2 illustrates a binary tree constructed through the Huffman algorithm.

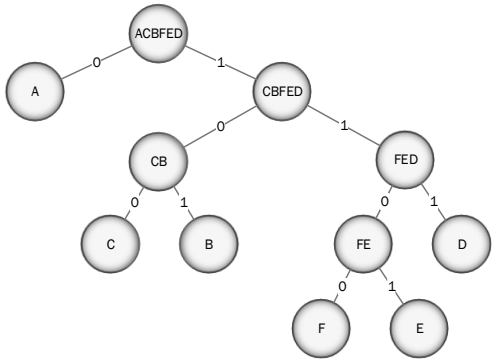


Fig. 2. Example of a Huffman binary tree.

As can be seen in Table I, when the binary tree is traversed from the root to each of its leaves, it is possible to form a variable length code assigned to each leaf, considering the binary values found at the edges covered. These codes are then assigned to the symbols represented by the leaves, forming a Huffman symbols dictionary. Since the generated codes have variable length, an also useful information of the dictionary is the length of each code. The detailed process of character encoding using the Huffman method can be found in [14].

TABLE I. EXAMPLE OF A HUFFMAN SYMBOLS DICTIONARY

Symbol	Huffman Code	Code length
A	0	1
B	101	3
C	100	3
D	111	3
E	1101	4
F	1100	4

To make possible the encoding of the detection rule set, we developed a set of scripts in the Matlab® computing environment to perform the following operations, as illustrated in the flowchart of Fig. 3:

1) General analysis of the text file containing the original rules and counting of incidence of each ASCII character, in order to calculate its probability of occurrence.

2) Construction of the Huffman symbols dictionary through the Huffman algorithm, based on the calculated probabilities for each ASCII character. This dictionary is then stored into a memory file whose data are disposed similarly to Table I, for later hardware decoding. The original file pointer is also "rewinded", in order to allow a second analisys of the rules for encoding.

3) Individual analysis of each rule in the original file and encoding through the Huffman method, using the dictionary previously generated. The "line feed" character (ASCII code 00001010) found at the end of each rule is also encoded, as it will be used to detect the rules boundaries during the decoding process. Each encoded rule is then stored into a binary file that can be used to load a hardware memory. This process is repeated until there are no more rules to be analyzed.

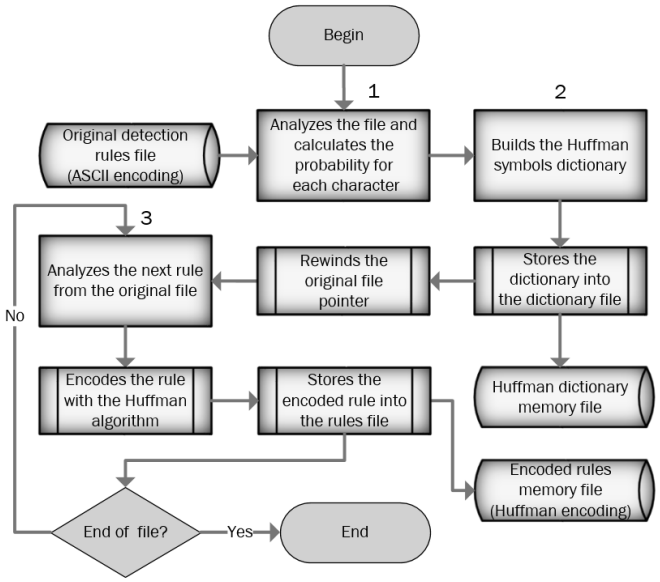


Fig. 3. Encoding process of the detection rules.

B. *Rearrangement of the encoded bits in a contiguous storage space*

According to our research, in the design of conventional hardware-based NIDS that employ a linear search technique, each detection rule usually occupies a distinct memory position [15]. This arrangement allows for a faster comparison of the incoming network packets with the rules stored, as the hardware requires only a few clock cycles to fetch each rule from the memory and perform the inspection. However, such design strategy causes wastage of memory resources, since its word length is determined based upon the largest rule of the set. Thus, the hardware synthesis tool completes each memory position with invalid padding bits (typically zeroes) whenever it is not completely filled by rule bits, as can be seen in Fig. 4 (a).

Once encoded with the Huffman method, the bits of the rules must be sequentially decoded so that the process does not result in ambiguities in the reconstruction of the original ASCII characters. To take advantage of such requirement and in order to reduce the wastage noticed, we propose the rearrangement of the encoded bits on a contiguous fashion, so that the entire space of the memory positions can be used optimally. That is, whenever a certain memory position contains unused space by a given rule, this space will be allocated to store the initial bits of the following rule and so on, as shown in Fig. 4 (b). Following the logic of this arrangement, invalid bits will only be admitted in the last memory position, where there are no more valid bits to be stored (Fig. 4 (c)).

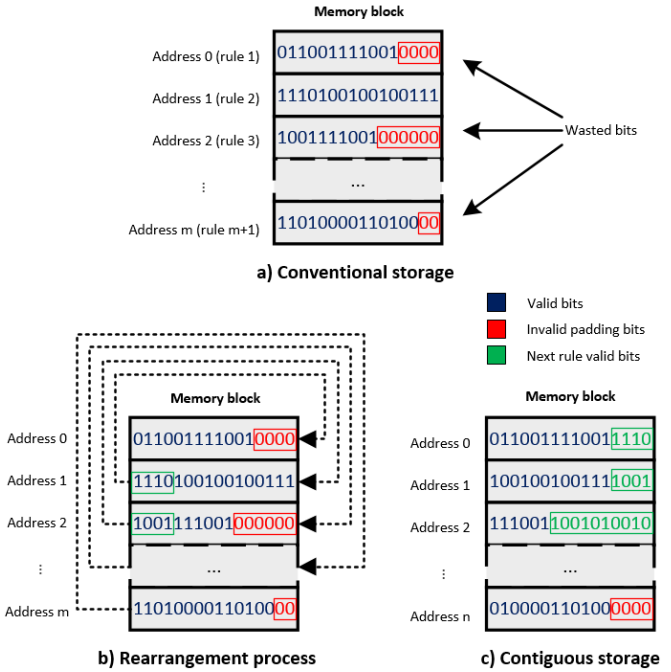


Fig. 4. Conventional versus contiguous memory storage.

As a result of this rearrangement, the total number of bits occupied by using the contiguous storage tends to be considerably smaller than the number of bits occupied by conventional storage, though both approaches have exactly the same number of valid bits.

The proposed rearrangement was implemented through scripts in the Matlab® environment, following the steps described in the flowchart of Fig. 5, which are summarized as follows:

1) Sequential analysis of each encoded rule (R) stored in the file containing the conventional arrangement, and calculation of its bit length (L).

2) For every rule analyzed, concatenation of its bits into an arrangement with the previous rule bits (A = [A R]) and increment of the total length of concatenated bits (T = T + L).

3) After the rules analysis and concatenation of all its bits, calculation of the number of memory positions required to

store all the rules (P = ceil (T / M)), where M is the predefined length for the memory positions and ceil() is an approximation function to the nearest integer greater than or equal to its argument. The number of valid bits in the last memory position is also registered, given the possibility of existence of invalid padding bits in this case.

4) Sequential reading of the bits arrangement in blocks of fixed length equal to M, and storage of these blocks into a new binary file (one block per line). The number of memory positions required is also decreased (P = P - 1) until the bits of the arrangement are exhausted.

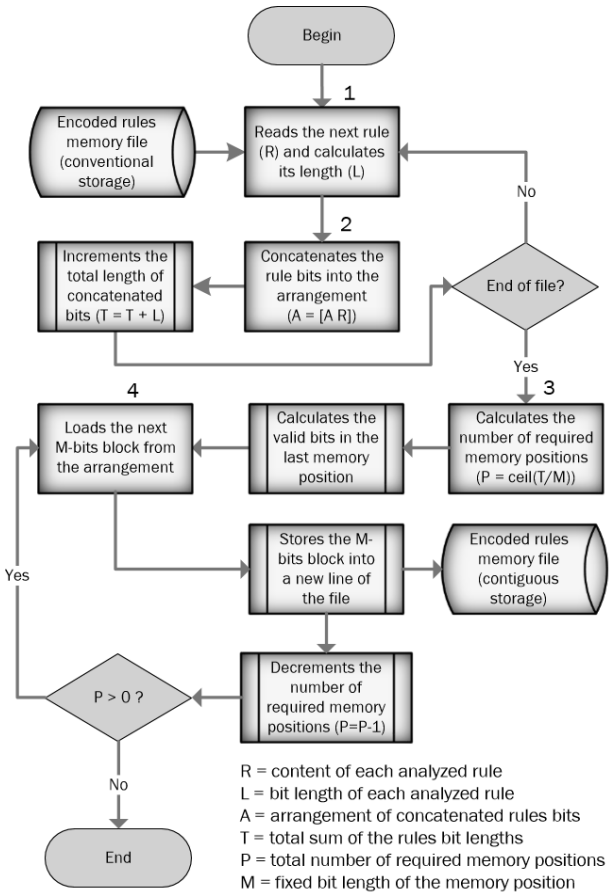


Fig. 5. Contiguous rearrangement of the memory space.

The implementation of the steps described in this subsection results in a contiguous binary file to be loaded into a hardware ROM memory with M x P dimensions, i.e. containing M bits of length by P positions of depth.

C. *Decoding and reconstruction of the intrusion detection rules*

In order that a hardware-implemented NIDS can analyze the rule set encoded with the proposed method, there must be a previous decoding process of each rule. To carry out this stage, we have developed and synthesized in digital logic a set of modules for decoding Huffman codes, by using the Altera Quartus II development environment together with the Verilog hardware description language. The resulting architecture is shown in Fig. 6.

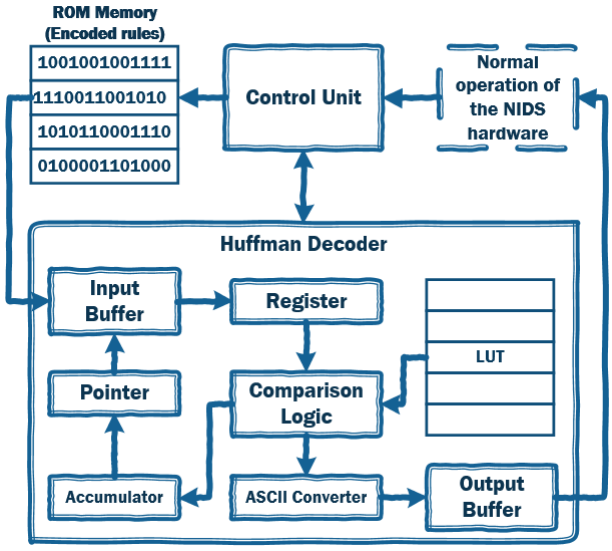


Fig. 6. Hardware architecture for decoding the intrusion detection rules.

The hardware-implemented architecture consists of three main modules: a control unit, responsible for driving the other modules and arbitrate the digital signals exchanged between them; a ROM memory containing the encoded rules and whose bits have been rearranged according to the method proposed in subsection III-B; and a Huffman decoder set based on Lookup Table, which was designed in a similar manner to that proposed by [13]. These modules work together to meet the following specifications:

1) When the intrusion detection module needs to compare a given network packet against the rule set, it requests its decoding to the control unit, which proceeds to transmit sequential fixed-length bit words from the ROM memory to the decoder input buffer.

2) The bits inserted in the input buffer of the decoder are sequentially shifted and stored into a temporary register for comparison.

3) For each clock cycle, the comparison logic checks whether the bits currently within the register corresponds to a known Huffman code. This comparison is done by querying a Lookup Table (LUT), which contains the entire Huffman symbols dictionary. To ensure that each query is performed in only one clock cycle, the LUT has been implemented by using a CAM (Content-Addressable Memory), which provides fast matching through the employment of parallel comparison circuitry.

4) If the bit sequence currently within the comparison register matches any record of the LUT, the converter module is triggered and the Huffman symbol is then converted to the corresponding ASCII character and stored into the output buffer.

5) If the register content does not match any record of the LUT, the comparison logic increments the pointer of the input buffer, so that its next bit is shifted to the temporary register for a new query to the LUT in the next clock cycle.

6) Finally, the decoding process of a rule is completed when a "line feed" character is decoded and detected. At this point, the rule stored in the output buffer is released to the main NIDS module for packet comparison, and a new bit sequence starts to be decoded for a subsequent comparison.

IV. Experiments and Analysis of Results

In order to evaluate the effectiveness of the Huffman encoding application in the context of a hardware-based NIDS, we used different subsets of Snort rules, released in May 2013 [3]. These rule subsets were encoded according to the procedures described in subsections III-A and III-B, whose results can be seen in Table II. This table contains the following information: category of the rule subset, number of rules in the subset, memory space required to store the rules when using the original encoding (ASCII), and memory space required for storage with the Huffman coding, when associated with the conventional and contiguous bit arrangements. At the right side of the column related to each arrangement, it is also possible to observe the percentage of the memory space saving obtained with it.

TABLE II. Encoding Results of Snort Rule Subsets

Snort Rule Subsets		**Memory consumption x encodings (Kilobits)**				
		ASCII encoding	***Huffman encoding – arrangements***			
Category	***# of rules***		***Conventional storage***	***Saving***	***Contiguous storage***	***Saving***
Browser-Plugins	1,726	21,789	15,613	28.34%	6,942	68.14%
Blacklist	1,693	8,749	6,317	27.81%	3,296	62.33%
Malware-CNC	1,484	13,926	9,468	32.01%	3,995	71.31%
Server-Webapp	1,353	9,006	6,956	22.76%	2,928	67.48%
File-Identify	1,028	6,267	4,356	30.50%	2,153	65.65%
Malware-Backdoor	694	5,369	3,872	27.88%	1,488	72.29%
Pua-Adware	619	5,829	3,822	34.42%	1,775	69.54%
Malware-Other	459	3,389	2,424	28.48%	1,281	62.21%
Exploit-Kit	345	3,028	1,969	34.97%	1,159	61.73%
Netbios	251	1,574	1,093	30.55%	736	53.26%
Policy-Spam	245	1,241	818	34.06%	330	73.42%
Malware-Tools	114	683	470	31.14%	303	55.59%
DoS	88	566	400	29.34%	180	68.11%

One can observe that among the rule subsets analyzed, the use of Huffman encoding reduced memory consumption between 22.76% and 34.97%, when maintaining the conventional storage of bits. By combining the Huffman encoding with the rearrangement of bits for better utilization of memory, the reduction of the space required for storing the rules was even more significant, ranging from 53.26% to 73.42%. In the first case, the category that most suffered a space reduction was *Exploit-Kit*, due to the larger amount of redundant characters in its original rules file and consequently to the higher yield of the compression method. In the second case, the highest reduction of memory requirements was observed in the *Policy-Spam* category, due to the wide variation in length between the longest rule and the average length of the other rules of the group, counteracted by the rearrangement of its bits. The graph with the saving percentage of memory space for each rule subset is depicted in Fig. 7.

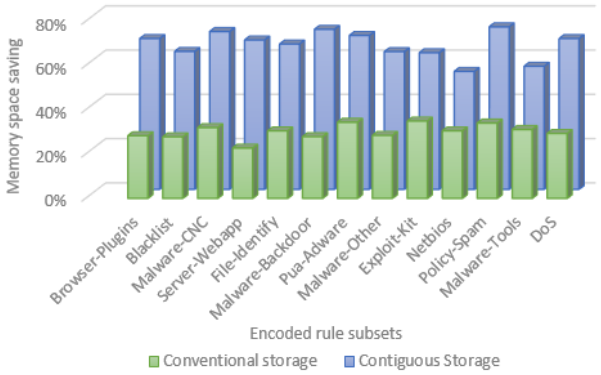


Fig. 7. Graph of memory saving percentages of the encoded rule subsets.

To simulate the hardware decoding architecture proposed in the subsection III-C, we used the ModelSim simulation tool. Once the digital logic module that performs the functions of a fully functional NIDS is still under development, we used a simpler module in its place, which performs the comparison of bit strings between the data from a buffer (representing the network packets intercepted by the NIDS) and the intrusion detection rules. Considering that the goal of this work is to demonstrate the optimization of memory resources obtained with the application of the Huffman encoding and not the details of a NIDS module operation, the experiments related to the applicability of this proposal were conducted without any side effects.

The hardware architecture proposed was synthesized in the Quartus II tool, along with a subset of 1,000 Snort rules of various categories, for an Altera Cyclone IV E FPGA chip (EP4CE115F29C7). The synthesis report showed a consumption of 17,600 logic elements (15.41% of the chip capacity) and a saving of 67.72% in the consumption of memory resources for this given rule subset. Similarly, the synthesis of the rule subsets listed in Table II confirmed the hardware memory saving estimated in the software encoding stage, demonstrating that the method is applicable in practice.

To measure the efficiency of the decoder architecture, we simulated the decoding process of rule samples from the mentioned 1,000 rules subset. Table III exemplifies the results obtained by the decoding process of five sample rules found between the shortest and the largest rule of the subset. This table shows the sample rule number, its bit length before and after the decoding process back to the ASCII standard, the number of clock cycles required to decode, and the decoding rate, measured in average clock cycles per character decoded.

TABLE III. DECODING EFFICIENCY OF SAMPLE RULES

Sample Rule	Bit length		Clock cycles to decode	Decoding rate (cycles/char)
	Huffman encoding	*ASCII encoding*		
1	1,077	1,584	504	2.55
2	2,953	4,376	1,358	2.48
3	4,811	7,072	2,235	2.53
4	7,011	9,768	3,443	2.82
5	9,046	12,624	4,448	2.82

As shown in Table III, the simulation indicated that the decoder design was able to decode a new ASCII character at an average rate ranging between 2.48 and 2.82 clock cycles, which represents an average decoding latency between 50 ns and 56 ns per character, when using the standard FPGA device frequency of 50 MHz. However, this performance can be further improved by 9.5 times on the same device, whereas its operation frequency can reach 472.5 MHz when powered with an external oscillator [16].

V. FINAL REMARKS AND FUTURE WORK

This paper proposes an architecture integrating software and hardware based on the Huffman algorithm for encoding, storage and decoding of detection rules of suspicious behaviors in computer networks, in order to optimize embedded memory resources in hardware design dedicated for this purpose.

The experiments conducted to evaluate the effectiveness of the proposed architecture used 13 subsets of current Snort rules. When combined the approaches of Huffman encoding with contiguous bits storage rearrangement, these experiments resulted in a significant reduction in the use of embedded memory resources of the FPGA chip. The hardware synthesis report indicated a modest consumption of logic elements on a device considered of low cost, given the memory space saving achieved. Is worth mentioning that, although it has been evaluated with detection rules of the Snort IDS, the proposed architecture can be easily adapted to projects of hardware-implemented NIDS based on other solutions available in the market, since the compression method used does not take into account the semantics of detection rules stored, but the frequency of occurrence of each character of the rule set.

It is also noteworthy that, although it is possible to use external memories to the chip (and therefore less expensive) like flash memories in the implementation of a dedicated hardware for network intrusion detection, there are several factors in favor of the use of embedded memories. As pointed out by [17], some of these factors are: reduced access time, lower power consumption and increased reliability of the device. In this sense, we also emphasize that, because it is a very valuable and limited resource in embedded systems applications, the optimization of embedded memory space is essential for reducing the cost of production and increasing the competitiveness of a hardware design in the industrial context.

The results reported in this paper are serving as a basis for the next stage of the project, already in progress, which is the prototyping in FPGA of a functional NIDS, with low memory consumption, to be used for network traffic analysis at the Federal University of Bahia. Once the activities of packet inspection and intrusion detection demand increasingly higher transfer rates, we are giving particular attention to the decoding process of the compressed rules, so that the NIDS prototype can achieve acceptable real transfer rates. Solutions like the optimization of the decoder module, the implementation of a cache memory to store the last decoded rules, and the integration of a superscalar architecture with multiple instances of parallel decoding are now being studied, and will be the subject of discussion in a future work.

REFERENCES

[1] J. Kurose and K. Ross, Computer Networking: A Top-Down Approach, 6th ed., Pearson. Addison-Wesley Longman Publishing Co., Inc., 2012.

[2] H. Song and J.W. Lockwood, "Efficient packet classification for network intrusion detection using fpga", in Proc. 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. Monterey, California, USA: ACM, 2005, pp. 238-245.

[3] Snort IDS. Available at: http://www.snort.org.

[4] Suricata IDS. Available at: http://suricata-ids.org.

[5] S. Sen, "Performance Characterization and Improvements of SNORT as an IDS", Bell Labs Report, 2006.

[6] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering", In Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 2002.

[7] I. Sourdis and D. Pneumatikatos, "Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system", in Lecture notes in computer science. Berlin; New York: Springer Berlin / Heidelberg, 2003, vol. Volume 2778/2003, pp. 880-889.

[8] S. Yusuf, W. Luk, M. Sloman, N. Dulay, and E. Lupu. "A combined hardware-software architecture for network flow analysis", in IEEE International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, USA, 2005.

[9] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc. IRE, Vol. 40, No. 9, pp. 1098-1101, September 1952.

[10] S. Yi, Kim, J. Oh, J. Jang, G. Kesidis, and C. R. Das, "Memory-Efficient Content Filtering Hardware for High-Speed Intrusion Detection Systems", Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 264-269, Seoul, Korea, 11~15 March, 2007.

[11] H. Chen, D.H. Summerville, and Y. Chen, "Two-stage decomposition of SNORT rules towards efficient hardware implementation", Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on, pp. 359-366.

[12] A. Nikitakis and I. Papaefstathiou, "A memory-efficient FPGA-based classification engine", Field-Programmable Custom Computing Machines, 2008, FCCM '08, pp. 53,62, 14-15.

[13] M. F. Mansour, "Efficient Huffman Decoding with Table Lookup", Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. vol.2, no., pp.II-53,II-56, 15-20 April 2007.

[14] T. H. Cormen, C. E. Leiserson, R. L Rivest, and C. Stein, Introduction to Algorithms, 3rd edition, MIT Press & McGraw-Hill, 2009.

[15] J. Loinig, J. Wolkerstorfer, and A. Szekely, "Packet filtering in gigabit networks using fpgas," in Austrochip 2007 Proc. 15th Austrian Workshop on Microelectronics, 2007, pp. 53-60.

[16] Altera Corporation. "Cyclone IV Device Handbook", Volume 1. May 2013. Available at: http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf

[17] K. Zhang, Embedded Memories for Nano-Scale VLSIs. Integrated Circuits and Systems Series, Springer, 2009.