ARQUITETURA MAR DE PROCESSADORES PARA DETECÇÃO DE INTRUSÃO EM REDES

Luís Carlos Mieres Caruso, Guilherme Montez Guindani, Hugo Webber Schimitt,
Ney Laert Vilar Calazans ,Fernando Gehm Moraes

Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS
Faculdade de Informática – FACIN
Av. Ipiranga, 6681 - Prédio 30 / Bloco 4 Telefone: +55 51 3320-3611 - Fax: +55 51 3320-3621
90619-900 - Porto Alegre - RS - BRASIL

lcaruso@inf.pucrs.br, calazans@inf.pucrs.br, moraes@inf.pucrs.br

ABSTRACT

The pressure for enhanced performance of systems stimulates the research of specific system architectures. Computer network security is an example subject where this pressure is pronounced. The use of conventional processor architectures for detecting network intrusion can seldom account for the needed throughput in applying all known attack detection rules. This is due to the extremely high throughput of current networks coupled to the sequential processing nature of these architectures. Reconfigurable systems technology offers a fast evolving field where new, specific architectures can be tested. One such architecture is the sea of processors approach. This work presents the development of specific architecture for network intrusion detection based on the replication of simple dedicated processors. The proposed processor architecture is dedicated to pattern detection, and performance can be increased by the massive use of parallelism. The proposed system design is particularly adequate to employ FPGAs with distributed memory blocks.

1. INTRODUÇÃO

A necessidade de assegurar o funcionamento de sistemas de computação tem exigido esforços crescentes no sentido de evitar o que se convencionou chamar de intrusão, ou seja, o uso não autorizado ou o mau uso de um sistema de computação. Isto levou à criação de sistemas de detecção de intrusão ou IDS (*Intrusion Detection System*). No contexto das redes de computadores foram criados os NIDS, IDSs especializados na vigilância sobre o tráfego de redes, como é o caso do programa Snort [CAS03][NOR93][NOR93b], aplicação de fonte aberto amplamente utilizado em todo mundo com esta finalidade.

O constante crescimento das redes e de sua utilização é acompanhado por um aumento das ameaças à segurança das redes, o que vem a exigir um volume cada vez maior de processamento em segurança de redes. Apenas o emprego de computadores mais poderosos para a execução dos NIDS em software não fez frente à velocidade com que se multiplicam as ameaças de segurança. O aumento de desempenho na área de segurança de redes tem sido, principalmente, conseqüência da evolução nas técnicas e algoritmos de NIDS [DES03]. Este foi o caso da evolução do *Snort*, onde o desempenho aumentou várias vezes até a versão atual, dada a aplicação de algoritmos mais eficientes.

A maioria dos sistemas de computação utilizados atualmente são sistemas mono-processados onde existe um único fluxo de processamento seqüencial, o que os limita em desempenho. A alternativa a estes sistemas são máquinas paralelas, entretanto estas máquinas são bastante mais caras devido à especialização necessária tanto da eletrônica, quanto do software empregado.

Por outro lado, a tecnologia de lógica reconfigurável, FPGAs, permite liberdade na concepção e teste de novas arquiteturas. Estes componentes são constituídos por um arranjo bidimensional que repete uma mesma célula básica de lógica configurável, o que representa um paralelismo nativo.

Henkel [HEN03] aponta a arquitetura "mar de processadores" como uma tendência para arquiteturas de alto desempenho. Nesta arquitetura vários processadores são integrados em um único componente, dividindo e especializando as tarefas confiadas ao sistema global.

O gargalo de desempenho do Snort é a comparação de conteúdos de pacotes de rede. Conforme é apresentado na Seção 4, diversos autores apontam que o componente do Snort sensível a incremento de desempenho, tanto em software como em hardware, é a máquina de detecção ou o comparador de padrões.

Este trabalho tem por objetivos: (*i*) implementar uma CPU para detecção de regras Snort; (ii) integrar estas CPUs na forma de um mar de processadores de forma a processar todos os pacotes oriundos da rede.

O presente documento é organizado como segue. A Seção 2 apresenta as principais características do SNORT. A Seção 3 apresenta definições relativas ao processo de comparação de padrões. A Seção 4 apresenta o estado da arte no campo de NIDS acelerados por hardware. As Seções 5 e 6 apresentar a arquitetura do sistema. Finalmente a Seção 7 apresenta as conclusões do presente trabalho e direções para trabalhos futuros.

2. SNORT

Snort é um NIDS capaz de realizar análise de tráfego em tempo real, seleção e registro de pacotes em redes IP. O Snort pode ser decomposto em quatro subsistemas [CAS03]: *sniffer*, preprocessador, máquina de detecção e o subsistema de saída. O Snort realiza detecção baseada em regras, sendo este um dos motivos que lhe proporcionam flexibilidade.

2.1 Operação do Snort

A priori, todo o conjunto de regras deveria ser verificado contra o tráfego de pacotes, mas este seria um método bastante ineficiente. Com o objetivo de tornar as verificações mais eficientes, as regras são classificadas segundo o tipo de tráfego detectado. Na inicialização do Snort é realizada uma tarefa de agrupamento das regras conforme o tipo específico de tráfego a que elas se destinam. Por exemplo, regras destinadas à análise de tráfego TCP, cuja porta destino seja a porta 21, constituirão um grupo de regras, enquanto regras TCP, cujo destino seja a porta 23, constituirá um outro grupo.

Durante a execução, conforme o tipo de tráfego detectado, o Snort aplica a análise segundo o grupo de regras específicas para aquele tráfego. Este procedimento simples diminui o número de regras a examinar tornando a execução mais eficiente. Regras genéricas podem ser estabelecidas. Estas regras pertencerão a um grupo especial onde nenhum tipo de especificidade com relação ao tráfego foi considerado. Este grupo de regras é aplicado a todos pacotes capturados. Estas regras, por impor-se a todos os pacotes, representam um alto custo de execução frente aquelas específicas [CAS03].

2.2 Formato das regras SNORT

Para ilustrar o formato básico das regras aceitas pelo Snort apresentamos a seguir uma regra que detecta um ataque FTP. As regras do Snort são escritas em formato texto em uma única linha, e constituem-se de duas sessões: o cabeçalho e as opções.

- O cabeçalho da regra

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21
```

- As opções da regra

```
(msg: "FTP EXPLOIT wu-ftpd 2.6.0 site exec format string overflow Linux"; flow:to_server, established;
content:"|31c031db31c9b046cd8031c031db|";
reference:bugtraq,1187; reference:cve,CAN-2000-0573; reference arachnids,287; classtype: attempted_admin; sid:344; rev4;)
```

A sintaxe usada no cabeçalho das regras compreende:

- **alert**. Formato utilizado para a saída. Os formatos de saída são: *alert*, *log*, *pass*, *dynamic* e *activate*.
- **TCP**. Protocolo em uso. Este campo pode aceitar os valores: TCP, UDP, IP e ICMP.
- **$EXTERNAL_NET**. Endereço IP de origem.
- **any**. Esta é a porta de origem selecionada como qualquer porta de origem.
- **->**. Esta seta indica a direção do fluxo de dados; neste caso, de $EXTERNAL_NET vindo de qualquer porta para $HOME_NET na porta 21.
- **$HOME_NET**. Endereço de destino.
- **21**. Esta é a porta destino, indicando potenciais ataques à porta 21. A porta 21 é a porta tipicamente utilizada para atividades FTP.

O principal componente nas opções da regra é **content**. O campo *content "|31 C0 31 BD 31 C9 B0 46 CD 80 31 C0 31 DB|"* é comparado com o pacote em análise usando um algoritmo de comparação de cadeias de caracteres.

2.3 Número de regras

O número de regras do Snort é ampliado dia a dia à medida que se verificam novos tipos de ataques. Hoje o Snort está em sua versão 2.11 e o conjunto de regras cobre 4 protocolos: IP, ICMP, TCP e UDP. A Tabela 1 ilustra o número de regras por protocolo para a versão atual do Snort.

Tabela 1 - Número de regras por protocolo - SNORT versão 2.11.

Protocolo	Número de Regras
TCP	1784
UDP	165
IP	44
ICMP	131
TOTAL	2124

3. DEFINIÇÕES RELATIVAS AO PROCESSO DE COMPARAÇÃO

Um NIDS cumpre sua função examinando os pacotes que trafegam na rede onde está instalado, buscando por características indicadoras de invasão, NIDS para redes IP deverão, portanto, analisar características de pacotes IP.

O problema de comparação de conteúdos do *payload* intrínseco a um NIDS é um problema de "*exact set matching*" [GUS97] com algumas peculiaridades inerentes à aplicação NIDS. Assim como no problema de "*exact matching*", a solução consiste em encontrar todas as ocorrências de um padrão P, de comprimento n, em um "texto" T, de comprimento m, onde $m > n$, no problema de "*exact set matching*" a solução consiste em encontrar em T todas as o ocorrências de qualquer um dos padrões P_i em um conjunto de padrões $\boldsymbol{P}$.

O processo de comparação de padrões utilizado por um NIDS caracteriza-se pela necessidade de comparar vários padrões com a carga útil do pacote, caractere a caractere. Para tanto, para cada caractere do *payload* do pacote deve-se realizar tantas comparações entre *payload* e padrões quantos sejam os diferentes padrões, e deve-se manter, para cada padrão comparado, um *estado atual da comparação*. Estas comparações individuais de um padrão contra o conteúdo do pacote referiremos como *comparações elementares*.

O estado da comparação reporta, para aquele padrão, se existe ou não uma coincidência dos bytes do padrão com os bytes do *payload* e em que extensão esta coincidência encontra-se, por exemplo, se a coincidência se deu apenas no último byte examinado ou se ela vêm mantendo-se pelos últimos "n" bytes.

O *estado da comparação*, para um dado padrão, pode ser assumido equivalentemente como sendo o próximo byte do padrão a ser comparado ao *payload*, ou o ponto de comparação. Enquanto não se estabelece uma seqüência de bytes coincidentes, o ponto de comparação mantém-se sobre o primeiro byte do padrão. Quando ocorre a primeira coincidência, então o ponto de comparação passa para o segundo byte do padrão e este evento vamos nos referir como *disparo de comparação*. Se as coincidências se mantiverem, o ponto de comparação se deslocará sempre para o próximo byte do padrão a ser comparado, se neste processo for atingido o

último byte do padrão, então deve ser sinalizada a *incidência do padrão* sobre o *payload*. Se, ao contrário, a seqüência de coincidências for interrompida, o ponto de comparação volta ao início do padrão e a este evento referiremos como *desarme de comparação*.

A principal restrição de um comparador para um NIDS é que todas as comparações requeridas por um pacote sejam efetuadas, ou seja, devem haver recursos de hardware e ou software para executar todas as comparações elementares exigidas pela maior das classes, no tempo médio em que cada um dos bytes de T ficam disponíveis para comparação. Este tempo será referido como *tempo de byte* e seu valor vai depender do tráfego da rede para qual o NIDS se destina. Exemplificando este processo, assumir:

- Rede de 100 Mbps, com tempo de byte igual a 200ns (taxa de bytes efetiva 5 MBps);
- Freqüência de operação da arquitetura: 200 MHz (T = 5ns);
- Número máximo de regras de uma determinada classe em uma CPU: 20;
- Número de CPUs em paralelo: 20;
- Tempo médio para tratamento de um byte do pacote: 2 ciclos de *clock* (10 ns);

Com esta topologia pode-se tratar simultaneamente 400 regras, pois cada uma das 20 CPUs tratará, em paralelo, 20 regras seqüencialmente. Sendo o tempo de 10 ns para tratar cada byte do payload, o tempo de byte para esta arquitetura é de 200 ns, atendo a taxa da rede. Para redes com taxas maiores, pode-se aumentar o número de CPUs, aumentar a freqüência de clock ou reduzir o número de regras por CPU.

No caso de um comparador em software, este tempo deve ser utilizado para realizar serialmente todas as comparações elementares, mesmo com o uso de esquemas de classificação refinados e com a aplicação de algoritmos baseados em comparações contra padrões em árvore, 200 ns não são suficientes para tratar serialmente centenas de padrões. E no caso de redes de 1 Gbps os tempos serão 10 vezes menores.

4. ESTADO DA ARTE

Esta Seção apresenta a contribuição recente de vários autores na área de aceleração por hardware de comparação de cadeias de caracteres.

4.1 Cho, Navab e Mangione-Smith

Este trabalho [CHO02] busca acelerar a tarefa de detecção de padrões necessária à execução do Snort por meio de um comparador implementado em FPGA. Segundo os autores, é possível processar pacotes com um tráfego de até 2,88 Gbps.

A máquina de detecção é composta por regras implementadas em hardware (*regras-hw*) dispostas em paralelo. Todas as regras-hw recebem a mesma cópia da palavra de 32 bits recebidas pelo FPGA. A Figura 1 ilustra este comparador. Internamente, cada regra-hw possui duas unidades distintas, tal como as próprias regras do Snort, com suas seções de dados de cabeçalho e de *payload*. A unidade de cabeçalho é um bloco projetado para comparar os campos do cabeçalho do pacote. Se a unidade de cabeçalho identificar completamente os dados de cabeçalho a que referencia, então será gerado um sinal de habilitação para que a unidade de payload, responsável por comparar o padrão buscado no conteúdo dos pacotes, seja ativada.

Esta arquitetura implica numa configuração "*hardwired*" do conjunto de regras, que para ser ampliado necessita que toda uma tradução das regras em lógica seja refeita e todo o hardware seja sintetizado e recarregado em uma parada do sistema.Os autores criaram um método para traduzir o arquivo de regras do Snort em VHDL segundo *templates* que permitem a manutenção da estrutura em *pipeline* requerida para a eficiência do projeto.

Os autores utilizaram um FPGA Altera da família EP20K para implementar o sistema. Sem o uso de nenhuma restrição de temporização obtiveram uma velocidade de 90 MHz, o que lhes permitiu-lhes filtrar uma taxa de dados de 2,88 Gbps a despeito do tamanho dos padrões das regras ou dos pacotes.A versão do Snort utilizada pelos autores, e segundo estes, propunha um conjunto de 105 regras, que à época deveria filtrar 95% dos ataques.

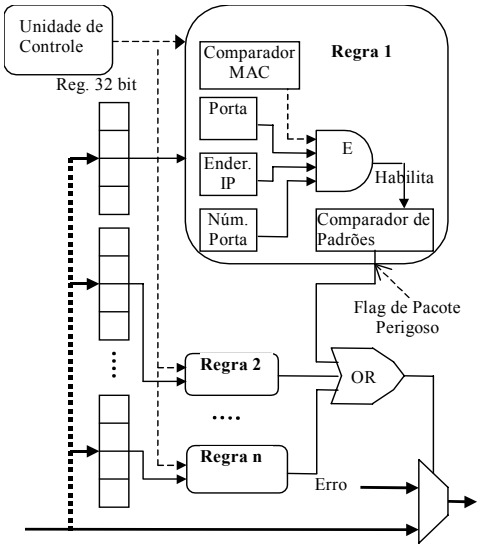

Figura 1 - Arquitetura de Cho, Navab e Mangione-Smith.

4.2 Sourdis e Pneumatikatos

O trabalho de Sourdis e Pnevmatikatos [SOU03] estabelece como meta o suporte para comparação de padrões em uma rede de 10 Gbps. Os autores dividem seu sistema em três partes:

- Uma árvore de registradores, que limita o fan-out dos dados proveniente da rede para as diversas unidades de comparação, evitando degradação elétrica ou atrasos;
- Lógica de comparação, semelhante à de [CHO02];
- Um circuito de codificação que, no caso do padrão de uma regra ser capturado, sinalize com o número da regra.

Nesta implementação também foi implementada apenas a comparação de padrões. O FPGA torna-se um subsistema conectado a uma CPU hospedeira onde o Snort está sendo executado.

Os autores salientam a importância do uso intensivo de *pipelines* e em seu trabalho descrevem a adaptação de

sua proposta à estrutura interna dos FPGAs Xilinx. Como diferencial de sua implementação os autores citam um estágio de codificação de alarme que permite repassar ao estágio posterior (pós-detecção), em geral software rodando na CPU hospedeira, qual regra foi ativada.

A atualização do conjunto de regras neste segundo caso segue igualmente o que já havia sido feito pelo grupo de [CHO02]. Os autores criaram programas para poderem extrair, dos arquivos de regras do Snort, os padrões de regras e aplicá-los em arquivos VHDL, automatizando o processo de atualização do conjunto de regras.

Foram utilizados três conjuntos de regras para as medições. Um conjunto pequeno, com 10 regras cada uma com 10 caracteres; um conjunto médio com 47 regras e em média 10,4 caracteres por regra e um conjunto grande com 210 regras e 11,7 caracteres por regra

Foram sintetizados todos estes conjuntos de regras para vários componentes Xilinx: Virtex 1000-6, VirtexE 1000-8, Virtex2 1000-5, Virtex 2600-8 e Virtex2 6000-5. Para o conjunto pequeno de regras os autores obtiveram 6 Gbps nos dispositivos mais simples e 12 Gbps nos mais avançados. No conjunto médio de regras foi que os autores obtiveram a taxa de 11 Gbps.

4.3 Carver, Franklin e Huntchings

O trabalho de [CAR02] visa realizar comparações de padrões em todos os pacotes em trânsito e à velocidade da rede. O Snort em execução no PC foi modificado para passar os dados dos pacotes para detecção diretamente ao hardware prototipado através da interface PCI. A máquina de comparação de padrões foi adaptada para um fluxo de dados em bytes ao invés de palavras de 32 bits como optaram os outros autores.

Em seu trabalho são utilizadas expressões regulares para descrever os padrões das regras do Snort e com isto proporcionar um método de converter regras em hardware, segundo a estratégia que os autores buscam em [SID01]. Pelo uso de expressões regulares fica facilitada a abordagem que busca poupar área em silício na implementação de regras que possuam padrões de caracteres semelhantes.

Este grupo utilizou o ambiente JHDL , um sistema que utiliza Java para produzir VHDL, que não apenas permite a criação de descrições HDL, como permite também a criação de programas em Java que proporcionem a automação desejada desde a captura das regras do Snort até a produção do netlist do circuito final.

Para casos de teste utilizando cadeias pequenas o desempenho de hardware e software foi aproximadamente o mesmo, no entanto, quando as cadeias crescem, o hardware chega a se tornar 600 vezes mais rápido, no melhor caso medido.

4.4 Coit, Stuart e McAlerney

O trabalho deste grupo de autores é anterior aos apresentados [COI01] e trata igualmente com o comparador de padrões do Snort, mas propõe aceleração exclusivamente em software.

Neste trabalho os autores exploram o fato de que existem muitos padrões de conteúdo semelhante entre as regras do Snort e que nenhum mecanismo no software impede de que parte de um padrão que já tenha sido rejeitado volte a ser novamente comparado. O artigo cita as heurísticas do algoritmo de Boyer-Moore, que é utilizado pelo Snort para acelerar as comparações e propõe variações deste algoritmo utilizando árvores de palavras de Aho-Corassik para obter ganhos sobre a repetição de padrões.

Como resultado deste trabalho, os autores apresentam um ganho em tempo, pelo uso do algoritmo proposto, onde as comparações se mostram de 1,02 a 3,32 mais rápidas que no Snort utilizando Boyer-Moore, sob as mesmas condições de execução. Como reflexo da importância dos resultados obtidos, na versão 2.0 do Snort, ocorreu a assimilação da sugestão do novo algoritmo de busca proposto pelos autores [DES03].

4.5 Baker e Prasanna

Estes autores [BAK04] propõe uma ferramenta de geração automática de arquiteturas destinadas a detecção de intrusão. A principal estratégia de comparadores utilizados neste trabalho é reduzir os bytes dos padrões e do *payload* a representações unárias, um sinal para cada byte. Os autores verificam que apenas 100 octetos distintos são encontrados entre os padrões do Snort, além disso, pelo uso de outros artifícios como a insensibilidade à caixa do texto, chegam a apenas 75 octetos, representados por 75 fios.

Os autores fizeram medições de área e freqüência, comparando valores entre exemplos de suas arquiteturas com e sem otimizações em árvore. Estes testes foram repetidos para cada arquitetura utilizando quatro subconjuntos, com 204, 361, 602 e 1000 padrões. Para cada subconjunto foi encontrado um número de partições ótimo, ficando entre 2 e 3 para o menor caso e por volta de 8 no caso maior. O particionamento em geral permitiu um ganho em freqüência máxima de operação, a custa, no entanto, de área em CI. A alternativa em árvore permite os ganhos em velocidade a custos menores em área.

A abordagem em árvore favoreceu como esperado a densidade dos circuitos, sendo que no subconjunto com 602 regras foi obtido um ganho de 50% em área.

4.6. Attig e Lockwood

Este trabalho [ATT04] fundamenta-se na aplicação de filtros Bloom, que são geradores de funções hash. A idéia de aplicação de funções hash para realizar a comparação de padrões é uma iniciativa inovadora destes autores na área de NIDS.

Por observarem que o conjunto de regras do Snort raramente apresenta padrões com extensão superior a 26 caracteres, a arquitetura proposta se atém a pesquisa destes padrões por meio de máquinas Bloom. Cada máquina Bloom busca por padrões de determinada extensão, entre 2 e 26 bytes. As 25 máquinas resultantes são instanciadas em paralelo. Os padrões encaminhados a cada uma das máquinas são submetidos a um filtro Bloom para verificar a incidência de padrões previamente programados.

A incidência é verificada pelo cômputo de k funções hash sobre a amostra analisada. Um vetor de bits é

preenchido com o resultado destas avaliações, um bit "1" para cada função hash detectada ou um "0" caso contrário. Ao final deste estágio, os vetores com todos os bits em "1", indicativos de um match de padrão, são encaminhados à um árbitro onde esperam por um acesso a uma SDRAM externa ao FPGA.

Operando a 62,8 MHz, processando um byte por ciclo de relógio, foi reportado um throughput de 502 Mbps, em um CI VirtexE 2000. Um sistema com múltiplas instâncias, que processe 32 bits por ciclo de relógio, poderia então chegar à taxas de 2 Gbps.

Outra característica da implementação é permitir a programação dos padrões dinamicamente. Um novo padrão é programado em 30 ciclos enquanto a carga de 35.000 padrões leva apenas 17 ms.

4.7 Crítica ao Estado da Arte

Do exame das diversas abordagens apresentadas, verifica-se que são aplicadas limitações ou simplificações ao problema da detecção de intrusão em rede. Estas limitações podem ser:

- Limitação no conjunto de regras - nos trabalhos apresentados como os de [CHO02] e [SOU03] o conjunto de regras do Snort utilizado fica muito aquém das 2124 regras hoje existentes.
- Limitação no tamanho dos padrões - o tamanho dos padrões não é livre, fica limitado em torno dos 20 caracteres por regra.
- Limitação no conjunto de caracteres dos padrões..

Outra restrição observada na maioria das propostas é a implementação "*hardwired*", ou seja, os padrões de comparação são fixos.

Diante da análise destas restrições que conjunto de características seriam desejáveis em uma arquitetura aplicável a um coprocessador para um NIDS prático?

- Aceitar o número máximo de regras suportadas pelo Snort.
- Aceitar padrões de comparação no tamanho requerido pelas regras do Snort.
- Aceitar padrões de comparação sem restrições de alfabeto.

Este trabalho apresenta um modelo de arquitetura paralela que permite a investigação do tráfego da rede no tempo em que ele ocorre. O desempenho da arquitetura não depende da extensão dos padrões comparados, mas apenas de seu número. Com relação ao número de padrões esta topologia é parametrizável, possibilitando o suporte para um número crescente de padrões a comparar.

5. PROCESSADOR pico-CPU

Esta Seção apresenta o projeto da unidade básica de comparação. Esta unidade, denominada *pico-CPU*, tem sua arquitetura desenvolvida segunda as características dos dispositivos programáveis com memória embarcada..

Os comparadores que se deseja empregar devem ter seus padrões de comparação programáveis. Com esse intuito, buscou-se utilizar os blocos de memória presentes em FPGAs comerciais. As memórias de dupla porta presentes nos dispositivos Virtex, denominadas Block SelectRAMs, referidas neste contexto como DPRAMs, possuem 18 K bits e reduzido tempo de acesso (~2,0 ns). Além da presença de DPRAMs, os dispositivos VirtexII Pro possuem CPUs PowerPC embarcadas, as quais poderão ser utilizadas futuramente como CPU mestre, responsável pelo controle dos processadores *pico-CPU* .

A *estratégia de classes* decide os critérios de classificação e por conseqüência o número das classes. A Figura 2 mostra um exemplo simplificado de estratégia de classes.

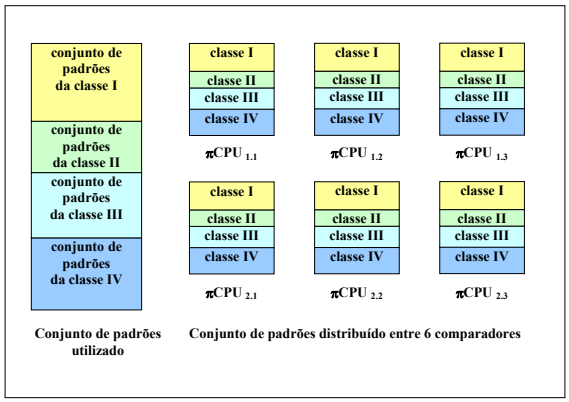

Figura 2 - Estratégia de classes.

O exemplo da Figura 2 define 4 classes de padrões de comparação. A coluna mais à esquerda da Figura representa o conjunto total dos padrões de comparação escolhidos, separados conforme o critério de classificação adotado e as resultantes classes de I a IV. As demais colunas representam os subconjuntos de padrões de cada uma das classes atribuídos a cada *pico-CPU*. É importante distribuir uniformemente os padrões de uma mesma classe na diversas *pico-CPUs*, uma vez que o tempo de processamento será determinado pela *pico-CPU* que contiver mais padrões a serem comparados.

A Figura 3 apresenta o algoritmo de comparação em uma *pico-CPU*, onde o segmento de texto "AROMAT", contido na carga útil de um pacote, é comparado com três padrões: "ROMA", "ROMANO" e "REI". Apenas o padrão "ROMA" está contido no texto "AROMAT" e, desta forma, apenas este padrão é detectado. Cada uma das fases de comparação (de "a" até "f") ocorre em um *tempo de byte*, e compreende no máximo três comparações, pois há três padrões.

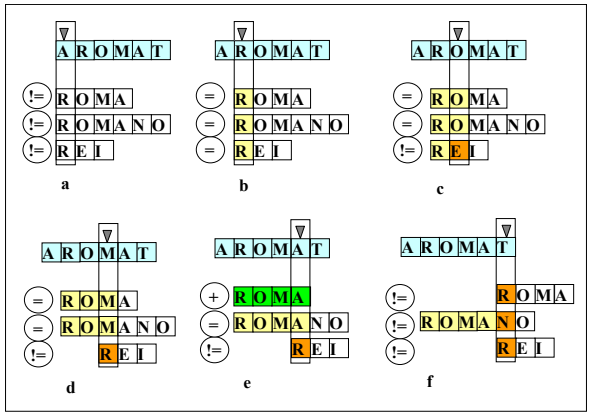

Figura 3 – O processo de comparação

(a) o caractere "A" do texto é comparado ao primeiro caractere de cada padrão, neste caso todas as

comparações resultam em descasamentos. O *ponto de comparação* de cada padrão está portanto na posição de seu primeiro caractere.

(b) passa-se ao próximo caractere do texto que é testado contra os *pontos de comparação* de cada padrão. Neste *tempo de byte*, entretanto, todas as comparações resultam em casamentos, em função disto cada uma das *posições de comparação* é avançada, indicando desta maneira o estado de comparação. Cada um destes primeiros casamentos ocorrido em cada padrão é um *disparo de comparação*.

(c) neste ponto, dois dos padrões produzem casamentos e seus pontos de comparação avançam mais uma posição. O padrão "REI" produz um descasamento e sua posição de comparação retorna ao seu primeiro caractere. A esta perda de pareamento com o texto é o que se chamou de *desarme de comparação*.

(d) neste tempo de byte, os mesmos padrões disparados produzem novos casamentos, agora para os caracteres "M". O terceiro padrão produz novo descasamento e permanece não disparado.

(e) repete-se a situação dos quadros "c" e "d", entretanto o padrão "ROMA" chega a sua última posição o que vai configurar uma incidência deste padrão no texto ou pacote em análise. Uma sinalização deve ser gerada indicando este evento.

(f) o padrão recém encontrado retorna ao seu ponto de comparação inicial, e verifica-se que todas as comparações resultam em descasamentos. Esta situação causará o desarme de comparação no padrão "ROMANO", de modo que no tempo de byte seguinte, não mostrado na figura 5, todos os pontos de comparação estejam nas posições iniciais de seus padrões.

Neste exemplo anterior tem-se uma idéia de como as diversas comparações avançam sobre os padrões à semelhança de uma "frente de onda" deslocando-se à superfície d'água. Em cada estágio de comparação tem-se um estado definido pelas posições em comparação em cada padrão e no próprio texto. A passagem de um estado para outro é determinada pelo próximo caractere do texto a comparar.

O estado de comparação obtido pela manutenção em memória do ponto de comparação de cada padrão é essencial para o funcionamento deste algoritmo, e ele deve ser alterado a cada novo ciclo de comparações, ou seja, a cada tempo de byte.

5.1 Organização da Memória

As DPRAMs armazenam subconjuntos do total dos padrões de comparação, organizados em classes. Os padrões serão armazenados seqüencialmente e serão acessados byte a byte. As DPRAMs são organizadas em 3 regiões, conforme ilustrado na Figura 4.

- O *bloco de vetores* armazena os endereços correspondentes ao primeiro registro de cada classe. Desta forma, cada classe pode ter um número arbitrário de registros.
- O *bloco de registros* contém as informações de acesso e controle de cada padrão. Cada registro contém 4 campos: endereço inicial do padrão, número de bytes do padrão, identificação do padrão e estado da comparação. O estado da comparação (S cx py) é um contador, que é incrementado a cada comparação positiva entre um byte do payload e o padrão em comparação.
- O *bloco de padrões* armazena os padrões. Este bloco contém diversos sub-blocos, um para cada padrão armazenado.

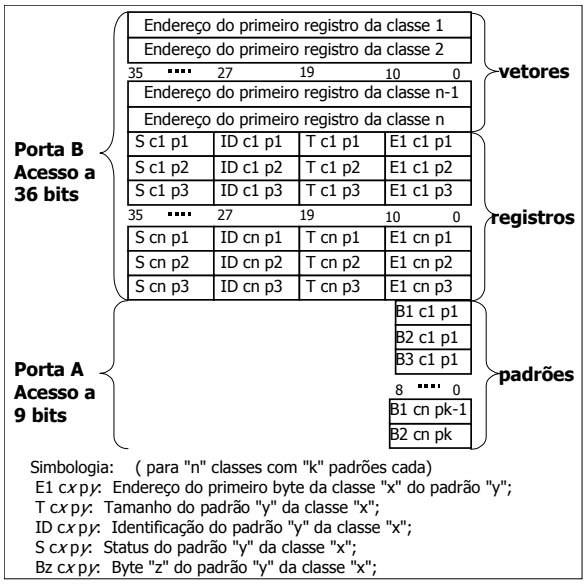

Figura 4 - Organização interna das DPRAMs.

A divisão de classes adotadas coincide com a divisão de protocolos reconhecidos pelo Snort quais sejam: TCP, UDP, ICMP e IP.

5.2 Interface externa da pico-CPU

A Figura 5 apresenta a interface externa da pico-CPU. Os sinais são agrupados por função.

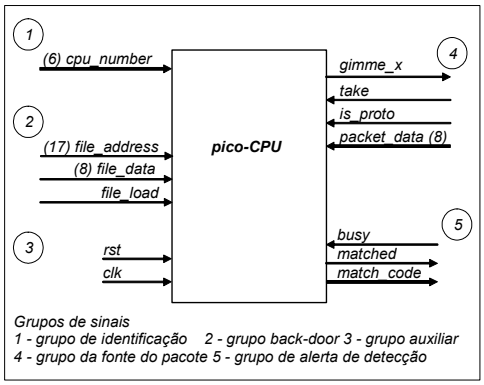

Figura 5 - Pico CPU.

- *Grupo 1*. Endereça uma dada CPU.
- *Grupo 2*. Permite o carregamento dos dados na DPRAM.
- *Grupo 3*. Sinais globais de reset e relógio.
- *Grupo 4*. Interface com a fonte dos bytes de payload e também de comandos, como é o caso do código de classe. O sinal **`ready`** indica que o último byte do payload oferecido à CPU para comparação foi examinado, e que o dispositivo está pronto à espera do próximo byte. O sinal **`take`** indica que existe uma informação válida nos fios **`packet_data`** e **`is_proto`**. Se "**`is_proto`**" for 1 trata-se de um

código de classe e do início de um novo pacote, se for 0 o byte em `packet_data` é payload.

- *Grupo 5*. Reúne os sinais de alerta. Quando há uma comparação positiva o sinal `matched` é ativado, e o número da regra é colocado em `match_code`. O sinal `busy` é gerado pela CPU-mestre indicando se o último alerta já foi tratado ou não. Enquanto o sinal `busy` estiver ativo, a pico-CPU fica em estado de espera.

5.3 Bloco de Controle

A Figura 6 ilustra a máquina de estados responsável pelo controle das pico-CPUs. O estado `Sask` é o ponto inicial da operação de comparação de cada byte do pacote sob análise. A partir deste estado diversos laços se encadeiam, detalhados nos parágrafos seguintes.

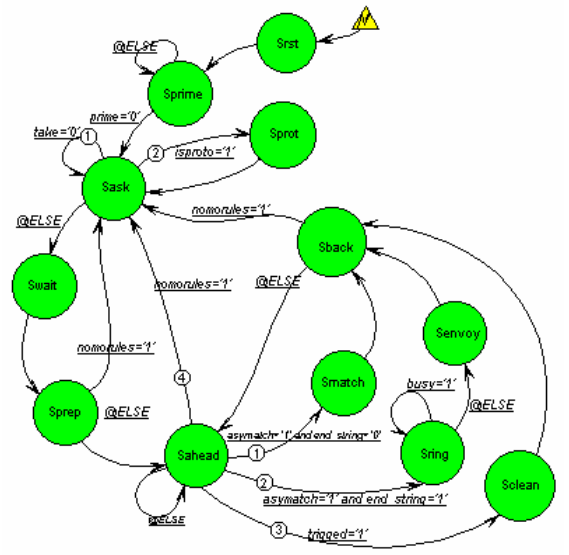

Figura 6 - Máquina de estados de controle das pico-CPUs.

A seqüência de estados ***Sask-Sprot***. é responsável pela sincronização das pico-CPUs no início de cada novo pacote e por capturar a informação de classe do pacote enviada pela CPU-mestre.

A seqüência de estados ***Sask-Swait-Sprep-Sahead***. é utilizada para preencher o *pipeline* de acesso à DPRAM.

O estado ***Sahead*** são feitas as diversas comparações com cada um dos padrões da classe. Permanece-se no estado ***Sahead*** se nenhuma das comparações com os padrões resultar positiva. Isto significa que cada byte do pacote pode ser processado em um único ciclo de clock, sempre que não houver *match* (comparação positiva).

A seqüência de estados ***Shead-Smatch-Sback-Shead*** ocorre quando há *match*. Neste caso, o ponto de comparação do padrão responsável pelo *match* deve ser atualizado, incrementado-se o contador de estado (8 bits mais significativos do registro – porta B).

A seqüência de estados ***Shead-Sclean-Shead*** quando não ocorre *match* e o padrão está em comparação (contador de estado diferente de zero). Este é ocaso de um desarme de comparação, quando o ponto de comparação retorna à posição inicial do padrão. Tal como no caso anterior o registro deste padrão deve ser atualizado, recebendo o valor zero.

A seqüência de estados ***Shead-Sring-Senvoy-Sback-Sahead*** ocorre quando há *match* e atinge-se o final do padrão (campo do contador de estado igual ao campo de tamanho do padrão). Neste caso a CPU-mestre é notificada. O estado ***Sring*** é onde a FSM espera pelo sinal de retorno da CPU-mestre, para então enviar o campo ID do registro correspondente ao padrão (estado ***Senvoy***).

5.4 Bloco de Dados

A Figura 7 ilustra a estrutura do bloco de dados, implementado ao redor da DPRAM. Observar que o sinal mais importante é a saída da porta B, DOB, que pode representar tanto endereço de vetor ou registro.

No estado `Sprot` a saída da porta B é armazenada no sinal `first_rule`. A partir dos estados seguintes o sinal rule é incrementado, permitindo o acesso a um novo padrão. O endereçamento da porta B pode receber diferentes valores, em função da máquina de controle.

O endereço da porta A é obtido somando-se dois campos da saída de dados da porta B: endereço inicial (E1) e estado de controle (stauts). Quando há igualdade entre o byte de entrada (packet) e o padrão atual (DOA), o sinal match é ativado.

O conteúdo de um determinado registro na porta B é atualizado sempre que há um comparação positiva, sinalizada pelo sinal `match`. A entrada `DINB` recebe o próprio conteúdo, havendo alteração apenas no campo relativo ao status de comparação.

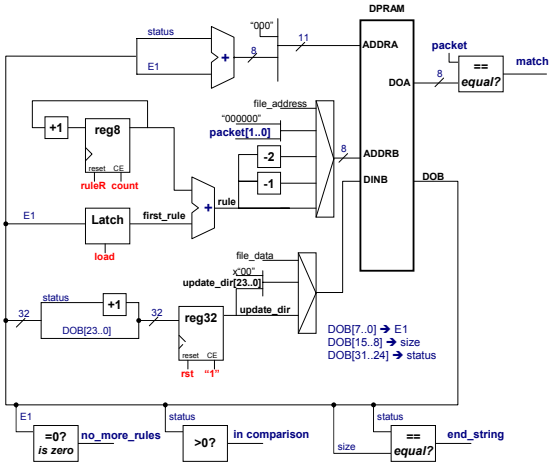

Figura 7 - Bloco de Dados da pico-CPU.

O sinal `no_more_rules` é ativado quando o endereço inicial é igual a zero. Os registros para uma determinada classe são armazenados em seqüência, e um registro adicional, com conteúdo igual à zero, é adicionado ao final destes. Na máquina de controle, este sinal é responsável pela volta ao estado `Sask`. A máquina de controle ao alcançar este estado habilita a pico-CPU a receber um novo pacote.

O sinal `in_comparison` indica um padrão em comparação, sendo utilizado na máquina de controle no estado Sclean.

O sinal `end_string` indica a incidência do padrão no pacote em análise.

5.5 Validação funcional

A Figura 8 ilustra o comportamento da pico-CPU para alguns estados da máquina de controle.

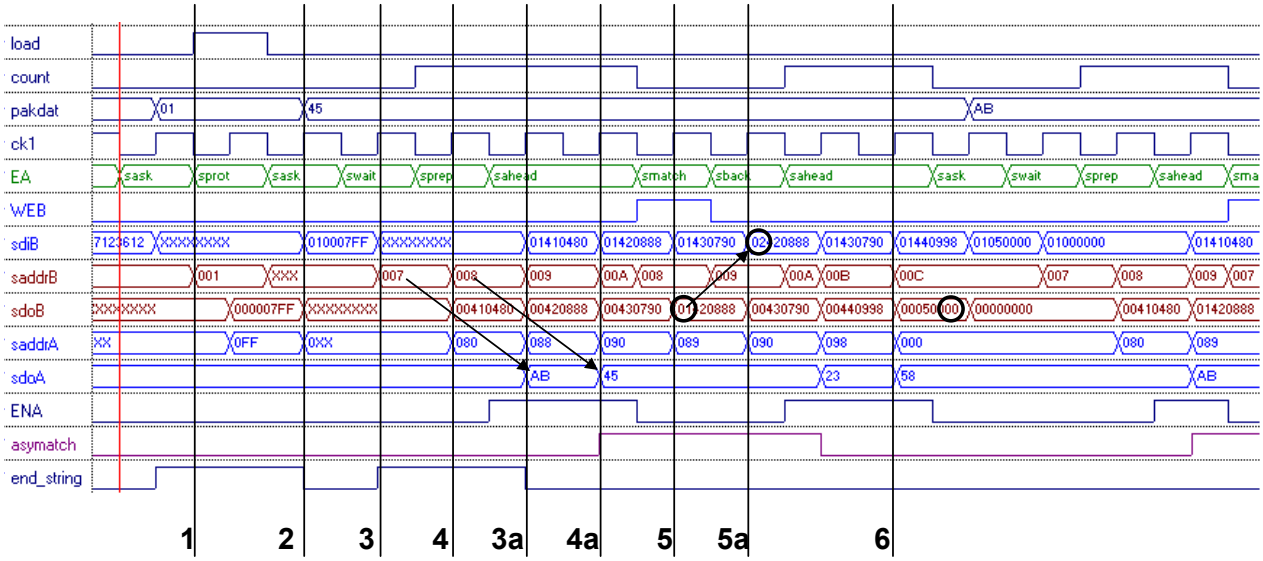

Figura 8 - Simulação da pico-CPU.

- Evento 1. Recebimento da classe à qual pertence o pacote que será recebido. Estado ***Sprot***.
- Evento 2. Recepção do primeiro byte do pacote.
- Eventos 3 e 3a. Leitura do primeiro registro da classe do pacote, obtendo-se o dado lido em sdoB 1 ciclo de clock após. Este dado inicia a leitura na porta A, obtendo-se o primeiro byte do padrão (AB), também um ciclo após, totalizando uma latência de dois ciclos de clock..
- Eventos 4 e 4a. Acesso ao segundo byte do padrão e ocorrência de *match*. Nesta situação a máquina de controle vai para o estado ***Smatch***.
- Evento 5 e 5a. Recupera o conteúdo do registro que ocasionou *match* (endereço 8), alterando o seu conteúdo quanto ao byte mais significativo.
- Evento 6. Continuação das comparações até que sdoB seja igual a zero, no campo de endereço inicial de padrão.

6. AGREGADO DE pico-CPUs

Apresenta-se nesta Seção uma arquitetura paralela composta por um agregado de pico-CPUs, caracterizando uma arquitetura tipo mar-de-processadores.

O sistema implementado inclui uma CPU-mestre, responsável por receber do meio externo o fluxo dos pacotes e repassá-lo aos comparadores, por identificar a classe de comparação do pacote e informar os comparadores, e por receber dos comparadores os alertas de detecção de padrões.

6.1 Interface externa do agregado

Várias pico-CPUs devem ser reunidas para que se obtenha o grau de paralelismo requerido para tratar do conjunto completo de regras do Snort. O agregado que se propõe é parametrizável em número de CPUs. A Figura 9 ilustra a interface externa do agregado de CPUs.

Os sinais são agrupados por função:

- *Grupo 1*. Permite o carregamento dos dados nas DPRAMs de cada pico-CPU.
- *Grupo 2*. Sinais globais de reset e relógio.
- *Grupo 3*. Interface com a fonte dos bytes de payload e também de comandos, como é o caso do código de classe. corresponde ao grupo da fonte de dados. O sinal "gimme" é obtido pela operação "E" sobre os sinais "gimme_x(n)" individuais das CPUs. Os outros sinais do grupo correspondem ao ponto de entrada dos sinais de mesmo nome das pico-CPUs.
- *O grupo 4* comunica-se com a CPU-mestre solicitando interrupção para o tratamento de um padrão detectado. O sinal "detected" indica o código binário da CPU onde se deu a detecção além do identificador deste padrão.

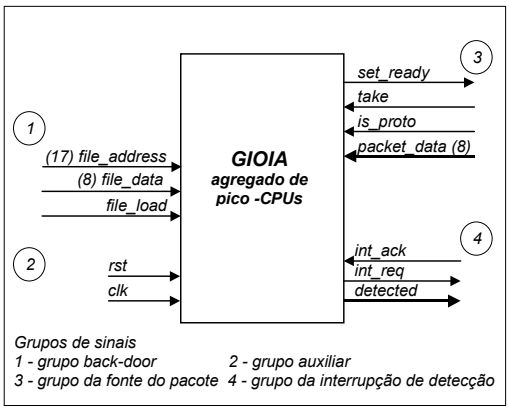

Figura 9 - Interface externa do agregado de CPUs.

6.2 Organização interna do agregado

O agregado de é constituído por módulos replicados, contando cada um: uma pico-CPU, um registrador com saída tristate, um *flip-flop* e uma porta "E".

A Figura 10 ilustra a organização interna do agregado, onde se vêem dois módulos. À direita da Figura, vêem-se os sinais dos grupos 4 e 3 conforme citados na interface externa.

Um grupo de sinais interno de controle conecta os módulos a uma máquina de estados. A função da máquina de estados (*FSM Control*) é enviar os códigos de incidência de padrões vindos de cada uma das pico-CPUs para a CPU-mestre, por meio da interrupção desta.

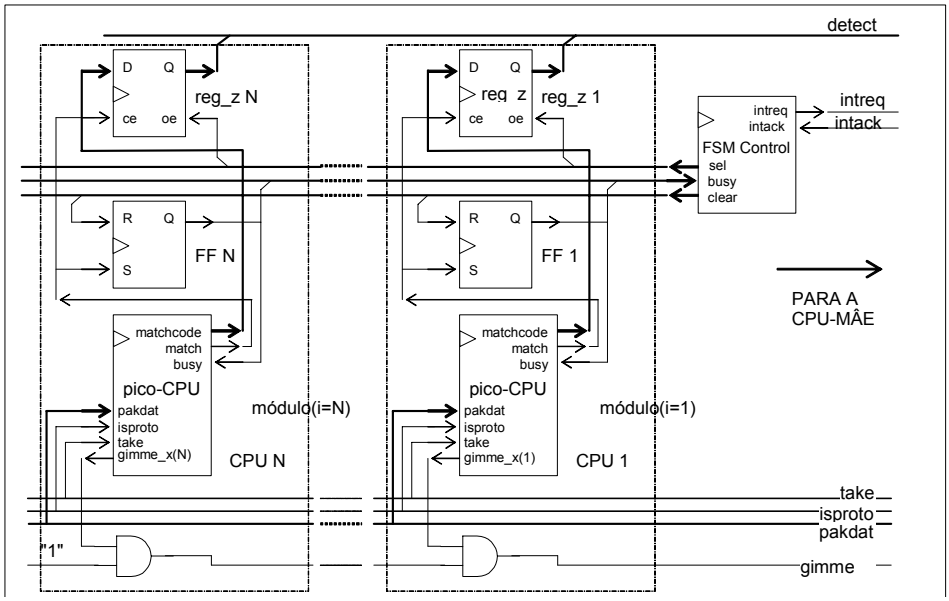

Figura 10 - Organização interna do agregado de pico-CPUs.

Esta máquina de estados percorre cada um dos *flip-flops*, os quais estão conectados como uma cadeia *daisy-chain*. Inicialmente todos os FFs têm saídas em zero, permitindo que a máquina de estados faça a varredura dos mesmos, até que um padrão seja detectado. O protocolo utilizado é descrito abaixo:

- Toda vez que uma pico-CPU encontrar uma incidência de padrão (*match*), esta entrará no estado `Sring`.
- Quando a cadeia daisy-chain detectar um match, sinalizado por um dado flip-flop em '1', a FSM gera uma solicitação de interrupção à CPU-mestre. Neste mesmo ciclo gera-se o sinal sel(i), sendo "i" o número da pico-CPU atendida, habilitando o tri-state reg_z(i). Esta habilitação coloca no barramento "detected" o número da regra que foi detectada (matchcode).
- Enquanto aguarda a resposta à interrupção, a FSM de controle permanecerá bloqueada. Notar entretanto que todas as pico-CPUs continuam a operar em paralelo. O único ponto de bloqueio que pode ocorrer é quando a mesma CPU que gerou a interrupção detectar um novo padrão.

A CPU ao atender a interrupção gera dois eventos. O primeiro é liberar a varredura da cadeia daisy-chain. O seguindo é gerar o sinal clear(i), liberando o FF, levando o sinal busy(i) a zero

Na porção inferior da Figura estão os sinais relacionados com a fonte dos bytes do pacote. Os sinais `pakdat`, `take` e `isproto` são entradas no agregado, copiadas para cada pico-CPU. Arranjos em árvore para redução de *fanout* é um trabalho futuro que deve ser considerado para evitar problemas de tempo de propagação.

O sinal `set-ready` é a operação "E" entre todos os sinais `ready` das pico-CPUs. Este procedimento força uma sincronização entre todas as CPUs antes de começar a transmissão de um novo pacote.

É previsto, para a posição de entrada destes sinais, uma estrutura FIFO, estabelecendo uma interface elástica entre a fonte dos dados e o agregado. Esta terá a finalidade de absorver picos de aporte de dados da rede.

6.3 Validação funcional do cluster de pico-CPUs

Para demonstrar a operação conjunta das pico-CPUs no *cluster*, foi criado um exemplo com quatro pico-CPUs.

Numa fase inicial as pico-CPUs devem ser configuradas, ou seja, devem receber seu conteúdo, constando do micro programa e dos padrões de comparação. A execução se inicia tão logo todas as pico-CPUs estejam configuradas. Na Figura 11 apresenta-se a simulação de um cluster com quatro CPUs.

Bem acima, na Figura, aparece o sinal de relógio dando idéia das relações de tempo entre os diversos sinais.

O sinal `gimme` é a função "E" dos sinais `gimme_x(n)`, esta operação entre estes sinais já havia sido referida na Figura 10.

Os sinais `gimme_x(n)` são solicitações de cada pico_CPU pelo próximo byte do fluxo de dados da rede. Estes sinais são ativados tão logo a pico-CPU tenha comparado cada uma das regras do protocolo atual contra o último byte do pacote fornecido.

Os sinais `take` e `isproto` são as respostas da CPU-mestre para o cluster em função do sinal `gimme`. A liberação do sinal `take` pela CPU-mestre leva o estado atual (EA) de todas as pico-CPUs do estado `sask` para o estado `swait`.

O fato do sinal `gimme` ser um "E" das requisições individuais das pico-CPUs faz com que as mesmas sincronizem-se na recepção do sinal `take`, onde todas mudam de estado conjuntamente para o estado `swait`. Esta situação está representada três vezes na Figura 11, na última vez o cursor vertical na descida do relógio em 9450 ns salienta esta ressincronização.

Em cada uma das situações o sinal `take` acompanha a chegada de um novo byte do fluxo de dados dos pacotes. Na figura, este fluxo de dados é representado `pakdat` (8 bits), onde se observa a recepção dos valores hexadecimais: "`0xEC`", "`0x83`" e "`0xAB`".

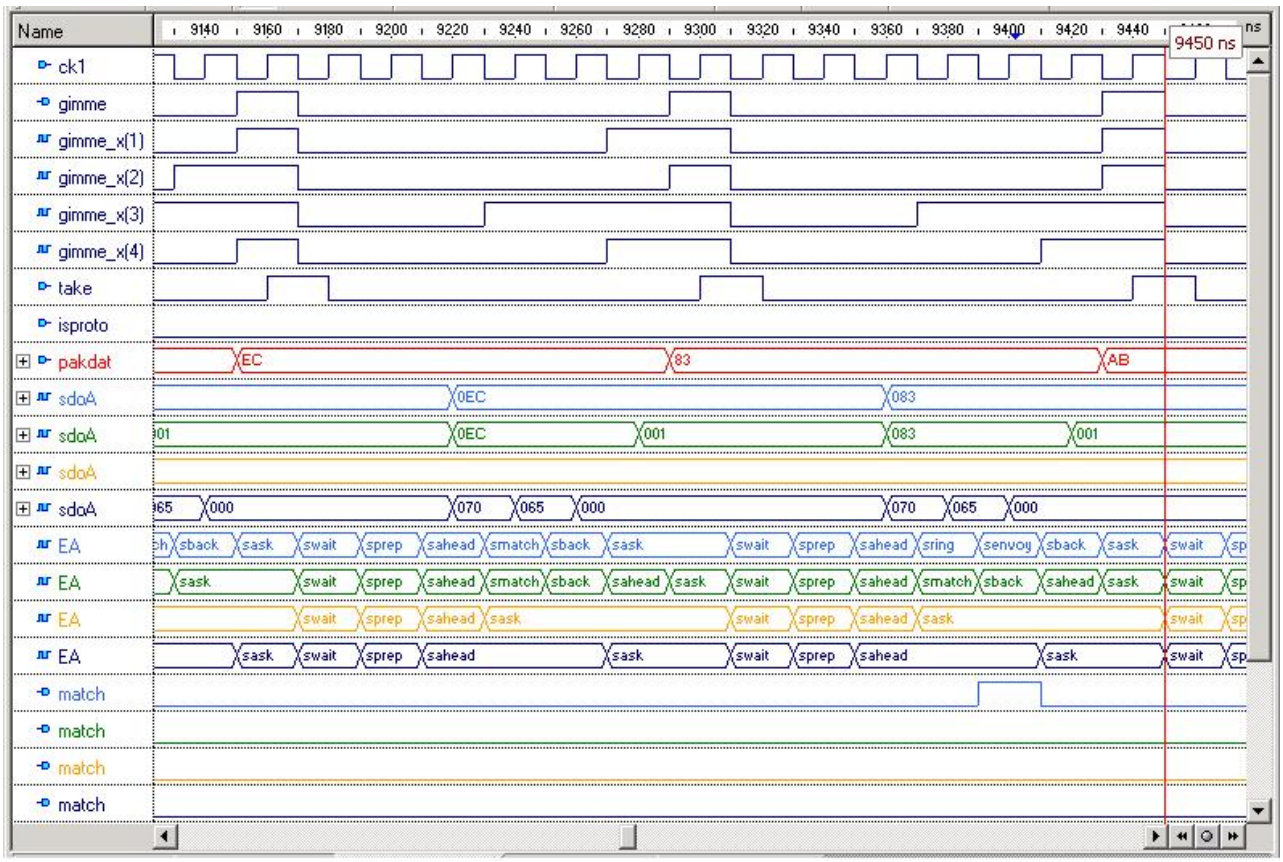

Figura 11 – 4 pico-CPUs em operação

O sinal `isproto` indica, quando ativo, que o byte recebido é um comando da CPU-mestre informando as CPUs do cluster que os próximos bytes pertencerão a um novo protocolo. No exemplo, `isproto` não é ativado indicando que todos os bytes pertencem ao último protocolo informado.

Os sinais `sdoA` (8 bits) são os valores dos padrões acessados pela porta A de cada DPRAM. Para uma mesma CPU, para um mesmo protocolo e para um mesmo byte do pacote a ser comparado haverá tantos acessos quantos padrões deste protocolo atribuídos a esta CPU.

Exemplificando, no caso aqui examinado, a Figura 11 mostra comparações para um mesmo protocolo. Para este protocolo, a Tabela 2 mostra os bytes iniciais de cada padrão em cada CPU. A pico-CPU 1 possui 1 padrão a ser comparado, com 4 bytes de extensão. A pico-CPU 2 possui 2 padrões um com 4 e outro com 16 bytes de extensão respectivamente. A pico-CPU 3 tem apenas 1 padrão com 14 bytes de extensão e a pico-CPU 4 tem 3 padrões com 4, 16 e 7 bytes de extensão.

Tabela 2 - Padrões do protocolo exemplo

picoCPU	Exten.	Padrões
1	4	0x83, 0x04, 0xEC, 0x83
2	4 16	0xEC, 0x83, 0xAB, 0x27 0x01, 0x02, 0x03, 0x04, ...
3	14	0x53, 0x6F, 0x6C, 0x61, ...
4	4 16 7	0x70, 0x6F, 0x6E, 0x67 0x65, 0x63, 0x68, 0x6F, ... 0x00, 0x01, 0x02, 0x03, ...

A CPU que possui mais padrões é a CPU 4. Observando o sinal `sdoA`(4) vemos os bytes iniciais dos três padrões sendo acessados em seqüência. No ciclo de comparações para o byte "`0xEC`" do pacote observa-se que durante o estado `sahead` da CPU4, a cada subida do relógio corresponde um acesso a um padrão diferente.

Neste caso nenhum dos bytes casou com o valor do byte do pacote.

No caso da CPU3 não vemos nenhuma transição do sinal `sdoA`(3). Isto porque esta CPU tem apenas um padrão para este protocolo e toda vez que acessado produz-se o mesmo valor. Este valor é o valor `0x53` que está à frente do padrão, tendo em vista que este padrão não teve um casamento em um acesso anterior.

Para as CPUs 1 e 2 os acessos `sdoA` de ambas produziram o mesmo valor "0xEC" do pacote, caracterizando um casamento de valores. Para estas CPUs pode-se notar pela figura a passagem pelos estados smatch (casamento) e sback.

Como a CPU1 não tem mais padrões à examinar, após `sback` vem o estado `sask` e a ativação de `gimme_x(1)`. A CPU2 ainda tem um padrão a examinar então após `sback` volta o estado `sahead` e o outro padrão é comparado. Como não ocorre o casamento, em seguida vem o estado `sask` para esta CPU igualmente.

A CPU1 está em uma situação especial a altura da comparação do byte de pacote `0xEC`, com este casamento três dos quatro bytes do pacote casaram com valores seqüenciais do pacote. Se no próximo byte do pacote o valor casar novamente haverá uma incidência de padrão e deverá ser gerado um alerta.

É exatamente esta situação que se estabelece durante o próximo ciclo de comparações. O valor do próximo byte do pacote é `0x83` e o acesso ao padrão da CPU1

produz exatamente este valor.

Ao invés de ocorrer um estado `smatch`, dado que é o último byte do padrão que esta sendo comparado, ocorre o estado `sring`. O estado `sring` verifica se os estágios coletores de alertas estão livres para receber um novo alerta. Caso afirmativo a CPU passa ao estado `senvoy` onde um alerta contendo o código do padrão incidente é coletado para um registrador intermediário onde aguardará leitura pela CPU-mestre.

Após o estado `senvoy` segue-se automaticamente o estado `sahead` se houver outros padrões para buscar ou `sask` em caso contrário.

Se durante o estado `sring` fosse constatado que o registrador intermediário ainda estivesse ocupado com um alerta anterior então vários ciclos de relógio se seguiriam no estado `sring`, impedindo a emissão de um próximo `gimme_x` e portanto segurando a execução em todo o cluster.

7 CONCLUSÕES

A análise do estado da arte mostrou a importância de se desenvolver arquiteturas dedicadas para a detecção de intrusão em redes. A evolução dos dispositivos FPGAs, tanto em número de elementos lógicos quanto de blocos de memória embarcada, torna possível a pesquisa de novas arquiteturas para este domínio de aplicação. A contribuição deste trabalho é o desenvolvimento de uma arquitetura mar de processadores para a detecção de padrões, segundo as regras do NIDS Snort. A arquitetura desenvolvida apresenta as seguintes características: (i) escalabilidade, pois o desempenho do sistema não é afetado com a inclusão de novas regras, uma vez que se pode incluir facilmente novas CPUs ao projeto; (ii) flexibilidade, pois as regras não são limitadas em extensão e tipos de caracteres; (iii) desempenho, pois o tempo para se processar um dado byte de entrada contra um conjunto com centenas de regras é na ordem de dezenas de ciclos de relógio. A arquitetura encontra-se validada funcionalmente. Trabalhos futuros incluem: (i) inclusão da CPU-mestre, responsável pelo controle do cluster; (ii) prototipação em FPGA; (iii) análise de desempenho frente a condições reais de ataque.

8. REFERÊNCIAS

[ATT04] Attig, M.; Dharmapurikar, S. Lockwood, J. "**Implementation Results of Bloom Filters for String Matching**". In: FCCM 2004.

[BAK04] Backer, Z; Prassana, V. "**Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs**". In: FPL 2004.

[CAR02] Carver, D; Hutchings, B.L.; Franklin, R. "**Assisting Network Intrusion Detection with Reconfigurable Hardware**". In: Proceedings of the IEEE Symposium on FPGA's for Custom Machimes (FCCM02), 2002, pp.111-120 .

[CAS03] Caswell, Brian; Beale, Jay; Foster, James C. "**Snort 2.0 Intrusion Detection**". Rockland, MA USA: Syngress Publishing, Inc, 2003, 523p.

[CHO02] Cho, Young H.; Navab, Shiva; Mangione-Smith, William H. "**Specialized Hardware for Deep Network Packet Filtering**". In: FPL 2002 .

[COI01] Coit, C. Jason; Staniford, Stuart; McAlerney, Joseph. "**Towards Faster String Matching for Intrusion Detection**". In: DARPA Information Survivability Conference & Exposition II, DISCEX '01), 2001, pp.367-373.

[DES03] Desay, Neil. "Increasing Performance in High Speed NIDS". Capturado em: http://www.linuxsecurity.com/resource_files/intrusion_detection/Increasing_Performance_in_High_Speed_NIDS.pdf, Agosto, 2003, 14p.

[GUS97] Gusfield, D. "**Algorithms on strings, trees, and sequences computer science and computational biology**". Cambridge University Press, 1997, 534p.

[HEN03] Henkel, J. "**Closing the SoC Design Gap**". Computer, v.36 (9), Sep. 2003, pp. 119-121.

[NOR03] Norton, Marc; Roelker, Daniel. "**Snort 2.0 Hi-Performance Multi-Rule Inspection Engine**". Capturado em: http://www.sourcefire.com/technology/wp_request.html, Agosto, 2003, 6p.

[NOR03b] Norton, Marc; Roelker, Daniel. "**Snort 2.0 Rule Optimizer**", Capturado em: http://www.sourcefire.com/technology/ wp_request.html, Agosto, 2003, 7p.

[SID01] Sidhu, R.; Prasanna, V. K. "**Fast Regular Expression Matching using FPGAs**". In: IEEE Symposium on FPGA's for Custom Machimes (FCCM01), 2001.

[SOU03] Sourdis, Ioannis; Pnevmatikatos, Dionisios. "**Fast, Large-Scale String Match for 10Gbps FPGA-based Network Intrusion Detectin System**". In: FPL 2003.