HardCore: Detalhando um HIDS por Assinatura em Hardware

Fernando de Barros Castro
fbc23@inf.ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

Raphael Kaviak Machnicki
rkmachnicki@inf.ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

André Grégio
gregio@ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

Marcos Zanata
zanata@inf.ufpr.br
Universidade Federal do Paraná
Curitiba, Paraná, Brasil

Abstract

Most Host Intrusion Detection Systems (HIDS) do not count with memory evaluation capabilities, being that the reason why they are not able to detect Advanced Volatile Threats, such as Fileless Malware, which can only be detected via memory scans. In this work, we propose HardCore, a signature based HIDS in hardware, able to perform signature matching with no overhead to the endpoint system. HardCore receives as input, at every memory write, a cache line, outputting any matches against known fileless malware present in the signature base. HardCore uses bloom filters and malware clustering to operate, having its signature matrices divided to obtain the gains originating from those implementation choices.

Keywords

DETECÇÃO DE INTRUSÃO EM HARDWARE, BLOOM FILTER, FILELESS MALWARE, HIDS.

1 Introdução

O número de ameaças a sistemas computacionais cresce ano a ano no Brasil. Em 2023, foram registrados no país 1,8 milhão de ocorrências de golpes e fraudes digitais, resultando em perdas financeiras que chegam a R$ 6 bilhões, segundo relatório do Comitê Gestor da Internet no Brasil [4] [15]. Ataques *phishing* e *trojan* são os principais responsáveis por esse aumento na quantidade de fraudes, sendo que o primeiro atua como um vetor de infecção, enquanto o segundo é responsável diretamente pela execução do golpe [7].

Tendo em vista a crescente quantidade de ameaças, soluções de segurança desempenham papel fundamental na detecção e prevenção de tentativas de fraude. Sistemas de Detecção de Intrusão em *hosts* (HIDS - *Host Intrusion Detection Systems*) são capazes de detectar atividades maliciosas ocorrendo no *endpoint* em que estão inseridos. Tais sistemas podem, por exemplo, detectar mudanças em arquivos, instrumentar funções em espaços de usuário ou *kernel* do sistema operacional, analisar pacotes de rede em busca de padrões sabidamente maliciosos, traçar comportamento dos usuário, entre outras ações de detecção.

Apesar de serem efetivos na detecção de algumas ameaças, os HIDS possuem limitações. Uma das principais desvantagem de um HIDS é não ser capaz de detectar os chamados *Fileless Malware*, aqueles que não são escritos em memória secundária em momento nenhum de sua execução. Por exemplo, um código malicioso escreve na memória de um outro processo, causando a instância de diversas *threads* que passarão a executar código arbitrário.

Como as rotinas de detecção dos HIDS passam somente pelas escritas em disco, esse tipo de sistema não é capaz de detectar a presença de *Fileless Malware*. Já os antivírus contam com rotinas de leitura de memória para detecção dessa sorte de ameaça. Porém, esse tipo de abordagem consome recursos computacionais em abundância, o que pode prejudicar a execução das tarefas normais do *host* que protegem [2].

Uma abordagem de detecção de *Fileless Malware* se dá pelo uso de *Harware* dedicado. Dessa maneira, o sistema hospedeiro não teria seu desempenho comprometido, além de toda escrita em memória ser verificada contra um conjunto de assinaturas sabidamente maliciosas pertencente a esse tipo de *malware*. O presente artigo propõe um sistema de detecção de intrusão em *host* por assinatura via *hardware*, que recebe como entrada uma linha de *cache* que será escrita em memória, verifica se há alguma correspondência na base indicando que a linha de *cache* corresponde à uma assinatura conhecida de *Fileless Malware*, e retorna se aquela escrita teve como autor um processo malicioso.

As principais contribuições deste trabalho são:

- a descrição da arquitetura de um HIDS por assinatura para avaliação de linhas de cache, capaz de fazê-lo sem consumir recursos do *endpoint* que está inserido;
- o detalhamento da arquitetura desse sistema, mostrando quais os principais componentes, técnicas e ferramentas necessários para a implementação.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta os conceitos fundamentais para assimilação da arquitetura proposta. A Seção 3 apresenta os trabalhos relacionados. A seguir, a Seção 4 detalha a arquitetura do sistema proposto, enquanto a Seção 5 mostra um estudos de 2 casos diferentes para o comportamento do sistema. Por fim, a Seção 6 traz as considerações finais com relação ao trabalho.

2 Fundamentação Teórica

Esta Seção apresenta os principais conceitos que fundamentam o presente trabalho.

2.1 Detecção de Intrusão e Antivírus

Detecção de Intrusão se refere ao processo de monitorar um sistema para detecção de atividades suspeitas via inspeção e análise de eventos [9]. Existem dois tipos de sistemas de detecção de intrusão (IDS - *Intrusion Detection Systems*): os NIDS (Network IDS) são acoplados a *switches*, e operam avaliando cópia do tráfego de

rede. Já os HIDS (Host IDS) são sistemas específicos para avaliar o *endpoint* que estão monitorando. Sistemas IDS podem ser definidos em *Hardware* ou *Software*.

Existem duas maneiras de implementar os sistemas IDS. A primeira delas é a detecção por anomalias, que propõe modelos que buscam entender o comportamento do usuário, tratando toda atividade que não está alinhada a esse comportamento como uma tentativa de intrusão. Apesar de ser uma abordagem capaz de detectar ataques desconhecidos, ela apresenta um alto índice de falsos positivos. A segunda diz respeito a detecção por assinatura: o IDS buscará por padrões suspeitos previamente estabelecidos em um conjunto de assinaturas de *malware* [10].

Já os antivírus (AVs) são um complemento aos sistemas de detecção de intrusão em *hosts* por assinatura, capazes não só detectar, mas também prevenir instalação e/ou execução de *malware*. Além das funções de HIDS, os antivírus ainda contemplam ações mais sofisticadas para mitigação de ameaças, por exemplo análises *system-wide* estáticas e dinâmicas que podem ocorrer devido a um evento específico que se passe no sistema, ou executadas pelo usuário. Ademais, os AVs ainda realizam *scans* de tráfego de rede, memória e processos, além de prover proteção para navegadores internet via *plugins* [2]. O escopo deste trabalho é limitado em sistemas de detecção de intrusão em *hosts* via *Hardware*, implementados via detecção por assinatura.

2.2 *Fileless Malware*

Fileless Malware são assim denominados porque não são escritos em disco em momento nenhum. Assim sendo, os Antivírus não os podem detectar realizando as rotinas de inspeção em disco [3]. Para contornar esse problema, os Antivírus implementam rotinas de avaliação de memória, que avaliam as imagens dos processos sendo executados buscando por assinaturas de *malware* conhecidas. Apesar de efetiva, essa abordagem exige que o Antivírus colha amostras de memória frequentemente, causando assim problemas de desempenho.

2.3 Codificador de prioridade

A entrada do codificador de prioridade é um vetor de bits, sendo a primeira posição do vetor a mais significativa. Dessa forma, o codificador tem por objetivo indicar na saída qual a posição mais significativa do vetor com indexação de 0 até N-1 (N elementos) está ativa. A Figura 1 mostra um exemplo de tabela verdade para um codificador de prioridade 4x2.

A fim de exemplificar, considere que A é o *bit* mais significativo e D o menos significativo, toda vez que o *bit* A estiver ligado, a saída será 00, ou seja, a posição de A, independentemente das demais posições do *bitmap*. Note também que caso todos os *bits* da entrada estejam desligados, a saída seria a mesma que quando o *bit* A está ativo. Para resolver essa ambiguidade, adiciona-se a saída E0, que indica se existe *bit* ativo na entrada.

2.4 *Bloom Filter*

É determinante para entender a motivação arquitetura o conceito do *Bloom Filter*: um algoritmo probabilístico que tem por finalidade retornar se um elemento está presente num conjunto, a partir de uma série de funções *hash* e um vetor de *bits* (*bitmap*).

Ao inserir um elemento no conjunto, aquele passa por uma sequência de funções *hash*, em que cada uma delas retornará um índice. O *bitmap* então receberá o valor 1 em cada um desses índices. Posteriormente, para realizar busca no conjunto, o elemento passa pelas mesmas funções *hash*. Se os índices gerados indicarem 1 no *bitmap*, o elemento provavelmente estará no conjunto. Se algum dos índices representar 0 no *bitmap*, certamente o elemento não pertence ao conjunto.

Portanto, o *bloom filter* permite afirmar que um elemento não pertence a um conjunto, porém não garante a presença do elemento. Em outras palavras, podem haver falsos-positivos em decorrência de outros elementos, quando inseridos, terem seu retorno pelas funções *hash* iguais a do elemento buscado.

Em outras palavras, existe sempre uma probabilidade maior que zero de haver falsos-positivos. Essa probabilidade pode ser calculada de acordo com a Equação 1, onde `m` representa o tamanho do *bitmap*, `n` o número de elementos a serem inseridos no filtro, e `k` o número de funções *hash* sendo utilizadas.

$$P = (1 - [1 - 1/m]^{kn})^k \tag{1}$$

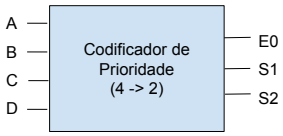
Para determinar o número ótimo de funções *hash* a serem utilizadas, segue-se a Equação 2

$$k = \ln 2 * n/m \tag{2}$$

3 Trabalhos Relacionados

Há mais de 20 anos, Sato and Fukase [14] investigaram o projeto e a implementação de um HIDS diretamente na placa de rede usando FPGA. O objetivo do trabalho era o de proteger o sistema na ponta do usuário, com baixo custo (o custo da FPGA), em tempo real, sem carga para a CPU, com alta capacidade de processamento de dados para detecção de ataques internos. Para mostrar a viabilidade da proposta, os autores limitaram o HIDS em hardware para detecção de acesso ilegal à certas portas (por exemplo, FTP, Telnet, SMTP, DNS, HTTP e POP). A técnica implementada alcançou velocidades que poderiam tratar adequadamente bandas de 1GBps, com atrasos entre 3,09 e 5,38ns.

Uma vez que os HIDS também podem ser utilizados para detecção de ameaças internas, isto é, ataques feitos por indivíduos que possuem acesso e podem modificar o sistema alvo. Nesse sentido, sistemas embarcados são críticos, e sua segurança implica na segurança das pessoas que os utilizam. Rahmatian et al. [13] discutem o problema temporal para se produzir e analisar registros de auditoria de HIDS, fazendo-os mais um dispositivo para auxiliar a perícia forense (após os ataques ocorrerem) do que um detector de ameaças em tempo real. Para lidar com esse problema, os autores apresentam um HIDS suportado por hardware cujo objetivo é detectar ataques em tempo real. Assim, foi implementado um HIDS em FPGA baseado em anomalia, cujo comportamento genuíno deve ser previamente modelado, o qual monitora chamadas de sistema e monta um grafo de controle de fluxo da execução em busca de sequências "ilegais". Os testes foram feitos com base na avaliação do HIDS proposto ao monitorar quatro aplicações (*bubblesort*, *inetd*, *telnet* e *FTP*). Considerando as quatro aplicações sendo monitoradas ao mesmo tempo, houve um *overhead* de área de 3,38% e de energia de 3,12%, mas o de desempenho foi negligenciável. Apesar de



A	B	C	D		S2	S1	E0
0	0	0	0		0	0	0
0	0	0	1		1	1	1
0	0	1	X		1	0	1
0	1	X	X		0	1	1
1	X	X	X		0	0	1

Figura 1: Codificador de Prioridade

promissor, o HIDS de FPGA apresentado requer a perfilização das aplicações a serem monitoradas, o que pode ser inviável, além de ter o potencial de gerar uma massa considerável de falsos positivos.

Damien et al. [5] discutem a implementação de um HIDS considerando os requisitos restritos da aviônica, tais como a preservação da ordem de execução em tempo real, pouca memória, resultados confiáveis e explicáveis, e a atualização frequente (com pelo menos 28 dias de distância entre elas). Sua proposta baseou-se em construir um modelo cujo perfil consiste do comportamento legítimo de cada aplicação monitorada, isto é, sua sequência de chamadas de API, e detectar em tempo real anomalias em relação a esse modelo. Os resultados mostraram-se promissores, com 0,014ms para processar os *logs* gerados por uma aplicação monitorada e 2,7% de *overhead* para cada chamada de API feita pela mesma aplicação. Entretanto, tal abordagem é limitada à chamadas de API e sua atuação não considera a monitoração de múltiplas aplicações.

Shone et al. [17] propõem um NIDS construído sobre um modelo que combina *deep learning* com *Random Forest*, implementado usando *TensorFlow* e avaliado em GPU com os *datasets* KDD Cup '99 [11] e NSL-KDD [19]. Apesar de relatar *f-score* de 87,37%, os testes foram feitos com ataques antigos e baseados somente em tráfego de rede. Nenhuma medida de eficácia/tempo foi apresentada no sentido em se avaliar a aplicabilidade da proposta em tempo real.

Al Neami et al. [1] apresentam um IDS para dispositivos na IoT baseado em FPGA e cujo foco é a proteção da rede em tempo real. O sistema aplica diversas técnicas de aprendizado de máquina, tais como XGBoosting, CNNs e LSTMs, e foi testado em *datasets* publicamente disponíveis (UNSW NB15 [12], CICIDS2017 [16], IOTID20 [18] e NSL-KDD). O IDS proposto foi implementado de maneira modular na FPGA, contendo componentes para pré-processamento de tráfego de rede, extração e seleção de características, motor de inferência e módulo de resposta e decisão. Cada conjunto de técnicas de aprendizado de máquina foi aplicado aos diversos *datasets*, ressaltando-se que para classificação de subcategorias de ataque, o modelo baseado em *deep learning* teve aproximadamente 50% de acurácia para o conjunto de dados específico para IoT (IOTID20). Entretanto, não foram apresentadas medidas de desempenho de tempo ou sobrecarga com o uso deste IDS.

No campo do aprendizado de máquina aplicado à segurança cibernética, Foudhaili et al. [6] apresentam uma metodologia para construir um IDS em FPGA, mais especificamente, considerando-se harware de borda (*edge*) reconfigurável. São avaliados modelos baseados em XGBoost, SVM (*Support Vector Machines*), Naive Bayes, *Random Forest*, árvores de decisão (DT) e três redes neurais de múltiplas camadas (MLP), os quais são aplicados nos seguintes tipos de tráfego de rede (advindos do dataset BOT-Iot [8]: Negação de Serviço (DoS), Reconhecimento, Roubo e Normal/Legítimo). Os autores concluem que as MLPs e o XGBoost superam os outros modelos em eficácia e medem características de desempenho e custo do hardware, como uso de recursos, vazão na inferência, eficiência energética, densidade computacional e flexibilidade. Com isso, concluem a viabilidade de se usar um IDS do tipo, limitado ao tráfego de rede, devido a sua eficiência e eficácia serem comparáveis ao estado da arte.

Pensando em auxiliar um antivírus (AV) via hardware na detecção de *Fileless Malwares*, Botacin et al. [3] propuseram adicionar ao controlador de memória uma unidade de verificação de dados sendo escritos. Considerando que apenas regiões de memória alteradas precisam ser avaliadas, o artigo reconheceu que há uma janela de tempo de tempo considerável entre uma escrita e posterior leitura a um mesmo endereço na memória que poderia ser aproveitada para identificar padrões maliciosos. A proposta consiste em diminuir a dependência nos recursos em software já estabelecidos para detecção de *Fileless Malwares* com o auxílio de um *Bloom Filter* implementado em harware, restando apenas para a aplicação avaliar aquilo que o filtro detectou como malicioso. Testes com assinaturas de 32 bytes com uma base de dados com 1K entradas e taxa de falso positivo desejada de 0.1% (considerando assinaturas de alta entropia e funções de *hash* com distribuição ideal) revelaram ótima eficiência aliado a baixo custo de memória: menos de 1% de *overhead* de tempo em todos os testes para aplicações não maliciosas.

4 Arquitetura do Sistema

O sistema proposto é definido como um mecanismo em *hardware* para detecção de processos maliciosos sem imagem binária presente no disco. Aproveitando-se da janela existente de alguns ciclos entre uma escrita e leitura a um mesmo endereço, o sistema opera realizando detecção por assinatura via *Bloom Filters*. Botacin et al. [3] também propõem um sistema capaz de detectar esse tipo de processo malicioso, no entanto, o que distancia o atual artigo da proposta supracitada é a filtragem dos dados. Um *Bloom Filter* convencional pode gerar muitos falsos positivos para uma base com muitas entradas, porque o filtro condensa todas as *hashes* obtidas para todas as assinaturas em um único *bitmap*. Assim, um cenário em que todas ou quase todas as posições do vetor de *bits* estão ativas não é improvável, como dita o princípio da casa dos pombos.

Este cenário pode afetar a proposta de Botacin et al. [3], quando há números cada vez maiores de *Fileless Malwares*.

Para contornar este problema, a solução proposta neste trabalho, denominada HardCore (**H**ardware **A**ssited **R**isk **D**etection **Core**) se aproveita de que dividindo as assinaturas em grupos, cada um com *bitmap* individual, é possível eliminar diversos grupos de malwares a cada aplicação de *hashes* distintas. Exemplificando: dado um conjunto com 1K assinaturas divididos em 10 grupos, caso o índice dado pela função de *hash* conter 1 em apenas um dos *bitmaps*, se faz necessário, então, verificar a linha de cache contra as assinaturas do único grupo em que houve a colisão. Além de separar os malwares em grupos, a proposta também prevê para cada grupo criado *bitmaps* únicos para as funções de *hash* utilizadas. Esse espalhamento dos dados permite a utilização de *bitmaps* menores e de menos *hashes*.

HardCore opera recebendo como entrada uma linha de *cache* e o endereço da escrita paralelamente com a escrita em memória. Após a análise probabilística, uma memória de resultados grava o veredito para aquela escrita. A Figura 2 mostra a arquitetura da solução. As Subseções a seguir descreverão cada etapa de processamento, desde a aquisição da linha de *cache*, até o veredito definido pelo sistema, passando pelos processamentos intermediários realizados por HardCore.

4.1 Inicialização: Conteúdo das Matrizes de Assinaturas

Primeiramente, as assinaturas (linhas de *cache*) ideais para a distinção dos códigos maliciosos são escolhidas e agrupadas em *clusters*. Para fins de demonstração, a arquitetura usa duas funções de *hash*, logo cada *cluster* origina dois *bitmaps*. Os vetores formados são armazenados em duas matrizes. Considerando o endereçamento das matrizes em memória, acessar uma coluna é mais custoso que uma linha, então a primeira matriz é construída de forma que cada linha de índice K tenha todos os valores dos N *bitmaps* na respectiva posição K, ou seja, os vetores são guardados coluna a coluna. Já os *bitmaps* formados da outra *hash* são armazenados linha a linha na segunda matriz de assinaturas.

Após obter a *hash* da linha de *cache* a ser escrita (entrada do sistema), o resultado é usado como índice para acessar uma linha da Matriz 1. Neste índice estará um *bitmap* que indica em quais grupos pode haver uma correspondência com uma assinatura de *malware*. Um codificador de prioridade (Vide Subseção 2.3), que recebe como entrada o *bitmap*, retorna uma linha da Matriz 2 para acesso ou, no melhor caso, nenhuma, obtendo-se o veredito para código não malicioso.

4.2 Busca: Aquisição de linha de *cache* e Escrita no *Buffer Circular*

Visto que o caminho crítico exigiria um período de relógio longo para fazer toda a análise em um ciclo, propõe-se um circuito sequencial que tem potencial para utilizar mais de uma batida de relógio para análise da linha de entrada. Assim, um *buffer* é apropriado. Sua ausência poderia levar uma série de escritas na memória a não passarem pelo análise probabilística, forçando que todas sejam consideradas suspeitas e uma sobrecarga do sistema, consumindo ciclos de processamento e prejudicando o desempenho da máquina. Um *Buffer Circular* é ideal para essa tarefa devido a sua alta eficiência: nenhum deslocamento de entradas é feito para remover ou inserir um elemento.

O *buffer* armazena o endereço da escrita, o valor obtido pela segunda função *hash* (guardar a linha para depois obter o valor não seria eficiente em quesito de armazenamento) e um ponteiro para a linha da primeira matriz. Na porção esquerda da Figura 2 (ponto 1) tem-se o recebimento da linha e do endereço. O *buffer* tem em sua construção um ponteiro para a entrada mais antiga que será tratada no ciclo atual e outro para a próxima posição livre. Vale ressaltar que o processamento da entrada mais antiga pode demorar vários ciclos: uma batida para cada grupo que pode conter a linha como uma de suas assinaturas.

4.3 Busca: Processamento das Entradas do *Buffer* Circular

Para efeitos práticos, considera-se que a cada ciclo do relógio, uma entrada do *Buffer* será processada. Lembrando que ele contém a linha obtida da primeira matriz, é preciso encontrar uma forma de alterar a linha após a execução do algoritmo para que o codificador aponte para o próximo grupo/*bitmap* a ser usado no filtro (isso é, se o algoritmo indicou que a linha modificada **não** pertence ao primeiro grupo). A solução para isso faz uso do próprio codificador: o *bitmap* armazenado no *Buffer* recebe zero na posição indicada pela saída dele. A etapa descrita é executada para a entrada mais antiga apontada até que todo o seu *bitmap* seja um vetor de zeros, quando todas as possibilidades foram avaliadas, ou antes se houve um positivo para *malware*. Pode-se pensar que a primeira etapa do HardCore seleciona *bitmaps* para o posterior uso do *Bloom Filter* usando o *buffer* como intermediário.

4.4 Busca: Veredito do Sistema

A arquitetura proposta possibilita alguns cenários:

(1) `Matriz1[Hash 1]` está zerada: significa então que certamente a linha não representa código malicioso de acordo com a base de assinaturas, pois nenhuma assinatura gerou a mesma saída após passar pela função de *hash*.
(2) `Matriz[Hash 1]` não está zerada. No entanto, `Matriz2[cluster][Hash 2] == 0` para todos os grupos: também leva a um resultado negativo. O que houve foi apenas uma ou mais colisões com a primeira *hash*.
(3) `Matriz[Hash 1]` não está zerada e `Matriz2[cluster][Hash 2] == 1` para um determinado *cluster*: Isso leva finalmente a um resultado positivo para *malware*. Houve colisão com a primeira e segunda *hash* para aquele grupo específico. Pode ser um falso positivo. Aqui já é possível encerrar a análise da linha e escrevendo 1 na Memória de Resultados na posição indicada pelo endereço armazenado e movendo o ponteiro do *buffer* uma posição adiante.

4.5 Considerações sobre a Arquitetura

Os autores reconhecem que uma outra solução possível seria fazer um `E` bit a bit de dois *bitmaps* e um `OU` dos bits do vetor resultado: dados os resultados x1 e x2 para as *hashes* usadas, os *bitmaps* seriam respectivamente um vetor de booleanos para ativação do valor x1 e x2 nos *clusters*, enquanto o *bitmap* resultado advém da operação de

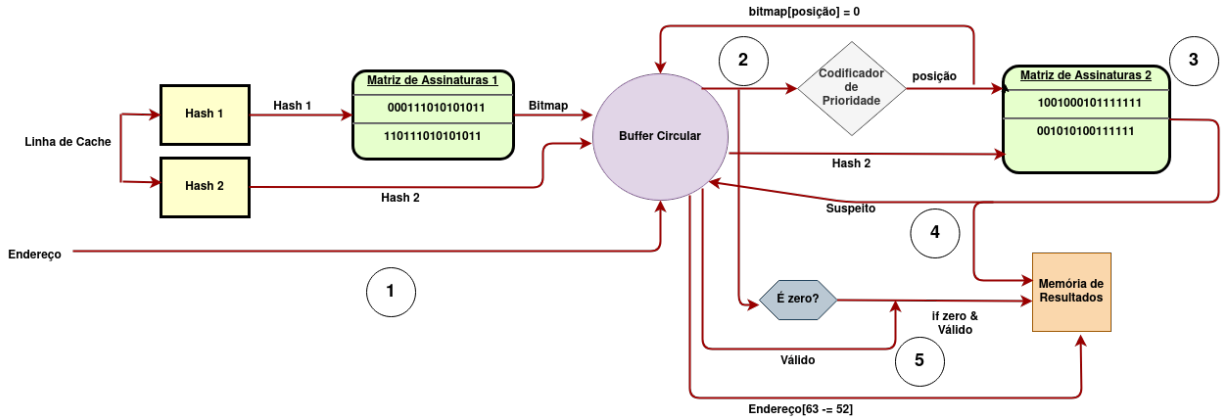


Figura 2: Arquiteura do HardCore

E *bitwise*. O resultado seria o mesmo retirando a complexidade da arquitetura descrita. No entanto, essa abordagem monociclo ignora que em um cenário com a utilização de várias funções de *hash*, haveriam muitos acessos a memórias para obter os *bitmaps*, levando a alta latência e diminuindo a vazão do sistema. Então, a construção de uma estrutura sequencial possibilita resolver esse problema e criar circuitos mais avançados a partir da base inicial descrita. A ideia central de divisão das assinaturas em grupos também permite liberdade criativa para achar métodos para agrupar visando menor taxa de falsos positivos.

O componente 'É zero' presente na Figura 2 determina se o *bitmap* de entrada está zerado, usado então para descartar a entrada mais antiga do *buffer* quando já se concluiu que a linha não pertence ao conjunto de assinaturas. Já o sinal 'Válido' determina se o *Buffer* Circular não está vazio, ou seja, é puramente um desambiguador que evita escritas inválida na Memória de Resultados.

Como previamente detalhado, o *Buffer* Circular é composto por três campos: um *bitmap*, um endereço e o resultado da segunda *Hash*. A Figura 3 mostra um exemplo do *buffer* preenchido. O endereço possui 52 *bits* porque dos 64 *bits* usados para endereçar páginas de 4 `Kib`, os 12 menos significativos são descartados, pois sinalizam o *offset* na página. Ao usar somente o endereço da página, tem-se garantia que todas as escritas nela realizadas serão direcionadas para a mesma posição na Memória de Resultados. É fato que apenas escritas em páginas com permissão de execução devem ser avaliadas, como mencionando por Botacin et al. [3].

Os ponteiros de início e fim seguem as técnicas clássicas para implementação de *buffers* circulares, usando o tamanho máximo para calcular a próxima posição apontada. Um registrador contendo a quantidade de entradas serve como desambiguador para obtenção do sinal 'Válido'. Os dois ponteiros apontam para mesma posição quando o *buffer* está vazio e cheio.

5 Estudos de caso

Esta Seção apresentará 2 estudos de caso para a linha de *cache* sendo avaliada: (i) considerada suspeita; e (ii) considerada legitima. Para tanto considere que o banco de dados contém quatro assinaturas a1, a2, a3 e a4 divididas em dois grupos ((a1 e a2) e (a3 e a4)). Usando duas funções f1 e f2 levando a valores de 0 a 3, temos então quatro *bitmaps* gerados: a1 e a2 ao passarem por f1 geram `0100`; a3 e a4 passando for f1 geram `1001`; a1 e a2 passando por f2 geram `1010` e a3 e a4 geram `0001` passando por f2.

A Figura 4 mostra o preenchimento das duas matrizes para o estudo de caso descrito acima. Note que, para a Matriz 1, o vetor de *bits* resultado de f1 é preenchido por coluna. Já a matriz 2 é completada linha a linha, com os *bitmaps* resultantes de f2.

5.1 Linha de *Cache* legítima

Suponha que a linha de *cache* a ser escrita (entrada do sistema) passa pela função f1 gerando o valor 1. Tem-se então que 1 deve ser usado como índice para a Matriz 1, resultando no *bitmap* `10`. Suponha também que para esta entrada, o resultado de f2 foi o valor 3.

Após um ciclo, o valor 'Válido' torna-se *true*. O *bitmap* `10` é usado como entrada para o codificador de prioridade. A saída desse aponta para a linha 0 da Matriz 2, justamente a linha que contém `1010`. Note que a saída do codificador é zero porque a posição 0 do *bitmap* possui um *bit* 1. Verifica-se que a terceira posição (o resultado de f2 foi 3) do *bitmap* está zerada, não ocorrendo escrita na memória. Mais um ciclo se passa e a posição 0 (saída do codificador) do vetor `10` recebe zero tornando-se `00`. O componente 'É Zero' fica ativo, sendo escrito na memória o valor zero na posição dada pelos 52 bits mais significativos do endereço armazenado no *buffer* e o ponteiro de entrada mais antiga é movido, efetivamente retirando a entrada.

5.2 Linha de *Cache* suspeita

Novamente ocorre uma requisição de escrita, com uma linha de cache e o endereço da alteração. Os valores obtidos das funções f1 e f2 são, respectivamente, 0 e 3.

A primeira matriz na posição 0 contém o *bitmap* `01`. A saída do codificador é 1, apontando para o grupo das assinaturas a3 e a4. Na próxima batida de relógio, com a nova entrada inserida no *buffer*, a segunda matriz é acessada na linha 1, a qual contém o vetor `0001`. Esse vetor, por sua vez, também tem 1 na terceira posição. Isso leva a conclusão de que os dados a serem escritos tiveram as mesmos resultados para ambas as funções de *hash* para o conjunto

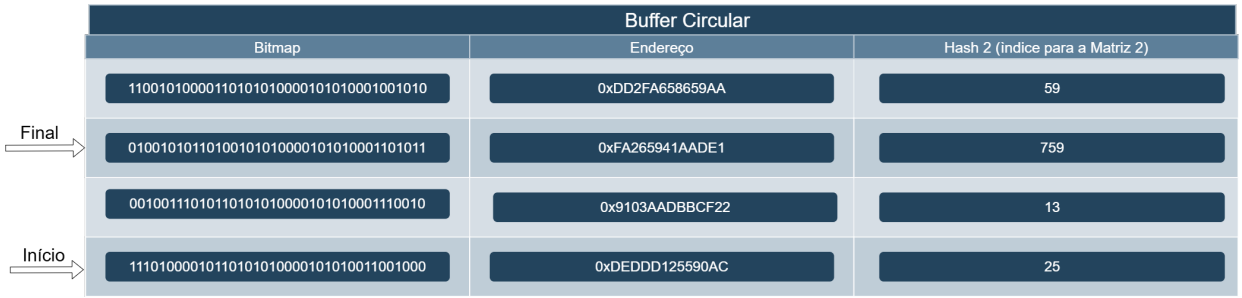


Figura 3: Buffer Circular

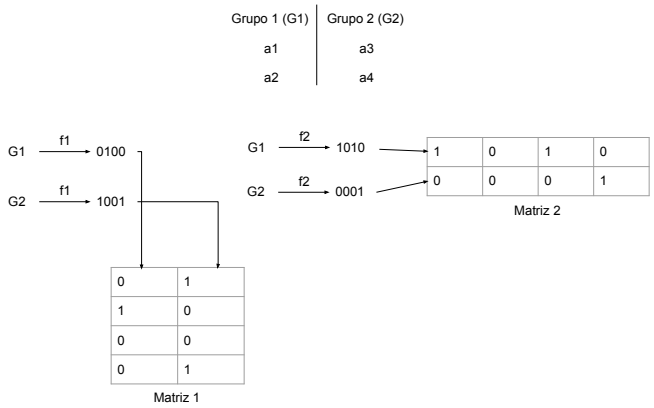


Figura 4: Preenchimento das matrizes para o estudo de caso.

das assinaturas a3 e a4. O algoritmo conclui que os dados pertencem ao conjunto e na posição indicada pelos 52 bits mais significativos do endereço de escrita, a memória de resultados recebe 1. Então, a linha é considerada suspeita e a entrada é retirada do *buffer*.

6 Conclusão e Trabalhos Futuros

A detecção de *Fileless Malware* pode ser considerada uma tarefa difícil e insólita, haja vista que grande parte das soluções não é capaz de identificar a execução de um processo malicioso que não faz interações em memória secundária. Uma das maneiras de realizar tal detecção se dá através da implementação de um sistema de detecção de intrusão em *harware* por assinatura, o que garante que o *host* contemplado não tenha seu processamento normal comprometido, em oposição aos HIDS usuais. Tais assinaturas na verdade representam *hashes* de linhas de *cache* sabidamente representativas de *Fileless Malware*.

Assim sendo, este trabalho propõe o HardCore, um HIDS capaz de avaliar uma linha de cache a cada vez que ocorre um escrita em memória. Ao contrário dos HIDS comuns, essa abordagem permite realizar, através do algoritmo *Bloom Filter*, a avaliação das assinaturas com um alto nível de granularidade, além de não onerar o sistema contemplado e permitir acesso eficaz e eficiente à base de assinaturas.

Para as futuras contribuições envolvendo o HardCore, pretende-se realizar testes comparando seu desempenho com aqueles trabalhos reportados do Seção 3, evidenciando qual *dataset* será abordado em cada teste. Além disso, o *bloom filter* pode ser configurado variando probabilidade de falsos-positivos, do tamanho do *bitmap* e da quantidade de funções *hash*.

Agradecimentos

Este trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) - Código de Financiamento 001. Os autores também agradecem o Programa de Pós-Graduação em Informática da Universidade Federal do Paraná.

Referências

[1] Israa Al Neami, Zaynab Saeed, and Abbas Zahraa. 2024. Adaptive FPGA-Based Intrusion Detection System for Real-Time Internet of Things Security. *Journal of Intelligent Systems and Internet of Things* 14 (09 2024), 278–292.

[2] Marcus Botacin, Felipe Duarte Domingues, Fabrício Ceschin, Raphael Machnicki, Marco Antonio Zanata Alves, Paulo Lício de Geus, and André Grégio. 2022. Antiviruses under the microscope: A hands-on perspective. *Computers & Security* 112 (2022), 102500.

[3] Marcus Botacin, André Grégio, and Marco Antonio Zanata Alves. 2020. Near-memory & in-memory detection of fileless malware. In *Proceedings of the International Symposium on Memory Systems*. ACM, Brazil, 23–38.

[4] CGI. 2024. CGI. https://cgi.br. Acessado em: 06/12/2024.

[5] Aliénor Damien, Michael Marcourt, Vincent Nicomette, Eric Alata, and Mohamed Kaâniche. 2019. Implementation of a Host-Based Intrusion Detection System for Avionic Applications. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 178–17809. https://doi.org/10.1109/PRDC47002.2019.00048

[6] Wadid Foudhaili, Anouar Nechi, Celine Thermann, Mohammad Al Johmani, Rainer Buchty, Mladen Berekovic, and Saleh Mulhem. 2024. Reconfigurable Edge Hardware for Intelligent IDS: Systematic Approach. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Iouliia Skliarova, Piedad Brox Jiménez, Mário Véstias, and Pedro C. Diniz (Eds.). Springer Nature Switzerland,

48–62.

[7] Kaspersky. 2024. Nova epidemia: phishing cresce mais de 5 vezes no Brasil com retomada das atividades econômicas e apoio da IA. https://www.kaspersky.com.br/about/press-releases/nova-epidemia-phishing-cresce-mais-de-5-vezes-no-brasil-com-retomada-das-atividades-economicas-e-apoio-da-ia. Acessado em: 06/12/2024.

[8] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. 2019. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Generation Computer Systems* 100 (2019), 779–796. https://doi.org/10.1016/j.future.2019.05.041

[9] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. 2013. Intrusion detection systems: A comprehensive review. *Journal of Network and Comp. Applications* 36, 1 (2013), 16–24.

[10] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. 2008. Using string matching for deep packet inspection. *Computer* 41, 4 (2008), 23–28.

[11] John McHugh. 2000. Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Trans. Inf. Syst. Secur.* 3, 4 (Nov. 2000), 262–294. https://doi.org/10.1145/382912.382923

[12] Nour Moustafa and Jill Slay. 2015. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*. 1–6. https://doi.org/10.1109/MilCIS.2015.7348942

[13] Mehryar Rahmatian, Hessam Kooti, Ian Harris, and Eli Bozorgzadeh. 2012. Hardware-Assisted Detection of Malicious Software in Embedded Systems. *Embedded Systems Letters, IEEE* 4 (12 2012), 94–97. https://doi.org/10.1109/LES.2012.2218630

[14] T. Sato and M. Fukase. 2003. Reconfigurable hardware implementation of host-based IDS. In *9th Asia-Pacific Conference on Communications (IEEE Cat. No.03EX732)*, Vol. 2. 849–853 Vol.2. https://doi.org/10.1109/APCC.2003.1274480

[15] SecurityReport. 2024. Golpes virtuais atingem 1,8 milhão de vítimas e geram R$ 6 bilhões em perdas no Brasil. https://securityleaders.com.br/golpes-virtuais-atingem-18-milhao-de-vitimas-e-geram-r-6-bilhoes-em-perdas-no-brasil/. Acessado em: 06/12/2024.

[16] Iman Sharafaldin, Arash Habibi Lashkari, Ali A Ghorbani, et al. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp* 1 (2018), 108–116.

[17] Nathan Shone, Tran Nguyen Ngoc, Vu Dinh Phai, and Qi Shi. 2018. A Deep Learning Approach to Network Intrusion Detection. *IEEE Transactions on Emerging Topics in Computational Intelligence* 2, 1 (2018), 41–50. https://doi.org/10.1109/TETCI.2017.2772792

[18] Imtiaz Ullah and Qusay H. Mahmoud. 2020. A Scheme for Generating a Dataset for Anomalous Activity Detection in IoT Networks. In *Advances in Artificial Intelligence*, Cyril Goutte and Xiaodan Zhu (Eds.). Springer International Publishing, Cham, 508–520.

[19] RUIZHE ZHAO. 2022. NSL-KDD. https://doi.org/10.21227/8rpg-qt98