UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Marcação de Pacotes IPv6 utilizando uma Cadeia de Confiança

Phyllipe de Souza Lima Francisco

Itajubá, Junho de 2016

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Phyllipe de Souza Lima Francisco

Marcação de Pacotes IPv6 utilizando uma Cadeia de Confiança

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Otávio A. Salgado Carpinteiro
Co-Orientador: Prof. Dr. Edmilson Marmo Moreira

Junho de 2016
Itajubá - MG

Este trabalho é dedicado a meus pais, Cezar e Helena

Agradecimentos

Agradeço primeiramente a Deus, que me deu força e sabedoria para sempre seguir em frente. Agradeço a meus pais que desistiram de seus sonhos para que eu pudesse realizar o meu, me forneceram toda estrutura familiar e a base na qual fui educado e por isso hoje, posso completar mais essa etapa da minha vida. Agradeço aos demais familiares por todo apoio recebido.

Agradeço o meu orientador Prof. Dr. Otávio Carpinteiro, que se comportou como um verdadeiro guia para mim neste período que estive na UNIFEI. Sempre esteve de portas abertas me orientando da melhor forma possível. Prof. Otávio teve papel fundamental na concretização deste trabalho, e na publicação que este gerou. Sempre procurou extrair o melhor de mim e nunca deixou de acreditar que eu poderia chegar ao fim. Um agradecimento também a meu co-orientador, Prof. Dr. Edmilson Moreira pela cooperação para a realização deste trabalho.

Agradeço também aos demais professores do grupo GPESC que deram ideias e sugestões para que pudesse realizar um bom trabalho. Agredeço a Engenheira Alessandra Adami pela revisão ortográfica do texto, e por fim, agradeço a todos os colegas do laboratório GPESC, no qual tive o prazer de dividir parte do meus dias de trabalho, além de serem exímios conhecedores da gastronomia de Itajubá.

Anduin, I now believe as you do, that peace is the noblest aspiration. But to preserve it, you must be willing to fight!
(Varian Wrynn - High King of the Alliance)

Resumo

Esta dissertação apresenta um novo método determinístico de rastreamento de pacotes IP - *Chain of Trust Packet Marking* (CTPM) - Marcação de Pacotes por cadeia de confiança. O CTPM cria uma cadeia de confiança composta pelos roteadores de borda de um sistema autônomo - SAs. Faz uso de um novo cabeçalho de extensão - *The Traceback Extension Header* (TEH) - Cabeçalho de extensão para rastreamento - que extende o pacote em até 168bytes (pior caso). O TEH contém dados criptografados que são capazes de rastrear o caminho percorrido por cada pacote IPv6. O gasto computacional e a degradação da largura de banda causados pela implementação do CTPM são apresentados ao final.

Palavras-chaves: IPv6. rastreamento. segurança. roteador de borda.

Abstract

This dissertation proposes a new deterministic traceback method — chain-of-trust packet marking (CTPM). CTPM establishes a chain of trust composed of the border routers of the autonomous systems (ASes). It makes use of a new IPv6 extension header — the traceback extension header (TEH) — which extends the datagram size in 168 bytes at most. The TEH contains encrypted marks which trace the path taken by each IPv6 datagram from its origin to the destination. The computational load on the border routers and bandwith degradation generated by CTPM is measured and presented.

Key-words: IPv6. traceback. security. border routers.

Sumário

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AES	Advanced Encryption Standard
AS	Autonomous System
ASN	Autonomous System Number
BGP	Border Gateway Protocol
CTM	Chains of Trust Mark
CBC	Cipher Block Chaining
CTPM	Chain of Trust Packet Marking
CS	Contador de Saltos
DES	Data Encryption Standard
DDoS	Distributed Denial of Service
DPM	Deterministic Packet Marking
LKM	Linux Kernel Module
HC	Hop Count
HL	Header Length
ICMP	Internet Control Message Protocol
ISP	Internet Service Provider
IPv6	Internet Protocol Version 6
IPv4	Internet Protocol Version 4
IETF	Internet Engineering Task Force
MA	Marca Autenticadora
MAC	Message Authentication Code
MI	Marca Identificadora
MTU	Maximum Transmission Unit

NAT	Network Address Translation
NH	Next Header
NIST	National Institute of Standards and Technology
PPM	Probabilistic Packet Marking
RFC	Request for Comments
SA	Sistema Autônomo
TEH	Traceback Extension Header
TFM	Transformation
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Introdução

A transmissão de dados na Internet ocorre através de pacotes IP (*Internet Protocol*). Cada pacote IP carrega dois endereços, o de origem e o de destino. O último serve para que o pacote seja, de fato, entregue, e o primeiro serve para identificar quem enviou a mensagem. O problema está justamente neste endereço de origem, pois pode ser modificado (*spoofed*) (BELENKY; ANSARI, 2003). No seu lugar, pode ser colocado um endereço de origem pertencente a outro dispositivo, ou mesmo um endereço que nem exista, completamente inválido.

Este *spoofing* do endereço IP permite o envio de pacotes de forma totalmente anônima, sendo que estes pacotes podem carregar conteúdos maliciosos (*spam* por exemplo), ou podem fazer parte de um ataque DDoS (*Distributed Denial of Service*). Este último é caracterizado pelo envio de quantidades gigantescas de pacotes IPs para um destino, que, no caso, é uma vítima. O objetivo do ataque é esgotar a capacidade da vítima de receber novos pacotes, impedindo assim usuários legítimos de acessarem os serviços da vítima. No DDoS, existe mais de uma fonte originadora de pacotes maliciosos (até mesmo milhares). Quando há apenas uma fonte, o ataque é chamado de DoS (*Denial of Service*). É um ataque simples, bastando enviar grandes quantidades de pacotes para sua vítima. Normalmente, estes ataques são realizados de forma anônima, forjando o IP de origem. Assim, não é possível saber os causadores do ataque (SINGH; SINGH; KUMAR, 2016).

Normalmente, o indivíduo ou grupo que está organizando o DDoS toma o controle de outras máquinas e as utilizam como escravos (*slaves*) ou zumbis (*zombies*). Essas máquinas *zombies* pertencem a outras redes e o seus adminstradores não têm ciência de que suas máquinas estão sendo controladas por terceiros e servindo de fonte de ataques DDoS, o que contribui ainda mais para a dificuldade de se encontrar os responsáveis (BELENKY; ANSARI, 2003).

Somando-se às máquinas *zombies*, os organizadores do ataque podem ainda usar outras máquinas que atuam como controladores, chamadas de *handlers*. Estas máquinas *handlers* também, normalmente, pertencem a outras redes e os administradores não têm ciência de que elas foram subvertidas. Os organizadores utilizam estas máquinas como um ponto intermediário entre suas próprias máquinas e as máquinas *zombies*. A ideia é utilizar os *handlers* para controlarem as máquinas *zombies* (SINGH; SINGH; KUMAR, 2016). A Figura 1 apresenta o esquema de um ataque DDoS.

Existem duas frentes de pesquisa para lidar com o problema de endereços IPs forjados, rastreamento e filtragem (SHUE; GUPTA; DAVY, 2008). O rastreamento utiliza técnicas que, de alguma forma, coletam e/ou inserem informações nos pacotes que trafe-

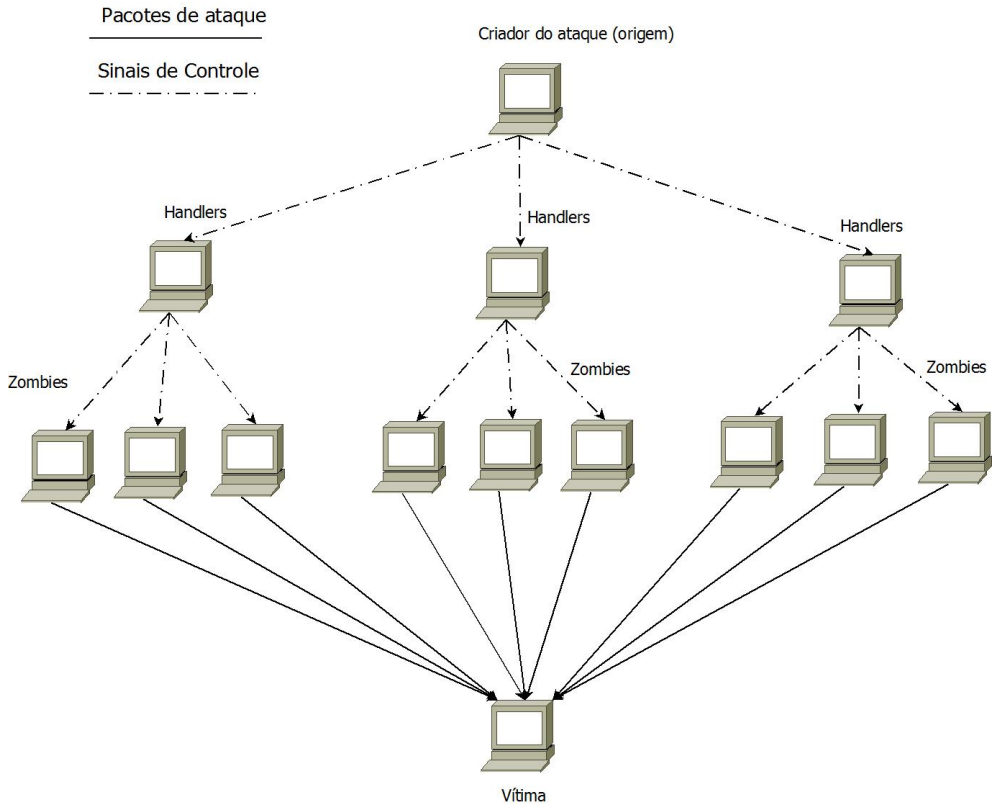

Figura 1: Esquema de ataque DDoS, adaptado de (SINGH; SINGH; KUMAR, 2016)

gam pela rede. A vítima deve utilizar estas informações para descobrir a origem do pacote. A filtragem pode também coletar ou inserir informações no pacote. Porém, as informações inseridas pelos roteadores nos pacotes são analisadas pelos próprios roteadores ao longo do percurso do pacote. O objetivo destas marcas em esquemas de filtragem permite detectar que o pacote está forjado e, portanto, é um pacote suspeito. Enquanto o rastreio busca identificar a origem, a filtragem busca eliminar pacotes antes destes chegarem no destino.

O rastreamento não tem a função de impedir o ataque de acontecer. Portanto, os possíveis danos de um ataque não serão minimizados. Apesar da filtragem tentar impedir de fato o ataque no momento em que está acontecendo, ela acaba ignorando a origem deste. Portanto, os causadores nunca serão descobertos e punidos. Além disso, a filtragem requer grande cooperação entre SAs (Sistemas Autônomos). Pacotes legítimos podem acabar sendo descartados neste processo (SHUE; GUPTA; DAVY, 2008).

Neste trabalho, é considerado que o responsável por um ataque cibernético deve ser identificado e punido. Pois, caso não seja descoberto, nada o impedirá de orquestrar novos ataques, ocupando recursos da rede com tráfego malicioso e causando prejuízos a sua vítima. Porém, se existir uma técnica de rastreio que detecte a origem, o responsável será descoberto. Além disso, dado que um sistema de rastreamento está vigente, o ataque pode nem mesmo acontecer, uma vez que não é interessante para o atacante ser descoberto.

Esta dissertação propõe uma nova técnica de rastreamento de pacotes IP. A marca-

ção dos pacotes é feita de forma determinística, envolvendo apenas os roteadores de borda. Para garantir a veracidade da marca nos pacotes, é utilizado um esquema de criptografia simétrica. O pacote IP carrega as marcas de todos os SAs que estão no caminho entre a origem e o destino. O restante desta seção apresenta quais foram as métricas utilizadas para se avaliar o CTPM (*Chain of Trust Packet Marking)*, e apresenta também como os esquemas de rastreamento de pacotes IPs podem ser classificados.

1.1 Classificação de esquemas de rastreamento

Os esquemas de rastreamento podem ser divididos em algumas categorias de acordo com Singh, Singh e Kumar (2016) e Belenky e Ansari (2003). A diferença fundamental entre as categorias baseia-se na forma como as informações de rastreio são coletadas ao longo do percurso do pacote IP. A seguir apresentam-se as categorias existentes.

1.1.1 Teste de enlace

Esta categoria de rastreamento usa o fato de que as rotas carregando pacotes de ataque estão, normalmente, sobrecarregadas. O teste de enlace se divide em outras duas subcategorias. Na primeira, usa-se uma amostra do pacote de ataque e inicia-se um trabalho de descoberta da rota de ataque a partir do roteador mais próximo da vítima. Todo roteador é capaz de saber por qual interface determinado pacote ingressou nele. Assim, ele consegue descobrir qual roteador anterior (*upstream router*) enviou aquele pacote. Este trabalho segue recursivamente nos *upstream routers* até que se descubra o primeiro roteador por onde o ataque passou.

Na segunda subcategoria, gera-se um tráfego grande de pacotes (uma espécide de DoS) e injeta sequencialmente nos enlaces que chegam até o roteador mais próximo da vítima. Quando este tráfego gerado é injetado no enlace por onde o ataque (que se deseja rastrear) está chegando, a vítima percebe que houve um declínio de pacotes maliciosos, pois agora o enlace foi sobrecarregado deliberadamente. Após detectado este resultado, conclui-se que esta rota é aquela por onde os pacotes de ataque estão chegando. Este trabalho é feito em todos os *upstream routers* até que se chegue no primeiro por onde passa o ataque.

No teste de enlace como um todo, é necessária forte cooperação dos ISPs para executarem este trabalho sobre os roteadores. Além disso, ela é inviável para ataques DDoS, visto que usam milhares de rotas.

1.1.2 Envio de Mensagens

Nesta categoria, todo roteador pode selecionar aleatoriamente, um pacote, e gerar uma mensagem ICMP (*Internet Control Message Protocol)* correspondente para o destino do pacote. Esta mensagem ICMP contém informações como, próximo salto do pacote, salto anterior do pacote, *timestamp* entre outros, e naturalmente, alguma informação do próprio pacote. Assim, no processo de rastreamento, deve-se identificar todas as mensagens ICMP pertencentes aos pacotes de ataque. Tendo as mensagens ICMP, é possível reconstruir o caminho do pacote.

O Envio de Mensagens gera um tráfego adicional pela rede, que deve ser controlado, para não trazer grande sobrecarga de banda. Necessita-se, também, de grandes quantidades de mensagens ICMP para ser possível reconstruir o caminho, e assim, é inviável para os padrões de ataques DDoS existentes.

1.1.3 Marcação

Na marcação, o roteador insere alguma informação no próprio pacote em questão. Esta informação pode ser referente ao roteador atual ou ao enlace (roteador atual e próximo salto). A marcação pode ser feita probabilisticamente ou deterministicamente. Neste trabalho, o esquema de rastreamento é baseado em marcação de pacotes de forma determinística, usando informações do próprio roteador.

A marcação de pacote tem como maior desafio otimizar a informação que deve ser marcada nos pacotes. No IPv4, existe um problema de espaço do cabeçalho IP, impedindo que informações precisas sejam salvas, tanto de identificação como de validação. Com o advento do IPv6 e seu conceito de cabeçalho de extensão, a marcação de pacotes se apresenta como um solução robusta e confiável para se detectar a origem de pacotes IPv6.

1.1.4 Logging

No *Logging*, cada roteador deve armazenar um resumo (*digest*) dos pacotes que encaminha. Assim, quando uma vítima detecta que está sob ataque, ela envia o pacote de ataque para os administradores dos *upstream routers*, para que detectem se aquele pacote passou por seus roteadores. Depois de detectados os roteadores que registraram um *digest* daquele pacote, monta-se o caminho percorrido por este. Novamente, há um grande envolvimento dos ISPs.

1.1.5 Overlay

Nesta categoria, é necessário existir um *hardware* adicional coordenando o tráfego dentro do ISP. Esse dispositivo se comporta como uma unidade central e todo o tráfego

passa por ele. Sendo detectado um ataque, a vítima envia para essa unidade central o pacote de ataque, e este dirá de qual enlace veio o ataque. Esse processo segue sucessivamente até que se atinja a origem do pacote. No *Overlay*, existe grande envolvimento dos ISPs durante o processo de reconstrução, além da exigência de adequarem sua topologia para incorporarem as unidades centrais.

1.2 Objetivos da Dissertação

Esta pesquisa tem como objetivo propor e implementar um esquema de marcação determinístico de pacotes IP para realizar o rastreamento de pacotes IPv6. O esquema será determinístico, pois, como será visto, os esquemas probabilísticos necessitam de muitos pacotes para detectar a origem, e com isso, aumentam a complexidade da reconstrução do caminho. No esquema que será proposto, basta um pacote IP para detectar a origem. Entende-se por origem do pacote IP, o SA (Sistema Autônomo) responsável por inserir este pacote na Internet. O esquema permite detectar qual foi o SA de origem do pacote. Cabe ao administrador do SA identificar qual de seus clientes está usando os recursos do SA para propagar pacotes com conteúdo malicioso.

Nesta proposta, as marcas inseridas no pacote são criptografadas para a integridade e segurança destas. Observa-se na comunidade e nos trabalhos utilizados como referência (apresentados no Capítulo 3) grande resistência em utilizar criptografia devido aos custos computacionais. Porém, nenhum trabalho fez a implementação para de fato verificar o custo exigido pela criptografia. Assim, nesta pesquisa também foram investigados esses custos envolvidos a fim de informar os operadores de SAs qual o investimento necessário para implementar a proposta. São medidos os custos de CPU, memória e a degradação da largura de banda.

1.3 Estrutura deste Trabalho

O restante deste trabalho é divido da seguinte forma: O capítulo 2 apresenta a fundamentação teórica e conceitos básicos para entender as tecnologias envolvidas na implementação deste trabalho. O Capítulo 3 traz uma revisão bibliográfica, onde apresenta demais trabalhos que serviram de inspiração para este. O capítulo quatro apresenta detalhadamente o CTPM, e o funcionamento completo do esquema de rastreamento. E finalmente o capítulo 5 traz a descrição dos experimentos e os resultados obtidos.

2 Revisão da Teoria

Este capítulo tem por objetivo fazer uma breve apresentação dos conceitos utilizados ao longo do trabalho, sem os quais não é possível a compreensão total da pesquisa desenvolvida. São apresentados conceitos sobre o IPv6, *Linux Kernel Module*, *Netfilter*, *Linux Kernel Crypto* e ao final do capítulo são apresentadas as métricas que foram utilizadas para avaliar a proposta apresentada.

2.1 IPv6

O IPv6 é um protocolo da camada de rede desenvolvido para suceder o IPv4. Dadas as proporções que a Internet alcançou no mundo, com mais de 3 bilhões de usuários, não é possível simplesmente desligar todos os equipamentos que suportam IPv4 e ligar os equipamentos que suportam IPv6. A transição deve ser gradativa e, portanto, as duas tecnologias devem coexistir durante esse tempo. Para entender a necessidade do IPv6, deve-se levar em consideração conceitos que existiam no início da Internet. Quando o IPv4 foi implantado, em 1983, a ideia era facilitar a troca de mensagens entre algumas Empresas e Universidades Americanas. Nenhum dos seus criadores imaginava o aumento explosivo de usuários na Internet ao longo dos anos seguintes. Não era previsto que ela se tornasse um meio, quase essencial, para realizar as mais diversas atividades, como transferências bancárias, pagamentos, assistir filmes, ouvir músicas, fazer compras e muitas outras. Assim, alguns aspectos de segurança e escalabilidade não foram levados em consideração. Ao longo dos anos, foram necessários alguns ajustes no IPv4 para que este se adaptasse melhor à crescente demanda de usuários e à forma como eles passaram a usar a Internet. O NAT (*Network Address Translation*) é um exemplo desses ajustes. Ele foi desenvolvido para minimizar o impacto da crescente escassez de endereços IPv4 e aumentar também a segurança. No entanto, é bom ressaltar que o IPv4 se mostrou robusto e flexível para ir se adaptando às demandas ao longo dessas décadas, mas já não suporta mais adaptações (HAGEN, 2014).

Quando foram feitas as primeiras distribuições de blocos de endereços IPv4, as Universidades e Empresas americanas adquiriram grande parte destes blocos, deixando o restante do mundo com os endereços que sobraram. Ao longo do tempo, esse procedimento mostrou-se ineficiente e contribuiu para o esgotamento de endereços IPv4 (que ocorreu em 2011) antes do previsto. Como resultado, tem-se regiões com uma quantidade de endereços maior do que precisam, e outras com escassez de endereços. Não sendo possível fazer uma relocação destes blocos, a saída é a transição para o IPv6.

A primeira característica do IPv6 é seu espaço de endereçamento. O IPv4 pos-

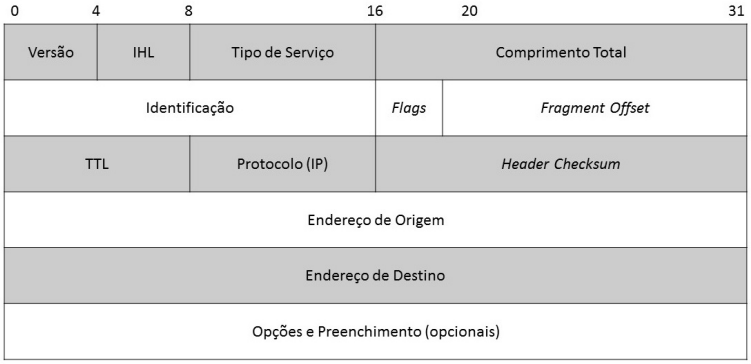

Figura 2: Cabeçalho IPv4 adaptado de (COMER, 2014)

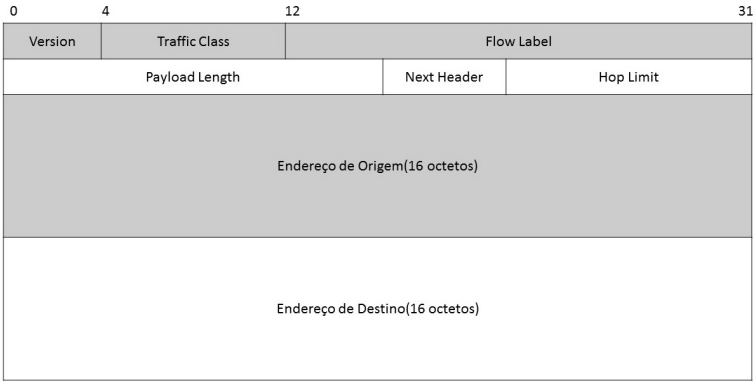

Figura 3: Cabeçalho IPv6 Fixo, adaptado de (COMER, 2014)

sui um endereço com tamanho de 32bits, possibilitando aproximadamente 4,3 bilhões de endereços diferentes. No IPv6, o endereço tem um tamanho de 128bits. Sendo difícil escrever este número por extenso, uma comparação mostra que seria possível fornecer vários endereços IPv6 para cada grão de areia existente na terra. Eletrônicos como geladeira, fogão, chuveiro e outros dispositivos já estão se conectando à Internet, e por isso precisam de um endereço IP. Mesmo que a distribuição de endereços IPv4 tivesse sido mais eficiente no início, ela não iria permitir acomodar a expansão destes novos equipamentos. Esta é uma das razões de migração para o IPv6, uma vez que tem capacidade de atender todo esse crescimento de dispositivos conectados a Internet. As Figuras 2 e 3 apresentam, respectivamente, o cabeçalho do IPv4 e do IPv6.

O IPv6 traz um cabeçalho simplificado se comparado ao IPv4. Alguns campos foram removidos tornando-o mais compacto e facilitando eventuais serviços administrativos. O campo *checksum* foi removido, pois agora considera-se que outras camadas são responsáveis por essa verificação, não mais a camada de rede. Este campo servia como uma verificação de integridade do pacote, pois caso algum bit tivesse sido corrompido, o cálculo de *checksum* não iria casar com o valor informado. Neste caso, o pacote era descartado. Esta remoção permite um aumento de desempenho, pois no IPv4, a cada nó

intermediário[1], é necessário este cálculo.

No IPv6 a fragmentação ocorre de forma diferente. No IPv4, o roteador fragmenta este pacote em outros pacotes cujos tamanhos sejam suportados neste enlace e os transmite. No destino, todos os pacotes têm de ser recebidos e remontados para recuperar o pacote original. Caso uma parte chegue com erro ou tenha sido perdida, é necessária a retransmissão completa de todas as partes. No IPv6, os roteadores não fazem mais fragmentação. É de responsabilidade do *host* gerador do pacote verificar qual o tamanho máximo suportado pelo enlance e fragmentar o pacote, se necessário. A este tamanho máximo dá-se o nome de MTU (*Maximum Transmission Unit*). Se o pacote for fragmentado, um cabeçalho de extensão é utilizado para a identificação dos fragmentos.

Os cabeçalhos de extensão são inseridos entre o cabeçalho IPv6 e o *payload*. Estes cabeçalhos não são obrigatórios. Assim, não havendo necessidade de utilizá-los, o processamento de um pacote IPv6 se torna bastante rápido. Os cabeçalhos de extensão carregam informações que podem ser processadas apenas no destino final ou em cada roteador intermediário. Cada pacote IPv6 pode ter 1, mais de 1, ou nenhum cabeçalho de extensão. Caso exista mais de um cabeçalho de extensão, eles são inseridos em sequência um logo após o outro. Foram definidos seis cabeçalhos de extensão. São eles, *Hop-by-hop, Destinations, Router, Authentication, Encapsulating Security Payload* e *Fragment Header*. Para reduzir gastos computacionais nos roteadores, todos estes cabeçalhos de extensão são processados apenas no destino ou nos roteadores intermediários especificados pelo *Routing Header*. O *Hop-by-hop* é uma exceção, pois ele é processado por todos os roteadores intermediários (DEERING; HIDEN, 1998).

O cabeçalho *Hop-by-Hop* carrega informações, ou opções, que devem ser processadas por todos os roteadores entre a origem e o destino. Este cabeçalho de extensão, caso exista no pacote IPv6, deve obrigatoriamente estar logo depois do cabeçalho IPv6. Dentre algumas opções usadas pelo *Hop-By-Hop*, tem-se o *Router Alert* e o *Jumbogram*. *Router Alert* são mensagens que devem ser processadas pelos roteadores, como exemplo o MLD (*Multicast Listener Discovery*). Estas são mensagens enviadas para detectar nós que desejam participar de grupos *multicast* e também qual endereço *multicast* interessa a esses nós receberem. Estes pacotes que contém a opção *Router Alert* não são pacotes de dados, são pacotes de controle. A opção *Jumbogram* é uma opção que indica que o pacote IPv6 está carregando um *payload* acima de 65.536 bytes e menor que 4.294.967.295 bytes.

O *Routing Header* é usado para indicar uma lista de roteadores intermediários que devem ser visitados pelo pacote IPv6 antes deste alcançar o destino. O *Destinations Header* é utilizado para carregar informações que devem ser processadas apenas no destino. Sabendo que todo pacote IPv6 pode ter mais de um cabeçalho de extensão, o *Destinations Header* pode aparecer em três situações distintas. Na primeira, existe apenas um

[1] Neste trabalho, nó intermediário ou dispositivo intermediário se refere a roteador.

cabeçalho *Destinations Header* no pacote e ele é o último na sequência de cabeçalhos de extensão existentes dentro do pacote IPv6. Neste caso, o conteúdo do *Destinations Header* é analisado apenas no destino. No segundo caso, existe um *Destinations Header* seguido de um *Routing Header* no pacote, e este *Routing Header* está especificando alguns roteadores que o pacote deve visitar antes do destino final. Como o *Destinations Header* se encontra antes do *Routing Header*, todos esses nós que devem ser visitados também devem processar o conteúdo encontrado no *Destinations Header*. No terceiro caso, existem dois cabeçalhos *Destinations Header*, um antes e outro depois do cabeçalho *Routing Header*. Os dados contidos no *Destinations Header* presentes antes do *Routing Header* devem ser analisados por todos os roteadores intermediários indicados pelo *Routing Header*, e os dados contidos no *Destinations Header* que se encontram após o *Routing Header* devem ser processados apenas no destino (DEERING; HIDEN, 1998).

O *Fragment Header*, como o próprio nome sugere, é responsável por armazenar as informações referentes à fragmentação do pacote IPv6. Como dito anteriormente, no IPv6, os roteadores não estão autorizados a realizar a fragmentação, cabendo esta tarefa ao dispositivo que cria o pacote. O IPv6 determina uma MTU mínima de 1280Bytes, ou seja, qualquer enlace que deseja trafegar dados IPv6 deve suportar, pelo menos, uma MTU de 1280Bytes. Porém, alguns enlaces podem suportar MTUs maiores, e os dispositivos podem se aproveitar deste valor para enviarem pacotes com tamanho superior à MTU mínima do IPv6. Para isso, usa-se o *Path Discovery*, onde um pacote é enviado da origem até o destino para descobrir qual a MTU deste enlace. Assim, o dispositivo que cria o pacote pode utilizar um valor superior a 1280Bytes, caso o enlace suporte. Se o dispositivo criador do pacote detectar que será necessário fragmentar, ele utiliza o cabeçalho de extensão *Fragment Header* para por informações necessárias para que o destino identifique e remonte os fragmentos para formar o pacote original. Caso um roteador IPv6 tenha de encaminhar um pacote cuja MTU é superior ao suportado pelo próximo enlace, o pacote será descartado e uma mensagem "*Packet Too Big*" é criada pelo roteador e enviada a quem criou este pacote IPv6.

Os cabeçalhos *Authentication* e ESP (*Encapsulation Security Payload*) estão relacionados com segurança e autenticação do pacote IPv6. O *Authentication* provê integridade dos dados, autenticação da origem e proteção contra pacotes repetidos. Esta proteção previne que pacotes capturados por usuários mal-intencionados sejam retransmitidos e considerados dados válidos. O ESP fornece também confidencialidade, autenticação da origem e proteção contra pacotes capturados, porém, ele provê segurança para o *payload* do pacote IPv6. Em contraste, o *Authentication* provê a segurança para o pacote IPv6 como um todo. Assim, ambos estes cabeçalhos podem ser utilizados em conjunto ou separados.

A especificação do IPv6 deixa aberta a possibilidade da criação de novos cabeçalhos

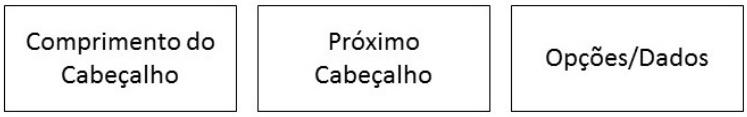

Figura 4: Cabeçalho de Extensão IPv6 Genérico, adaptado de (HAGEN, 2014)

de extensão, sendo esta uma das formas do IPv6 se adaptar a futuras modificações. Nesta dissertação, é proposta a criação de um novo cabeçalho de extensão, chamado de *Traceback Extension Header* (TEH), que tem a função de marcar, no pacote IPv6, o caminho por onde ele está trafegando. No Capítulo 4, é dada mais atenção à proposta deste novo cabeçalho de extensão. Existem algumas regras para se propor um novo cabeçalho de extensão. A primeira diz que nenhum cabeçalho com comportamento do *Hop-by-Hop* deve ser criado, isto é, um cabeçalho que deve ser processado por todos os roteadores entre origem e destino. Se necessário, deve-se propor uma nova opção para ser utilizada pelo já existente *Hop-By-Hop*. A ideia desta restrição é evitar gasto computacional nos roteadores. A segunda restrição diz para evitar a criação de qualquer cabeçalho, priorizando propor novas opções para o *Destinations Header*. Caso seja necessário, deve-se provar o porquê de um novo cabeçalho. Existe uma estrutura básica que todo cabeçalho deve seguir, ilustrado na Figura 4 e detalhado a seguir:

- Comprimento do Cabeçalho(*Header Length*): Campo de 1Byte que indica o tamanho total do cabeçalho de extensão, incluindo os campos *Header Length* e *Next Header*. Este campo é necessário, pois as opções que são inseridas nos cabeçalhos de extensão possuem tamanhos variáveis.

- Próximo Cabeçalho(*Next Header*): Campo de 1Byte que identifica qual é o cabeçalho de extensão que está logo em seguida. Caso este pacote IPv6 não tenha mais cabeçalhos de extensão, o campo *Next Header* informa qual o protocolo da camada superior está presente no *payload* do pacote IPv6, como por exemplo, TCP.

- Dados(*Options*): Contém os dados a serem carregados por este cabeçalho de extensão, por isso é de tamanho variável.

A exceção do *Hop-By-Hop*, que deve ser obrigatoriamente o primeiro cabeçalho de extensão logo após o cabeçalho fixo IPv6, a documentação do IPv6 não exige que os cabeçalhos de extensão sejam usados em uma ordem obrigatória, sendo os dispositivos geradores de pacotes IPv6 livres para inserirem os demais cabeçalhos de acordo com sua conveniência. Os cabeçalhos de extensão são processados sequencialmente. Portanto, não é possível a um roteador pular cabeçalhos até chegar no desejado, pois ele precisa ler o campo *Next Header* de cada cabeçalho para saber qual o próximo. Por este motivo, existe uma ordem de inserção dos cabeçalhos de extensão proposta, mas não obrigatória, pela

documentação oficial do IPv6, apresentada a seguir (DEERING; HIDEN, 1998; HAGEN, 2014):

- Cabeçalho IPv6 Fixo: sempre existe;
- *Hop-By-Hop*: obrigatório ser o primeiro a seguir o cabeçalho fixo, caso exista;
- *Destinations Header*: para dados a serem processados por roteadores listados pelo Routing Header;
- *Routing Header*
- *Fragment Header*
- *Authentication Header*
- *Encapsulating Security Payload*
- *Destinations Header*: para dados a serem processados apenas no destino final;

O IPv6 representa um grande avanço em relação ao IPv4, sendo um protocolo robusto, confiável, escalável, que permite uma transição suave do IPv4, e que se adapta a futuras mudanças que podem ocorrer na Internet. Está fora do escopo desta dissertação comentar sobre todas as características do IPv6. Para isso, recomenda-se a leitura de Hagen (2014), que fornece uma excelente abordagem sobre o IPv6.

2.2 Linux Kernel Module

Quando deseja implementar alguma modificação no *kernel* do Linux, o programador deve baixar o código fonte do *kernel* mais recente, fazer todas as modificações que deseja, compilar o código, instalar e reiniciar a máquina para que suas modificações tenham efeito. Este procedimento deve ser executado toda vez que ele deseja testar alguma nova modificação, ou seja, compilar e reiniciar a máquina. Esta repetição pode tornar o trabalho de desenvolvimento e *debug* improdutivo ou até mesmo cansativo.

Outra alternativa para fazer modificações no *kernel* é fazê-las em um módulo do *kernel* (*Linux Kernel Module* - LKM) (SALZMAN; BURIAN; POMERANTZ, 2007). LKMs são carregados e removidos sob demanda do usuário ou automaticamente pelo próprio *kernel*. Os módulos são códigos escritos na linguagem C que estendem o *kernel*. A ideia é utilizar o módulo para adicionar uma nova funcionalidade ao *kernel*, e a partir do momento que este módulo é carregado, a funcionalidade passa a fazer parte do *kernel*. Portanto, este código executa no espaço do *kernel*, em contraste com códigos que executam no espaço de usuário. O Linux já possui uma série de módulos em seu *kernel*, sendo que os mais comuns são *device drivers* que permitem que o sistema operacional se comunique

com algum dispositivo conectado ao computador. O próprio fabricante do dispositivo pode, também, fornecer o módulo de *kernel* que contém o *device driver* necessário. Para visualizar quais módulos estão carregados, basta digitar `lsmod` no terminal de comandos.

Outra tarefa comumente implementada em um LKM é a filtragem de pacotes IP, semelhante a um *firewall*. Através do desenvolvimento de um módulo de *kernel*, é possível capturar pacotes, fazer a análise, descartar o pacote ou encaminhá-lo para seguir o fluxo normal na pilha TCP/IP. É possível também fazer modificações nos pacotes, inserir dados e remover dados caso seja necessário. Tudo isso pode ser feito sem a necessidade de modificar o código diretamente no *kernel*, mas simplesmente desenvolvendo um módulo de *kernel* para esta função. Como exemplo, durante o desenvolvimento desta dissertação, foi criado um LKM que insere um novo cabeçalho de extensão em cada pacote IPv6. Ao longo deste trabalho, este LKM específico é discutido em detalhes (SALZMAN; BURIAN; POMERANTZ, 2007).

Normalmente, um programa em C inicia-se com a chamada da função *main*, executa todas as suas instruções, encerra e é removido da memória. O módulo de *kernel* funciona de forma um pouco diferente. Ele é carregado no *kernel* e fica à disposição para ser invocado e realizar sua tarefa a qualquer momento em que o *kernel* o requerer. Depois que o *kernel* o utiliza, ele não é removido, pois permanece carregado à espera de uma nova requisição. Ele só é removido se o usuário explicitamente emitir o comando para tal, ou se houver algum *script* executando no *kernel* que solicite a remoção do módulo.

Analogamente à função *main* de um programa, todo módulo também possui um ponto de entrada. Este ponto é a função *init*, que é executada uma única vez e somente no momento em que o módulo é carregado no *kernel*. Esta função serve para apresentar ao *kernel* qual tarefa este módulo executa, registrando qual função e em qual situação o *kernel* deve chamar este módulo. Depois de carregado, o módulo fica à espera da sua invocação pelo *kernel*. Todo módulo também tem obrigatoriamente um ponto de saída que é executado uma única vez quando o módulo é removido do *kernel*. Este ponto é a função *exit*. Ela desfaz o que a *init* fez, cancelando qualquer função registrada. Em outras palavras, é como se ela dissesse para o *kernel* que não está mais disponível para ser executada.

Depois que o programador escreveu o módulo, basta compilá-lo e carregá-lo. Deve-se observar que não é necessário compilar o *kernel* todo, apenas o módulo. Também não é necessário reiniciar a máquina para que o *kernel* passe a utilizar este módulo. No momento em que ele é carregado, imediatamente está disponível para ser executado. Também pode-se remover o módulo a qualquer momento, desde que não esteja sendo utilizado. Fica bastante evidente as suas vantagens, principalmente com relação a *debug*. O programador faz sua modificação, compila e carrega o módulo. Em seguida, observa o comportamento e, se algo não executar como esperado, pode remover o módulo, fazer as modificações

no código, compilá-lo e carregar novamente. Este procedimento é bem mais rápido se comparado ao processo de compilar todo o código fonte do *kernel* e reiniciar a máquina. Este módulo executa no espaço do *kernel* e, portanto, qualquer falha no código pode resultar em *kernel panic* ou mesmo travamento total da máquina.

Como exemplo, a Figura 5 apresenta um módulo bem simples. Segue-se a análise deste código.

```
/*LKM mais simples possível*/
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
static int __init ola_init(void){
    printk("Olá Mundo do Kernel");
    return 0;
}
static void __exit adeus_exit(void){
    printk("Adeus Mundo do Kernel");
}
module_init(ola_init);
module_exit(adeus_exit);
```

Figura 5: Módulo de Kernel do Linux, adaptado de (SALZMAN; BURIAN; POMERANTZ, 2007)

Os pontos de entrada e saída do módulo são definidos pelos macros `module_init()` e `module_exit()` respectivamente. Estes recebem como argumento o nome das duas funções que serão usadas para inicializar e terminar o módulo. No código exemplo, a função de entrada é a `ola_init` e a função de saída é a `adeus_exit`. Um detalhe importante é que essas duas funções devem ser declaradas antes dos macros `module_init` e `module_exit`, caso contrário, ocorrerá um erro de compilação. Neste exemplo, quando o módulo é carregado, a função `printk()` imprime *Olá Mundo do Kernel* no arquivo de *log*, e quando o módulo é removido a função `printk()` imprime *Adeus Mundo do Kernel* no *log*.

Como já mencionado, módulos de *kernel* executam no espaço de *kernel*. Assim não têm acesso às bibliotecas presentes no espaço de usuário. Por exemplo, a função `printf()` não está disponível e, em seu lugar, usa-se a função `printk()`, que funciona de forma bastante semelhante. Diferentemente do `printf()`, `printk()` não tem o objetivo de imprimir mensagens para o usuário e sim mensagens de *log* para *debug* dos módulos. Ele possui um argumento opcional de prioridade da mensagem, ou nível de *log*. O *kernel* oferece macros para facilitar o seu uso. No código exemplo, é usado a macro `KERN_INFO`, que indica que a mensagem é uma simples informação. Tem-se macros para mensagens de erro, alertas, entre outras, num total de oito níveis diferentes de prioridade. Se nenhuma prioridade for usada, é considerada a `KERN_WARN`, que é um aviso. Digitando

dmesg no terminal é possível ver o que está sendo impresso por printk(). Esta função foi a principal ferramenta de *debug* utilizada no desenvolvimento deste trabalho.

Outra macro importante é a __init e __exit. A primeira é utilizada na função que é executada quando o módulo é carregado. Ela diz para o compilador que esta função pode ser removida da memória depois de executada. É uma técnica para economizar memória, pois a função de entrada é executada uma única vez. Logo, não tem porque mantê-la na memória. A macro __exit, utilizada na função executada quando o módulo é removido, indica que esta função só deve ser carregada quando for solicitada a remoção do módulo.

Para compilar este código exemplo, cria-se, primeiro, o arquivo *Makefile* apresentado na Figura 6.

```
obj-m += nome_do_modulo.o
all:

    make -C /lib/modules/$(shell uname -r)/build M=$ (PWD)
modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$ (PWD)
clean
```

Figura 6: Makefile, adaptado de (SALZMAN; BURIAN; POMERANTZ, 2007)

A palavra-chave make deve estar a exatamente um TAB de distância do início da linha. Este arquivo deve estar no mesmo diretório que o módulo. No terminal, digita-se make e o módulo é compilado e está pronto para ser inserido. O arquivo nome_do_modulo.ko é criado e deve ser carregado usando o comando insmod nome_do_modulo.ko no terminal. Imediatamente após carregado, o módulo já passa a se comportar como parte do *kernel*. Sua função de entrada já foi executada. Assim, digitando dmesg no terminal, já é possível visualizar a mensagem "Olá Mundo do Kernel". Como este módulo não realiza nenhuma outra tarefa, nada mais acontecerá. Para removê-lo, basta digitar no terminal rmmod nome_do_modulo.ko. Antes de ser removido pelo sistema, a função de saída é executada e, digitando dmesg no terminal, aparece a mensagem "Adeus mundo do Kernel".

É importante ressaltar que o arquivo "*Linux-headers*" deve estar instalado no *kernel* que se deseja compilar o módulo. A ausência deste impede que o módulo seja compilado. Para instalar este arquivo, no Ubuntu ou Debian, basta digitar no terminal apt-get install linux-header-x.y, onde x.y é a versão do *kernel* executando no computador que se deseja compilar o módulo.

Existem algumas práticas para se evitarem erros na escrita do módulo. Por exem-

plo, para alocar-se memória no espaço de usuário, usa-se `malloc(size_t)`. No espaço de *kernel*, o comando é semelhante, `kmalloc(size_t, y)`, onde y pode assumir o valor `GFP_ATOMIC` ou `GFP_KERNEL`. Dependendo do lugar onde uma função do módulo seja executada, usar o valor `GFP_KERNEL` pode causar um *kernel panic*, pois este pode iniciar o trabalho de alocação e, logo em seguida, sofrer uma preempção. No caso de se usar LKM juntamente com *Netfilter* (descrito na seção a seguir), deve-se usar `GFP_ATOMIC`, pois o *Netfilter* funciona como uma interrupção (LOVE, 2010).

Algumas experiências adquiridas neste trabalho mostram que é interessante alocar memória somente quando o módulo for carregado, ou seja, na função chamada pela macro `module_init`. Não é vedado alocar memória durante a execução do módulo, mas dependendo da quantidade de vezes que isso ocorrer, um *kernel panic* pode acontecer. Assim, para este trabalho, a técnica foi alocar memória somente no carregamento do módulo. Durante a execução da tarefa do módulo, esta área de memória é reaproveitada. Quando o módulo for removido, a função `__exit` pode se encarregar de liberar a memória. No espaço de *kernel*, usa-se a função `kfree()` para isto (LOVE, 2010).

Muitas das bibliotecas normalmente utilizadas em programas C, no espaço de usuário, tiveram uma versão implementada também no espaço de *kernel*. Como exemplo, a biblioteca *string.h* pode ser usada em programação de módulos também, mas incluindo `linux/string.h`. Sendo muito vasta a quantidade de bibliotecas que tiveram uma versão implementada no espaço de *kernel*, faz-se necessário uma busca em base de dados *Linux*, como Electrons (2015), para localizar se há uma implementação da biblioteca desejada no *kernel* e, se existir, como ela deve ser incluída.

Trabalhar com módulos permite focar exclusivamente na nova tarefa que se quer estender no kernel, sem precisar ficar compilando tudo e tendo de reiniciar a máquina a cada modificação que se deseja fazer. Não é necessária uma imersão no código fonte do *kernel* para se implementar a modificação desejada. Basta escrever o módulo, compilá-lo e carregá-lo no *kernel*. Faz-se os testes, observações, e análise de *log*. Para continuar desenvolvendo, basta escrever as novas linhas de código, remover o módulo, compilá-lo e carregá-lo novamente. As mudanças são imediatas (SALZMAN; BURIAN; POMERANTZ, 2007).

Dentre as desvantagens de se utilizar módulos de *kernel* ao invés de modificar o código fonte do *kernel* diretamente, cita-se a possível fragmentação que pode ocorrer na memória, introduzindo uma leve perda de desempenho. Porém, os resultados obtidos neste trabalho, mencionados no Capítulo 5 (Experimentos e Resultados), mostram que utilizar o LKM não introduz praticamente nenhum impacto computacional.

2.3 Netfilter

Netfilter é um *framework* dentro do *kernel* do Linux que fornece pontos de acesso (ganchos) para filtragem e/ou modificação de pacotes da pilha de protocolos TCP/IP. Para o protocolo IP, o *Netfilter* disponibiliza cinco ganchos distribuídos ao longo do trajeto que um pacote IP percorre dentro da camada IP. Os módulos de *kernel* podem registrar funções que se conectam a estes ganchos. Toda vez que um pacote atravessar a camada IP dentro do *kernel* e este pacote atingir um destes ganchos, o *kernel* verifica se existem módulos de *kernel* conectados neste gancho. Se não houver nenhum módulo, o pacote prossegue o seu fluxo de execução normal. Caso exista mais de um módulo carregado, sua prioridade é verificada. Cada programador define explicitamente a prioridade do seu módulo. Depois de verificado qual módulo tem a maior prioridade, o pacote IP é desviado para este módulo de *kernel*. Este módulo passa a ter total controle do pacote IP e pode manipulá-lo completamente, inclusive descartá-lo (RUSSEL; WELTE, 2002).

Se o módulo não descartar o pacote IP, ele o devolve para a camada IP do *kernel*. Assim, se existir outro módulo conectado ao gancho, este também tem a oportunidade de manipular o pacote IP. Este procedimento se repete até que não haja mais módulos conectados ao gancho. Assim, o pacote segue o seu fluxo de execução normal, até que outro gancho seja encontrado. Se isto ocorrer, todo o processo inicia-se novamente.

O *iptables* é um exemplo de uma ferramenta conhecida que utiliza o *Netfilter*. Essa ferramenta é utilizada para a criação de regras de *firewall* para o Linux. É muito utilizada por administradores de rede quando estes desejam adicionar alguma segurança e regras aos pacotes trafegando em sua rede.

A Figura 7 apresenta os cinco ganchos disponibilizados pelo *Netfilter* na camada IP. Após ser recebido pela placa de rede, o pacote é verificado pela camada de enlace. Em seguida, a camada IP recebe este pacote e faz alguns testes iniciais para ver se não há dados corrompidos nele. Após estes testes iniciais, o pacote passa pelo primeiro gancho, conhecido como *pre-routing*. Quando o pacote atinge este gancho, ainda não foi analisado o seu destino. Portanto, não se sabe se o pacote é para a máquina em questão ou se deve ser apenas encaminhado. Se houver módulos de *kernel* que tenham registrado funções conectadas a este gancho, o *kernel* desvia este pacote para que ele seja manipulado por todos os módulos que estão registrados, seguindo a ordem de prioridade determinada pelos próprios módulos. Depois dos módulos terem manipulado o pacote IP no primeiro gancho, este retorna ao *kernel* para ser processado na pilha TCP/IP. Neste momento, é verificado o destino do pacote, se é para a própria máquina ou se deve ser encaminhado para outra máquina. Se o destino do pacote for a própria máquina, o fluxo de execução chega no segundo gancho, *input*. Este gancho só é chamado por pacotes que foram destinados à máquina que está analisando o pacote, e novamente o *kernel* desvia o fluxo de execução para módulos que se conectam a este gancho. Após o retorno do pacote ao *kernel*, este

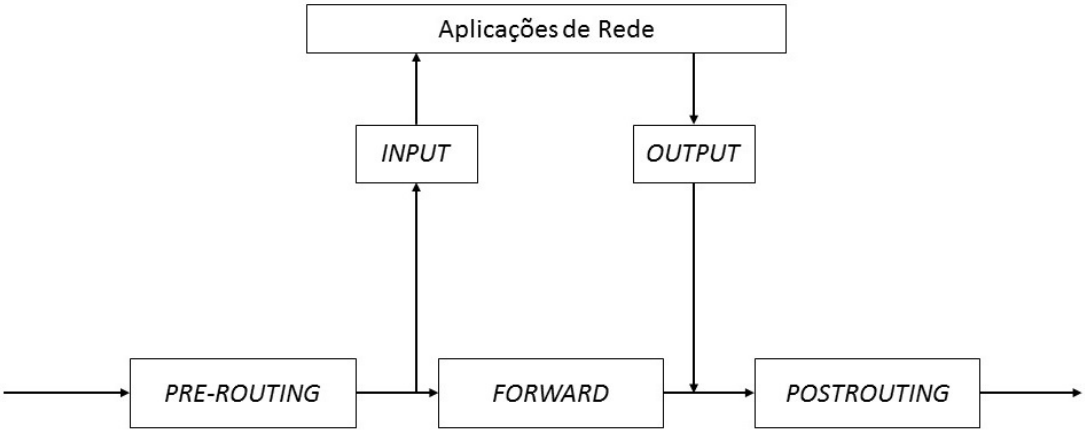

Figura 7: Ganchos(*Hooks*) do Netfilter, adaptado de (YOSHIFUJI, 2007)

segue para ser processado pela camada de transporte, encerrando o uso da camada IP do *kernel*.

Quando um pacote IP chega a um computador que não é o seu destino, ele é encaminhado para o gancho de *forward*, logo após o *pre-routing*. Neste momento, o *kernel* verifica se há módulos conectados e repete o procedimento descrito anteriormente. O pacote retorna ao *kernel* e passa por mais um gancho, o *post-routing*. Este gancho é chamado toda vez que um pacote IP está deixando a máquina, não importa se foi gerado localmente ou se está apenas sendo encaminhado (RUSSEL; WELTE, 2002).

Em se tratando de pacotes gerados localmente, a camada de transporte é responsável por enviar os pacotes para a camade de rede. No instante em que esses pacotes chegam na camada IP, encontram o gancho *output*, e o *kernel* repete o procedimento de desvio do pacote IP para módulos que se conectam a este gancho, caso existam. Em seguida, o pacote retorna ao *kernel* e continua sendo processado normalmente pela camada IP. Antes do pacote ser despachado para a camada de enlace, o gancho *post-routing* é encontrado. Caso existam módulos conectados neste gancho, o *kernel* faz o desvio e, ao retornarem são enviados para a camada de enlace. Neste momento, está encerrada o uso da camada IP dentro do *kernel* do Linux.

O conteúdo apresentado acima não trata os detalhes de funcionamento da pilha TCP/IP dentro do *kernel* do Linux, a ideia é a compreensão dos ganchos *Netfilter* presentes dentro da camada IP do Linux. Recomenda-se a leitura de referências como Benvenuti (2006) ou Rosen (2014) para mais informações sobre a implementação da pilha TCP/IP dentro do *kernel* do Linux.

Para manipular o pacote IP dentro do *kernel* do Linux, ou em algum módulo, é necessário o conhecimento da estrutura `sk_buff`. Esta estrutura é a unidade fundamental da pilha TCP/IP dentro *kernel*. Ela representa o pacote nas camadas de enlace, rede e transporte. A estrutura se modifica dependendo de qual ponto da pilha TCP/IP o pacote

se encontra. Por exemplo, na camada de enlace, o `sk_buff` representa o quadro Ethernet. Depois que esse quadro é desencapsulado e chega na camada de rede, o `sk_buff` passa a representar o datagrama IP e assim por diante. Para facilitar a escrita e a generalização, neste trabalho o `sk_buff` sempre se refere a pacote, não importando em qual camada ele está. Toda vez que um pacote é gerado localmente, a camada de transporte reserva uma área de memória para a estrutura `sk_buff` conter este pacote. De forma análogo, para todo pacote que chega na placa de rede, a camada de enlace faz o processo de alocação de memória para a estrutura `sk_buff` conter o pacote que acaba de ser recebido. Esta estrutura é definida como uma lista duplamente encadeada, facilitando a pesquisa na estrutura quando necessário (ROSEN, 2014).

Dentro da estrutura `sk_buff`, existem quatro variáveis que são ponteiros para diferentes áreas do pacote (`head`, `data`, `tail` e `end`). Dependendo da camada (enlace, rede ou transporte) onde o pacote está sendo processado, alguns destes ponteiros são ajustados para apontarem para novas áreas. Por exemplo, o ponteiro `data` aponta sempre para o início do cabeçalho do protocolo da camada corrente. Isto quer dizer que no momento em que o pacote é manipulado pela camada de rede, dentro do *kernel*, o ponteiro `data` marca o início do cabeçalho IP. Se o pacote está na camada de enlace, `data` indica o início do cabeçalho Ethernet. Conforme o pacote vai transitando pelas camadas dentro do kernel, é de responsabilidade de cada camada atualizar o ponteiro *data* para que erros não ocorram. Os cinco ganchos do *Netfilter* estão na camada IP. Portanto, os módulos que manipulam estes pacotes sempre recebem a estrutura `sk_buff` tendo o ponteiro `data` marcando o início do cabeçalho IP. Tendo acesso a este ponteiro, os módulos de *kernel* podem inserir e/ou remover dados do pacote.

A estrutura `sk_buff` possui três áreas: `headroom`, `data` e `tailroom`. A primeira é uma área reservada para os cabeçalhos dos protocolos presentes no pacote. A segunda contém os dados propriamente ditos e a última área é uma região de cauda. Para criar a estrutura em memória, a camada de transporte usa a função `alloc_skb()`. A estrutura recém-criada se apresenta como ilustrado na Figura 8. Inicialmente, os ponteiros `head`, `data` e `tail` apontam para o início da área de `headroom`, pois ainda não há dados nesta estrutura. O ponteiro `end` aponta para o final da estrutura. Tanto o ponteiro `head` quanto o `end` não mais se alteram, mantendo este valor inicial até que o pacote seja enviado ou descartado. Os ponteiros `data` e `tail` vão sendo ajustados conforme os dados sejam inseridos no pacote e conforme o trânsito do pacote pelas camadas.

Com a estrutura `sk_buff` criada em memória, a camada de transporte reserva um espaço dedicado para o `headroom`, usando a função `skb_reserve()`. A reserva é feita considerando uma área que o *kernel* julga ser suficiente para conter o cabeçalho MAC, cabeçalho IP e o cabeçalho da camada de transporte (TCP ou UDP por exemplo). Após definir a `headroom`, os ponteiros `data` e `tail` são atualizados de acordo com

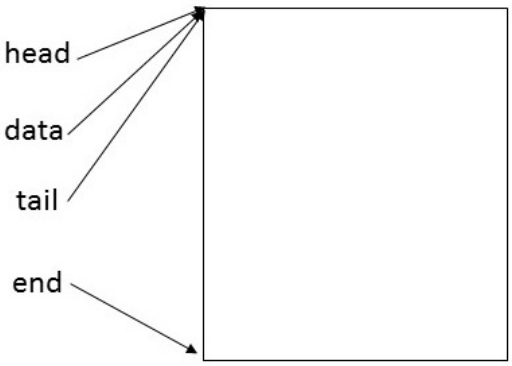

Figura 8: A estrutura SK_BUFF recém criada, adaptado de (YOSHIFUJI, 2007)

o tamanho da área reservada, apontando ambas para o fim da área de `headroom`. A própria função `skb_reserve()` trata de atualizar estes ponteiros. A Figura 9 apresenta a estrutura nesse estágio.

A estrutura já possui a área reservada para os cabeçalhos e então a camada de transporte insere os seus dados nesta área de memória. O ponteiro *data* continua na mesma posição, mas agora apenas ele representa o início da área de dados. O ponteiro `tail` é realocado para o fim da área de dados. Assim, neste estágio, já está bem definida as áreas dentro da estrutura `sk_buff`. O ponteiro `tail` é movido usando a função `skb_put()`, invocado pela própria camada de transporte. O resultado final da estrutura está ilustrado na Figura 10.

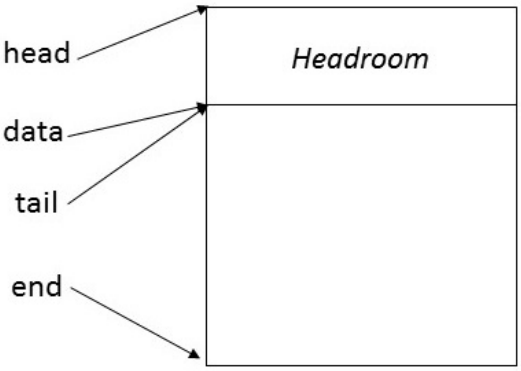

Figura 9: A estrutura SK_BUFF com o headroom alocado, adaptado de (YOSHIFUJI, 2007)

Supondo que, em algum gancho *Netfilter*, algum módulo deseja adicionar dados no pacote IP, dado que as áreas da estrutura já foram previamente definidas. Existem duas opções. A primeira é verificar se existe espaço no `headroom` para ser usado para dados. Existindo espaço, basta realocar o ponteiro `data` para expandir a área de dados utilizando a área de `headroom` cedida. A segunda opção é expandir a área de `headroom` usando a função `pskb_expand_head()`. Na verdade, neste caso, a estrutura como um todo está sendo aumentada, pois teve de se expandir a área de memória total contendo esta

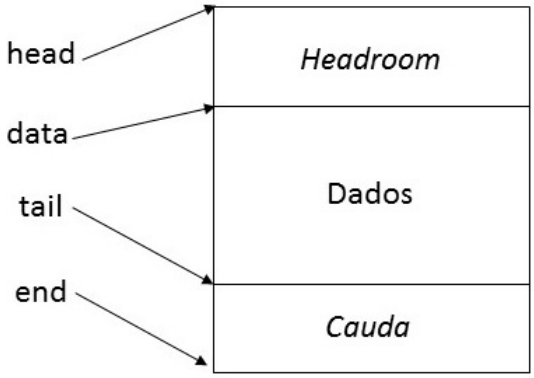

Figura 10: A estrutura SK_BUFF com dados, adaptado de (YOSHIFUJI, 2007)

estrutura. Logo após, é necessário mover o ponteiro `data` com a função `skb_push(X bytes)`, onde X é o número de bytes extra que a área de dados vai passar a ocupar (RUSSEL; WELTE, 2002).

2.4 Linux Kernel Crypto

O *kernel* do Linux possui um *framework* dedicado à criptografia, conhecido como *Linux Kernel Crypto*. Este consiste em implementações de algoritmos de criptografia que podem ser utilizados em qualquer parte do *kernel* ou por módulos de *kernel*. Existem também algoritmos de compressão e é possível também implementar novos algoritmos de criptografia. No início do desenvolvimento deste *framework*, um dos primeiros objetivos era suportar o IPsec. O número de algoritmos de criptografia presente no *Linux Kernel Crypto* é bastante vasto, sendo fora do escopo deste trabalho entrar em detalhes de todos estes algoritmos e como utilizá-los. Assim, o foco desta seção está em explanar como a criptografia foi implementada neste trabalho utilizando o *Linux Kernel Crypto*. Para uma leitura mais completa, recomenda-se a documentação presente em Mueller e Vasut (2014) e Morris (2003).

O AES (*Advanced Encryption Standard*) foi escolhido para ser o algoritmo de criptografia utilizado neste trabalho. O AES é um padrão de cifra de bloco baseado no algoritmo de Rijndael. O AES nasceu de uma competição organizada pelo NIST (*National Institute of Standards and Technology*) dos EUA, em 1997. O objetivo era definir um sucessor para o DES (*Data Encryption Standard*), algoritmo de criptografia desenvolvido na década de 1970 que já não era mais considerado seguro. O algoritmo de Rijndael foi declarado vencedor em 2001, sendo assim anunciado como AES e adotado pelo Governo dos Estados Unidos. Em relação aos algoritmos que participaram da competição, todos deveriam atender às seguintes exigências (WELSCHENBACH, 2013; WIKIPEDIA, 2016):

- Cifra de bloco (nada mais é do que um cifrador que opera em cima de blocos de

tamanho fixo);

- Blocos de 128bits;
- Suporte a chaves de 128,192 e 256 bits;
- Rápida execução via *software*.

O *Linux Kernel Crypto* implementa o AES tanto em *software* quanto em *hardware*. Neste último, é necessário que a máquina possua, na CPU, instruções de criptografia. Caso não tenha as instruções, uma versão padrão por *software* é utilizada pelo *kernel*. A escolha da implementação é feita de forma automática, ou seja, se houver instruções de criptografia, elas serão utilizadas pelo *Linux Kernel Crypto*, a menos que o programador seja explícito em informar que deseja utilizar a implementação em *software*.

Para que os programadores possam acessar estes algoritmos de cifra e usá-los de dentro do *kernel*, é fornecida uma API com funções que facilmente podem criptografar e descriptografar dados no espaço do *kernel*. A API se refere a todo algoritmo de cifra como um "transformador", então, ao longo do texto, "transformador"ou "cifrador"sempre se referem a algoritmos de criptografia. Existe o conceito de *template* também, o que pode ser visto como uma cifra e um modo de operação. Por exemplo, no caso do AES pode-se tê-lo executando no modo ECB (*Electronic Codebook*) ou CBC (*Cipher Block Chaining*), entre outros.

Antes de se usar qualquer criptografia presente no *kernel*, é necessário saber quais algoritmos a versão do *kernel* executando na máquina suporta. Para isso, verifica-se no arquivo `/proc/crypto` quais as cifras que são implementadas na máquina em questão. O resultado apresenta todas as transformações suportadas pelo *kernel* e outras informações como as descritas em Mueller e Vasut (2014):

- nome: o nome genérico da transformação. Quando se usa este nome para invocar a transformação, o *kernel* utiliza também o item prioridade para decidir qual a melhor transformação deve ser utilizada. Isto quer dizer então que várias transformações possuem o mesmo nome genérico.
- prioridade: define a prioridade da transformação. Quanto maior o valor, maior sua prioridade. Como exemplo, o AES na implementação em *Assembly* em arquiteturas x86, possui prioridade 300, e em *hardware* possui valor 400.
- driver: o nome específico da transformação. Assim como o nome genérico, o *driver* também pode ser usado para inicializar o objeto transformação, mas neste caso o *kernel* não verifica a prioridade, pois a chamada foi feita de forma explícita.

- tipo de cifra: define se é cifra de bloco síncrona ou assíncrona, ou se é algoritmo de compressão, geração de números aleatórios, além de outros.
- módulo: define o módulo de *kernel* responsável pela implementação da transformação.
- tamanho do bloco (quando aplicável): se for uma cifra de bloco, define o tamanho do bloco de dados que deve ser usado. No caso do AES, este valor é de 128bits
- tamanho da chave (quando aplicável): define o tamanho da chave que deve ser usada na cifragem. Como exemplo, o AES tem este campo preenchido como 128, 192 ou 256bits.
- tamanho do valor inicial (*Initialization Value*) (quando aplicável): alguns modos de operação requerem o uso deste valor de inicialização.

Depois de definido qual a criptografia e qual o *template* a usar, deve-se inicializar a transformação, para que ela fique disponível para criptografar dados. Após o uso, é necessário destruir a transformação para liberar memória. Neste trabalho, optou-se por usar o AES no modo CBC. O AES operando no modo ECB possui uma falha de segurança, onde blocos de dados iguais produzem a mesma saída criptografada, e no AES em modo CBC isto não ocorre.

Neste exemplo, o nome da transformação presente no *kernel* é *cbc(aes)*. Apenas a título de complementação, no modo CBC é necessário utilizar um valor inicial, chamado *Initialization Value* (valor inicial) no primeiro bloco de dados a ser criptografado. Cada bloco de dados sucessivo utiliza o resultado da criptografia do bloco anterior para compor o resultado, por isso o seu nome. Esta fórmula garante que, mesmo dados de entrada (texto plano) iguais produzem saídas diferentes, aumentando a segurança do esquema de criptografia.

A API disponibiliza o algoritmo de criptografia para ser usado pelo programador como um objeto do tipo `Transformação` (*Transformation* - TFM). Assim, deve-se primeiro alocar memória para o objeto TFM. A linha de código a seguir mostra a estrutura usada para indicar onde na memória o transformador está salvo. Neste exemplo, o objeto TFM é chamado de *blkcipher*.

```
struct crypto_blkcipher blkcipher = NULL;
```

Esta estrutura é genérica, pois ainda não foi definido qual tipo de algoritmo de criptografia será utilizado. Depois de escolhido o algoritmo, ele é definido através de uma variável *char*. Como já mencionado, este trabalho utiliza AES no modo CBC, assim a criptografia usada é definida como:

```
char *cipher = "cbc(aes)"; //nome lido em /proc/crypto.
```

Tendo alocado memória para o TFM e definido o algoritmo, cria-se de fato o TFM para ser utilizado. Isto é feito utilizando o espaço de memória previamente reservada para o objeto TFM através da chamada:

```
blkcipher = crypto_alloc_blkcipher(cipher, 0, 0);
```

Neste momento, a estrutura `blkcipher` é, de fato, uma TFM e pode criptografar e descriptografar dados em programas executando dentro do *kernel*. As linhas de código apresentadas acima foram retiradas de Mueller e Vasut (2014). Recomanda-se uma leitura desta documentação para mais detalhes.

Os dados são criptografados ou descriptografados em *buffers* que são apontados por estruturas especiais, conhecidas como `scatterlists`. As estruturas `scatterlist` funcionam como uma tabela de endereços de memória, onde cada entrada desta tabela marca o início e o tamanho de cada segmento de memória que contém parte dos dados que devem ser escritos/lidos. Assim, a criptografia e descriptografia é feita utilizando esta estrutura, eliminando a necessidade de se copiar os dados para uma área contínua de memória. Adicionalmente, este esquema, permite-se o uso da criptografia acelerada por *hardware* acessar os dados nesta estrutura `scatterlist`.

O código presente na Figura 11 é um exemplo onde a TFM *blkcipher* executa a operação de criptografia e descriptografia utilizando os dados salvos em `buffer_plaintext`. Os dados descriptografados são salvos no `buffer_decryptedText`. Primeiro, é necessário inicializar as estruturas `scatterlists`, utilizadas pelo TFM, para realizar as operações. Cada `scatterlist` é associada a um `buffer` específico. O resultado disso é que toda operação de escrita na `scatterlist` provoca na verdade a escrita dos dados no `buffer` a qual ela foi associada, ou inicializada. No código apresentado, existem três `scatterlists` (`sg1`, `sg2` e `sg3`), associadas ao `buffer_plainText`, `buffer_cryptedText` e `buffer_decryptedText` respectivamente.

A operação de criptografia é executada nos dados apontados pela `sg1` e salvos em `sg2`. O processo de descriptografia utiliza os dados em `sg2` e os salva em `sg3`. Dado que `sg3` foi associado com o `buffer_decryptedText`, os dados descriptografados podem ser lidos diretamente deste `buffer_decryptedText`. A função `crypto_blkcipher_blocksize(blkcipher)` tem, como retorno, o tamanho do bloco a ser usado na entrada do cifrador. Como `blkcipher` foi criado para ser uma TFM do tipo AES, essa função retorna o valor de 16, uma vez que o AES trabalha com blocos de 16Bytes.

O *Linux Kernel Crypto* oferece muitas opções de cifras para que programadores do *kernel* possam adequá-las às suas necessidades. O exemplo apresentado acima teve o objetivo de ilustrar a simplicidade de se utilizar esta funcionalidade presente no *kernel*. A documentação, Mueller e Vasut (2014), fornece mais informações sobre o *Linux Kernel*

Crypto, com outros exemplos de outras implementações de algoritmos de cifragem, e até mesmo de como implementar uma solução de cifra customizada.

```
struct scatterlist sg1, sg2;
int blockSize = crypto_blkcipher_blocksize(blkcipher);
unsigned char *buffer_plainText = NULL;
unsigned char *buffer_cryptedText = NULL;
unsigned char *buffer_decryptedText = NULL;
struct blkcipher_desc desc;
buffer_plainText = kmalloc(blockSize,GFP_ATOMIC);
buffer_cryptedText = kmalloc(blockSize,GFP_ATOMIC);
buffer_decryptedText = kmalloc(blockSize,GFP_ATOMIC);
get_random_bytes(buffer, blockSize);
sg_init_one(&sg1, buffer_plaintext, blockSize);
sg_init_one(&sg2, buffer_cryptedText, blockSize);
sg_init_one(&sg3, buffer_decryptedText, blockSize);
desc.flags = 0;
desc.tfm = blkcipher;
crypto_blkcipher_encrypt(&desc, &sg2, &sg1);
crypto_blkcipher_decrypt(&desc, &sg3, &sg2);
```

Figura 11: Código simples de criptografia, adaptado de (MUELLER; VASUT, 2014)

2.5 Métricas de avaliação de esquemas de rastreamento

Uma série de métricas foram propostas por Belenky e Ansari (2003) para analisar e avaliar um esquema de rastreamento de pacotes IP. De acordo com os autores, atender a estes critérios é essencial para que o esquema possa ter aceitação da comunidade e vir a ser implantado na atual estrutura da Internet. Essas métricas fazem sentido, pois alterar a estrutura atual da Internet é tarefa complicada, sendo talvez até mesmo inviável. A seguir são apresentadas algumas dessas métricas, juntamente com a análise feita por Singh, Singh e Kumar (2016).

2.5.1 Custo Computacional

Todo esquema de rastreamento, que usa a marcação de pacotes, tem como protagonista o roteador. Assim, ele deve atuar nos pacotes IPv6 que está roteando. Normalmente, estes roteadores inserem alguma marca no pacote que será usada futuramente no processo de rastreamento. Logo, é exigido um trabalho adicional, e o gasto computacional mede o custo extra exigido da CPU do roteador para que ele consiga exercer esta nova função sem comprometer a sua tarefa principal. Um esquema que apresenta alto gasto computacional pode enfrentar resistência para ser aceito, pois exigirá investimento de *hardware* dos proprietários das redes. Na seção de Experimentos desta dissertação, é feita uma aná-

lise completa do custo computacional dos roteadores quando submetidos ao esquema de marcação proposto.

2.5.2 Envolvimento dos ISPs (*Internet Service Provider)*

Os ISPs são grandes SAs responsáveis por fornecer serviços de Internet para seus clientes, e assim, a Internet pode ser vista como redes de SAs. Os ISPs possuem relutância em implementar esquemas de rastreamento, pois são eles que administram os roteadores envolvidos no esquema. Primeiramente, ISPs não possuem interesse em adquirir novos equipamentos que podem ser necessários para implementar o esquema. Em segundo lugar, durante o processo de rastreamento, é interessante que os administradores dos ISPs sejam pouco ou nada acionados, uma vez que exigir cooperação deles pode ser tarefa burocrática e demorada. Idealmente, a própria vítima, ou adminstradora de seu domínio, deve ter condições de realizar o rastreamento e achar a origem do pacote.

2.5.3 Privacidade dos ISPs

ISPs também não gostam de revelar a sua topologia interna e nem informações sobre seus roteadores. Todas essas informações são essenciais para sua segurança e devem ser mantidas em sigilo. Os esquemas de rastreamento devem utilizar técnicas que não revelem a estrutura interna do ISP e também não necessite conhecer endereços IPs dos roteadores e suas interfaces.

2.5.4 Implantação incremental

Qualquer modificação na estrutura atual da Internet não pode ser feita de forma binária, ou seja, exigindo que 100% dos SAs presentes na Internet passem a implementar imediatamente um esquema de rastreamento de pacotes IPs. Tal esquema está fadado a nunca entrar em funcionamento. É preciso um plano de implantação incremental até que todos os SAs tenham condições de implementar o esquema de rastreamento.

2.5.5 Complexidade do processo de rastreamento

Alguns esquemas de rastreamento podem exigir grandes quantidades de pacotes maliciosos para serem capazes de reconstruir o caminho até a origem. Quanto mais pacotes necessários, mais complexo o algoritmo de reconstrução, podendo até nunca convergir a uma resposta. Complexidade essa que aumenta ainda mais se o ataque for do tipo DDoS, onde pode existir até milhares de fontes causadoras de ataque. O esquema ideal deve ser capaz de identificar a origem do pacote, ou reconstruir o caminho, usando a quantidade mínima de pacotes possíveis. Um esquema que depende de muitos pacotes

não tem condições de rastrear um e-mail *spam*, que pode ser enviado em apenas um pacote IP.

3 Revisão da Bibliografia

Este capítulo apresenta uma revisão dos trabalhos mais importantes que usam marcação de pacotes. Para cada trabalho, é apresentado seu funcionamento, suas vantagens e desvantagens. Os métodos baseados em marcação de pacotes podem ser divididos em duas grandes áreas — prevenção (*prevention*) ou filtragem e rastreamento (*traceback*) (SHUE; GUPTA; DAVY, 2008).

3.1 Métodos de prevenção

Métodos de prevenção *(prevention methods)* buscam filtrar pacotes com endereço IP forjado antes que alcancem a vítima. Visam, portanto, impedir o ataque ou minimizar seus efeitos. Dentre os métodos de prevenção, situam-se os métodos desenvolvidos por Lee et al. (2007), Liu et al. (2008), Wu, Ren e Li (2007), Wu, Ren e Li (2008), Shue, Gupta e Davy (2008) e Parashar e Radhakrishnan (2014).

Lee et al. (2007) propõem o método BGP *Anti-Spoofing Extension* (BASE). O BASE funciona em quatro fases. Na primeira, os roteadores propagam entre si, através de mensagens BGP *update*, os valores de suas marcas. Cada marca é criada por um algoritmo MAC (*Message Authentication Code*) que utiliza, como entrada, a chave secreta de cada sistema autônomo e o valor anterior da marca ou o prefixo da rede. A segunda fase é disparada pela vítima. Esta, ao detectar que está sob um ataque, solicita aos roteadores, através de mensagens BGP *update*, a marcação e a filtragem dos pacotes. Na terceira fase, os roteadores passam efetivamente a marcar pacotes que saem de seus SAs e a filtrar pacotes que entram em seus SAs. A quarta fase é também disparada pela vítima. Esta, ao detectar que não mais está sob um ataque, solicita aos roteadores, através de mensagens BGP *update*, o retorno ao seu modo normal de operação. Uma das desvantagens do BASE consiste no fato deste depender de mensagens BGP para a comunicação entre os roteadores, uma vez que alguns SAs podem ter políticas de não propagarem tais mensagens. Uma outra desvantagem consiste no fato de que cada roteador deve manter permanentemente uma tabela com os valores atualizados das marcas para cada prefixo de rede existente.

Liu et al. (2008) propõem o método *passport*. No *passport*, cada pacote leva marcas específicas para cada SA que se encontra entre sua origem e seu destino. As marcas são geradas por meio de um algoritmo MAC que utiliza, como entrada, a chave secreta do SA de origem do pacote e de cada SA pelo qual o pacote irá passar. Assim, todas as marcas são criadas no SA de origem, cabendo a todos os SAs no percurso do pacote apenas validarem-nas. Durante a validação, se algum roteador detectar que a marca é inválida,

o pacote é marcado como suspeito, mas só será descartado no destino, pois podem ter ocorrido mudanças na rota. Uma das desvantagens do *passport* consiste no fato de ele exigir que cada SA conheça previamente as rotas para os demais SAs na Internet. Exige, igualmente, que cada SA troque chaves simétricas com os demais SAs, bem como o seu armazenamento. Exige, por fim, que cada SA saiba quais SAs implementam o *passport*. Este conhecimento é necessário, pois o último roteador que implementa o *passport* na rota para o destino é responsável por retirar todas as marcas do pacote para evitar que sejam roubadas.

Wu, Ren e Li (2007), Wu, Ren e Li (2008) propõem o método denominado Arquitetura de Validação de Endereços de Origem (*Source Address Validation Architecture* - SAVA). O SAVA valida o endereço IPv4 de origem de cada pacote, descartando, assim, pacotes que possuam endereços de origem forjados. Em sua abordagem básica, o SAVA estabelece que cada roteador deve manter uma tabela de entrada que relaciona cada uma de suas interfaces com endereços IPv4 de origem de pacotes que trafegam por ela. Estas tabelas são criadas e atualizadas por meio de mensagens SAVA *update*, entre os roteadores. Uma das desvantagens do SAVA consiste na sobrecarga da banda de rede devido ao tráfego de mensagens SAVA *update*. Uma outra consiste no armazenamento das tabelas de entrada, posto que estas, dependendo da localização do roteador, podem alcançar tamanhos consideráveis. Por fim, uma outra desvantagem do SAVA advém de sua arquitetura, que dificulta seu emprego em outros paradigmas de roteamento, tais como os usados em endereços móveis IP e tunelamento.

Shue, Gupta e Davy (2008) propõem um método baseado em marcação de datagramas IPv4. Neste método, cada roteador possui uma tabela que contém um ou mais valores de marcas, conhecido como *tags*, para cada par [prefixo de endereço de origem, interface de entrada]. O pacote é descartado pelo roteador, caso a *tag* contida no datagrama não esteja presente na tabela do roteador ou caso o datagrama não possua uma *tag* quando, de fato, deveria possuí-la. A marca, gerada aleatoriamente, possui 64 bits e é inserida no campo opções do datagrama IPv4. Uma das desvantagens do método consiste no fato de que é possível o roubo de marcas. Além disto, o método possui funcionalidades similares às do método *ingress filtering* (FERGUSON; SENIE, 2000), método este que, embora muito conhecido, é, ao mesmo tempo, muito pouco implementado pelos SAs da Internet.

Parashar e Radhakrishnan (2014), por sua vez, propõem um método baseado em marcação de pacotes IPv6. O método utiliza o cabeçalho de extensão *hop-by-hop* para armazenar, como marca, os 128 bits do endereço IPv6 da interface do roteador de borda do SA pela qual o pacote ingressa na Internet e um valor *hash* deste endereço. Cada roteador no percurso do pacote avalia este valor *hash* para decidir a validade ou não do pacote. Caso não seja válido, o pacote é descartado. Segundo os autores, o método é capaz

de descartar, em um ataque DDoS, 50% dos pacotes que são enviados à vítima. Dentre as desvantagens do método, destacam-se a sobrecarga em todos os roteadores pelos quais o pacote transita, bem como a possibilidade de envio de pacotes com marcas forjadas, caso a função *hash* seja descoberta. Além disto, os autores não apresentam qualquer proposta para a troca segura de chaves da função *hash* entre os SAs.

Métodos de prevenção apresentam três sérias desvantagens. Primeiro, podem não reconhecer pacotes legítimos e, consequentemente, descartá-los (SHUE; GUPTA; DAVY, 2008). Segundo, exigem a cooperação entre diversos sistemas autônomos na Internet, necessitando a troca de informações entre eles e, consequentemente, aumentando gastos extras dos SAs com recursos de armazenamento e largura de banda. Terceiro, ao filtrar pacotes forjados, impedem não só a detecção do real causador do ataque como também a obtenção de meios para provar que o atacante de fato cometeu o ataque. Assim, atacantes continuam produzindo sucessivos ataques na Internet, consumindo indevidamente seus recursos, sem serem identificados, responsabilizados e, eventualmente, penalizados.

3.2 Métodos de rastreamento

Métodos de rastreamento buscam identificar a origem dos pacotes forjados. Visam, portanto, identificar a origem do ataque. Os métodos mais conhecidos são os métodos de marcação probabilística, que seguem a filosofia do PPM (*Probabilistic packet marking*) e os métodos determinísticos que seguem o DPM (*Deterministic packet marking*). A diferença básica entre PPM e DPM consiste no momento de inserção da marca. Enquanto que no PPM o roteador decide, aleatoriamente, se deve ou não marcar um pacote, no DPM existe uma classe de roteadores, previamente definidos, que fazem a marca deterministicamente em todo pacote roteado por eles. Dentre os métodos PPM, situam-se os métodos desenvolvidos por Savage et al. (2001), Paruchuri et al. (2004) e Goodrich (2008). Dentre os métodos DPM, situam-se os métodos desenvolvidos por Belenky e Ansari (2003), Belenky e Ansari (2007), Xiang, Zhou e Guo (2009) e Sun et al. (2011).

Savage et al. (2001) propõem pela primeira vez um método baseado em marcação probabilística, nomeado de CEFS (*Compressed Edge Fragment Sampling*). O CEFS utiliza tuplas [*start-edge*,*end-edge*,distância] como marcas, onde *start-edge* e *end-edge* são os endereços IPv4 dos dois roteadores adjacentes que representam uma aresta do caminho percorrido pelo pacote e "distância", é a distância desta aresta até o destino do pacote. As marcas são autenticadas por uma função *hash* e transformadas por uma função XOR. Não há espaço para o armazenamento integral da marca em um datagrama. Assim, ela é fragmentada em partes e cada parte é enviada separadamente em um datagrama. Uma das desvantagens do CEFS consiste no fato de que é possível, mesmo com autenticação hash, a criação de marcas falsas. Além disto, o CEFS exige uma grande quantidade de pacotes

para reconstrução do caminho entre a origem e o destino. Como consequência, o destino (ou a vítima) precisa dispor de grande quantidade de processamento computacional para identificar o caminho percorrido pelos pacotes.

Paruchuri et al. (2004) propõem um método PPM no qual somente roteadores de borda marcam pacotes que saem dos ASs. Os autores propõem o uso de chaves simétricas para encriptografar as marcas. Assim, cada AS tem sua própria chave e a de seus vizinhos de forma a poder descriptografar as marcas nos pacotes recebidos. Se o roteador não conseguir validar alguma marca, ele sobrescreve-a com sua própria marca para evitar que marcas falsas sejam propagadas. Mesmo que o roteador opte por não marcar algum pacote, é necessário que ele descriptografe a marca do pacote e torne a encriptografá-la com a chave do AS que receberá o pacote. Uma das desvantagens do método consiste no fato de que os autores não definem um protocolo para a troca confiável de chaves simétricas entre os roteadores de borda.

Goodrich (2008) propõe um método PPM no qual cada roteador possui uma marca, definida por um valor de *checksum*, que o identifica. O método proposto é mais eficiente que o PPM original (SAVAGE et al., 2001) em termos da menor quantidade de pacotes necessária para a reconstrução do caminho, da menor quantidade de falsos positivos gerados e da maior dificuldade para geração de marcas falsas nos pacotes. Com 1000 fontes de ataque a 10 saltos de distância até a vítima e com o valor de 0.1 como probabilidade de marcação, o método proposto necessita de cerca de 2 milhões de pacotes para reconstruir o caminho percorrido pelo ataque. O autor enfatiza o fato de que o PPM original, com apenas 30 fontes de ataque, necessita de 650 trilhões de pacotes para reconstruir o caminho. Entretanto, apesar de reduzir sensivelmente o número de pacotes necessários para a reconstrução dos caminhos percorridos pelo ataque, o método proposto não consegue rastrear os atuais ataques DDoS, posto que estes empregam, usualmente, mais de 1000 fontes de ataque.

Belenky e Ansari (2003), Belenky e Ansari (2007) propõem, pela primeira vez, o método DPM. Em 2003 foi proposta a ideia e em 2007 o esquema foi formalizado. No DPM, cada pacote possui, como marca, o endereço IPv4 da interface do roteador de borda do AS pela qual o pacote ingressa na Internet. Dentre as vantagens do DPM, destacam-se a pequena quantidade de pacotes necessária para a reconstrução da origem do tráfego de pacotes e a sua robustez quanto a marcações falsas. Dentre as desvantagens, destacam-se o fato de que o recipiente (ou vítima) necessita conhecer endereços IPv4 de interfaces de roteadores de borda dos SAs para identificar a origem dos pacotes, bem como conhecer quais SAs implementam-no e quais não o implementam. Além disto, a implementação do DPM é complexa, tanto no lado do recipiente (ou vítima), como no lado do AS pelo qual o pacote ingressa na Internet. São necessários a criação e manutenção de algumas tabelas, o uso de algoritmos para percorrê-las, bem como o uso de famílias de funções *hash*.

Xiang, Zhou e Guo (2009) propõem o FDPM (*Flexible Deterministic Packet Marking*). No FDPM, as marcas podem variar em tamanhos de 16, 19 e 24 bits. A marca armazena os 32 bits do endereço IPv4 da interface do roteador de borda do AS pela qual o pacote ingressa na Internet bem como um valor *hash* deste endereço. Não há espaço para o armazenamento integral da marca em um datagrama. Assim, ela é fragmentada em partes e cada parte é enviada separadamente em um datagrama diferente. Segundo os autores, o FDPM é capaz de detectar 128 vezes mais fontes de ataques que o DPM original. No entanto, caso haja colisões nos valores *hash*, são gerados falsos positivos. Uma das desvantagens do FDPM consiste no fato de que o recipiente (ou vítima) necessita conhecer endereços IPv4 de interfaces de roteadores de borda dos SAs para identificar a origem dos pacotes. Além disto, os autores não apresentam qualquer proposta para a troca segura de chaves da função *hash* entre os SAs.

Sun et al. (2011) propõem um método DPM para o IPv6. Os autores optaram por utilizar o cabeçalho de extensão *Destinations* para armazenar, como marca, os 128 bits do endereço IPv6 da interface do roteador de borda do AS pela qual o pacote ingressa na Internet. Assim, o recipiente (ou vítima) consegue saber por qual interface ingressou na Internet cada pacote que recebe. Uma das desvantagens da proposta consiste no fato de que o recipiente (ou vítima) necessita conhecer endereços IPv6 de interfaces de roteadores de borda dos SAs para identificar a origem dos pacotes. Além disto, os autores não especificam qualquer política para que os ISPs implementem a proposta. Por fim, não propõem qualquer esquema de autenticação das marcas de forma a garantir a validade das mesmas.

Os métodos PPM propostos na literatura apresentam algumas desvantagens. Primeiro, exigem grandes quantidades de pacotes para a reconstrução correta do caminho entre a origem e o destino do tráfego. Segundo, exigem que o destino (ou vítima) disponha de recursos computacionais significativos para que possa identificar corretamente o caminho percorrido pelos pacotes. Terceiro, não definem protocolos para a troca confiável de chaves entre os roteadores quando fazem uso de criptografia ou de funções *hash*. Por fim, métodos PPM não conseguem rastrear *e-mails* com conteúdo *spam* nem ataques DDoS.

Os métodos DPM propostos na literatura também apresentam algumas desvantagens. Primeiro, exigem que o recipiente (ou vítima) conheça endereços IP de interfaces de roteadores de borda dos SAs para identificar a origem dos pacotes. Segundo, exigem que o recipiente (ou vítima) conheça quais SAs implementam o DPM e quais não o implementam. Terceiro, não especificam qualquer política para que os ISPs implementem, gradualmente ou não, o DPM. Por fim, os métodos DPM não propõem qualquer esquema de autenticação das marcas de forma a garantir a validade das mesmas.

Belenky e Ansari (2003) descrevem os métodos de rastreamento de pacotes IPv4 e avaliam-nos segundo algumas métricas, tais como envolvimento do ISP, quantidade

de pacotes necessária para o rastreamento, sobrecarga de processamento, sobrecarga de largura de banda, escalabilidade, dentre outras. Singh, Pundir e Pilli (2013) descreve os métodos recentes de rastreamento de datagramas IPv6. Vários destes métodos são extensões para IPv6 de métodos originalmente propostos para IPv4. O autor menciona que os métodos de rastreamento de datagramas IPv6 precisam levar em consideração três desafios. Primeiro, precisam detectar marcas falsas nos pacotes. Segundo, precisam lidar satisfatoriamente com a situação de transição entre os protocolos IPv4 e IPv6. Por fim, precisam ser eficientes mesmo com implementação parcial na Internet ou precisam definir políticas para sua implementação, gradual ou não, pelos ISPs

3.3 Considerações Finais

Para uma Internet mais segura, é importante saber a origem real do pacote trafegando pela rede. Assim, simplesmente descartando um pacote malicioso não resolve o problema de ter uma Internet mais limpa e segura. Quando um esquema opta por filtrar e descartar o pacote, ataques futuros não estão sendo impedidos. Por isso, este trabalho propõe um esquema de rastreamento de pacotes IPv6, se baseando fortemente no trabalho proposto por Belenky e Ansari (2007). O objetivo é criar marcas nos pacotes que servirão para identificar a origem destes. Adicionalmente, esta pesquisa investiga a viabilidade de se utilizar criptografia na infra-estrutura atual da Internet.

4 Marcação de Pacotes por cadeia de Confiança

Este capítulo apresenta um novo esquema de marcação de pacotes IPv6, denominado Marcação de Pacotes por Cadeia de Confiança (*Chain of Trust Packet Marking* - CTPM). É feita uma análise completa deste esquema, abordando todo o modelamento teórico até a forma como foi implementado no laboratório.

O CTPM funciona de forma determinística, onde apenas roteadores na borda do Sistema Autônomo fazem parte do esquema. Toda vez que um pacote entra em um SA, o roteador na entrada cria uma marca e a insere no pacote IPv6. Quando este pacote deixa o SA em direção a um SA vizinho, o roteador na saída criptografa a marca usando uma chave simétrica específica conhecida pelo SA de destino. Portanto, somente o SA vizinho em questão, tem condições de ler a marca e validá-la. Ao longo do percurso entre origem e destino, cada roteador na entrada do SA insere a sua marca, assim, esta marca vai se expandindo. Quando o pacote atingir o destino final, o campo responsável por carregar as marcas no pacote conterá informações de todos os SAs por onde este pacote passou, sendo assim possível fazer um rastreamento completo. Ao longo deste capítulo são discutidos tópicos como: criação da marca, onde a marca é inserida, o que a marca contém de informação, quem é o responsável por inseri-la e, finalmente, como se garante a integridade da marca.

4.1 Traceback Extension Header

Esta seção apresenta onde a marca é inserida. O Capítulo 2 fez uma apresentação dos cabeçalhos de extensão existentes no IPv6. Foi apresentado, de forma resumida, o seu funcionamento, e quem faz a análise destes cabeçalhos. A exceção do *Hop-by-Hop*, os cabeçalhos são analisados apenas no destino.

Os cabeçalhos de extensão existentes não permitem a implementação do CTPM, pois os roteadores responsáveis pela inserção e análise das marcas são apenas os roteadores de borda. Os roteadores internos do SA não participam deste esquema de marcação. Por isso, deve existir um cabeçalho de extensão exclusivo para roteadores de borda, e este cabeçalho é responsável por carregar as marcas inseridas por estes mesmos roteadores. A este cabeçalho de extensão é proposto o nome de *Traceback Extension Header* (TEH), ou Cabeçalho de Extensão para Rastreamento.

Como mencionado no Capítulo 2, o *Hop-By-Hop*, se existir, precisa necessariamente

ser o primeiro cabeçalho de extensão, pois todo roteador deve fazer análise do conteúdo presente nele. Se o TEH deve ser analisado por todo roteador de borda, e não somente no destino, faz sentido o TEH ser obrigatoriamente o segundo cabeçalho de extensão. Sabendo que a análise dos cabeçalhos de extensão é feita de forma sequencial, o primeiro cabeçalho a ser analisado é o *Hop-By-Hop*, obrigatoriamente por todos os roteadores. Ao final de sua análise, é verificado qual o próximo cabeçalho de extensão. Se for o TEH, apenas os roteadores de borda fazem o seu processamento. Os demais roteadores podem ignorar o TEH e todos os demais cabeçalhos, se existirem. A lista a seguir apresenta a nova ordem sugerida para a inserção dos cabeçalhos de extensão:

- Cabeçalho IPv6 Fixo: sempre existe;
- *Hop-By-Hop*: obrigatório ser o primeiro a seguir o cabeçalho fixo, caso exista;
- *Traceback Extension Header*: obrigatório ser o segundo. Caso *Hop-By-Hop* não exista, é obrigatório ser o primeiro;
- *Destinations Header*: para dados a serem processados por roteadores listados pelo *Routing Header*;
- *Routing Header*;
- *Fragment Header*;
- *Authentication Header*;
- *Encapsulating Security Payload*;
- *Destinations Header*: para dados a serem processados apenas no destino final;
- Cabeçalho da camada superior.

Para se adequar as normas do IPv6, o TEH é proposto seguindo o formato genérico de qualquer cabeçalho de extensão, já apresentado na Figura 4 do Capítulo 2. Como foi discutido, existem duas regras para a criação de um novo cabeçalho de extensão:

- Nenhum cabeçalho com comportamento *Hop-by-Hop* deve ser criado. Em último caso, devem ser criadas novas opções para este.
- De preferência, nenhum novo cabeçalho de extensão deve ser criado, e sim proposto uma nova opção para o *Destinations Header*.

Se não for possível seguir as regras acima, deve-se emitir uma justificativa que comprove a necessidade de um cabeçalho totalmente novo. A justificativa da proposta do

novo cabeçalho de extensão é que nenhuma nova opção para o *Hop-By-Hop* se enquadra nas necessidades do TEH, pois não é desejável que o conteúdo da marca seja analisado por todo roteador dentro do SA. Isso significaria um aumento do gasto computacional. Também nenhuma nova opção para o *Destinations Header* atende aos requisitos do TEH, pois não é somente o destino que faz a análise do conteúdo da marca. Assim, o TEH surge da necessidade de se ter um cabeçalho de extensão exclusivo para análise dos roteadores de borda, além do seu nome definir o conteúdo a ser carregado, ou seja, marcas referentes a rastreamento de pacotes.

Porém, o TEH não precisa se limitar à marcação de pacotes. Nada impede que outros pesquisadores definam novas opções a serem carregadas pelo TEH, porém ele continuará sendo analisado apenas por roteadores de borda. Isto é semelhante à proposta de novas opções para o *Destinations Header*, onde novas opções podem ser propostas, mas somente o destino faz a análise destes dados.

4.2 A Marca da Cadeia de Confiança - Chain of Trust Mark

Esta seção apresenta o conteúdo da marca inserida dentro do TEH. Esta marca, denominada *Chain of Trust Mark* (CTM), Marca da Cadeia de Confiança, possibilita ao CTPM funcionar. Ela é a unidade fundamental do processo de rastreamento da origem do pacote. De uma maneira geral, o TEH carrega como conteúdo somente a marca, CTM, inserida nos pacotes pelos roteadores de borda. Assim, caso não seja especificado nada, TEH e CTM se referem à mesma coisa.

A CTM inserida nos pacotes IPv6 contém duas partes: *Identification Mark* (IM), ou Marca Identificadora (MI) e *Authentication Mark* (AM), ou Marca Autenticadora (MA). Naturalmente, o TEH também possui os campos obrigatórios de qualquer outro cabeçalho de extensão, *Next Header* - Próximo Cabeçalho e *Header Length* - Tamanho do Cabeçalho, já mencionados no Capítulo 2 - Referencial Teórico. A Marca Identificadora, como o nome diz, contém informações que identificam a interface do roteador que a inseriu. A Marca Autenticadora garante a autenticidade e integridade da marca, impedindo assim que pacotes capturados tenham sua marca copiada e utilizada para marcar outros pacotes.

4.2.1 Marca Identificadora

Para que a identificação do SA seja a mais completa possível, a MI, ilustrada na Figura 12, contém três campos. Segue a sua descrição:

- *ASN*: AS *Number*, ou Número do Sistema Autônomo. É um valor de 4Bytes/32bits que identifica globalmente um sistema autônomo (VOHRA; CHEN, 2007);

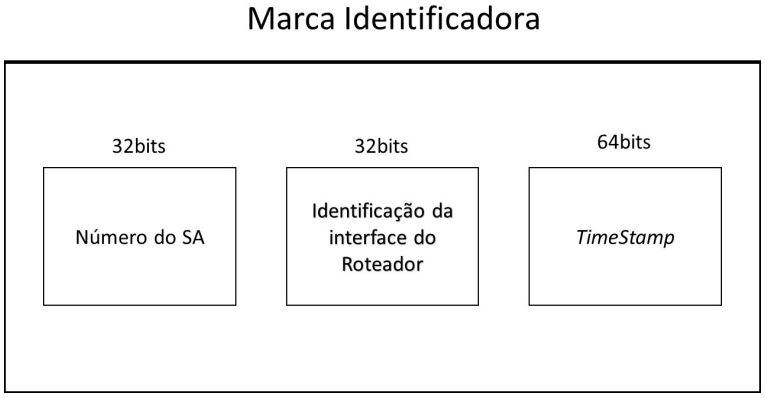

Figura 12: Marca Identificadora dividos em seus três campos

- *Router ID*: Identificação da interface do roteador. Com os dois campos anteriores tem-se informações do SA, data e hora no qual o pacote trafegou. Porém, ainda não se tem informações da interface pela qual o roteador recebeu o pacote. Este campo, *Router ID*, possui 32bits e não tem uma regra de como o adminstrador da rede deve usá-lo. A ideia deste campo é ter uma informação precisa que identifique o roteador e sua interface, porém sem expor identificações que só interessam ao adminstrador do SA. O objetivo é que este campo contenha uma informação de identificação da interface do roteador, porém esta informação só faz sentido para o administrador daquela rede. Esta informação pode ser exposta na marca sem comprometer as informações reais do roteador como IP e outros dados de identificação. O uso desta técnica faz sentido, pois, somente o adminstrador da rede pode tomar providências com relação ao mau uso de um de seus roteadores. Assim, não faz sentido o adminstrador de outra rede conhecer detalhes deste roteador ou até mesmo de sua interface.

- *TimeStamp*: é um campo que registra a hora e data. Neste campo, o roteador informa em qual horário e data a marca foi criada. O objetivo desta informação é adicionar mais rigidez e controle nas marcas, facilitando o processo de rastreamento do pacote. No *kernel* do Linux essa informação possui tamanho de 8Bytes, dada pela estrutura `timeval`;

Normalmente, a CTM não é construída por completo em um único roteador. Isso quer dizer que o roteador responsável por criar a MI da CTM pode não ser o mesmo que cria a Marca Autenticadora (MA). Este último é descrito na próxima seção. A MI é criada pelo roteador na borda de entrada do pacote IPv6. No momento em que o pacote ingressa no sistema autônomo pela primeira vez, este roteador insere a MI referente a ele, e portanto, o pacote circula por dentro do SA já carregando a parte de identificação da CTM.

A cada novo SA que o pacote ingressa, uma nova MI é criada e anexada ao TEH. No processo de rastreamento, todas as MIs presentes no cabeçalho TEH informam por qual interface do roteador o pacote ingressou no SA especificado, além da data e horário deste ingresso.

4.2.2 Marca Autenticadora - Authentication Mark

A Marca Autenticadora (MA) garante que a CTM presente no pacote é legítima e portanto, que aquele pacote é válido. O processo de criação da MA leva em consideração informações presentes no pacote onde ela é inserida. Por isso, se algum usuário utilizar um *sniffer* [1]para capturar um pacote e tentar reutilizar a marca em outro pacote com conteúdo malicioso, este não conseguirá ingressar nas redes seguintes, pois a CTM não será válida para este pacote malicioso. A MA, ilustrada na Figura 13, é dividida em duas partes descritas a seguir:

- Valor do Bit - *Bit Values*: para garantir que a marca só é válida para um pacote IPv6, 12 bits são escolhidos aleatoriamente deste pacote, incluindo o *payload* e o cabeçalho fixo. Depois de escolhidos esses bits, o valor individual de cada um destes é salvo no campo *Bit Values*. No processo de validação da CTM, o roteador utiliza esse campo para verificar se a marca presente no pacote IPv6 recebido é válida. Este roteador verifica se os valores individuais dos 12 bits presentes no pacote recebido são os mesmos valores informados no campo *Bit Values*. O motivo de se utilizar 12 bits foi uma solução de compromisso entre segurança e também atender ao requisito de que todo cabeçalho de extensão IPv6 precisa ser múltiplo de 8bytes. Além disso, todo bloco a ser cifrado pelo AES precisa ser múltiplo de 16Bytes. Assim chegou-se ao valor de 12bits.

- Posição do Bit - *Bit Position*: depois que o campo *Bit Values* é criado com os valores dos 12 bits escolhidos, é preciso informar quais 12 bits foram escolhidos. Caso contrário, fica impossível fazer a validação. Assim, o campo *Bit Position* carrega a informação de quais foram os bits escolhidos, e não os seus valores. Sabendo que o *payload* do pacote IPv6 pode ter até 64KBytes, ou 524288 bits, é necessário uma palavra de 19bits para endereçar completamente qualquer valor. Como a CTM precisa informar os 12 bits escolhidos, o campo *Bit Position* precisa carregar 228bits (19 bits para cada um dos 12 escolhidos). No processo de validação, o roteador primeiro consulta o campo *Bit Position* para saber qual a posição do primeiro bit, faz a leitura deste valor e compara com o valor salvo no campo *Bit Values*. Esse processo se repete para todos os 12 bits.

[1] ferramenta que é capaz de analisar e interceptar tráfego em uma rede de dados.

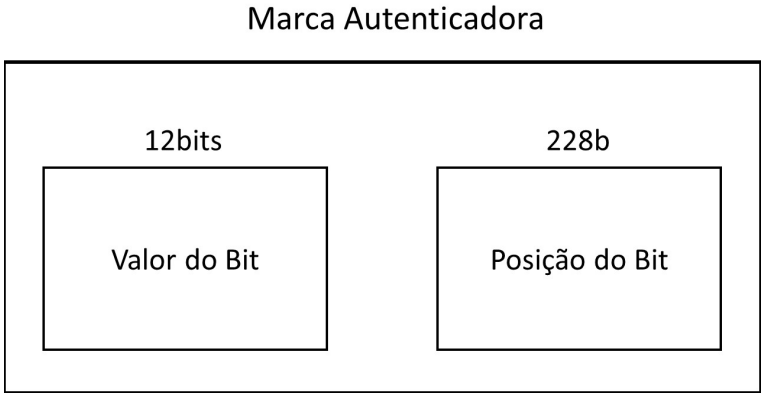

Figura 13: Marca Autenticadora

Exemplo de criação da Marca Autenticadora

1. Construção do campo *Bit Values*: primeiro são escolhidos aleatoriamente 12 bits e depois o roteador busca no pacote IPv6 qual o valor de cada um destes bits. Neste exemplo, os bits escolhidos e seus valores foram:

   - Bit 25 - Valor '0'
   - Bit 49 - Valor '0'
   - Bit 112 - Valor '1'
   - Bit 164 - Valor '1'
   - Bit 166 - Valor '1'
   - Bit 190 - Valor '0'
   - Bit 450 - Valor '1'
   - Bit 456 - Valor '0'
   - Bit 512 - Valor '0'
   - Bit 589 - Valor '1'
   - Bit 601 - Valor '1'
   - Bit 602 - Valor '1'

   Assim, o campo *Bit Values* contém a palavra 001110100111.

2. Construção do campo *Bit Position*: tendo os bits já escolhidos, basta salvar o valor de sua posição. Cada palavra contém, 19 bits, mas os bits de valor '0' a esquerda foram omitidos a fim de proporcionar uma lista mais clara. Neste exemplo os valores são:

   - Bit 25 - Valor '11001'

- Bit 49 - Valor ‘110001’
- Bit 112 - Valor ‘1110000’
- Bit 164 - Valor ‘10100100’
- Bit 166 - Valor ‘10100110’
- Bit 190 - Valor ‘10111110’
- Bit 450 - Valor ‘111000010’
- Bit 456 - Valor ‘111001000’
- Bit 512 - Valor ‘1000000000’
- Bit 589 - Valor ‘1001001101’
- Bit 601 - Valor ‘1001011001’
- Bit 602 - Valor ‘1001011010’

O campo *Bit Position* fica sendo, então, todos os valores binários acima em sequência. Esta palavra final de 228bits não é escrita aqui devido ao seu tamanho.

Quando a CTM é recebida, o roteador segue os seguintes passos para validá-la:

1. Extrai os valores do campo *Bit Position* e *Bit Values*;
2. Busca o primeiro bit apontado pelo *Bit Position*. Neste exemplo é o bit 25;
3. Compara o valor do bit 25 presente no pacote IPv6 com o valor do primeiro bit informado pelo *Bit Values* (neste exemplo valor ‘0’). Caso o valor seja igual, o algoritmo prossegue para o próximo bit. Caso contrário, o pacote deve ser descartado, pois o pacote está corrompido, ou pode ter tido sua marca roubada e inserida em outro pacote IPv6 na tentativa de validar este pacote.

O exemplo acima tem como objetivo deixar claro a função dos campos *Bit Values* e *Bit Position*: o primeiro carrega o valor do bit escolhido (‘0’ ou ‘1’), e o segundo informa qual foi o bit escolhido (bit 25, por exemplo).

Na subseção 4.2.1, foi informado que o roteador na borda de entrada é o responsável pela criação e inserção da MI. A MA é criada e inserida dentro do TEH pelo roteador de borda no momento em que o pacote for deixar o SA corrente em direção a um SA vizinho. Observe que, quando o pacote IPv6 for deixar o SA, ele já contém a sua MI inserida no TEH. Portanto, quando o roteador na borda de saída for criar a MA, todo conteúdo presente no TEH é removido por completo do pacote temporariamente. Esse processo é feito, pois, quando a MA é criada, alguns bits do pacote IPv6 são escolhidos para serem utilizados. Quando a MA é inserida, naturalmente, há uma modificação nos

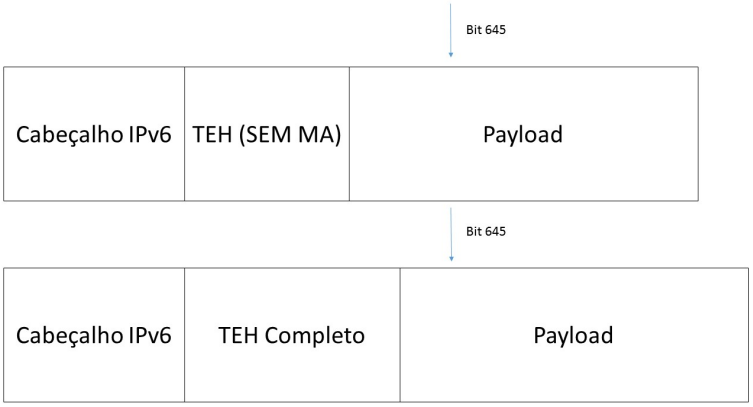

Figura 14: Inserção Incorreta da MA

valores dos bits, pois novos dados foram adicionados no pacote. Isto causaria um erro na validação da CTM. Assim, a solução dada foi toda vez que a MA for criada, remove-se o TEH por completo, cria a MA, retorna com o TEH (que havia sido removido) e insere a MA recém criada no pacote. No processo de validação da marca, remove-se o TEH e extrai os campos da CTM. Assim, o pacote se encontra da mesma forma que estava no momento da criação da MA no roteador na borda de saída, e todo o processo de validação pode ser feito.

Para ilustrar a situação mencionada no parágrafo anterior, considere a Figura 14. O pacote IPv6 já está saindo do SA e, portanto, o TEH ainda não contém a MA (primeira parte da Figura). Suponha que o primeiro bit escolhido, dentro dos 12, seja o 645. Nesse momento, esse bit se encontra em uma posição dentro do pacote IPv6. No momento em que a MA for inserida, haverá deslocamento para a direita de dados (parte de baixo da Figura 14), e o bit 645 já não é mais o mesmo.

A fim de evitar o uso de técnicas de *offset*, pois bits do cabeçalho fixo também podem ser usados, optou-se por remover o TEH temporariamente, cria-se a MA e depois insere-se o TEH completo, como mostra a Figura 15. Nesta Figura, tem-se 3 etapas. Na primeira parte, o TEH está incompleto (sem MA) e o pacote está no último roteador de borda antes de deixar o SA. Na segunda parte, o TEH é removido, e os bits para compor a MA são escolhidos. Novamente como o exemplo, o primeiro é o bit 645. Depois de buscado os outros 11 bits, constrói-se a MA, reinsere-se o TEH e anexa-se a MA, como mostra a última parte da Figura.

4.2.3 Criptografia

As duas subseções anteriores (4.2.1 e 4.2.2), definem as duas partes fundamentais da CTM, a MA e a MI. Porém, somente elas não podem garantir a segurança do sistema. É preciso também utilizar um esquema de criptografia para proteger o conteúdo da CTM,

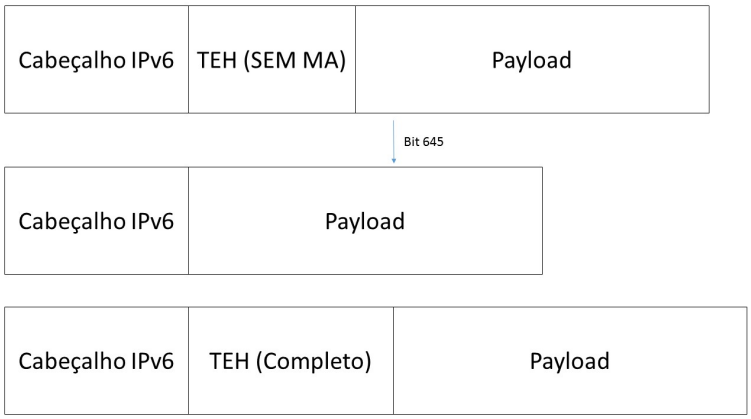

Figura 15: Inserção Correta da MA

em especial, quando o pacote IPv6 trafegar para fora de um SA em um canal não confiável.

Como mencionado na seção 2.4, a criptografia utilizada é a AES-128 no modo CBC. O AES é uma cifra de bloco de 16Bytes/128bits, ou seja, é preciso fornecer uma entrada com esse tamanho de bloco, e a saída criptografada também possui 16Bytes. Na seção 4.2.4 será comentado sobre os bits de preenchimento inseridos na CTM para atender a este critério.

O esquema proposto pelo CTPM tem como princípio básico a confiança entre SAs vizinhos. Para isso, o esquema propõe que cada roteador de borda de um SA tenha uma chave secreta para cada uma de suas interfaces. Adicionalmente, cada roteador deve conhecer a chave secreta das interfaces de outros roteadores (de outros SAs) conectadas diretamente a ele.

Considere o esquema apresentado na Figura 16. Nesse exemplo, tem-se três Sistemas Autônomos com um roteador cada. O fluxo de informação está saindo de R_1 em direção a R_2 e R_3. Cada um destes roteadores deve concordar com o uso de uma chave secreta no formato, K_I. É necessário uma chave secreta para cada interface que conecta estes roteadores. Como na Figura 16, tem-se dois enlaces, são necessárias duas chaves simétricas. Para o enlace 1, que conecta R_1 até R_3, foi definida a chave K_1. Para o enlace 2, que conecta R_1 até R_2, foi definida a chave K_2. Como R_1 se conecta a dois roteadores, ele precisa conhecer duas chaves.

Supondo uma situação onde há mais de uma interface conectando R_1 a R_2, o procedimento é o mesmo. Cada um dos enlaces que conectam estes roteadores possuirá uma chave secreta, portanto a chave usada deve ser a definida para o enlace em questão.

É necessário também definir um esquema para a troca de chaves, que ocorrerá periodicamente de acordo com os interesses dos administradores dos SAs. O CTPM não determina nenhuma periodicidade para a troca de chaves. Esta troca ocorrerá através de criptografia assimétrica. Apesar de ser mais lenta que a criptografia simétrica, ela será

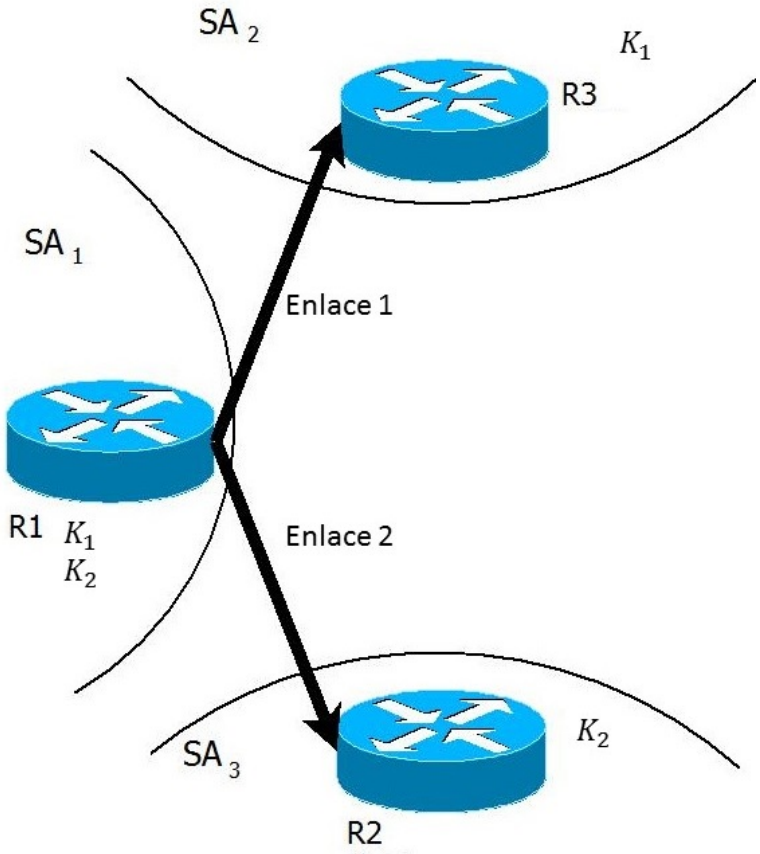

Figura 16: Esquema de Criptografia

usada com menos frequência, além de permitir que um canal não seguro seja usado para troca de chaves simétricas. Um exemplo prático pode ser o protocolo *Diffie-Hellman*, que consiste em um esquema para a troca de chaves simétricas em um canal não seguro. (DIFFIE; HELLMAN, 1976).

4.2.4 Complementos da CTM

Neste estágio, a CTM possui dois campos, a MA e a MI. A MA possui tamanho de 30Bytes e a MI tem tamanho de 16Bytes, totalizando 46Bytes. Porém, é necessário atender a um requisito do IPv6 que diz que todo cabeçalho de extensão precisa ter um tamanho múltiplo de 8Bytes, já considerando os campos obrigatórios *Next Header* e *Header Length*, que possuem 1 Byte cada (DEERING; HIDEN, 1998). Assim, o TEH tem tamanho total de 48Bytes. Porém, O TEH define também um campo de controle de 1 Byte denominado *Hop Count* (HC), ou Contador de Saltos (CS). Este campo é incrementado toda vez que o pacote ingressa em um SA. Portanto, o mesmo roteador que cria a MI também faz este incremento do HC. Sua função é manter registro de quantos SAs o pacote trafegou e também serve de *offset* para um roteador inserir a sua MI.

O TEH, já com a CTM inserida, possui agora tamanho de 49Bytes. Logo, é ne-

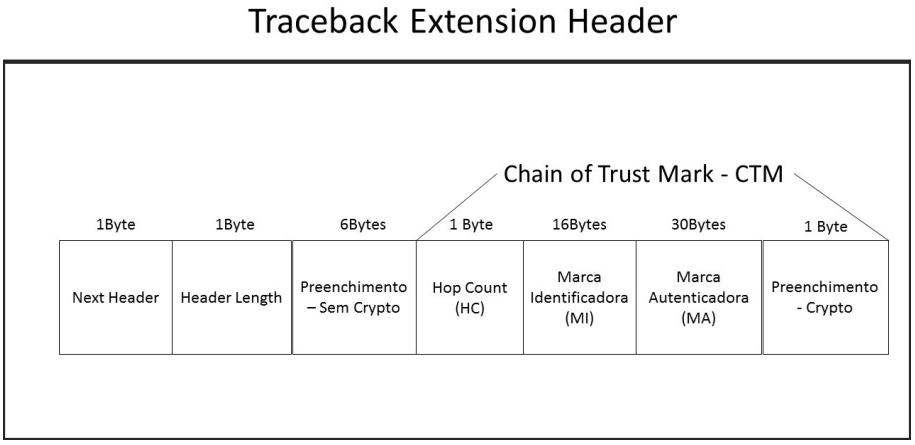

Figura 17: Traceback Extension Header (TEH) - Completo

cessário alguns bytes de preenchimento para que o TEH se torne múltiplo de 8Bytes. Porém, como a CTM é criptografada usando AES-128, ela deve ser múltipla de 16Bytes. Atendendo a todos esses requisitos, chega-se na configuração final do TEH, apresentada na Figura 17.

Foram necessários bytes de preenchimento em duas situações. Na primeira, buscava-se que a CTM se tornasse múltipla de 16Bytes. Portanto, 1 Byte foi adicionado ao final da CTM, chamado de "Preenchimento - Crypto". Sabendo que o TEH, como um todo, deve ser múltiplo de 8Bytes, e a CTM, somada aos campos obrigatórios do TEH, passou a ter 50Bytes, outro preenchimento se tornou necessário. Mais 6Bytes foram adicionados na parte externa da CTM, chamada de "Preenchimento - Sem Crypto". Como será visto mais adiante nesse capítulo, o TEH possui este exato formato no momento em que o pacote deixa o primeiro SA.

4.3 Exemplo de Funcionamento do CTPM

As duas seções anteriores apresentaram o que é a marca, CTM, gravada nos pacotes IPv6 durante o processo de roteamento nas bordas de um SA. Esta seção apresenta detalhadamente o sistema completo em funcionamento. É mostrado o que acontece quando o pacote entra no primeiro SA encontrado, até o momento em que o pacote chega ao seu destino.

Considere o ambiente apresentado nas Figuras 18 e 19. Ele tem três sistemas autônomos, cada um com cinco roteadores. A análise será feita apenas nos roteadores de borda, os demais são irrelevantes para o CTPM. Suponha que um pacote IPv6 foi gerado pelo computador "Origem", e trafega por estes três SAs até atingir o computador "Destino". A ideia é mostrar como se dá o processo de funcionamento do CTPM usando

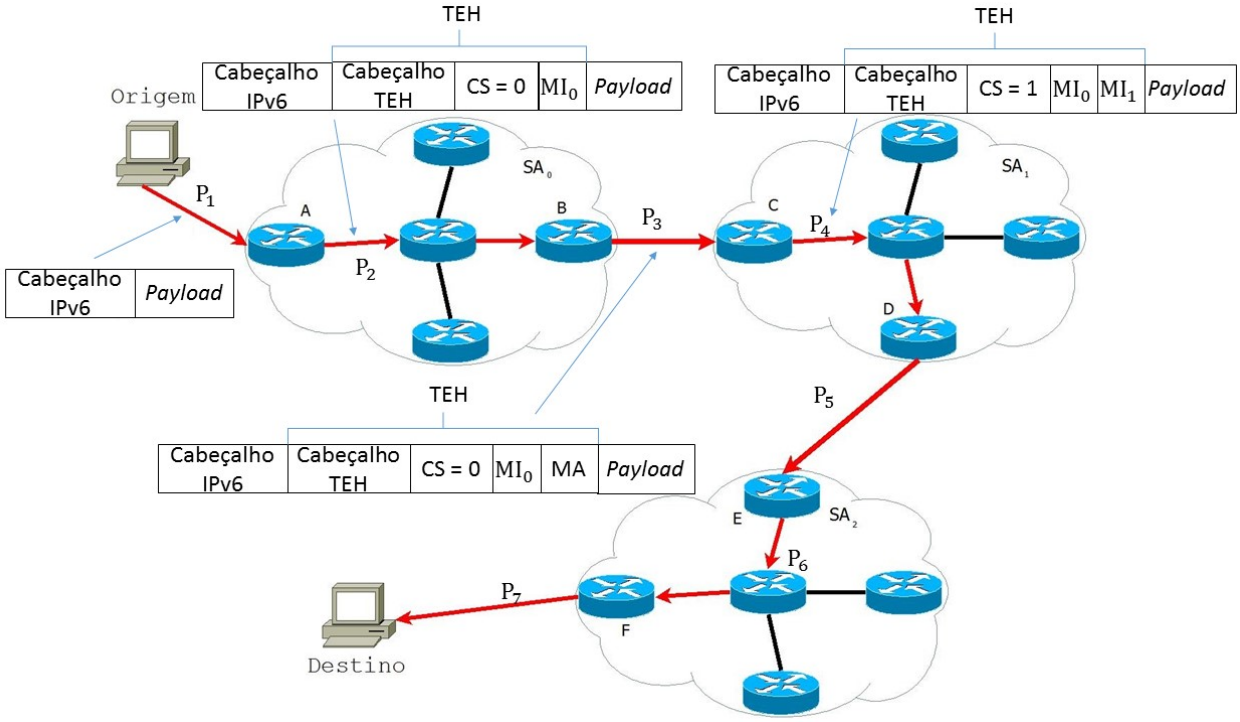

Figura 18: Ambiente para apresentação do CTPM - Parte 1

este ambiente hipotético. Na primeira parte da análise, considere apenas a Figura 18.

No computador "Origem", um pacote IPv6 é gerado com destino ao computador "Destino", e este pacote ingressa na Internet pelo SA_0. No ponto P1, este pacote não possui nenhuma informação referente a CTM e nem possui o cabeçalho de extensão TEH. Quando este pacote ingressa no SA_0, o roteador de borda 'A' executa o algoritmo de entrada. Este algoritmo é apresentado na Figura 20.

Seguindo o algoritmo, o roteador 'A' verifica que o pacote foi originado dentro da sua zona adminstrativa e assim, inicia o processo de criação do TEH. Primeiro inicializa o valor de HC com '0', e depois cria a marca identificadora (MI), completando assim, parte do CTM. Essa CTM é inserida dentro do TEH, e os campos obrigatórios NH e HL são preenchidos adequadamente. Este dois campos, NH (*Next Header*) e HL (*Header Length)*, estão sendo chamados neste trabalho de "Cabeçalho do TEH". O ponto P2 apresenta o conteúdo do pacote IPv6 neste momento.

Em seguida, o pacote IPv6 trafega normalmente dentro do SA_0 e nenhum roteador interno faz qualquer tipo de verificação ou alteração no conteúdo presente dentro do TEH. Ao atingir o roteador de borda 'B', este aplica o algoritmo de saída no pacote, ilustrado na Figura 21.

O roteador 'B' verifica que o destino do pacote não se encontra na zona administrativa do SA_0, e portanto, o pacote deve ser enviado para outro SA. Assim, o algoritmo entra no laço *else*, criando a MA e anexando-a ao CTM já existente no pacote IPv6.

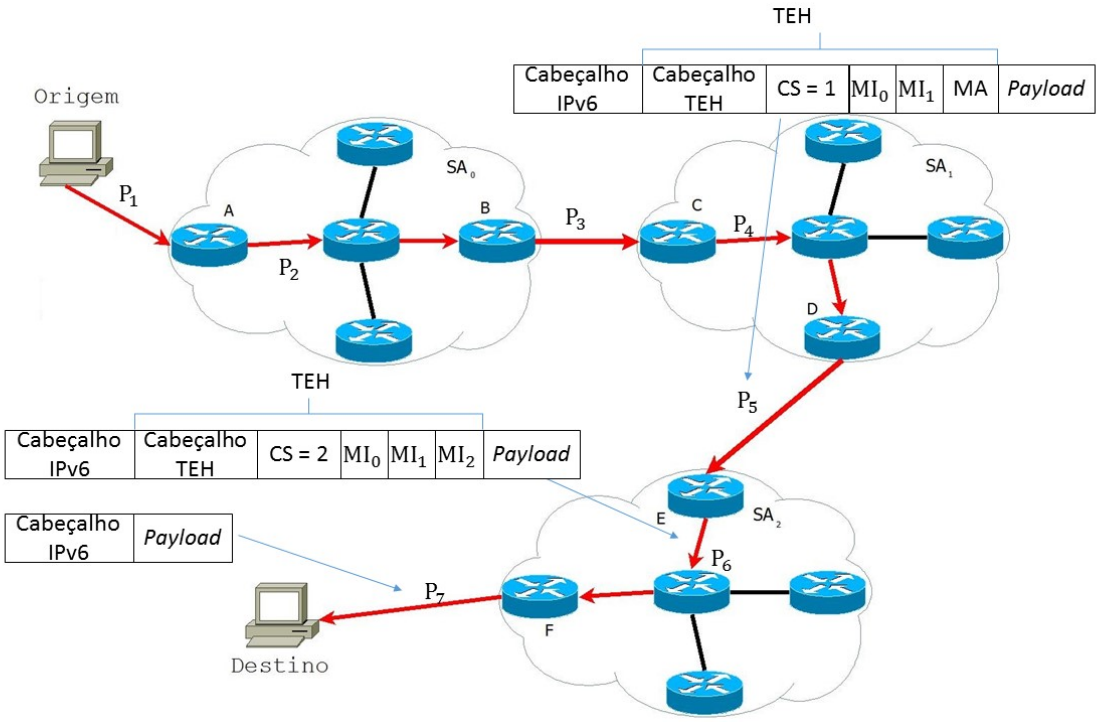

Figura 19: Ambiente para apresentação do CTPM - Parte 2

Em seguida, a CTM passa pelo processo de criptografia AES-128, onde a chave simétrica utilizada pertence ao enlace P3. A chave, neste exemplo, é a K_{P3}. Lembrando que apenas a CTM dentro do TEH é criptografada. Os demais campos NH e HL não são criptografados, pois os roteadores internos do SA devem ser capazes de ler este campo. O ponto P3 mostra como está o conteúdo do TEH no pacote IPv6 logo que deixa o SA_0. O TEH está completo, e contém todas as informações necessárias para deixar o SA. Este é o tamanho mínimo de um TEH quando um pacote deixa a sua zona administrativa. Apenas por questões de clareza na imagem, é omitido os campos de preenchimento, mas estes estão presentes também.

Quando o pacote alcançar o roteador 'C', este aplica o algoritmo de entrada. É detectado que o pacote já possui TEH, e portanto, a CTM precisa ser validada. Primeiramente, o roteador executa a descriptografia para recuperar a CTM em texto plano. Depois, utilizando os dados presentes na MA recebida, faz as buscas dos bits e compara os valores presentes no pacote com os valores informados na marca. Caso o valor de qualquer bit não coincida, então, algo está errado e o pacote é considerado suspeito. Cabe ao administrador da rede decidir o que fazer. Ele pode descartar o pacote, ou colocá-lo em uma fila de pacotes suspeitos.

Ocorrendo a validação sem problemas, a marca autenticadora é removida do TEH e descartada. Agora, o roteador 'C' deve criar a sua MI e anexá-la ao pacote, além de incrementar o campo HC e atualizar o campo HL (que faz parte do Cabeçalho do TEH). O ponto P4 na Figura 18, apresenta o pacote logo após este sair do roteador 'C'. Observe

```
if (pacote originado dentro da zona adminstrativa do SA) {
    HC = 0;
    cria TEH;
    cria CTM com HC e MI;
    insere CTM dentro do TEH;
}
else {
    descriptografar CTM usando AES da interface;
    // verifica MA, compara valor de HC com HL,
    // verifica ASN da última MI
    if (MA, HC ou ASN estão incorretos) erro;
    remove MA e campos de preenchimento do CTM ;
    HC++;
    adiciona MI do SA corrente ao CTM;
    calcula o novo valor do HL;
    reconstroi o novo TEH;
}
```

Figura 20: Algoritmo de Entrada TEH

```
if (destino do pacote se encontra na zona adminstrativa do SA) {
    salva o cabeçalho IPv6 e o TEH para análise posterior;
    // (se necessário)
    remove TEH do pacote;
}
else {
    insere MA e bits de preenchimento no CTM;
    encriptografa CTM usando AES-128 com a chave do enlace;
    calcula e insere HL;
    reconstroi TEH com a MA e MI;
}
```

Figura 21: Algoritmo de Saida TEH

que não existe mais a MA e uma nova MI foi criada e anexada. Seguindo o padrão já apresentado, o pacote trafega normalmente por dentro do SA_1 até atingir o roteador 'D'. Este roteador executa o algoritmo de saída, anexando uma nova MA (criada a partir de novos bits) e faz a criptografia da CTM utilizando a chave simétrica K_{P5}. O ponto P5 apresenta o pacote após este procedimento.

Ao chegar no roteador E do SA_2, este aplica o mesmo procedimento realizado pelo roteador 'C'. Se a marca for válida, o MA é removido e um novo MI é anexado, como mostra o conteúdo do pacote no ponto P6 na Figura 19. No momento em que o pacote deixa o SA_2, ele detecta que o destino está em sua zona administrativa, e é na verdade o destino final do pacote. Portanto, todo o cabeçalho TEH é removido e o pacote é enviado para o destino com o conteúdo apresentado no ponto P7.

Nessa seção fica claro o funcionamento do CTPM, e quais as modificações os pacotes IPv6 sofrem ao longo de sua rota. Fica evidente também, que apenas os roteadores na borda são envolvidos no esquema. Este modelo de funcionamento, além de trazer segurança à infraestrutura da *Internet*, atende ao requisito da não exposição da topologia interna de um SA. Os administradores de rede não têm interesse em divulgar para a comunidade qual é a estrutura interna das redes sob sua adminstração, pois isto pode comprometer a segurança do SA (SINGH; SINGH; KUMAR, 2016). Esquemas de rastreamento como (SAVAGE et al., 2001) expõem completamente a estrutura interna do SA no momento em que o rastreamento é realizado. Como o CTPM identifica apenas quais roteadores de borda estão envolvidos, o que acontece internamente dentro do SA não é exposto, mantendo assim a topologia protegida.

O campo *Router ID* também contribui para manter sigilo das estruturas internas de um SA. Como explicado na seção 4.2.1, este campo não força o adminstrador do SA a inserir dados como interface e endereço IP de um roteador de borda. Estas informações são sigilosas, e se forem expostas, podem comprometer a segurança da rede (SINGH; SINGH; KUMAR, 2016). A especificação do campo *Router ID* deixa aberta para o adminstrador da rede decidir como melhor usá-lo. Assim, ele pode construir uma tabela interna, onde cada entrada da tabela possui um código correspondente para cada IP e interface do roteador. No campo *Router ID* estes códigos podem ser utilizados mantendo os valores reais em sigilo.

4.4 Realizando o Rastreamento

O CTPM permite um rastreamento instantâneo, diferentemente de outros esquemas que dependem de mais de um pacote, ou até mesmo milhares de pacotes, para executarem o seu algoritmo de reconstrução do caminho percorrido pelo pacote IP. No esquema proposto neste trabalho, apenas um pacote é suficiente para detectar a origem. Com isso,

um único *e-mail spam* pode ser facilmente rastreado.

Não há necessidade de enviar o TEH para o usuário final, pois ele não faz uso desta informação. É um conteúdo de interesse apenas para os administradores de rede. Se um cliente detectar que esta sendo vítima de um ataque DDoS, ele deve entrar em contato com o administrador do seu SA, para que este comece a ler o TEH presente nos pacotes maliciosos, e assim verificar a sua origem. Em seguida, o administrador da rede, referente a primeira MI presente no TEH, deve ser contactado e informado que sua rede está gerando tráfego malicioso. O CTPM não prevê, mas o SA tem autonomia para implantar um esquema de detecção de ataque de forma pró-ativa, ou seja, sem esperar o cliente se queixar. Com o auxílio do TEH os próprios administradores de rede podem detectar o ataque e já iniciar o processo de rastreio.

Existe também a possibilidade do cliente ter sido vítima de um ataque, mas somente tomar ciência após o ataque já ter sido finalizado. Neste caso, não tem como o SA verificar os pacotes em direção a vítima para extrair o TEH corrente, pois não há mais um fluxo de pacotes de ataque. Visando este tipo de rastreamento, os roteadores podem armazenar, por um período de 24 horas, todos os TEHs dos pacotes que saem dele, assim tendo um histórico de TEHs. Se um cliente receber um pacote malicioso, ele pode solicitar para o administrador da rede verificar em seu histórico recente, qual o TEH do pacote em questão, para assim fazer o rastreamento.

Observa-se então, que no CTPM não existe algoritmo de rastreamento. Em outros esquemas como (BELENKY; ANSARI, 2007) e (SAVAGE et al., 2001), é necessário extrair uma informação (no caso a marca) de uma quantidade de pacotes, e utilizar estas informações como entrada do algoritmo de reconstrução do caminho percorrido. Em alguns esquemas, o algoritmo poderia executar por dias (dependendo do número de fontes originadoras de pacotes maliciosos), ou nem mesmo convergir, sem fornecer nenhuma resposta sobre o caminho percorrido pelo pacote de ataque. Como mencionado em (SINGH; SINGH; KUMAR, 2016), umas das métricas de avaliação de um esquema de rastreamento é a carga computacional exigida no processo de rastreamento. No CTPM não existe tal carga, pois a CTM inserida no pacote já contém toda a informação necessária para saber a origem do pacote.

4.5 Implantação do CTPM

Para que o CTPM entre em funcionamento, é necessário primeiro uma mudança nos padrões IPv6 com relação ao TEH. Este cabeçalho de extensão proposto deve ser aceito e inserido na lista oficial de cabeçalhos de extensão, tornando assim um padrão. Isso deve ser feito, pois as normas do IPv6 dizem que se um roteador encontrar um cabeçalho desconhecido, este pacote deve ser descartado. Obviamente, isso não é desejável.

Logo que o TEH for inserido na norma IPv6, todos os *softwares* dos roteadores devem ser atualizados para conhecerem este novo cabeçalho. Além disso, os *softwares* devem também serem atualizados para implementar o CTPM (criar a CTM, validá-la, executar a criptografia etc.). Também, quando necessário, o *hardware* deverá ser atualizado.

Porém, como todo esquema novo, ele deve suportar uma implantanção incremental. Não pode ser uma exigência que o esquema seja implantado por todos os SAs em uma única fase, pois isso inviabilizaria o sucesso. A implantação deve começar pelos grandes *backbones* ou *players* da internet, conhecidos como ISPs de Nível 1 (*Tier 1 ISPs*) (BELENKY; ANSARI, 2007), que concentram grande parte do tráfego da Internet (WINTHER, 2006). Como exemplo de ISP Nível 1, tem-se o AT&T, com ASN de 7018. Posteriormente, a implantação deve seguir um plano descendente, onde SAs de níveis menores começam a implantar o esquema sucessivamente até que se atinja o menor nível de SA.

No primeiro momento somente os grandes *players* estariam usando o CTPM para trocar dados entre si. Em seguida deve ser estipulado uma data, onde todos os demais SAs de nível 2 conectados diretamente aos de nível 1 devem passar também a implementar o CTPM. Caso o prazo para estes se adaptarem seja extrapolado, os SAs de nível 1 podem se negar a encaminhar pacotes vindo de SAs nível 2. O administrador de uma rede de um SA nível 1, pode optar por conta própria, inserir uma marca e encaminhar o pacote, porém neste caso ele passa a assumir a responsabilidade por este pacote. O esquema de implantação segue desta forma, sempre tendo um prazo para que SAs de nível inferior implementem o CTPM. Assim, ao final, toda a infraestrutura da Internet estará implemenando o CTPM.

5 Experimentos

5.1 Ambiente de Teste

Para realizar os experimentos foram utilizados quatro computadores pessoais, sendo dois se comportando como computadores de usuário final e dois se comportando como roteadores. O ambiente simula a existência de dois sistemas autônomos, onde cada sistema possui um par (PC-Roteador). A Figura 22 ilustra o ambiente. R1 e R2 são computadores executando *Debian Wheezy* 7 e, além da placa de rede *on-board*, possuem uma outra placa de rede *off-board* para se comportarem como roteadores. A placa de rede *off-board* utilizada era ora o modelo *Realtek* ora o modelo *Intel Server 1000/1*.

Em relação aos computadores de usuário final, o PC2 possui o sistema operacional *Debian Wheezy* 7 e executa o *Iperf3* (ROSEN, 2014) como servidor. O PC1 possui o sistema operacional *Windows* 8.1 executando a versão cliente do *Iperf3*. Tanto o R1 quanto o R2 tiveram duas configurações diferentes. Na primeira, ambas eram máquinas AMD E-350 1.6GHz com 2GB de *RAM*. Na segunda, ambas eram máquinas i7 *octa-core* 3,6GHz com 8GB *RAM*.

Sendo um dos objetivos desta pesquisa medir o custo computacional de se aplicar criptografia em dados trafegando na Internet, é fundamental que o ambiente utilizado seja real e não virtual. O ambiente utilizado já serve para tal propósito. Assim, os custos são medidos em máquinas que realmente implementam o CTPM. Nas demais seções, são descritos o objetivo destes experimentos, as configurações utilizadas e como é executado cada teste em determinada configuração. Ao final, os resultados são apresentados.

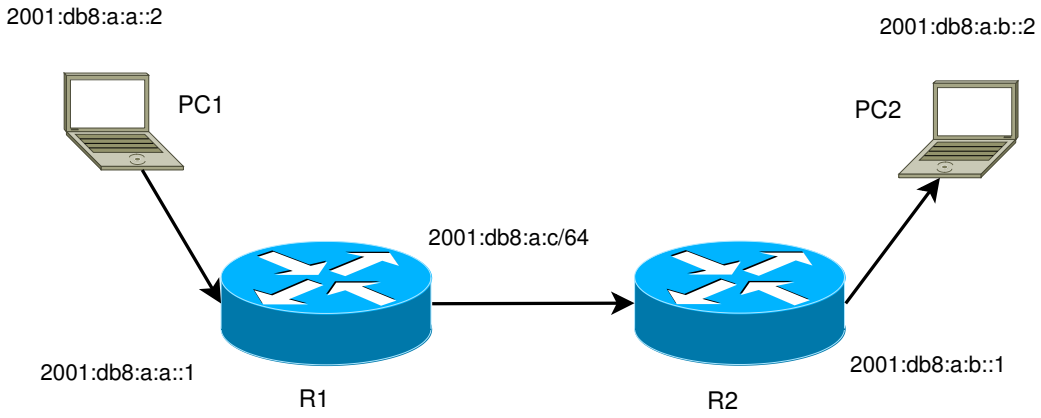

Figura 22: Ambiente de Testes

5.2 Objetivo dos experimentos

Feita a modificação da pilha TCP/IP do Linux, através da criação do novo cabeçalho TEH, deseja-se analisar quais são os impactos que essa modificação provoca no desempenho dos roteadores. As métricas analisadas são: largura de banda, uso de CPU e memória. A largura de banda é medida entre PC1 e PC2 usando a ferramenta *Iperf3* (ROSEN, 2014). O uso de CPU e memória são medidos em R1 e R2, uma vez que é o roteador que faz o processamento necessário para incluir o novo cabeçalho de extensão. Para fazer essa medição, utilizou-se a ferramenta SAR do pacote SYSSTAT, presente em sistemas UNIX/Linux.

5.3 Configuração dos ambientes

Quatro ambientes diferentes de execução foram usados. Deve-se notar que, independentemente da configuração utilizada, fisicamente o ambiente está de acordo com a figura 22. A seguir, uma descrição das configurações de *hardware* utilizadas para os testes.

- Configuração 1: R1 e R2 são máquinas AMD E-350, e os roteadores usam, como placa de rede *off-board*, a *Realtek*.
- Configuração 2: R1 e R2 são máquinas AMD E-350, e os roteadores usam, como placa de rede *off-board*, a *Intel Server*.
- Configuração 3: R1 e R2 são máquinas Intel i7, e os roteadores usam, como placa de rede *off-board*, a *Realtek*.
- Configuração 4: R1 e R2 são máquinas Intel i7, e os roteadores usam, como placa de rede *off-board*, a *Intel Server*.

Para cada configuração de ambiente descrita acima, seis testes distintos foram executados. Toda vez que este trabalho mencionar "configuração", está se referindo a qual *hardware* está sendo utilizado. Estes testes foram executados por um tempo de 60 segundos, e ao final observou-se os valores. Os testes são detalhados posteriormente ainda nessa seção, porém, vale ressaltar que quando este trabalho mencionar que "o teste está sendo executado"significa que o *Iperf3* está gerando tráfego entre PC1 e o PC2, e ao final, observa-se os valores de CPU, memória e largura de banda. O tempo de 60 segundos foi escolhido, pois observou-se que testes sendo executados com duração superior não apresentam alteração significativa em seu valor médio. Ou seja, os valores obtidos após a execução de 60 segundos, podem ser considerados estáveis. Assim, não há necessidade de se executar testes exaustivamente.

Visando detectar pontos de maior gasto computacional, como também queda na largura de banda durante o processo de criação do cabeçalho de extensão, foram criados vários módulos de *kernel* (LKM) que isolam tais pontos. Assim, pôde-se observar quais etapas impactaram mais na queda de largura de banda e aumento de uso de CPU e memória. A seguir, tem-se uma descrição dos diferentes módulos de *kernel* utilizados em cada teste, ou seja, cada módulo foi criado para a realização de um teste. Cada módulo é executado em cada uma das quatro configurações de *hardware* descritas no início desta seção. O cabeçalho de extensão é utilizado unicamente para inserir a marca. Assim, toda vez que esta seção mencionar criação da marca ou criação do cabeçalho de extensão, está se referindo à mesma coisa. Estes módulos de *kernel* são carregados apenas em R1 e R2.

1. Teste 1 - Ausência de Módulo: neste teste, nenhum módulo é inserido em R1 e em R2. O objetivo é verificar a largura de banda, uso de CPU e Memória no sistema original, sem modificação. Os dados obtidos neste teste são a referência para todos os demais testes dentro da mesma configuração de *hardware*.

2. Teste 2 - O TEH é criado através de um módulo de *kernel*, por isso é preciso saber qual o impacto causado pelo uso de um LKM. Neste teste, deseja-se verificar qual o impacto causado apenas pela inserção de um módulo vazio em R1 e R2. O módulo carregado não executa nenhuma operação nos pacotes IPv6, ele apenas provoca o seu desvio, e depois, retorna os pacotes para seguirem seu fluxo na pilha TCP/IP. Com isso, pôde-se medir o impacto causado apenas pelo desvio dos pacotes.

3. Teste 3 - Módulo Completo: neste teste é inserido em R1 e em R2 o módulo TEH completo. Este módulo é a implementação completa do CTPM. Ambos os algoritmos de entrada e saída apresentados no Capítulo 4 são implementados dentro deste módulo. No caso do R1, cada pacote que está saindo, é desviado para o módulo TEH Completo, onde é feita toda a operação de criação do cabeçalho de extensão (usando a criptografia e geração de números aleatórios) que é então inserido no pacote IPv6. No R2, a operação é bem mais simples. Para cada pacote que está chegando em R2, executa-se a descriptografia da marca, faz-se a sua validação e anexa uma nova MI na CTM. Caso a validação seja positiva, o pacote é encaminhado para o seu destino final, neste caso o PC2. Observe que o módulo carregado em R1 e em R2 é o mesmo, não há diferença entre módulo para roteadores ingressantes e um módulo para roteadores egressantes. Este módulo TEH Completo foi desenvolvido já pensando em uma implementação comercial, sendo que ele já detecta automaticamente qual algoritmo que deve ser utilizado, o de entrada ou de saída.

4. Teste 4 - Módulo de Criptografia: neste teste, o módulo TEH cria o cabeçalho de extensão sem usar geração de números aleatórios. Os dados do cabeçalho de

extensão, porém, são criptografados antes de serem inseridos nos pacotes IPv6. Assim, este teste foca nos impactos gerados pela criptografia.

5. Teste 5 - Módulo de Números Aleatórios: neste teste, o módulo TEH cria o cabeçalho de extensão e o insere no pacote IPv6 sem realizar a criptografia, porém, é utilizada a geração de números aleatórios. Assim, verifica-se o impacto causado pela geração de números aleatórios isoladamente.

6. Teste 6 - Módulo da Marca sem Criptografia e sem Números Aleatórios : cria-se o cabeçalho de extensão, porém não se utiliza geração de números aleatórios e os dados não são criptografados antes de serem inseridos no pacote IPv6.

Os testes, descritos acima, detectam qual etapa é aquela que mais degrada o desempenho dos roteadores no que diz respeito à largura de banda, uso de memória e CPU. Dessa forma, futuramente, pode-se concentrar o desenvolvimento do TEH em uma parte mais específica a fim de obter-se um melhor desempenho.

5.4 Execução e Medição dos Testes

A execução dos testes consiste em gerar um tráfego de dados entre PC1 e PC2 usando o *Iperf3*. O que distingue cada teste é o módulo TEH que vai estar carregado em R1 e R2. Definido qual módulo será utilizado e o carregando devidamente, iniciam-se os testes. Para começar, executa-se no PC2 o *Iperf3* Servidor, que fica aguardando uma conexão de um cliente para iniciar a troca de dados. O servidor é inicializado digitando o seguinte comando no terminal do PC2.

- *root: iperf3 -s*

Assim que o servidor é iniciado, deve-se iniciar na máquina cliente (PC1) o *Iperf3 Client* para o início da troca de dados. Ao final, um relatório é emitido contendo qual a largura de banda entre cliente e servidor, ou seja, entre PC1 e PC2. Para iniciar o cliente, e consequentemente, a troca de dados, digita-se o seguinte comando no terminal do PC1:

- *root: iperf3 -c 2001:db8:a:b::2*

O IP 2001:db8:a:b::2 é o endereço IPv6 da máquina servidora. Depois de executado o *iperf3*, a seguinte resposta, apresentada na Figura 23, aparece no terminal de ambos PC1 e PC2. No exemplo executado, foi usada no final a opção *-t 30* para dizer que o *iperf3* deve ser executado por 30 segundos. Por padrão, ele é executado por 10 segundos. Neste exemplo, a largura de banda detectada ao final de 30 segundos foi de 739Mbps.

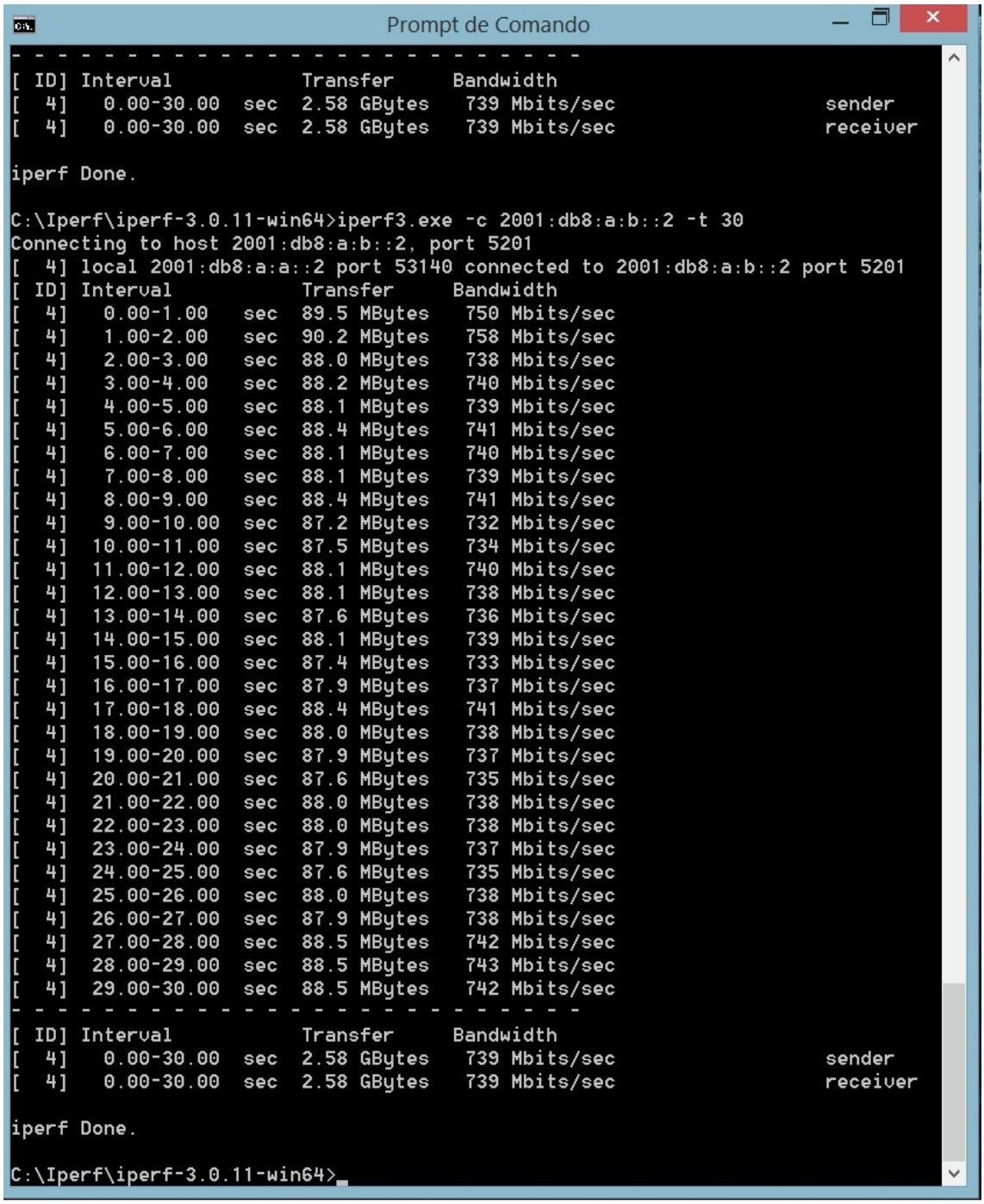

Figura 23: Resultado da Execução do Iperf

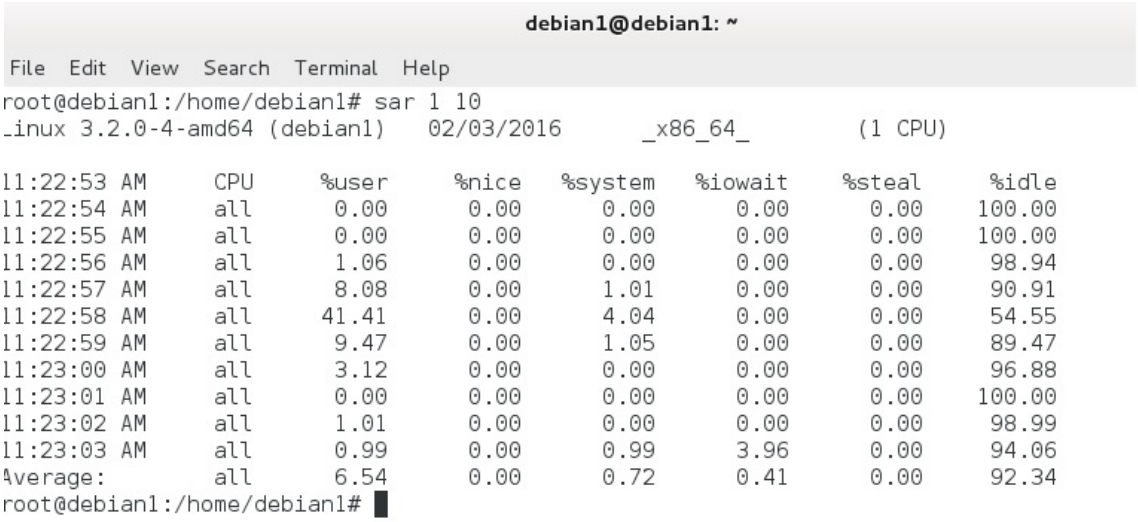

Figura 24: Resultado da Execução do SAR para CPU

Esta execução é apenas uma ilustração do *Iperf3*, pois nos testes, o *iperf3* executa por 60 segundos.

Para medir o uso de CPU usa-se o comando SAR da seguinte forma:

- *root: sar 1 X*, onde X é o tempo total de execução do SAR. Neste exemplo, ilustrado na Figura 24, utilizou-se 10 segundos. E o valor '1' simplesmente é o ponto de ínicio, dizendo que a CPU será medida do segundo 1 até o segundo X.

Assim, o comando SAR é executado por 10 segundos, e a cada segundo de execução o terminal apresenta, entre outras informações, qual o uso de CPU naquele segundo. Ao final dos 10 segundos, apresenta também o valor médio, que é o valor de interesse neste trabalho. O valor médio apresentado no terminal, pode ser visto como a reposta para a pergunta "qual foi o uso de CPU nos últimos 10 segundos?". Se omitirmos a opção *"1 10"* no comando SAR, o uso de CPU é medido por apenas 1 segundo. O uso de CPU é medido tanto no R1 quanto no R2. O comando SAR deve ser executado juntamente com o *Iperf3*, pois o objetivo é medir o uso durante uma grande carga de pacotes atravessando os roteadores.

De maneira similar, para medir o uso de memória usa-se o comando descrito abaixo:

- *root: sar -r 1 X*, e, neste exemplo, também utilizou-se 10 segundos.

Este comando mostra no terminal, entre outras informações, a quantidade de memória livre, memória utilizada e memória total nos últimos 10 segundos. Portanto, novamente, este comando deve ser executado coincidindo com a execução do *iperf3* para que seu resultado seja relevante. A Figura 25 apresenta o resultado da execução do comando *sar -r*.

```
root@debian1:/home/debian1# sar -r 1 10
Linux 3.2.0-4-amd64 (debian1)   02/03/2016      _x86_64_

11:24:43 AM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact
11:24:44 AM    296352    730656     71.14     69868    427724   1059332     54.22    275540    359552
11:24:45 AM    296352    730656     71.14     69868    427724   1059332     54.22    275540    359552
11:24:46 AM    296352    730656     71.14     69868    427724   1059332     54.22    275540    359552
11:24:47 AM    296352    730656     71.14     69876    427716   1059332     54.22    275540    359552
11:24:48 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
11:24:49 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
11:24:50 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
11:24:51 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
11:24:52 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
11:24:53 AM    296352    730656     71.14     69876    427724   1059332     54.22    275540    359560
Average:       296352    730656     71.14     69874    427723   1059332     54.22    275540    359557
root@debian1:/home/debian1#
```

Figura 25: Resultado da Execução do SAR para Memória

5.5 Resultados e Análises

Depois de apresentadas as configurações de *hardware*, os módulos de *kernel* utilizados e o procedimento de execução destes testes, esta seção faz a apresentação e a análise dos resultados obtidos. Como já mencionado, cada teste é executado em todas as quatro configurações e, para melhor apresentar os resultados, foi criada uma tabela para cada configuração. Cada tabela contém o resultado de todos os testes para uma configuração específica e, ao final, uma tabela sumariza alguns dados para efeitos de comparação. Para todos os testes, os roteadores foram inicializados sem nenhuma interface gráfica, ou seja, apenas com terminal de comandos. A ideia é reduzir ao máximo os aplicativos executando em R1 e R2. Também, todo o módulo usado para os testes, foi carregado em tempo de inicialização do roteador. O arquivo de configuração */etc/modules* já foi pré-configurado para carregar os módulos de *kernel* no momento de inicialização dos roteadores.

5.5.1 Configuração 1

Tanto na configuração 1 quanto na 2, os roteadores eram máquinas AMD E-350 *DUAL CORE* 1,6GHz com 2GB de *Ram*. A diferença está na placa de rede *off-board* que foi utilizada. No primeiro caso, foi uma simples *Realtek*. Já no segundo, a placa de rede *Intel Server* foi utilizada. Esta é uma placa com preço bem superior à placa *Realtek*, pelo fato dela possuir uma CPU própria. Esta máquina E-350 é relativamente fraca para os padrões atuais de processadores, e tem alguns problemas de desempenho quando é utilizada para realizar tarefas de um usuário comum (edição de texto e navegar na Internet). No entanto, tem sua função em laboratórios de teste, pois é interessante colher o máximo de dados possíveis para gerar uma tabela comparativa. Também, mede-se o quanto é necessário gastar mais em *hardware* para atingir um desepenho satisfatório. Estas máquinas AMD E-350 não suportam a criptografia AES-NI por *hardware*. Então, o *Linux Kernel Crypto* utilizou a versão por *software* escrita na linguagem *Assembly* presente no próprio *kernel*.

No Teste 1, tem-se uma largura de banda de 496 Mbps. Isso quer dizer que, sem

nenhuma modificação em nenhum roteador, o sistema consegue funcionar a no máximo 496Mbps. O uso de CPU é bem pequeno e o de memória também. Estes valores podem ser considerados o "melhor caso"para esta configuração. No teste 2, é inserido um módulo que não faz alterações no pacote IPv6. Este teste indica que o desvio de pacotes IPv6 para o módulo de *kernel* não comprometeu o desempenho, pois a redução da largura de banda e o acréscimo do uso da CPU e Memória foram desprezíveis.

O Teste 3 coloca o módulo completo para executar. O módulo cria a marca completa, utiliza números aleatórios, faz a criptografia de todos os dados e insere esta marca no pacote IPv6. O resultado apresentado foi considerado muito ruim. A enorme queda na largura de banda, cerca de 95% de queda, tornou inviável utilizar na prática, da forma como foi desenvolvido, o novo cabeçalho de extensão. Houve também grande aumento do uso de CPU, que passou de 4,72% para 43,15%.

O teste 3 mostrou que era necessário identificar, no módulo, qual etapa de construção da marca estava provocando a queda na largura de banda, ou pelo menos qual parte é a maior responsável por isto. Assim, seguiram-se os demais testes.

O teste 4 verificou o custo da criptografia. Neste teste, a marca é criada normalmente, mas não se utilizam números aleatórios para localizar os bits no pacote IPv6. O resultado mostra que, apesar de uma queda na largura de banda, ela certamente não é a maior responsável pelo resultado apresentado no teste 3. Nota-se também o acréscimo do uso de CPU, porém, é bastante pequeno se comparado ao resultado apresentado no teste 3. Certamente, não é a criptografia a maior responsável pelo impacto causado pela criação do novo cabeçalho de extensão.

O próximo teste, o quinto, verificou o custo da geração de números aleatórios. Para cada pacote IPv6 que deixa o R1 em direção a R2, doze números aleatórios são gerados via *software*, uma vez que nenhuma configuração de *hardware* utilizada oferece suporte a geração de números aleatórios por *hardware*. Neste teste, a marca não foi criptografada, porém, foram utilizados números aleatórios para buscar os bits do pacote IPv6. Observou-se uma grande queda na largura de banda, cerca de 42%, e aumento do uso de CPU para 36%. Estes valores mostram que, de fato, a geração de números aleatórios possui o maior custo computacional.

No teste 6 foi criada a marca sem números aleatórios e sem uso da criptografia. Este teste mostra que não há praticamente nenhum impacto na largura de banda, e apenas, um pequeno acréscimo de uso de CPU. Mas, obviamente, sem a criptografia e também sem utilizar os números aleatórios, o esquema de segurança do TEH estaria comprometido. A criptografia garante a confidencialidade, e os números aleatórios garantem a integridade das marcas.

Com relação ao uso de memória, nota-se que não há uma diferença significativa em

Tabela 1: Resultados para Configuração 1

E-350 + Realtek	BW(Mbps)	CPU(%)	Memória(%)	CPU RX(%)	Memória RX(%)
Teste 1 – Ausência de Módulo	496	4,72	6,75	1,11	7,09
Teste 2 – Módulo Vazio	492(↓0,8%)	4,78	6,78	1,95	7,09
Teste 3 – Modulo Completo	22(↓95,56%)	43,15	8,06	7,16	7,16
Teste 4 – Modulo Sem Num. Aleatórios	420(↓15,32%)	9,13	6,78	7,16	7,16
Teste 5 – Modulo Sem Criptografia	283(↓42,94)	36,7	7,88	2,23	7,09
Teste 6 – Marca Sem Crip. E Num. Aleatórios	489(↓1,41%)	4,81	6,78	2,23	7,09

seu valor, independentemente do teste executado. No pior caso, que acontece no teste 3, o uso de memória sobe de 6,75% para 8,06% de uso. Portanto, foi um acréscimo aceitável. Com relação a CPU e memória na máquina R2, também observa-se que o uso é bastante menor que na máquina R1, uma vez que, a parte que mais consome CPU é a geração de números aleatórios, que não existe na recepção do pacote. O único gasto no R2 é a descriptografia, que tem um aumento de 1,11% para 7,16% com relação a CPU. A tabela 1 apresenta os resultados completos para a Configuração 1. Na tabela tem-se 6 colunas. A primeira coluna informa qual módulo estava em uso, a segunda a largura de banda entre PC1 e PC2, a terceira e quarta informam, respectivamente, uso de CPU e memória para o R1. A duas últimas colunas informam o valor de CPU e memória para R2.

5.5.2 Configuração 2

Nesta configuração 2, apenas trocou-se a placa de rede *off-board* presente nos roteadores R1 e R2 para a placa *Intel Server* e repetiram-se todos os testes. A Tabela 2 mostra os resultados. Iniciando pelo teste 1, ou também teste de referência, nota-se que houve um melhor desempenho se comparado ao da Configuração 1. Na Configuração 1, obteve-se uma largura de banda de 496Mbps e agora, na Configuração 2, o valor passou para 775Mbps aliado a uma queda no uso de CPU de 4,72% para 2,32%. Estes valores melhores são reflexo direto do uso da placa de rede Intel Server.

No teste 2, verifica-se o impacto causado pelo desvio dos pacotes IPv6 para o módulo de *kernel*. Assim como para a Configuração 1, o impacto é bastante pequeno, cerca de 0,9% de queda da largura de banda, e pode ser desprezado. No teste 3, que usa o módulo completo, a queda na largura de banda foi de 61,16%, o que, se comparada à Configuração 1, é um valor inferior, porém ainda insuficiente para ser considerado um

Tabela 2: Resultados para Configuração 2

E350 + Intel Server	BW(Mbps)	CPU(%)	Memória(%)	CPU RX(%)	Memória RX(%)
Teste 1 – Ausência de Módulo	775	2,32	6.75	1,11	7,09
Teste 2 – Módulo Vazio	768(↓0,9%)	2,89	6.75	1,95	7,09
Teste 3 – Modulo Completo	301(↓61,16%)	41,27	8.06	6,88	7,16
Teste 4 – Modulo Sem Num. Aleatórios	623(↓19,61)	9,26	6.75	6.88	7,16
Teste 5 – Modulo Sem Criptografia	349(↓54,96)	39,27	7.88	2,18	7,09
Teste 6 – Marca Sem Crip. E Num. Aleatórios	688(↓11,22%)	4,11	6.75	2,18	7,09

bom resultado. Em valores absolutos, a queda foi de 775Mbps para 301Mbps, enquanto que, na Configuração 1, a lagura de banda caiu de 496Mbps para 22Mbps.

No teste 4 observa-se uma queda de 15,32% provocada pela criptografia e, no teste 5, verifica-se uma queda de 42,94% na largura de banda causada pela geração de números aleatórios. Estes resultados confirmam, mais uma vez, que o maior responsável pela queda de desempenho do sistema é a geração de números aleatórios utilizada para a construção da marca.

5.5.3 Configuração 3

Como já mencionado, nas configurações 3 e 4, os roteadores são agora máquinas i7 3,46 GHz *Octacore*. Máquinas i7 são mais poderosas que as AMD E-350 1,6GHz, presentes nas configurações 1 e 2. Outro detalhe importante é que máquinas i7 suportam criptografia AES-NI, que é a criptografia AES acelerada por *hardware*. O *Linux Kernel Crypto* presente no *kernel* detecta o suporte da criptografia por *hardware* e já faz uso dela. Assim, espera-se um desempenho bem superior ao encontrado nas configurações anteriores. A Tabela 3 mostra os resultados.

No teste 1, o valor da largura de banda obtido foi de 491Mbps, que é o maior valor possível nesta configuração. Este valor é inferior ao da configuração 2, mesmo tendo evoluído a CPU para um i7. Isto ocorre justamente por se estar utilizando a placa de rede *off-board Realtek*, enquanto na configuração 2 foi usada a placa de rede *Intel Server*. Pode-se concluir então que a placa de rede limita o valor máximo de largura de banda. Assim como nas configurações anteriores, este valor serve de base para todos os outros testes desta configuração. No teste 2, a largura de banda caiu para 485 Mbps, uma queda de 1,22%. Igualmente, como nas configurações anteriores, esta queda também pode ser

Tabela 3: Resultados para Configuração 3

Intel i7 + Realtek	BW(Mbps)	CPU(%)	Memória(%)	CPU RX(%)	Memória RX(%)
Teste 1 – Ausência de Módulo	491	0,11	1,72	0,09	1,72
Teste 2 – Módulo Vazio	485(↓1,22%)	0,12	1,73	0,10	1,72
Teste 3 – Modulo Completo	332(↓32,38%)	13,16	1,79	1,45	1,73
Teste 4 – Modulo Sem Num. Aleatórios	452(↓7,94%)	0,32	1,75	0,96	1,73
Teste 5 – Modulo Sem Criptografia	362(↓26,27%)	9,78	1,77	0,10	1,72
Teste 6 – Marca Sem Crip. E Num. Aleatórios	482(↓1,83%)	0,14	1,73	0,10	1,72

desprezada, pois é insignificante.

O teste 3 apresentou uma queda na largura de banda de 32,38%, que foi o melhor resultado obtido em todas as configurações até o momento. O uso de CPU, de 13,16%, também foi o melhor resultado obtido. Nota-se porém, que as máquinas presentes na Configuração 1 e 2 possuíam 2 núcleos, enquanto as máquinas presentes nas configurações 3 e 4 possuem 8 núcleos. O uso da memória RAM teve pouca ou nenhuma alteração. Os demais testes (4, 5 e 6) nesta configuração apenas provam que, assim como nas configurações anteriores, o maior responsável pela queda na largura de banda é a geração de números aleatórios. Pode-se ver também que a criptografia sendo executada por *hardware* apresenta uma desempenho superior ao visto nas configurações 1 e 2, pois nestas configurações a criptografia era executada por *software* presente no *kernel* do Linux. O gasto computacional extra no R2 pode se desconsiderado, sendo irrelevante.

5.5.4 Configuração 4

A quarta e última configuração utiliza como placa de rede *off-board* a placa *Intel Server*. É esperado que os testes executados nesta configuração tenham o melhor resultado, uma vez que o *hardware* disponível é o melhor em relação às outras configurações. Os resultados dos testes para a quarta configuração se encontram na Tabela 4.

No teste 1, o valor de largura de banda obtido foi de 765Mbps, que é o maior valor possível nesta configuração. Assim como nas configurações anteriores, este valor serviu de base para todos os outros testes desta configuração. No teste 2, a largura de banda caiu para 751 Mbps, uma queda de 1,83%. Assim como nas configurações anteriores, está queda é insignificante.

O Teste 3 apresentou uma queda na largura de banda de 29,9%, sendo esse o melhor

Tabela 4: Resultados para Configuração 4

Intel i7 + Intel Server	BW(Mbps)	CPU(%)	Memória(%)	CPU RX(%)	Memória RX(%)
Teste 1 – Ausência de Módulo	762	0,03	1,72	0,08	1,72
Teste 2 – Módulo Vazio	760(↓0,26%)	0,086	1,72	0,10	1,72
Teste 3 – Modulo Completo	534(↓29,9%)	12,24	1,74	1,22	1,73
Teste 4 – Modulo Sem Num. Aleatórios	739(↓3,01%)	0,29	1,72	0,86	1,73
Teste 5 – Modulo Sem Criptografia	576(↓24,41%)	9,24	1,74	0,10	1,72
Teste 6 – Marca Sem Crip. E Num. Aleatórios	758(↓0,52%)	0,12	1,72	0,10	1,72

valor obtido dentre as configurações. O uso de CPU de 12,24% também foi o melhor resultado obtido dentre as demais configurações. Os valores de memória não tiveram grandes modificações e, portanto, podem ser desprezados. O Teste 5 isolou a geração de números aleatórios e confirma todos os resultados anteriores de que esta etapa é mais custosa e tem o maior impacto na queda da largura de banda. Do lado da recepção (R2), não há grandes alterações nos valores de CPU e memória e, portanto, não são analisados.

5.6 Análise das Métricas

No capítulo 2 são apresentadas cinco métricas que são utilizadas para medir um esquema de rastreamento de pacotes IP. Esta seção apresenta brevemente a avaliação do CTPM de acordo com essas métricas.

- Gasto Computacional: Considerando a Configuração 4, onde se obteve os melhores resultados, o CTPM gera um custo computacional de aproximadamente 12% na CPU dos roteadores. Provoca também uma queda na largura de banda de 29,9%. Valores esses que podem ser melhorados utilizando configurações de *hardware* mais poderosas.
- Envolvimento dos ISPs: O administrador de um SA somente é acionado quando seu cliente (uma vítima) detecta que está sob ataque. Nesse momento, o adminstrador do SA começa a verificar os TEHs presentes nos pacotes de ataque, para descobrir sua origem e contactar o adminstrador do SA de origem. Portanto, baixo envolvimento.
- Privacidade dos ISPs: O CTPM se baseia apenas nos roteadores de borda. Também não envolve os endereços IPs das interfaces desses roteadores. Portanto, essas in-

formações são mantidas em sigilo. São expostas somente informações previamente selecionadas pelo adminstrador. Logo, a privacidade do SA é mantida.

- Implantação Incremental: O CTPM prevê um plano de implantação incremental, onde inicialmente os ISPs Nível 1 iniciam a implantação. Este procedimento segue de forma incremental até que os ISPs de Nível superior possam se adaptar para implementarem o CTPM

- Complexidade do Processamento de Rastreamento: Como o TEH já carrega toda a informação das interfaces dos roteadores e os respectivos SAs, a complexidade é praticamente nula. Não é necessário executar um algoritmo de reconstrução do caminho.

5.7 Conclusão

Conforme o *hardware* evolui nos experimentos, os resultados também apresentam melhorias gradativas. Dentre as quatro configurações possíveis, o que se alterava entre elas era a CPU, a placa de rede *off-board* e a quantidade de memória em cada máquina. Analisando os resultados, é possível detectar como a CPU e a placa de rede *off-board* contribuíram para um melhor desempenho na quarta configuração.

Na primeira configuração, quando o teste 3 foi executado (cabeçalho de extensão completo), observou-se grande queda na largura de banda (cerca de 90%). Por isso, foi necessário detectar a parte do módulo que causa a maior queda na largura de banda. Assim, dois testes foram criados, um que media o custo da criptografia e outro que media o custo computacional da geração dos números aleatórios. Ao final foi constatado que a geração dos números aleatórios possui o maior custo computacional na criação do TEH. O *kernel* do *linux* já possui mecanismos de geração de números aleatórios por *software*, usando a função `get_random_bytes()`. É verificado que esta função utiliza bastante a CPU, e uma vez que o módulo todo é processado na CPU, o pacote não deixa a máquina enquanto todo o TEH não estiver completo. Assim, aumento do uso de CPU implica diretamente na queda de largura de banda.

Na configuração 2, a placa de rede *Realtek* foi trocada pela placa *off-board Intel Server*. O efeito conseguido foi um aumento no valor absoluto na largura de banda, de 496Mbps para 765Mbps. Porém, quando o teste 3 foi executado, a queda foi de 60,70%. Apesar do valor absoluto ter sido superior ao obtido na configuração 1, a queda ainda foi bastante grande. Assim, foi necessário evoluir na CPU. Na Configuração 3, foi feita uma troca de CPU, mas foi utilizada a placa de rede *off-board* Realtek. Nesta configuração, o objetivo foi ver como a evolução isolada da CPU contribuiu para o resultado final. Pelos resultados, observou-se que a CPU contribuiu para uma menor queda na largura de

Tabela 5: Comparação dos Resultados

	BW(Mbps)	CPU(%)	Memória(%)	CPU RX(%)	Memória RX(%)
Configuração 1 – E350 + Realtek	496 -> 22 (↓95,56%)	4,72 -> 43,19	6,75 -> 8,06	1,11 -> 7,16	7,09 -> 7,16
Configuração 2 – E350 + Intel Server	775 -> 301 (↓61,16%)	2,32 -> 41,17	6,75 -> 8,06	1,11 -> 6,88	7,09 -> 7,16
Configuração 3 – Intel i7 + Realtek	491 -> 332 (↓32,38%)	0,11 0 -> 13,16	1,72 -> 1,79	0,09 -> 1,45	1,72 -> 1,73
Configuração 4 – Intel i7 + Intel Server	762 -> 534(↓29,9%)	0,03 -> 12,24	1,72 -> 1,74	0,08 -> 1,22	1,72 -> 1,73

banda. Isso que dizer que a CPU mais poderosa refletiu positivamente no custo da criação do cabeçalho de extensão.

Na quarta Configuração, utilizou-se a CPU e a placa de rede mais poderosa. Os resultados obtidos foram, de todas as configurações, os melhores. O valor nominal da largura de banda foi de 765Mbps e, quando submetido ao teste completo, a queda foi de cerca de 30%. Ou seja, o impacto menor foi consequência da CPU mais poderosa, e o valor absoluto maior foi consequência da placa de rede *Intel Server*.

Sabendo que o maior responsável pela queda no desempenho do sistema foi justamente a geração de números aleatórios, trabalhos futuros podem investigar a implementação de geração de números aleatórios por *hardware*, ou o uso de alguma arquitetura computacional que já o faça. Como exemplo, os processadores da sexta geração da Intel i7 6700K já possuem instruções para gerar números aleatórios (6TH. . . , 2015). Acredita-se que, com isso, os custos computacionais para a criação do TEH passem a ser insignificantes, ou no pior caso, sejam aceitáveis e exijam um pequeno investimento de *hardware*. Assim, pode-se afirmar que as instruções de *hardware* tornam o uso da criptografia uma opção viável para esquemas de rastreamento. Além disso, sabendo que as máquinas utilizadas são de propósito geral, acredita-se que em dispositivos específicos (roteadores, neste caso) os resultados possam ser ainda melhores.

6 Conclusão

A estrutura de roteamento atual na Internet não faz usualmente nenhum tipo de verificação do endereço de origem IP, que pode ser forjado. Isso compromete a segurança na rede, viabilizando ataques cibernéticos totalmente anônimos. Este ataque pode ser um DDoS, que consiste em enviar grandes quantidades de pacotes IP para uma vítima, esgotando a sua capacidade de processar novos pacotes, inviabilizando usuários legítimos de poderem utilizar os serviços da vítima. Um *e-mail* com *spam* também é um pacote com conteúdo malicioso e pode ser enviado com um domínio forjado ou inexistente. Todos esses ataques mostram a fragilidade da atual estrutura de rede e a dificuldade em combatê-los. Hoje não existe uma forma de se impedir estes ataques e uma forma segura de detectar suas origens. Sabendo a origem destes ataques é possível implantar medidas adminstrativas onde os atacantes são punidos ou impedidos de usar a Internet.

Este trabalho propôs o CTPM, um esquema de rastreamento de pacotes IP, utilizando a marcação determinística como forma de rastreio. Essas marcas são feitas e analisadas exclusivamente pelos roteadores de borda de um SA. Os roteadores internos ignoram-nas. As marcas possuem duas informações, sendo uma identificadora e a outra autenticadora. A identificadora registra qual roteador acaba de criar e inserir a marca no pacote. Em cada novo SA que o pacote ingressa, o roteador anexa a sua identificação às demais existentes. Portanto, a marca se expande conforme o pacote trafega pela Internet. A informação autenticadora é responsável por garantir que a marca seja válida e que pertença ao pacote em questão. No caso de algum usuário mal-intencionado utilizar um *sniffer* para roubar o pacote e tentar usar a marca em outro pacote, a parte autenticadora irá revelar que a marca na verdade não pertence àquele pacote. Ao deixar o SA, a marca é descriptografada, para garantir que apenas o roteador na borda do outro SA tenha condições de ler o seu conteúdo.

Os resultados mostram que, considerando a Configuração 4 utilizada para os experimentos, existe um gasto computacional de cerca de 12% nos roteadores implementando o CTPM. A criptografia AES por *hardware* reduz significativamente o custo da criação do TEH.

No CTPM não existe custo para reconstrução do caminho percorrido pelo ataque, pois, com apenas um pacote, é possível rastrear a origem do mesmo. De fato, o TEH já carrega toda a informação necessária para reconstruir o caminho. Como o TEH registra informações apenas dos roteadores de borda, a estrutura interna do SA não é revelada. O CTPM também não faz uso dos endereços IPs das interfaces dos roteadores, mantendo-os em sigilo.

O envolvimento do SA só ocorre no momento da implantação do CTPM. É necessário fazer a modificação da pilha TCP/IPv6 nos roteadores além de implementar a troca de chaves AES com seus SAs vizinhos. No processo de reconstrução do caminho, o SA só é acionado quando a vítima verifica que está sob ataque. Nesse momento, o adminstrador da rede do SA da vítima passa a analisar o TEH presente nesses pacotes de ataque, a fim de encontrar a origem. O adminstrador do SA de origem deve ser contactado e informado de que sua rede está originando tráfego malicioso. Em qualquer esquema, esse procedimento deve ocorrer, pois a vítima não possui privilégios administrativos sobre o SA onde foi originado o ataque.

A implementação do CTPM depende da aceitação dos grandes *players* da Internet. É necessário mostrar os prejuízos causados pelos ataques cibernéticos a prestadores de serviço na Internet. Considerando-se apenas ataques DDoS, estima-se que 7000 ocorram diariamente (SINGH; SINGH; KUMAR, 2016). Além disso, a quantidade de emails *spam* é responsável por 70% do tráfego de *e-mails* na Internet (VERGELIS; SHCHERBAKOVA; DEMIDOVA, 2015). Uma Internet segura e com técnicas de rastreio permite auditorias, identificar causadores de ataques e também inibir ataques futuros, pois os potenciais causadores sabem que estão sendo rastreados. A adoção do CTPM produz uma série de benefícios para a segurança da Internet. O maior deles é a possibilidade de conhecer a origem precisa de todo pacote que circula pela Internet, a origem de *e-mails Spam* e de outras fontes de *malware*. Outro benefício advém da possibilidade de conhecer a rota trilhada pelos pacotes. Assim, o CTPM pode auxiliar na correção de erros de configuração BGP, garantindo que as rotas informadas pelos SAs são as rotas utilizadas de fato, para o tráfego de dados.

Trabalhos futuros

Como trabalho futuro, propõe-se a implementação do CTPM em roteadores cujas arquiteturas suportem criptografia e geração de números aleatórios por *hardware*. Este trabalho é essencial para mostrar que o CTPM pode ser totalmente viável como solução de segurança para a estrutura atual da Internet.

Existe também a possibilidade de se usar um FPGA (*Field Programmable Gate Array*) para a criação e manipulação do TEH. Esta seria uma solução inteiramente por *hardware*. A ideia seria ter o FPGA integrado aos roteadores de borda fazendo toda a parte de criação do TEH.

Este trabalho gerou a publicação "Carpinteiro, O.A.S., Francisco, P.S.L., Moreira E.M., Chain of Trust Packet Marking, *International Conference on Networks* (ICN), Lisboa, 21 - 25 de Fevereiro, 2016. Foi condecorado com o prêmio "Best Paper Award".

Referências

6TH Generation Intel® Core™ i7-6700K and i5-6600K Processors. *Intel Corporation - Product Brief*, 2015. Citado na página 66.

BELENKY, A.; ANSARI, N. On ip traceback. *IEEE Communications Magazine*, p. 142–153, julho 2003. Citado 6 vezes nas páginas 1, 3, 25, 31, 32 e 33.

BELENKY, A.; ANSARI, N. On deterministic packet marking. *Computer Networks 51*, p. 2677–2700, Janeiro 2007. Citado 5 vezes nas páginas 31, 32, 34, 50 e 51.

BENVENUTI, C. *Understanding Linux Network Internals*. Sebastopol, EUA: O'REILLY, 2006. Citado na página 18.

COMER, D. E. *Internetworking With TCP/IP Vol I: Principles, Protocols, and Architecture*. Nova Jersey, EUA: PEARSON, 2014. Citado 2 vezes nas páginas xi e 8.

DEERING, S.; HIDEN, R. Internet protocol, version 6 (ipv6) specification. *IETF - Request for Comments*, dec. 1998. Disponível em: <https://www.ietf.org/rfc-/rfc2460.txt>. Acesso em: 21 mar. 2016. Citado 4 vezes nas páginas 9, 10, 12 e 44.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE Transactions on Information Theory*, p. 644 – 654, nov 1976. Citado na página 44.

ELECTRONS, F. Linux cross reference. *Free Electrons*, 2015. Disponível em: <http://lxr.free-electrons.com/ident>. Acesso em: 21 mar. 2016. Citado na página 16.

FERGUSON, P.; SENIE, D. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. *IETF - Request for Comments*, mai 2000. Disponível em: <https://www.ietf.org/rfc/rfc2827.txt>. Acesso em: 21 mar. 2016. Citado na página 30.

GOODRICH, M. T. Probabilistic packet marking for large-scale ip traceback. *IEEE/ACM Transactions on Networking*, v. 16, p. 15–24, fevereiro 2008. Citado 2 vezes nas páginas 31 e 32.

HAGEN, S. *Ipv6 Essentials*. Sebastopol, EUA: O'REILLY, 2014. Citado 4 vezes nas páginas xi, 7, 11 e 12.

LEE, H. et al. Base: An incrementally deployable mechanism for viable ip spoofing prevention. *ASIA Conference on Computer and Communications Security*, p. 20–31, março 2007. Citado na página 29.

LIU, X. et al. Passport: Secure and adoptable source authentication. *5th USENIX Symposium on Networked Systems Design and Implementation*, p. 365–378, april 2008. Citado na página 29.

LOVE, R. *Linux Kernel Development*. Crawfordsville, EUA: Developer's Library, 2010. Citado na página 16.

MORRIS, J. The linux kernel cryptographic api. *Linux Journal*, abr. 2003. Disponível em: <http://www.linuxjournal.com/article/6451>. Acesso em: 21 mar. 2016. Citado na página 21.

MUELLER, S.; VASUT, M. Linux Kernel Crypto API. 2014. Disponível em: <http:/-/www.chronox.de/crypto-API/>. Acesso em: 21 mar. 2016. Citado 5 vezes nas páginas xi, 21, 22, 24 e 25.

PARASHAR, A.; RADHAKRISHNAN, R. Improved deterministic packet marking algorithm for ipv6 traceback. *Electronics and Communication Systems (ICECS), 2014 International Conference on*, p. 1–4, feb 2014. Citado 2 vezes nas páginas 29 e 30.

PARUCHURI, V. et al. Authenticated autonomous system traceback. *International Conference on Advanced Information Networking and Application*, v. 1, p. 406–413, março 2004. Citado 2 vezes nas páginas 31 e 32.

ROSEN, R. *Linux Kernel Networking*. Nova Iorque, EUA: APRESS, 2014. Citado 4 vezes nas páginas 18, 19, 53 e 54.

RUSSEL, R.; WELTE, H. Linux Netfilter Hacking HOWTO. jul. 2002. Disponível em: <http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html>. Acesso em: 21 mar. 2016. Citado 3 vezes nas páginas 17, 18 e 21.

SALZMAN, P. J.; BURIAN, M.; POMERANTZ, O. The Linux Kernel Module Programming Guide. mai. 2007. Disponível em: <http://www.tldp.org/LDP/lkmpg-/2.6/html/>. Acesso em: 21 mar. 2016. Citado 6 vezes nas páginas xi, 12, 13, 14, 15 e 16.

SAVAGE, S. et al. Network support for ip traceback. *IEEE/ACM Transactions on Networking*, v. 9, p. 226–237, junho 2001. Citado 4 vezes nas páginas 31, 32, 49 e 50.

SHUE, C. A.; GUPTA, M.; DAVY, M. P. Packet forwarding with source verification. *Elsevier Computer Networks 52*, p. 1567–1582, feb 2008. Citado 5 vezes nas páginas 1, 2, 29, 30 e 31.

SINGH, K.; SINGH, P.; KUMAR, K. A systematic review of ip traceback schemes for denial of service attacks. *Computer & Security*, v. 56, p. 111–139, fevereiro 2016. Citado 8 vezes nas páginas xi, 1, 2, 3, 25, 49, 50 e 68.

SINGH, R. K.; PUNDIR, S.; PILLI, E. S. Ipv6 packet traceback: A survey. *International Journal of Computer Applications*, v. 74, p. 31–35, julho 2013. Citado na página 34.

SUN, Y.-y. et al. Modified deterministic packet marking for ddos attack traceback in ipv6 network. *IEEE International Conference on Computer and Information Technology*, p. 245–248, agosto 2011. Citado 2 vezes nas páginas 31 e 33.

VERGELIS, M.; SHCHERBAKOVA, T.; DEMIDOVA, N. Kaspersky security bulletin. spam in 2014. *Secure List*, mar. 2015. Disponível em: <https://securelist.com/analysis-/kaspersky-security-bulletin/69225/kaspersky-security-bulletin-spam-in-2014/>. Acesso em: 21 mar. 2016. Citado na página 68.

VOHRA, Q.; CHEN, E. Bgp support for four-octet as number space. *IETF - Request for Comments*, mai. 2007. Disponível em: <https://tools.ietf.org/html/rfc4893>. Acesso em: 21 mar. 2016. Citado na página 37.

WELSCHENBACH, M. *Cryptography in C and C++*. Nova Iorque, EUA: APRESS, 2013. Citado na página 21.

WIKIPEDIA. Advanced encryption standard. mar. 2016. Disponível em: <https://en-.wikipedia.org/wiki/Advanced_Encryption_Standard>. Acesso em: 21 mar. 2016. Citado na página 21.

WINTHER, M. Tier 1 isps : What they are and why they are important. *IDC - Analyze the Future*, mai 2006. Disponível em: <http://www.us.ntt.net/downloads-/papers/IDC_Tier1_ISPs.pdf>. Acesso em: 21 mar. 2016. Citado na página 51.

WU, J.; REN, G.; LI, X. Source address validation: Architecture and protocol design. *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, p. 276 – 283, oct 2007. Citado 2 vezes nas páginas 29 e 30.

WU, J.; REN, G.; LI, X. Building a next generation internet with source address validation architecture. *Sci China Ser F-Inf Sci*, v. 51, p. 1681–1691, nov 2008. Citado 2 vezes nas páginas 29 e 30.

XIANG, Y.; ZHOU, W.; GUO, M. Flexible deterministic packet marking: An ip traceback system to find the real source of attacks. *IEEE Transactions on Parallel and Distributed Systems*, v. 20, p. 567–580, abril 2009. Citado 2 vezes nas páginas 31 e 33.

YOSHIFUJI, H. Basic functions for sk_buff. jul. 2007. Disponível em: <http://www-.skbuff.net/skbbasic.html>. Acesso em: 21 mar. 2016. Citado 4 vezes nas páginas xi, 18, 20 e 21.