Pontifícia Universidade Católica Do Rio De Janeiro

Extensão das interfaces de Sockets do Linux com Lua

Pedro Caldas Tammela

Projeto Final de Graduação

Centro Técnico Científico - CTC

Departamento de Informática

Curso de Graduação em Ciência da Computação

Rio de Janeiro, janeiro de 2019

Pedro Caldas Tammela

Extensão das interfaces de Sockets do Linux com Lua

Relatório de Projeto Final, apresentado ao programa **Ciência da Computação** da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Ciência da Computação.

Orientador: Prof. Noemi Rodriguez

Rio de Janeiro

janeiro de 2019.

"The most effective debugging tool is still careful thought,
coupled with judiciously placed print statements."
– Brian Kernighan

Agradecimentos

À minha família, meus amigos, meus professores e a comunidade de software livre pelo apoio ao longo desses anos.

Resumo

Caldas Tammela, Pedro. Rodriguez, Noemi. Extensão das interfaces de Sockets do Linux com Lua. Rio de Janeiro, 2019. 28 p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho foi implementado um módulo de kernel para o Linux com o intuito de estender chamadas de sistemas. Este módulo de kernel estende as chamadas de sistema *accept()*, *setsockopt()*, *getsockopt()*, *recvmsg()* e *close()* com uma versão adaptada da linguagem de programação Lua.

Palavras-chave

Sistemas Operacionais, Kernel, Lua, Scripting, Sockets

Abstract

Caldas Tammela, Pedro. Rodriguez, Noemi. Extension of the Linux's Socket interface with Lua. Rio de Janeiro, 2019. 28 p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

In this project a Linux kernel module was implemented aiming to extend system calls. This kernel module extends the system calls *accept()*, *setsockopt()*, *getsockopt()*, *recvmsg()* and *close()* with a modified version of the Lua programming language.

Keywords

Operating Systems, Kernel, Lua, Scripting, Sockets

Sumário

1 Introdução

A arquitetura de sistemas operacionais modernos[22] gira em torno de filosofias alternativas: sistemas com o núcleo monolítico[30] e sistemas com micro-núcleo[29]. Historicamente existem sistemas que misturam as duas filosofias, como o XNU[35], e filosofias extremistas como o exo-núcleo[7].

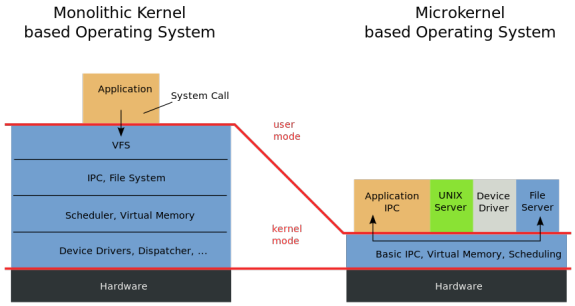

Figura 1: Estrutura de um kernel monolítico e micro-kernel, respectivamente [33].

Em ambas filosofias, a comunicação entre o modo privilegiado (modo de kernel) e o modo de usuário de um sistema operacional é feita através de chamadas de sistemas. Uma chamada de sistema é um mecanismo programático através do qual um programa de computador requisita um serviço de um sistema operacional[34].

Neste projeto, criamos um artefato para estender algumas chamadas de sistemas, no sistema operacional Linux, em estruturas sockets utilizando Lua[10] no modo privilegiado do sistema operacional. Utilizamos um artificio recentemente incorporado ao Linux que permite a modificação das chamadas de sistemas sobre estruturas sockets sem a necessidade de modificar a implementação original.

Para ter acesso a Lua dentro do sistema operacional, utilizamos um *port* da implementação da linguagem para o kernel do Linux[12] e fazemos uso da API da linguagem C[11] oferecida pela própria linguagem Lua[10].

O artefato pode ser utilizado para delegar pequenas tarefas para serem realizadas a nível de kernel. Neste projeto utilizamos o artefato para validar mensagens HTTP no nível de kernel em um servidor HTTP.

2 Situação Atual

Atualmente, a possibilidade de adaptar funções de um sistema operacional a uma aplicação vem sendo debatida e investigada extensivamente[14][13][8]. No kernel do Linux, em 2014 foi introduzido o primeiro *patch* para a generalização da máquina

virtual BPF, dando início a um novo trabalho que ficou conhecido como eBPF (*Extended Berkeley Packet Filter*)[1]. No kernel do NetBSD, em 2011, foi introduzido um *port* da linguagem Lua[16], possibilitando a extensibilidade do sistema operacional usando Lua e a API C de Lua.

Apesar de já existirem os meios para se estender as funcionalidades das chamadas de sistemas de diversos sistemas operacionais, apenas o Linux possui uma arquitetura apropriada para a extensibilidade através de módulos de kernel.

2.1 Lua no Kernel

Em 2011, a tese de mestrado "Lunatik: Scripting de Kernel de Sistema Operacional com Lua"[19] apresentou uma implementação da linguagem Lua, versão 5.1, para o kernel do Linux. O interpretador da linguagem teve de ser modificado para recusar numerais de ponto flutuante e a máquina virtual de Lua teve de ser modificada para suportar a representação interna de numerais utilizando inteiros. Nas versões anteriores a 5.3, a máquina virtual de Lua representava internamente todos os numerais utilizando precisão dupla de ponto flutuante. Em modo de kernel, os registradores de ponto flutuantes não podem ser acessados.

Com o lançamento, em janeiro de 2015, da versão 5.3 da linguagem Lua, foi introduzido o suporte a representação de numerais com inteiros para a máquina virtual de Lua, e foi necessária a atualização da primeiro *port* da linguagem. A nova implementação para o Linux[12] foi atualizada com base na implementação feita para o kernel do NetBSD[16][24] e passou apenas a restringir numerais em ponto flutuante pelo interpretador de Lua.

Apesar de algumas restrições, a maior parte da biblioteca padrão de Lua, como corotinas e manipulação de strings, está disponível. A função *require()* em Lua, que permite o carregamento de módulos dinâmicamente, ainda está indisponível. Essa função é implementada usando funções da biblioteca de linkagem dinâmica (*libdl*) no caso do carregamento de uma biblioteca *shared object*, e utiliza funções em arquivos para o carregamento de scripts Lua. Atualmente, Lua no Kernel oferece o carregamento de bibliotecas através apenas da API de C de Lua, utilizando a função *luaL_requiref()*. Estão indisponíveis, também, funções que operam sobre arquivos, tais como *loadfile()* e a biblioteca *io*.

No trabalho *lunatik-ng*[5], uma continuação do trabalho apresentado na tese "Lunatik: Scripting de Kernel de Sistema Operacional com Lua"[19], é apresentado

o *port* da linguagem Lua e uma nova chamada de sistema para o kernel Linux. A chamada de sistema *sys_ lua()* recebe um script Lua e o executa dentro do kernel do Linux. Existe a opção de realizar a execução síncrona e assíncrona, isto é, o script é executado durante a chamada de sistema ou é postergado para ser executado em uma fila de trabalhos (*workqueues*). Em outros trabalhos que se baseiam na versão original do *lunatik*, como o *lunatik* desenvolvido no LabLua[12], é apresentado apenas o *port* da linguagem, sem mecanismos de controle e execução de scripts pelo espaço de usuário.

2.2 eBPF

A máquina virtual BPF está presente nos variantes do Unix há 20 anos. Seu principal uso é a filtragem de pacotes utilizando filtros fornecidos por aplicações no nível de usuário. Desde de que foi descrita em 1992 por Jacobson et al., a implementação passou por poucas mudanças e não acompanhou as evoluções dos hardwares de rede, caindo em desuso em prol de outras soluções como o Netfilter e o DPDK.

O Linux oferece uma máquina virtual dentro do kernel chamada de eBPF. A máquina virtual eBPF é uma modificação da máquina virtual BPF com o objetivo de fornecer uma arquitetura moderna, mais eficiente e mais abrangente. A máquina virtual originalmente era usada apenas em filtros de pacotes, possuía 2 registradores de 32 bits e executava 22 instruções diferentes. A arquitetura atual possui 11 registradores de 64 bits[9], estende o conjunto de instruções, é até 4 vezes mais rápida do que a arquitetura antiga e pode ser usada em diversas aplicações[17] no kernel do Linux.

O bytecode eBPF pode ser gerado utilizando a infraestrutura de compilação LLVM[9] ou através de ferramentas mantidas pela comunidade open source, como o *bcc*[3]. Um programa em espaço de usuário passa um bytecode eBPF para o kernel através da chamada de sistema *bpf()*. O verificador eBPF analisa estaticamente[9] o bytecode recebido em busca de violações de regras e retorna indicações de erro ao espaço de usuário em caso de alguma violação.

O verificador de bytecode eBPF proíbe *loops*, resultando num modelo de computação não Turing completo[17] e bem restrito. O verificador analisa também, a existência de instruções inalcançáveis (*unreachable instructions*), marca registradores não inicializados e recusa programas que os leem e simula a execução do programa, verificando o estado da pilha e dos registradores da máquina virtual[9]. Isso tudo

é para garantir que um programa eBPF não vá interferir negativamente no kernel do Linux, seja reiniciando o sistema com falhas de segmentação, *starvation* de uma CPU ou o vazamento de informações com um programa eBPF malicioso.

O verificador de bytecode eBPF também proíbe programas com mais de 4096 instruções. Para contornar esta limitação, é oferecido um mecanismo de *jump* para outro programa eBPF, porém restrito a poucos *jumps*.

2.3 Lua no Kernel e eBPF

Lua no Kernel é composto por duas partes, a primeira parte é a linguagem de programação Lua e a segunda parte é a máquina virtual Lua. O eBPF também é composto por duas partes, a primeira parte é um subconjunto da linguagem C[11] e a segunda parte é a máquina virtual eBPF. Lua no Kernel e eBPF se diferenciam na expressividade e poder de computação. Lua é uma linguagem Turing completa, multi paradigma e extensível através de bibliotecas, já o subconjunto de C utilizado pelo eBPF não é Turing completo, não é multi paradigma e é sintaticamente desconfortável. Lua no Kernel é uma abordagem de cima para baixo, onde se oferece uma linguagem de programação para um ambiente de baixo nível. eBPF é uma abordagem de baixo para cima, onde se oferece uma máquina virtual para um ambiente de baixo nível mas com uma abstração pobre.

```
int handle_ingress(struct __sk_buff *skb)
{
    void *data = (void *)(long)skb->data;
    struct eth_hdr *eth = data;
    struct iphdr *iph = data + sizeof(*eth);
    struct udphdr *udp = data + sizeof(*eth) + sizeof(*iph);
    void *data_end = (void *)(long)skb->data_end;

    /* single length check */
    if (data + sizeof(*eth) + sizeof(*iph) + sizeof(*udp) > data_end)
        return 0;

    if (eth->h_proto != htons(ETH_P_IP))
        return 0;
    if (iph->protocol != IPPROTO_UDP || iph->ihl != 5)
        return 0;
    if (ip_is_fragment(iph))
        return 0;
    if (udp->dest == htons(DEFAULT_PKTGEN_UDP_PORT))
        return TC_ACT_SHOT;
    return 0;
}
```

Cód. 1: Um programa eBPF para filtrar pacotes UDP com destino a porta 9. [2]

Lua não implementa um verificador de bytecode e espera que o programador que expõe os estados Lua ao usuário tome as devidas medidas, como chamadas pro-

tegidas[15], para a construção de um ambiente *sand-boxed*. Por ser uma linguagem de alto nível naturalmente não sofre com problemas encontrados pelo subconjunto de C utilizado pelo eBPF, tais como acesso indevido a regiões da memória, corrompimento da pilha de execução e *buffer overflows*. Ao interagir com C, tanto Lua quanto eBPF podem estar sujeitos a falhas de segurança. [25] [4]

Programas eBPF garantidamente param pois a linguagem não é Turing completa. Para forçarmos um programa Lua a parar, podemos utilizar a função *debug.sethook()* da biblioteca *debug* e assinalar um cronômetro que quando zerado força o término do programa.

3 Objetivos

Neste projeto, temos como objetivo estender a interface de sockets oferecida pelo kernel do Linux com scripts Lua, de maneira que seja adaptável as necessidades da aplicação que utiliza o artefato. Dentre as alternativas usadas em diversas aplicações na comunidade de software aberto, escolhemos Lua no Kernel por ser a alternativa melhor documentada, mais fácil de realizar *debugging*, possuir boas abstrações e uma biblioteca nativa para manipulação de strings, além de contar com uma API C para interação com o estado Lua.

Para o *port* de Lua no Kernel, decidimos escolher o projeto *lunatik* desenvolvido no LabLua[12]. Escolhemos este pois além de estar atualizado para a versão de Lua 5.3.4, não precisamos de uma nova chamada de sistema para realizar a comunicação entre o espaço de usuário e Lua no Kernel.

Vemos como possíveis usuários deste projeto programadores, ou aplicações, que gostariam de delegar tarefas para o sistema operacional ao invocar uma chamada de sistema que opera sobre a estrutura socket, tal como a validação de mensagens da camada de aplicação do modelo TCP/IP[28] e o *cache* estático de requisições HTTP em servidores web.

Esperamos que a arquitetura proposta neste projeto consiga trazer os benefícios de uma linguagem dinâmica, com coletor de lixo e baixa curva de aprendizado para um ambiente de baixo nível com um desempenho aceitável.

4 Atividades realizadas

Ao longo deste projeto realizamos diversos estudos para definirmos a arquitetura do artefato. Assim que definimos o modelo de comunicação entre o espaço de usuário e o espaço de kernel comparamos possíveis soluções para desenharmos a arquitetura inicial.

Depois que desenhamos a arquitetura inicial, iniciamos o desenvolvimento de um protótipo para verificarmos as limitações e assegurarmos a viabilidade do projeto.

Dedicamos as seções seguintes para os assuntos que mais demandaram tempo e que influenciaram profundamente a arquitetura do artefato produzido.

4.1 Estudo dos modelos de comunicação

O kernel do Linux oferece APIs para que o espaço de usuário e o espaço de kernel possam realizar troca de dados. Temos a opção de usar objetos que simulam arquivos em pseudo sistemas de arquivos especiais, como o *procfs* e o *devfs*. Existe também um protocolo para socket especialmente desenvolvido para a comunicação entre ambos os espaços chamado de Netlink. Por último, temos as chamadas de sistemas multiplexadoras, como *ioctl()*, *prctl()* e *setsockopt()*, que oferecem uma assinatura genérica para manipular estruturas internas do sistema operacional, dependendo dos argumentos.

4.1.1 Pseudo Sistema de Arquivos

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    /* ... */
};
```

Cód. 2: A struct de operação sob arquivos

O *procfs*[32] é um pseudo sistema de arquivos, geralmente montado no diretório *'/proc'*, que expõe uma interface para estruturas internas do kernel. Comumente usado para a obtenção de informações sobre estruturas internas do kernel como mapeamento de memória de um processo, informações sobre sockets criados e uso de memória pelo SLAB allocator, a interface permite a comunicação duplex através dos nós expostos. O criador do nó no *procfs* pode definir funções de callback utilizando

a estrutura *file_ operations*. As funções serão chamadas pelo kernel quando o nó é operado por chamadas de sistemas como *open()*, *read()*, *write()* e *close()*.

Um character device é o nome dado a um nó no pseudo sistema de arquivos *devfs*[27], geralmente montado em *'/dev'*, que representa um dispositivo de caracteres, ou um processador de caracteres como terminais e portas seriais. O *devfs* é um meio de expor um pseudo dispositivo diretamente ao usuário. Cada nó possui suporte a chamadas de sistemas como *open()*, *read()*, *write()* e *close()* caso o criador as disponibilize através da estrutura *file_ operations*.

O *procfs* e o *devfs* são idênticos no quesito funcionalidade. Semanticamente são diferentes, o *procfs* pode ser desabilitado via configuração e representa um sistema de arquivo para a configuração do sistema operacional. O *devfs* representa um sistema de arquivo para a interação com os dispositivos expostos como nós.

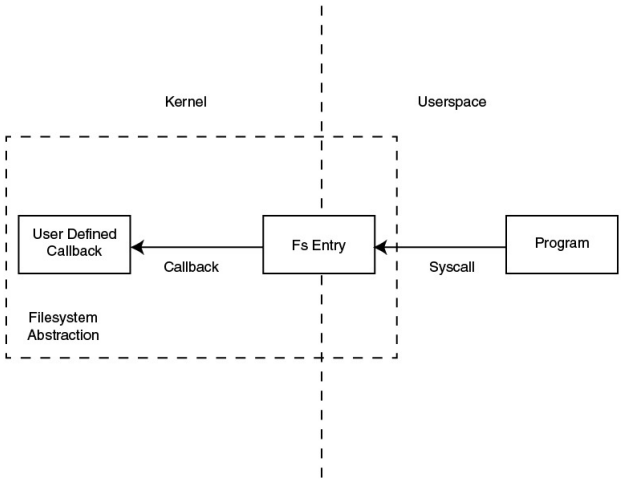

Figura 2: Abstração de um sistema de arquivos.

4.1.2 Netlink

O Netlink[31] foi originalmente desenvolvido para uso na pilha de rede do Linux, tendo suporte nativo a protocolos voltados para a configuração do firewall do Linux (Netfilter[26]) e a obtenção de informações da tabela de rotas. O socket Netlink é exclusivamente desenhado para a comunicação entre kernel e espaço de usuário, o que faz com que os pacotes sejam endereçados utilizando Port IDs, que são identificadores virtuais de pontos de comunicação. As mensagens são passadas como datagramas de dados, com um limite de *maximum transmission unit* (MTU) de 32 kilobytes.

O Netlink oferece ao usuário a opção de estender o tratamento de mensagens,

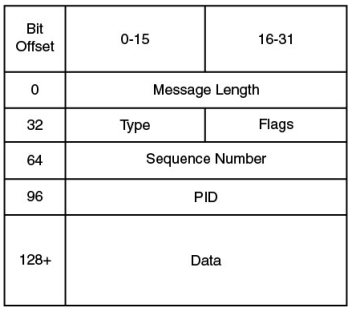

Figura 3: Cabeçalho de um datagrama Netlink. [31]

assim criando sub protocolos de comunicação. O espaço de usuário cria um socket especificando o tipo de tratamento de cada mensagem e no kernel cria o socket tratador, especificando a função de tratamento usando a estrutura *netlink_ kernel_ cfg*.

Através dos Port IDs, as mensagens enviadas pelo socket tratador dentro do kernel são entregues a seus respectivos sockets de espaço de usuário. Esse mecanismo é abstraído do modelo de mensagens, visto que para o usuário mandar uma mensagem para o kernel basta usar o Port ID 0. Os Port IDs são escolhidos pelo socket de usuário. A passagem de mensagem pelo usuário é feita de forma transparente usando os métodos *send()* ou *sendto()*, assim como o recebimento de mensagens pelo usuário, que também é feito de forma transparente usando os métodos *receive()* e *receivefrom()*. Toda a comunicação interna realizada pelo Netlink é abstraída na implementação do protocolo.

O Netlink conta com mecanismos de comunicação que não estão presentes em pseudo sistema de arquivos como multicasting de mensagens, hierarquia de grupos de comunicação e um limite maior de transmissão de dados entre o kernel e o espaço de usuário, que propiciam um ambiente sem a necessidade de serialização de operações.

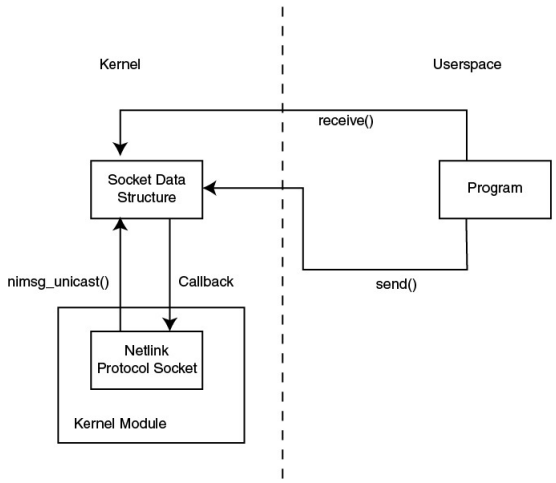

Figura 4: Um modelo de comunicação com Netlink, em alto nível.

4.1.3 Chamadas de Sistema

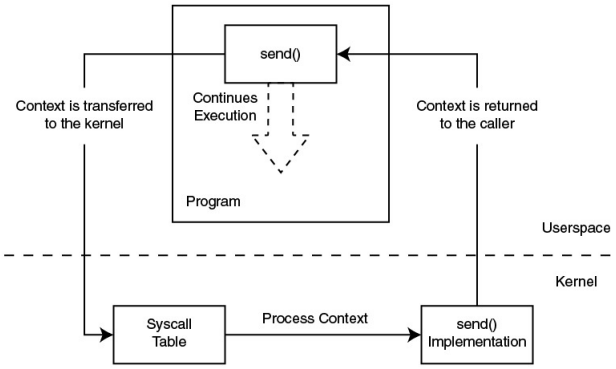

Figura 5: O modelo de chamada de sistemas do Linux.

Uma chamada de sistema é o método mais transparente de comunicação com o kernel, o contexto de processamento é passado para o kernel imediatamente diferentemente dos métodos anteriores, mas impõe muito mais esforço de implementação e de design[34].

Chamadas de sistema são dependentes de plataforma e precisam ser mantidas indefinidamente. O mal design de uma chamada de sistema leva a aparição de assinaturas de funções com a mesma semântica, mas com parâmetros diferentes, como *accept()* e *accept4()*.

Para permitir a criação de novas funcionalidades utilizando chamadas de sistema, o kernel disponibiliza chamadas de sistema capazes de multiplexação como *ioctl()* e *prctl()*. Ambas são complexas o suficiente para serem evitadas em favor

dos métodos anteriores. Apesar disso a arquitetura de ambas é bem parecida. A definição da função segue a lógica de um pipeline para o kernel, onde é necessário apenas o preenchimento do código da operação requisitada e de eventuais dados para processamento. No kernel o processamento é multiplexado para o tratador necessário.

```
int ioctl(int fd, unsigned long request, ...);
int prctl(int option, unsigned long arg2, unsigned long arg3,
    unsigned long arg4, unsigned long arg5);
```

Cód. 3: assinaturas das chamadas de sistema *ioctl()* e *prctl()*

No caso particular de sockets, existe outra chamada de sistema multiplexadora, a *setsockopt()*. A *setsockopt()* é uma chamada de sistema para mexer em configurações internas de um socket, como alterar os valores do tamanho dos buffers de envio e recebimento de mensagens. Recentemente foi demonstrado o uso da *setsockopt()* como um mecanismo de passagem de informação para o processamento de requisições TLS[6] dentro do kernel[23].

Por operar em diversos níveis da estrutura socket, a *setsockopt()*, é a escolha natural para se adicionar extensibilidade com Lua, pois além de suporte a essa nova funcionalidade, propicia que operações como *accept()* e *receive()* sejam feitas de forma transparente no nível de usuário, recebendo seus respectivos retornos após o processamento em Lua[10].

```
int setsockopt(int socket, int level, int option_name,
    const void *option_value, socklen_t option_len);
```

Cód. 4: Assinatura da chamada de sistema *setsockopt()*

4.2 Protótipo

Levamos em consideração que um processo no espaço de usuário não deve interferir na semântica da chamada de sistema do sistema operacional para todos os processos, assim qualquer scripting da chamada de sistema deve afetar apenas estruturas criadas e pertencentes ao processo de espaço de usuário que deseja modificar suas chamadas de sistema. Concluímos que a melhor forma de comunicação, para o projeto, era utilizarmos um artifício da chamada de sistema *setsockopt()*, o *Upper Layer Protocol* (ULP), que permite ao espaço de usuário modificar as chamadas de sistemas sobre as estruturas socket sem afetar a implementação original do sistema operacional. Qualquer modificação é contida nas estruturas sockets pertencentes ao processo.

Após a definição do método de comunicação entre os dois espaços, estudamos a arquitetura ULP e concluímos que é possível criar um módulo de kernel para estender as chamadas de sistemas que operam sobre a estrutura socket utilizando Lua dentro do kernel do Linux. Para rascunhar a arquitetura final, foi desenvolvido um protótipo de um servidor HTTP com base na arquitetura ULP que invoca Lua dentro da chamada de sistema *recvmsg()*. O protótipo prepara o socket de requisições com os scripts Lua através de chamadas de sistema *setsockopt()*, porém o preparo do socket de requisição a cada nova conexão impõe um custo significativo. Cerca de 50% do tempo total de execução é gasto nessa operação.

Neste protótipo, há um custo significativo para carregar os scripts Lua. Como ativamos o ULP apenas no socket de requisição, temos a preparação do socket dentro do loop de processamento de requisições, realizando uma cópia entre espaço de usuário e espaço de kernel, uma alocação dinâmica para comportar o script e a execução do script no estado Lua correspondente ao socket a cada nova conexão.

5 Projeto e especificação do sistema

5.1 Inicialização

```
static struct tcp_ulp_ops ss_tcpulp_ops __read_mostly = {
    .name           = "lua",
    .uid            = TCP_ULP_LUA,
    .user_visible   = true,
    .owner          = THIS_MODULE,
    .init           = sk_init
};

static int __init ss_tcp_register(void)
{
    tcp_register_ulp(&ss_tcpulp_ops);
    return 0;
}

static void __exit ss_tcp_unregister(void)
{
    tcp_unregister_ulp(&ss_tcpulp_ops);
}

module_init(ss_tcp_register);
module_exit(ss_tcp_unregister);
```

Cód. 5: Implementação da inicialização do módulo e do ULP *lua*.

O projeto produziu como artefato um módulo de kernel para o Linux. O módulo de kernel é carregado utilizando a ferramenta *insmod*. No código 5, temos a definição do tipo estruturado para o registro do ULP "lua" e a função de inicialização. A função de inicialização é automaticamente chamada quando o módulo é carregado.

Ao ser carregado, o módulo registra um novo ULP chamado de "lua" e uma função de callback para ser invocada sobre o socket que ativa o ULP. A função de callback inicializa um novo estado Lua, associa um estado Lua ao socket e registra as chamadas de sistemas do módulo no socket. O código 6 exibe a implementação da função de callback *sk_ init()*. Associamos também ao estado Lua um contexto, que atualmente é uma estrutura que contém apenas o campo com o nome da função Lua que deve ser invocada na chamada de sistema *recvmsg()*.

```
static struct proto *sys;

static int sk_init(struct sock *sk)
{
   lua_State *L;
   struct context *ctx;
   struct context **area;

   if (sk->sk_family != AF_INET)
      return -ENOTSUPP;

   ctx = kmalloc(sizeof(struct context), GFP_KERNEL);
   if (ctx == NULL)
      return -ENOMEM;

   ctx->entry[0] = '\0';

   L = luaL_newstate();
   if (L == NULL)
      return -ENOMEM;

   luaL_openlibs(L);
   inet_csk(sk)->icsk_ulp_data = (void *)L;
   area = (struct context **)lua_getextraspace(L);
   *area = ctx;
   sys = sk->sk_prot;
   register_funcs(&sk->sk_prot);

   return 0;
}
```

Cód. 6: Implementação da função *sk_ init()*

Para registrarmos as novas chamadas de sistema, precisamos antes salvar a estrutura *proto* original, que guarda ponteiros para as chamadas de sistema do protocolo associado ao socket, na variável global *sys* para utilizarmos as chamadas de sistema originais. Depois basta registrarmos as implementações das chamadas de sistemas modificadas, demonstrado no código 7.

```
static void register_funcs(struct proto **skp)
{
   new = *sys;
   new.accept = ss_accept;
   new.recvmsg = ss_recvmsg;
   new.setsockopt = ss_setsockopt;
   new.getsockopt = ss_getsockopt;
   new.close = ss_close;
   *skp = &new;
}
```

Cód. 7: Implementação da função *register_funcs()*

5.2 Preparação dos sockets

Com base nas atividades realizadas, desenhamos a arquitetura do projeto para evitarmos os problemas encontrados no primeiro protótipo. Para isso, passamos a permitir a carga de scripts apenas em sockets no estado *TCP_LISTEN*, ou seja, sockets que apenas escutam por novas conexões em uma porta. Assim retiramos do loop de processamento de requisições a necessidade de preparar todo socket de requisição a cada nova conexão.

```
/* setup lua */
err = setsockopt(listener, SOL_TCP, TCP_ULP, "lua", sizeof("lua"));
if (err == -1) {
   return -1;
}
/* load scripts to kernel */
err = setsockopt(listener, SOL_LUA, SS_LUA_LOADSCRIPT, buff, sz);
if (err == -1) {
   return -1;
}
/* set Lua entry point inside the system call */
err = setsockopt(listener, SOL_LUA, SS_LUA_ENTRYPOINT, "process", sizeof("process"));
if (err == -1) {
   return -1;
}
```

Cód. 8: Preparação de um socket por uma aplicação de usuário

O código 8 é executado por uma aplicação no nível de usuário. Iniciamos o ULP "lua" no socket que aceitará as novas conexões através da função *setsockopt()*. A partir deste ponto da aplicação do usuário, todo socket que é derivado do socket com o ULP "lua", incluindo o mesmo, possui as chamadas de sistemas modificadas pelo módulo de kernel. Após a inicialização, carregamos os scripts para o kernel, onde são executados dentro do estado Lua associado ao socket, e estabelecemos a função de entrada no estado Lua.

```
static struct sock *ss_accept(struct sock *sk, int flags, int *err, bool kern)
{
   struct sock *reqsk = sys->accept(sk, flags, err, kern);

   if (reqsk == NULL)
      return NULL;

   try_module_get(THIS_MODULE);
   inet_csk(reqsk)->icsk_ulp_data = sk_ulp_data(sk);

   return reqsk;
}
```

Cód. 9: Implementação da chamada de sistema *accept()*

A chamada de sistema *accept()* é modificada para que o socket retornado numa nova conexão bem sucedida herde o estado Lua já preparado pela aplicação no nível de usuário.

A arquitetura atual assume que o processamento dos sockets é realizado de maneira serializada pelo espaço de usuário. O estado Lua associado a todos os sockets de requisição é o mesmo, logo qualquer paralelismo introduzido pelo espaço de usuário poderá acarretar em condições de corrida.

5.3 Lua dentro de uma chamada de sistema

No artefato, scripts Lua são diretamente invocados na chamada de sistema *recvmsg()*.

```
static int ss_recvmsg(struct sock *sk, struct msghdr *msg, size_t len,
      int nonblock, int flags, int *addr_len)
{
   lua_State *L = sk_ulp_data(sk);
   struct context *ctx = sk_ctx(sk);
   int perr = 0;
   int data = LUA_NOREF;
   struct sk_buff *skb;
   struct tcphdr *hdr;

   lock_sock(sk);

   if (skb_queue_empty(&sk->sk_receive_queue) || !ctx->entry[0])
      goto out;

   if (lua_getglobal(L, ctx->entry) != LUA_TFUNCTION)
      goto out;

   skb = skb_peek_tail(&sk->sk_receive_queue);
   hdr = tcp_hdr(skb);
   if (!hdr->psh)
      goto out;
   lua_pushlstring(L, skb->data, skb->len);
   perr = lua_pcall(L, 1, 1, 0);
   if (perr) {
      pr_err("%s\n", lua_tostring(L, -1));
      goto out;
   }

   if (lua_toboolean(L, -1) == false)
      goto bad;

out:
   release_sock(sk);
   return sys->recvmsg(sk, msg, len, nonblock, flags, addr_len);

bad:
   release_sock(sk);
   return -ECONNREFUSED;
}
```

Cód. 10: Implementação da chamada de sistema *recvmsg()*

Na implementação dessa função, Lua só é invocada em certas condições. A primeira condição é de que o usuário tenha definido um ponto de entrada, uma função Lua, para ser invocada. Verificamos também se o pacote de rede TCP reconstruído é do tipo PSH, indicando que o cliente enviou todos os dados, evitando assim uma chamada a Lua para buffers da rede vazios ou incompletos. Quaisquer erros reportados pelo estado Lua, na chamada protegida, são logados no buffer de mensagens do kernel do Linux, podendo ser visualizados usando a ferramenta *dmesg*.

A chamada de sistema *recvmsg()* tem de lidar com diversos casos especiais como pacotes urgentes, flags e cópia para o espaço de usuário. Para ainda mantermos a

funcionalidade correta da chamada, quando os scripts Lua retornam *true* é invocada a implementação original da chamada de sistema.

O código de erro *ECONNREFUSED* indica que a conexão foi recusada, ou seja o estado Lua retornou *false*, e o socket deve ser fechado pela aplicação. Ao usar o artefato, a aplicação servidora tem de tratar essa possibilidade de retorno.

5.4 Exemplo de aplicação

Nesta seção vamos demonstrar a execução do artefato do ponto de vista da aplicação de usuário.

O código 11 aceita conexões a partir do socket *listener* e recebe uma mensagem do socket de requisições. O socket *listener* foi previamente preparado para usar o ULP "lua" com o script Lua 12, assim os sockets retornados a variável *sock* possuem o ULP "lua" e invocam as chamadas de sistemas modificadas pelo artefato.

```
struct sockaddr_in cl;
socklen_t len;
while(1) {
   int sock = accept(listener, (struct sockaddr *) &cl, &len);
   if (sock == -1) {
      raise_err();
      return -1;
   }

   size_t msgsz = recv(sock, msg, 8192, 0);
   if (msgsz == 0) {
      close(sock);
      continue;
   }

   if (msgsz == -1) {
      if (errno == ECONNREFUSED) {
         close(sock);
         continue;
      }
      raise_err();
      break;
   }

  close(sock);
}
```

Cód. 11: *while* loop de um servidor com o ULP "lua"

```
function process()
  print("Hello, Kernel World!")
  return true
end
```

Cód. 12: Um script Lua simples

Ao invocar a chamada de sistema *recv()* no socket *sock*, a função *process()* do código 12 é executada se uma mensagem completa está presente, imprimindo

no buffer de mensagens do kernel do Linux a frase “Hello, Kernel World!”. Neste exemplo, o script Lua sempre retorna *true* e não pula a execução da chamada de sistema *recv()* original. No caso de um retorno *false* ou *nil*, a aplicação precisa tratar o caso do erro *ECONNREFUSED* separadamente.

6 Implementação e avaliação

6.1 Testes funcionais

O artefato desenvolvido neste projeto foi testado com dois objetivos em mente. O primeiro objetivo é a estabilidade. Em artefatos voltados para sistemas operacionais qualquer falha tem como consequência direta uma reinicialização não esperada do sistema. O segundo objetivo é a análise de desempenho do artefato quando comparado a um artefato que utiliza apenas o espaço de usuário para processar as mensagens recebidas pelo socket.

Para testarmos a estabilidade do artefato, consideramos o cenário de um servidor TCP simples atendendo muitas requisições de vários clientes. Desenvolvemos o servidor para criar um socket com o ULP “lua” e criamos um script simples para ser invocado junto ao recebimento de uma mensagem no socket de requisição. Desenvolvemos, também, um servidor que não usa o ULP “lua” mas invoca Lua a cada mensagem recebida pela chamada de sistema *recvmsg()*. Ambas as implementações fecham o socket sem o envio de qualquer mensagem ao cliente.

A ferramenta *curl* se conecta ao servidor e faz um requisição HTTP GET, permitindo que o usuário defina campos do cabeçalho HTTP. Usamos essa funcionalidade para enviarmos diversos pacotes mal formados intencionalmente com o objetivo de testar a estabilidade e as funcionalidades dos scripts desenvolvidos.

6.2 Aplicações

Uma grande vantagem que o artefato produzido proporciona é a possibilidade da camada de aplicação inspecionar os dados sem necessidade de copiá-los dos buffers de rede. Delegando tarefas para o sistema operacional, como servimento de requisições estáticas e validação de cabeçalhos HTTP, podemos dividir o trabalho entre o sistema operacional e aplicações no espaço de usuário, evitando cópias para o espaço de usuário.

Todo servidor precisa realizar a validação do cabeçalho HTTP recebido. Neste

projeto desenvolvemos um script Lua para validar cabeçalhos HTTP e comparamos o desempenho da solução utilizando os 2 servidores desenvolvidos para os testes de estabilidade.

6.2.1 Validação de cabeçalho HTTP

Lua possui em suas bibliotecas padrão uma biblioteca para o processamento de strings. Utilizando essa biblioteca, implementamos heurísticas para as validações de requisições HTTP. A validação de requisições HTTP é muito importante devido a falhas de segurança na especificação do protocolo.

Um exemplo é o método de requisição HTTP *PUT*. O método *PUT* cria um novo recurso ou reescreve um recurso com o conteúdo da mensagem enviada[18]. É extremamente perigoso quando não autenticado corretamente, podendo criar brechas de seguranças gravíssimas.

O código 13 implementa 3 funções de sanitização, além de construir uma tabela com os elementos que foram analisados. Não foi necessário o uso de nenhuma biblioteca externa, apenas as bibliotecas padrão da linguagem Lua.

A função *parse.requestmethod()* faz a captura do caminho requerido pelo cliente. Na implementação estamos considerando apenas o método HTTP GET e quaisquer outros métodos são considerados falha.

A função *parse.requestfields()* preenche uma tabela com todos os campos de requisição do cabeçalho HTTP. O padrão de captura garante que não será considerado campos mal formados.

A função *parse.sanitycheck()* verifica se a requisição recebida tem alguma marca de fim de linha de um cabeçalho HTTP.

```lua
local parse = {}

function parse.sanitycheck(s)
   if not string.find(s, "\r\n") then return end
   return true
end

function parse.requestmethod(s)
   local ok = string.match(s, "^GET")
   if not ok then return end
   local path = string.match(s, "GET (%g+) HTTP")
   return path
end

function parse.requestfields(s, header)
   local size = 0
   for k, v in string.gmatch(s, "(%g*):%s*([^%c]+)") do
      header[k] = v
      size = size + 1
   end
   return size ~= 0, size
end

function parse.http(s)
   local header = {}
   if parse.sanitycheck(s) then
      header.GET = parse.requestmethod(s)
      local ok, size = parse.requestfields(s, header)
      if not ok then return end
      header.size = size
      return true, header
   end
   return
end

function process(buff)
   local ok, header = parse.http(buff)
   if ok then
      return true
   end
end
```

Cód. 13: Implementação do script de sanitização em Lua

6.2.2 Testes de desempenho

Para analisarmos o desempenho do artefato produzido no projeto utilizamos os dois servidores utilizados nos testes de estabilidade rodando o script Lua de sanitização. Utilizamos a ferramenta *ab* para realizarmos muitas requisições HTTP em um período de 15 segundos. Os servidores rodam numa máquina virtual com 2 cores, mas como não realizam processamento *multi-threading* estamos consumindo apenas uma vCPU da máquina.

Realizamos os testes de desempenho em 3 cenários. No primeiro cenário todas as requisições feitas pelo cliente são inválidas, assim visamos medir o desempenho de ambas as soluções em detectar um cabeçalho mal formado e fechar a conexão. No

segundo cenário todas as requisições feitas pelo cliente são válidas, assim visamos medir o impacto de invocar Lua dentro da chamada de sistema. No terceiro cenário misturamos ambos os cenários anteriores com o objetivo de simular tráfego de servidores reais.

Para cada cenário realizamos 4 medidas. As medidas foram realizadas consecutivamente, mantendo o mesmo estado Lua, para analisarmos o impacto do coletor de lixo de Lua.

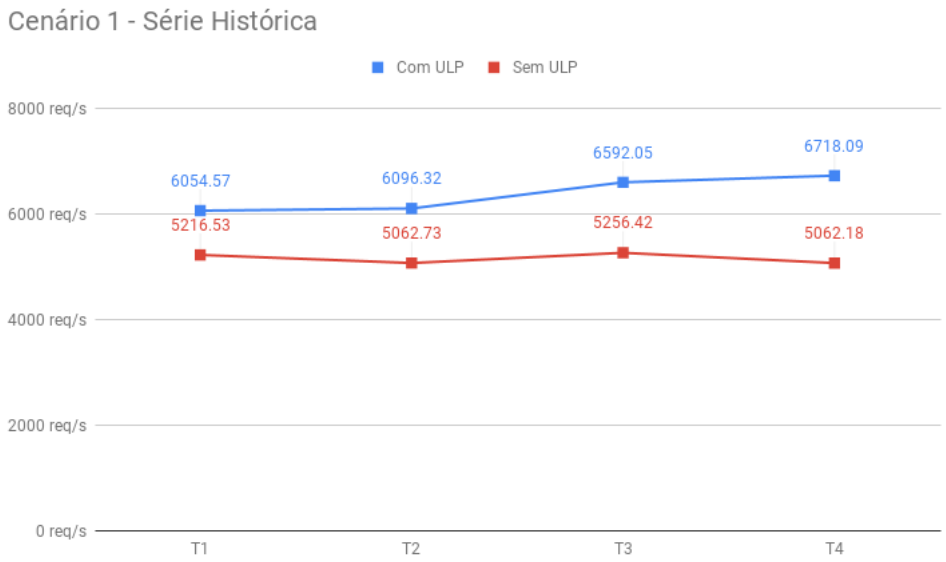

Figura 6: Teste de desempenho do primeiro cenário

No primeiro cenário observamos um ganho de desempenho expressivo. Em média a solução que utiliza o artefato produzido conseguiu processar 23% a mais de requisições do que a solução convencional.

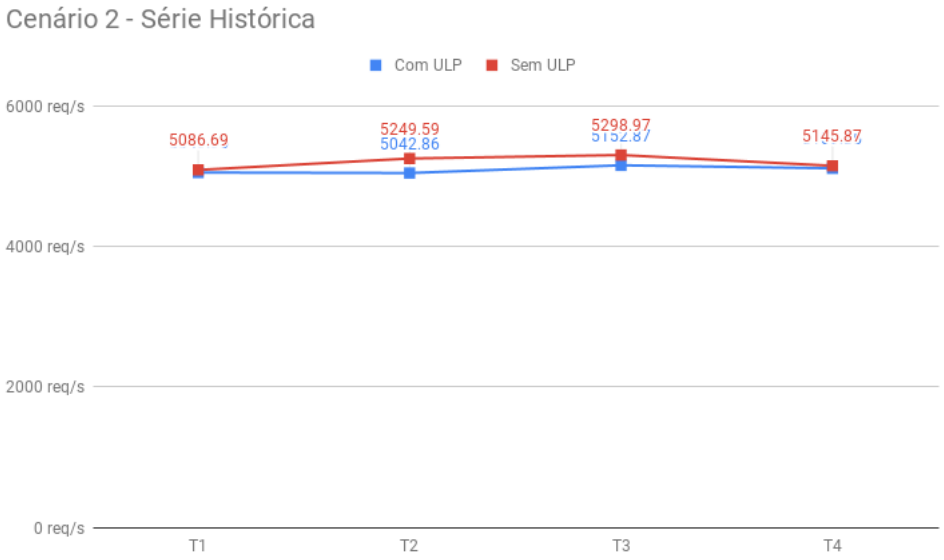

Figura 7: Teste de desempenho do segundo cenário

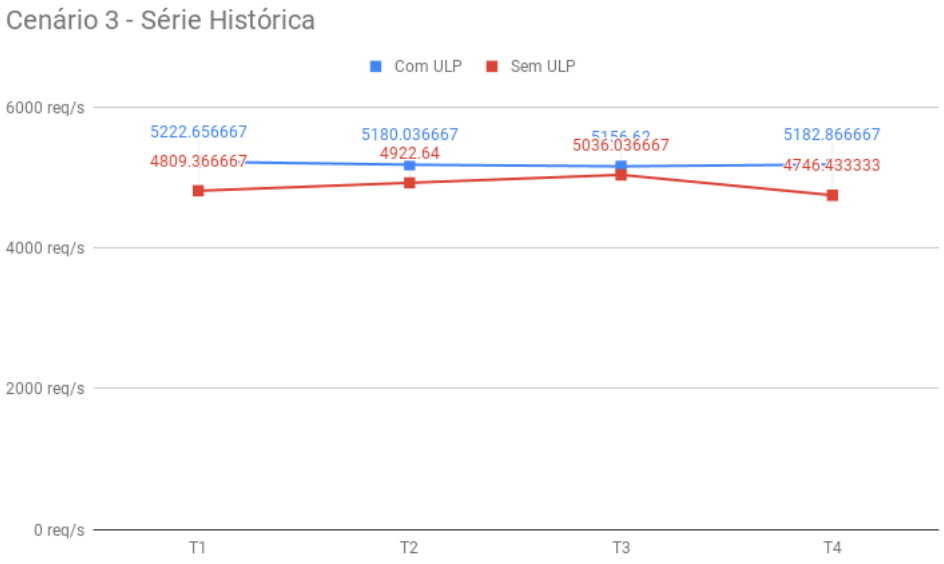

Figura 8: Teste de desempenho do terceiro cenário

No segundo e terceiro cenários não obtivemos resultados expressivos. O segundo cenário apresentou uma degradação de 3% na solução que utiliza o artefato. O terceiro cenário apresentou um ganho de desempenho de 6% na solução que utiliza o artefato.

A degradação no cenário 2 pode estar associada ao alocador de memória utilizado pelo lunatik. Exploramos essa possibilidade na seção 7.3.

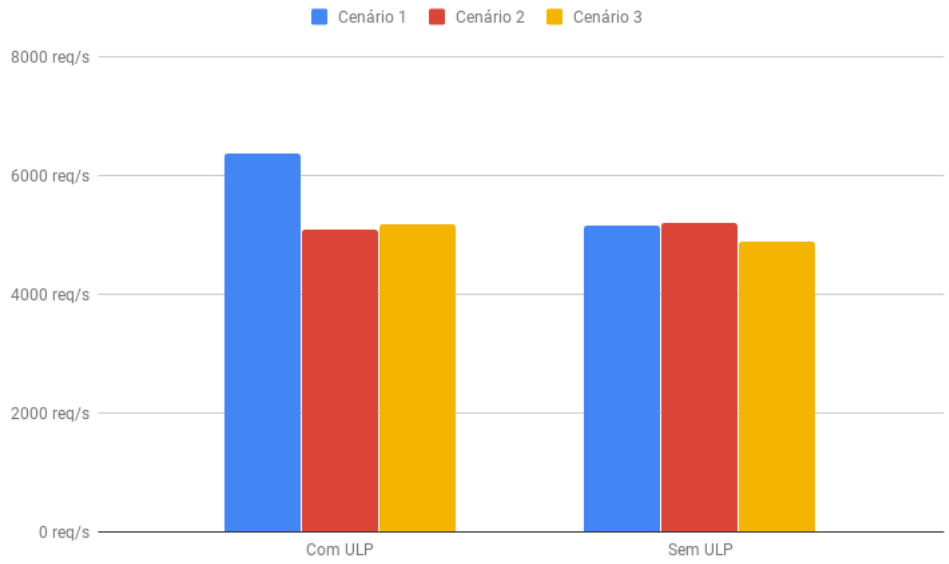

Figura 9: Comparação de todos os cenários

7 Considerações finais

Neste projeto apresentamos uma implementação para o *scripting* de chamadas de sistema usando Lua em sockets. Apesar disso, o artefato ainda impõe restrições nas soluções em que pode ser introduzido, havendo grandes margens para ser evoluído em trabalhos futuros.

7.1 Melhorias na arquitetura

A arquitetura do artefato produzido ainda é muito restrita quanto aos casos de uso. É essencial que os sockets não compartilhem o estado Lua associado pelo ULP para que se possa introduzirmos o *scripting* de chamadas de sistemas em programas de uso profissional. Para contornamos esta limitação poderia ser feito um cache de estados Lua, com reciclagem de recursos no fechamento de um socket.

A implementação da função *recvmsg()* ainda é frágil a situações fora do comum. O protocolo TCP permite a fragmentação de mensagens da camada de aplicação, onde é reconstruída na função *recvmsg()*. Como a nossa implementação verifica apenas o primeiro buffer de dados recebido pela rede, cabeçalhos HTTP mal-formados ao final podem passar desapercebidos. Em trabalhos futuros, é interessante percorrer a fila de recebimento do socket e agregar todos os buffers recebidos em uma estrutura de dados, de maneira que os scripts Lua vejam os buffers como um buffer contínuo. Um detalhe é que nem sempre o sistema operacional é encarregado de reconstruir os

fragmentos TCP, em drivers de placas de redes modernas o sistema operacional pode delegar essa tarefa para a placa de rede, tanto em *output* quanto em *input*, nesse caso o servidor sempre recebe o buffer completo.

Neste artefato exploramos pouco a customização do estado Lua e do módulo de kernel pelas chamadas de função *setsockopt()* e *getsockopt()*. Em trabalhos futuros, podemos usar informações de antemão providas pela aplicação do usuário para pré alocar possíveis recursos, como tabelas em Lua. A chamada de sistema *getsockopt()* também pode ser usada para a inspeção de informações guardadas dentro do kernel, pela aplicação de usuário.

Um fator que não consideramos neste projeto é a latência das mensagens. A implementação assume que ao aceitar a conexão temos no buffer de dados do socket pelo menos uma mensagem, caso a mensagem seja incompleta ou não há mensagem invocamos a chamada de sistema original. Em servidores reais praticamente todas as mensagens estão sujeitas a latência entre cliente e servidor e portanto não estariam sujeitas ao artefato.

7.2 Comparação com outras soluções

7.2.1 Servidores HTTP

No mercado, e na comunidade open source, exitem diversas soluções para servidores web. Alguns dos mais famosos são o Apache HTTP server, Nginx e o lighttpd. A maioria são escritos em C, pois utilizam recursos avançados da GNU libc para conseguirem o melhor desempenho possível.

A arquitetura de processamento das requisições varia bastante de servidor para servidor. Em testes de desempenho recentes, é possível constatar que arquiteturas recentes tem se mostrado mais eficientes.

Em trabalhos futuros, podemos atualizar parte da arquitetura de processamento de um servidor já existente para utilizar o projeto desenvolvido. Assim, poderemos comparar o desempenho com a versão original com aquele obtido com as outras soluções.

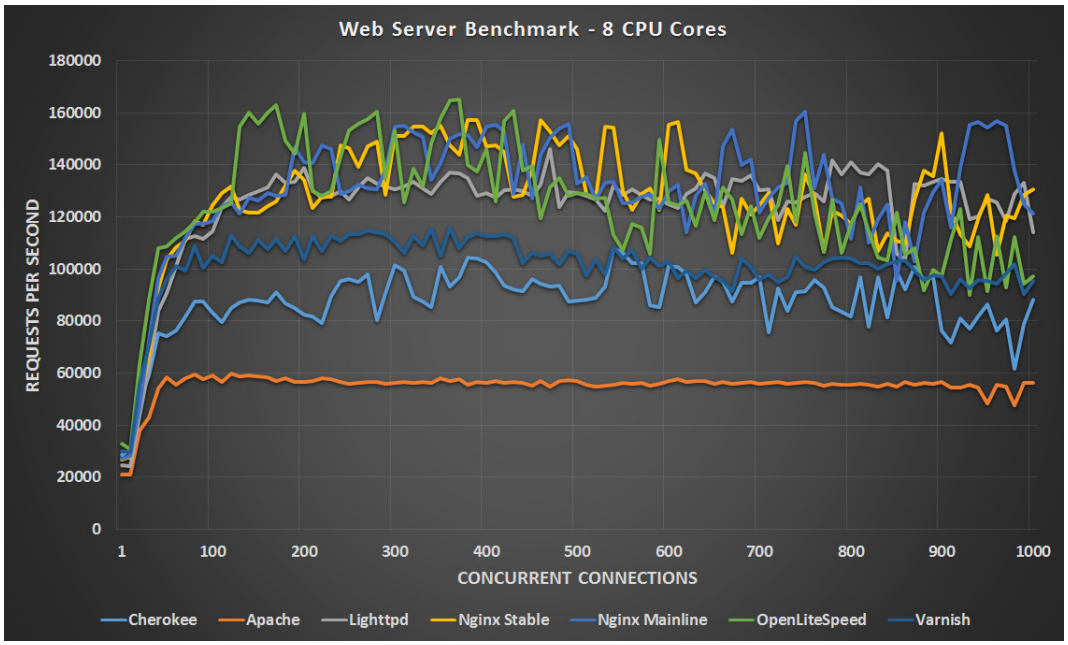

Figura 10: Testes de desempenho de servidores web no Linux [21]

7.2.2 eBPF

Códigos compilados para a máquina virtual eBPF não tem acesso a funções de uma biblioteca padrão[9]. No entanto o kernel do Linux provê *helpers*, funções ou funcionalidades que são usadas com frequência por usuário do eBPF, como consulta em mapas e inspeção de cabeçalhos nos buffers de rede[20].

A abstração e expressividade desses *helpers* ainda é pobre. Funcionalidades complexas, como casamento de padrões em strings (*pattern matching*), precisam ser implementadas. Em nosso caso de uso, grande parte das funcionalidades que utilizamos não está presente atualmente nos *helpers* do eBPF mas está presente em Lua através das bibliotecas padrão.

Utilizar eBPF, quando comparado a utilizar Lua, é uma solução propensa a reescrita de código. Lua é claramente superior na expressividade e suas bibliotecas padrão são bastante poderosas, além do fato de ser extensível via bindings em C. Já o eBPF é pouco expressivo e implementa um sistema de binding através de *helpers* mas focado em aplicações e casos de usos específicos.

Ambos os artefatos são escritos em C, extensíveis via C e podem ser executados tanto em espaço de usuário quanto no kernel, mas apresentam ao programador um estilo de programação muito diferente. É possível perceber que o foco do projeto do eBPF, atualmente, vem sendo a aplicação das funcionalidades na inspeção da

camada TCP/IP através de projetos como o XDP (*eXpress Data Path*)[9].

7.3 Melhorias para Lua no Kernel

Apesar de ter sido utilizado apenas como uma ferramente neste projeto, o *port* da linguagem Lua para o kernel demonstrou um gargalo para o nosso caso de uso. Em scripts que fazem o coletor de lixo ser bem atuante, como o código 14, existe um gargalo que degrada bastante o desempenho em relação à execução do mesmo script em espaço de usuário.

```
function process()
   local tb = {}
   for i = 1, 131072 do
      tb[i] = i
   end
   return false
end
```

Cód. 14: Implementação do script que exercita o coletor de lixo de Lua

O gargalo pode ser devido ao alocador de memória genérico escolhido pela implementação oficial do *port*, que aloca páginas de memórias contíguas fisicamente. Devem ser considerados outros alocadores de memória, como o alocador de memória virtual, para trabalhos futuros.

8 Referências

[1] Alexei Starovoitov. *BPF syscall, maps, verifier, samples*. 2014. URL: https://lwn.net/Articles/604043/.

[2] Alexei Starovoitov. *eBPF program to parse udp packets*. URL: https://github.com/torvalds/linux/blob/master/samples/bpf/parse_simple.c.

[3] Brenden Blanco. *BPF Compiler Collection (BCC)*. URL: https://github.com/iovisor/bcc.

[4] CVE Details. *CVE-2014-5461*. URL: https://www.cvedetails.com/cve/CVE-2014-5461/.

[5] Daniel Bausch. *lunatik-ng*. URL: https://github.com/lunatik-ng/lunatik-ng.

[6] Tim Dierks. "The transport layer security (TLS) protocol version 1.2". Em: (2008).

[7] Dawson R Engler, M Frans Kaashoek et al. *Exokernel: An operating system architecture for application-level resource management*. Vol. 29. 5. ACM, 1995.

[8] EuroBSDcon. *EuroBSDcon 2018*. URL: https://2018.eurobsdcon.org/talks-speakers/#avondoll.

[9] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern e David Miller. "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". Em: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: ACM, 2018, pp. 54–66. ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. URL: http://doi.acm.org/10.1145/3281411.3281443.

[10] Roberto Ierusalimschy, Luiz Henrique De Figueiredo e Waldemar Celes Filho. "Lua-an extensible extension language". Em: *Software, Practice & Experience* 26.6 (1996), pp. 635–652.

[11] Brian Kernighan e Dennis M Ritchie. *The C programming language*. Prentice hall, 2017.

[12] LabLua. *Lua in Kernel for Linux x86*. URL: https://github.com/luainkernel/lunatik.

[13] Linux Plumbers 2017. *The Linux Plumbers 2017 Conference*. URL: https://blog.linuxplumbersconf.org/2017/ocw/events/LPC2017/schedule.html.

[14] Linux Plumbers 2018. *The Linux Plumbers 2018 Conference*. URL: https://linuxplumbersconf.org/event/2/timetable/?view=lpc.

[15] Lua.org. *Lua 5.3 Reference Manual - lua_pcall()*. URL: https://www.lua.org/manual/5.3/manual.html#lua_pcall.

[16] Marc Balmer. *Lua in the NetBSD Kernel*. 2013. URL: https://www.netbsd.org/gallery/presentations/mbalmer/fosdem2012/kernel_mode_lua.pdf.

[17] Matt Fleming. *A thorough introduction to eBPF*. 2017. URL: https://lwn.net/Articles/740157/.

[18] Mozilla Developer Network. *PUT*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT.

[19] Lourival Vieira Neto. "Lunatik: Scripting de Kernel de Sistema Operacional com Lua". Diss. de mestrado. PUC-Rio, 2011.

[20] Quentin Monnet. *BPF-HELPERS*. URL: https://github.com/iovisor/bpf-docs/blob/master/bpf_helpers.rst.

[21] RootUsers. *Linux Web Server Performance Benchmark – 2016*. URL: https://www.rootusers.com/linux-web-server-performance-benchmark-2016-results/.

[22] Andrew S Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[23] *TLS in Kernel*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/Documentation/networking/tls.txt?h=v4.17.3.

[24] Lourival Vieira Neto, Roberto Ierusalimschy, Ana Lúcia de Moura e Marc Balmer. "Scriptable operating systems with Lua". Em: *ACM SIGPLAN Notices*. Vol. 50. 2. ACM. 2014, pp. 2–10.

[25] Wei Wu. *Kernel heap overflow in bpf leading to LPE (exploit provided)*. URL: https://seclists.org/oss-sec/2018/q4/170.

[26] Harald Welte. "The netfilter framework in Linux 2.4". Em: *Proceedings of Linux Kongress*. 2000.

[27] Wikipedia, the free encyclopedia. *Device file*. URL: https://en.wikipedia.org/wiki/Device_file.

[28] Wikipedia, the free encyclopedia. *Internet protocol suite*. URL: https://en.wikipedia.org/wiki/Internet_protocol_suite.

[29] Wikipedia, the free encyclopedia. *Microkernel*. URL: https://en.wikipedia.org/wiki/Microkernel.

[30] Wikipedia, the free encyclopedia. *Monolithic Kernel*. URL: https://en.wikipedia.org/wiki/Monolithic_kernel.

[31] Wikipedia, the free encyclopedia. *Netlink*. URL: https://en.wikipedia.org/wiki/Netlink.

[32] Wikipedia, the free encyclopedia. *Procfs*. URL: https://en.wikipedia.org/wiki/Procfs.

[33] Wikipedia, the free encyclopedia. *Structure of monolithic and microkernel-based operating systems, respectively*. URL: https://en.wikipedia.org/wiki/Microkernel#/media/File:OS-structure.svg.

[34] Wikipedia, the free encyclopedia. *System Call*. URL: https://en.wikipedia.org/wiki/System_call.

[35] Wikipedia, the free encyclopedia. *XNU*. URL: https://en.wikipedia.org/wiki/XNU.