UNIVERSIDADE FEDERAL DE OURO PRETO
DEPARTAMENTO DE COMPUTAÇÃO

Título: Semântica formal e validação de filtros de pacotes linux expressos na linguagem eBPF

Aluno: Rafael Diniz de Oliveira
Orientador: Dr. Rodrigo Geraldo Ribeiro

Relatório Final, referente ao período 08/2020 a 07/2021, apresentado à Universidade Federal de Ouro Preto, como parte das exigências do programa de iniciação científica do edital EDITAL

Ouro Preto - Minas Gerais - Brasil
31 de Agosto de 2021

Resumo:

Semântica formal e validação de filtros de pacotes linux expressos na linguagem eBPF

O Extended Berkeley Packet Filter (eBPF) é um subsistema Linux que permite executar extensões não confiáveis definidas pelo usuário dentro do kernel. Para proteger o kernel contra código malicioso, o eBPF utiliza técnicas de análise estática simples. Porém, a medida que a linguagem eBPF cresce em popularidade, o seu ecossistema evolui para oferecer suporte a extensões mais complexas e diverisificadas. Porém, as limitações presentes em seu verificador atual, como a sua alta taxa de falsos positivos e falta de suporte a comandos de repetição, tornaram-se um grande empecilho para sua ampla adoção por desenvolvedores de aplicações de rede. Neste sentido, o presente projeto pretende aplicar técnicas de semântica formal e sistemas de tipos de maneira a aumentar a suporte do verificador eBPF, permitindo assim que programadores desenvolvam aplicações de maneira simples e garantindo a segurança do kernel estendido.

https://github.com/lives-group/racket-ebpf

Bolsista: Rafael Diniz de Oliveira

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro

1 Introdução

Os sistemas operacionais modernos implementam a maior parte de sua funcionalidade por meio de extensões carregadas dinamicamente que proveem suporte para dispositivos de E / S, sistemas de arquivos, redes, etc. Essas extensões são executadas no modo privilegiado da CPU e, portanto, devem ser verificadas para assegurar a ausência de código malicioso. Tradicionalmente a validação destas extensões é estabelecida através do uso de testes para eliminar bugs. Além disso, as ferramentas de verificação formal são usadas, em alguns casos, para obter maior segurança [2, 7]. Apesar de tais ferramentas serem eficazes para encontrar bugs, estas não possuem fortes garantias de correção.

As extensões do kernel são um tipo especial de aplicação que se originam de fontes não confiáveis e, portanto, não podem ser consideradas seguras. Essas extensões permitem que aplicativos personalizem o kernel com algoritmos específicos para processamento de pacotes, políticas de segurança, profiling e até modifiquem como os principais subsistemas do kernel [1].

No passado, os sistemas operacionais utilizavam técnicas como sandboxing para a execução de extensões não confiáveis dentro do kernel. Além disso, algumas abordagens se valiam de linguagens de domínio específico de domínio [3, 5] e interpretadores de bytecode [8]. Porém, essas abordagens se mostraram restritivas para alguns casos em que o desempenho é um fator crítico.

Como uma forma de amenizar essas falhas, o Linux adotou uma abordagem baseada no uso de um verificador de código, chamado de Exteded Berkeley Package Filters (eBPF). Programas eBPF consistem de bytecode simples que é compilado em instruções nativas da CPU quando carregado pelo kernel. Ao contrário dos bytecodes da Java Virtual Machine, o compilador e o run-time de eBPF não impõe nenhum restrição de tipo. A validade de programas eBPF é imposta por um verificador estático que impede que programas possam acessar estruturas de dados arbitrárias do kernel.

Atualmente, programadores eBPF enfrentam quatro grandes problemas. O primeiro deles é que o verificador de eBPF apresenta uma taxa elevada de falsos positivos, isto é, rejeita como inseguros programas que não oferecem risco algum ao kernel. Essa limitação obriga desenvolvedores a reescrever seu código de forma a este ser aceito pelo verificador. Muitas vezes, essas alterações envolvem a inserção de acessos e verificações redundantes. O segundo problema é que o algoritmo utilizado pelo verificado é sensível ao número de caminhos presente no programa analisado. Terceiro, atualmente o verificador impõe a séria restrição de que programas não devem possuir loops. Finalmente, o verificador atual não é especificado formalmente e isso dificulta a predição de quando um código será ou não por ele recusado.

Diante do apresentado, o presente projeto pretende especificar formalmente um verificador de programas eBPF de forma a solucionar as limitações apresentadas. Para isso, utilizaremos técnicas conhecidas do projeto de linguagens de programação, a saber: semântica formal e sistemas de tipos [9, 6, 4].

2 Revisão da Literatura

2.1 A linguagem eBPF

Sitemas Unix se tornaram um sinônimo de uso eficiente de redes e usuários UNIX passaram a depender de acesso seguro e responsivo. No entanto, essa dependência faz com que problemas de conexão podem fazer com que seja impossível realizar tarefas realmente úteis. A solução de problemas requer ferramentas de análise e diagnóstico apropriadas e, de preferência, essas ferramentas devem ser acessíveis de onde os problemas acontecem, em estações de trabalho UNIX. E para permitir que essas ferramentas sejam criadas, o kernel deve ter alguma forma de fornecer dados não processados da rede para programas no nível de usuário [8]. Assim sendo, foi criado o BPF (Berkeley Packet Filter) uma nova arquitetura do kernel para captura de pacotes.

No entanto, na medida que processadores evoluíram para registradores de 64 bits e a arquitetura de instruções utilizada no BPF foi sendo deixada de lado, o foco do filtro de pacotes em fornecer um pequeno número de instruções RISC não satisfaziam a realidade de processadores modernos. Então, Alexei Starovoitov introduziu o modelo do extended BPF (eBPF) para tirar proveito dos novos avanços de hardware.

A versão oficial do kernel que adicionou o suporte ao eBPF mostrou que este era até quatro vezes mais rápido em arquitetura x86-64 do que o antigo classic BPF, para alguns microbenchmarks de filtro de rede e a maioria era 1,5 vezes mais rápido.

Programas eBPF são escritos usando bibliotecas em linguagens de mais alto nível, como C, python ou Go.

2.2 A linguagem Racket

Racket é uma linguagem da família de linguagens Lisp, que foi com o objetivo principal de servir como uma plataforma para criação de linguagens, ainda que seja usada em diversos contextos. Para tal objetivo, Racket conta com um sistema de macros extremamente poderoso, o que facilita a definição de síntaxe a partir transformação de um padrão de código em código Racket.

A linguagem é composta por s-expressions, que geram um valor atômico ou uma lista de valores, uma função é um valor atômico do tipo procedure, o significa que funções podem ser passadas como qualquer outro valor em Racket. Além disso, as funções só são avaliadas quando invocadas em tempo de execução, o que permite que funções permaneçam como um procedure, não calculadas, caso não estejam em forma de s-expression (entre parênteses). Funções em Racket são definidas usando define.

```
(define (my-sum x y)
  (+ x y))
(my-sum 3 5)
```

Racket é uma linguagem forte e dinâmicamente tipada, o que significa que apesar de não ser necessário fazer declarações de tipos, erros de tipo serão apontados seja em tempo de compilação, ou execução. Ademais, racket conta com alguns dialetos que podem ser escolhidos para o desenvolvimento, entre eles é possível optar pela declaração estática de variáveis usando Typed Racket.

Além disso, é importante ressaltar o uso de macros em Racket, pois são o motivo de esta ser tão eficiente na criação de liguagens de domínio específico como dialetos de Racket com diferentes semânticas. Macros definem uma forma sintática e associam a um transformador, que expande o formato original em uma série de operações, no entanto, diferente de funções, macros são avaliadas antes da compilação, ou seja, o trabalho de uma macro é literalmente, transformar um pedaço de código em outro pedaço de código.

A forma mais simples de se criar uma macro é usando define-syntax-rule:

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Dessa forma, o expressão (swap fst lst) gera o código

```
(let ([tmp fst])
  (set! fst lst)
  (set! lst tmp))
```

O compilador trata a redeclaração de variáveis caso, uma variável usada na macro tenha o mesmo nome de uma variável do escopo da mesma, a expressão (swap tmp var) funcionaria normalmente, por exemplo.

****Ainda não terminado****

Na sequência você pode explicar a implementação daquela máquina virtual simples que fiz.

3 Desenvolvimento

3.1 Construção da arquitetura eBPF

O interpretador simula uma arquitetura eBPF a partir de uma maquina virtual com três elementos básicos:

- Um acumulador A de 32 bits.
- Um registrador X de 32 bits.
- Um conjunto M de 16 registradors de 32bits.

Além disso a maquina virtual precisa de outros campos para correr o programa, começando por um pc, para controlar o fluxo do programa; um campo para armazenamento de todas as instruções; e um campo que mapeia os labels do programa eBPF ao seu deslocamento a partir do ínicio do programa.

```
(struct ebpf-state
  (pc             ;; program counter
   code           ;; instructions
   regs-m         ;; registers M[]
   reg-x          ;; register X
   acc-a          ;; accumulator A
   labels)        ;; hash table mapping labels to its
                  ;; line offset in the program
   #:transparent)
```

O interpretador inicia a maquina virtual, recebendo a lista de instruções do programa, enquanto, outros campo são iniciados vazios ou zerados. Para garantir que os registradores e o acumulador tenham os tamanhos apropriados (32bits), o interpretador inicia seus respectivos campos com o tipo de dado bitvectors e realiza todas as operações básicas em sua implementação para bitvectors.

```
(define (initial-state instrs)
  (ebpf-state
  0
  instrs
  (make-vector 16 (bv 0 32))
  (bv 0 32)
  (bv 0 32)
  '#hash()))
```

3.2 Execução da máquina virtual

O primeiro comando a ser executado depois de gerada a lista de intruções é aquele que dará ínicio a máquina virtual, chamado pelo interpretador de exec. A função desse comando é, a partir de um estado representado por ebpf-state, realizar a próxima instrução do programa e retornar um novo estado para execução até que o programa se encerre.

```
(define (exec st)
  (...)
  (let* ([next-instr (list-ref (ebpf-state-code st)
                               (ebpf-state-pc st))]
         [next-state (next-instr st)])
    (exec next-state)))
```

O encerramento do programa pode ocorrer de duas formas, caso não haja mais intruções a serem executados ou caso uma instrução termine o programa. Em qualquer dos casos, o contador do programa deve exceder a quantidade de instruções, logo, é essencial que a função que encerra o programa incremente o contador para garantir o encerramento do programa.

```
(define (ret st)
```

```
(define code-size (length (ebpf-state-code st)))
(match st
  ((ebpf-state pc instrs m x a labels)
   (ebpf-state code-size instrs m x a labels))))
```

Por fim, a máquina virtual deve verificar sempre que receber um novo estado, se o contador excedeu o tamanho do programa, para isso dentro da função de execução existem duas variáveis de controle, uma para o número atual da instrução, outro para o tamanho total do programa.

```
(define (exec st)
(define current-pc (ebpf-state-pc st))
(define code-size (length (ebpf-state-code st)))
(if (>= current-pc code-size)
    st
    (let* ([next-instr (list-ref (ebpf-state-code st)
                                 current-pc)]
           [next-state (next-instr st)])
      (exec next-state))))
```

3.3 Labels

O campo labels é iniciado como uma tabela hash vazia que deve, ao ínicio da execução, ser preenchida, isso é feito percorrendo a lista de instruções e mapeando a chave de cada Label a seu deslocamento de linhas no programa. Para isso, ainda é necessário implementar os métodos que geram a lista de instruções e definir a macro que gera a hash de cada Label.

3.4 Instruções

O programa será processado pelo interpretador e a partir de um algoritmo do novo modelo eBPF, deve ser gerada uma lista de instruções em Racket, que a partir do uso de macros poderão ser transformadas em funções aplicáveis a um estado da máquina virtual, essas funções, por sua vez representam instruções eBPF originais. No entanto, ainda é necessário fazer a implementação dos métodos que interpretam o programa em ebpf-sim para Racket, o método main, além das macros que transformam as instruções em funções Racket.

Todas as funções que manipulam palavras, ou seja um Bitvector de 32bits, estão implementadas arquivo expander.rkt dentro do diretorio ebpf-sim, o arquivo por sua vez, deve importar a biblioteca bv, para fazer uso destes tipos de dados. As funções que representam as intruções ldh,ldb e ldbxb ainda não foram implementadas.

Todas as instruções de load para o acumulador A passam pela função ld, a partir de um dado k e um estado, altera o valor salvo no acumulador A para o dado recebido.

```
(define (ld k st)
```

```
  (match st
    ((ebpf-state pc instrs m x a labels)
     (ebpf-state (add1 pc) instrs m x k labels))))
```

Outras funções que carregam palavras para A, só precisam formatar seu endereçamento para a função anterior. A máquina virtual ainda não consegue encontrar o deslocamento em bytes no programa. Com isso, resta apenas tornar possível carregar palavras guardadas em qualquer dos registradores M.

Carregar palavras para o registrador X, segue exatamente o mesmo padrão de carregar palavras para o acumulador, utilizando a função ldx, e também deve conseguir carregar palavras dos registradores M.

```
(define (ld-add3 k st)
  (ld (vector-ref (ebpf-state-regs-m st) k) st))

(define (ldx-add3 k st)
  (ldx (vector-ref (ebpf-state-regs-m st) k) st))

(define ldi ld)
```

Guardar as informações do registrador X e do acumulador é trabalho das funções st, que assim como ld existe uma referente ao acumulador outra referente ao registrador, ambas as funções requerem que seja passado um deslocamento k, referente a posição do registrador no vetor M de registradores.

```
(define (stx-add3 k st)
  (match st
    ((ebpf-state pc instrs m x a labels)
     (ebpf-state (add1 pc)
                 instrs
                 (vector-set! m k x)
                 x
                 a
                 labels))))
```

Porém o interpretador eBPF ainda não tem uma função que permite transferir dados do registrador X para o acumulador A nem o contrário, isto é feito nas especificações do eBPF através da instrução tax que transfere de A para X e da instrução txa que transfere de X para A.

```
;tax
(define (tax st)
  (match st
    ((ebpf-state pc instrs m x a labels)
     (ebpf-state (add1 pc) instrs m a a labels))))

;txa
(define (txa st)
  (match st
```

```
    ((ebpf-state pc instrs m x a labels)
     (ebpf-state (add1 pc) instrs m x x labels))))
```

Instruções de salto dependem da construção prévia da tabela hash e utiliza os Labels como keys da tabela para encontrar a posição adequada do pacote em que deve apontar o contador do programa. Assim como nas instruções de load, para salto também é interessante definir uma função generalizada para reduzir a repetição do código e essa função é o salto incondicional jmp ou ja.

```
(define (jump-to-label st key)
  (define new-pc (hash-ref (ebpf-state-labels st) key))
  (match st
    ((ebpf-state pc instrs m x a labels)
    (ebpf-state new-pc instrs m x a labels))))
(define jmp jump-to-label)
(define ja jump-to-label)
```

Os saltos condicionais também seguem um outro padrão próprio, além de fazerem a alteração do contador para a posição do Label desejado, esses saltos aplicam uma condição para tomar a decisão de qual salto tomar, logo é fácil generalizar essa definição para evitar repetição de código. No entanto, é necessário se atentar que alguns endereçamentos não recebem um direcionamento caso a condição não se satisfaça. Por isso, quando a condição não é satisfeita, a função geral deve verificar se qual tipo de endereçamento foi recebido.

```
(define (jmp-cond st cond k lt lf)     ;cond needs
                                       ;;a logical operator
  (if (cond (ebpf-state-acc-a st) k)
      (jump-to-label st lt)
      (if (null? lf)
          (nop st)
          (jump-to-label st lf))))
```

Além disso, documentação de eBPF define 12 formas de endereçar instruções, das quais os saltos só usam quatro, onde k é um campo genérico de uma instrução eBPF, Lt um Label para salto quando a condiçãofor verdadeira e Lf um Label para salto quando a condição for falsa.

Modo de endereçamento	Descrição
7	compara um valor k e salta para Lt quando verdade ou Lf quando falso
8	compara o registrador x e salta para Lt quando verdade ou Lf quando falso
9	compara um valor k e salta para Lt quando verdade
10	compara o registrador x e salta para Lt quando verdade

Levando em consideração as informações acima, a tarefa de implementar as funções para os saltos condicionais é quase que intuitiva, tudo que resta é

definir quais operadores serão usados para verificar cada condição. Como os valores serão interpretados em Racket como Bitvectors de 32 bits, é necessário que os operadores sejam capazes de trabalhar com esse tipo de dados.

Modo de endereçamento	Instrução	Operador Lógico
7, 8, 9 e 10	jeq	bveq
9 e 10	jneq	!bveq
9 e 10	jne	!bveq
9 e 10	jlt	bvslt
9 e 10	jle	bvsle
7, 8, 9 e 10	jgt	bvsgt
7, 8, 9 e 10	jge	bvsge
7, 8, 9 e 10	jset	bvand

Esses operadores lógicos estão implementados na biblioteca bv que é importada para utilizar o tipo de dado já citado. No entanto, é necessário definir um operador bvneq como a negação de bveq, desta forma, as funções jneq e jne só precisam passar o operador para a função jump-cond, evitando um pouco a repetição de código.

```
(define (bvneq x y)
  (not (bveq x y)))
```

Estas informações são importantes pois o fato de que as funções serão extraídas da lista esperando um estado da maquina virtual, limita cada função a ter um número esperado de parâmetros e cada instrução é definida no seguinte padrão utilizando seu próprio nome e operador.

```
;;;addressing mode 7
(define (jeq-add7 k lt lf st)
  (jmp-cond st bveq k lt lf))

;;;addressing mode 8
(define (jeq-add8 lt lf st)
  (jmp-cond st bveq (ebpf-state-reg-x st) lt lf))

;;;addressing mode 9
(define (jeq-add9 k lt st)
  (jmp-cond st bveq k lt null))

;;;addressing mode 10
(define (jeq-add10 lt st)
  (jmp-cond st bveq (ebpf-state-reg-x st) lt null))
```

O restante das instruções, são operações aplicadas aos dados, isto é operações aritméticas e operações bit a bit, essas operações por definição precisam de dois valores, o primeiro será sempre o valor armazenado no acumulador A, o segundo valor poderá ser endereçado pelo usuário de duas formas diferentes. Lembrando que k é um campo genérico de uma instrução eBPF.

Modo de endereçamento	Descrição
0	Registrador x
4	Valor literal em k

Seguindo o pensamento utilizado na implementação das funções de salto o próximo objetivo foi definir uma função genérica que realiza essas instruções, as funções para que utilizam os modos de endereçamento acima serão executadas por uma função arith-instr ou uma função bitwise-ops como definição genérica conforme lexicalmente mais adequado, as instruções de operações aritméticas utilizam arith-instr e operações bit a bit usam bitwise-ops.

```
(define (arith-instr k op st)
  (match st
    ((ebpf-state pc instrs m x a labels)
    (ebpf-state (add1 pc) instrs m x (op a k) labels))))

(define bitwise-ops arith-instr)
```

A partir das funções auxiliares criadas acima, tudo que resta a ser feito é definir as funções que realizam as instruções restantes.

```
;;addressing mode 0
(define (INSTRUCAO-add0 st)
    (FUNCAO (ebpf-state-reg-x st) OPERADOR st))

;;addressing mode 4
(define (INSTRUCAO-add4 k st)
    (FUNCAO k OPERADOR st))
```

Basta seguir a tabela e substituir de forma adequada, esta tarefa seria realizada de forma mais elegante utilizando macros, este seria o próximo passo do desenvolvimento, refatorar o código usando macros.

FUNCAO	INSTRUCAO	OPERADOR
arith-instr	add	bvadd
arith-instr	sub	bvsub
arith-instr	mul	bvmul
arith-instr	div	bvsdiv
arith-instr	mod	bvsmod
bitwise-ops	and	bvand
bitwise-ops	or	bvor
bitwise-ops	xor	bvxor
bitwise-ops	lsh	bvshl
bitwise-ops	rsh	bvlshr

4 Conclusão e Trabalhos Futuros

Neste trabalho, foi construído um interpretador eBPF com propósito de usá-lo para a validação de programas eBPF. Para isso, foi utilizada a linguagem

Racket, devido a suas facilidades envolvendo a metaprogramação, linguagens de programação que manipulam linguagens de programação.

Como trabalhos futuros pretendemos formalizar o mecanismo de verificação de programas eBPF realizados pelo kernel Linux, de forma a facilitar o desenvolvimento de aplicações de rede de forma segura.

Referências

[1] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks. *SIGOPS Oper. Syst. Rev.*, 51(1):54–59, September 2017.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, December 1995.

[4] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. *SIGPLAN Not.*, 52(1):694–705, January 2017.

[5] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, April 2006.

[6] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. *SIGPLAN Not.*, 47(1):285–296, January 2012.

[7] Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 202–212, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[9] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.