University of Rome Tor Vergata
ICT and Internet Engineering

Network and System Defense

Alessandro Pellegrini, Angelo Tulumello

A.A. 2023/2024

Lecture 5: Firewall and Packet Classification Algorithms

Angelo Tulumello

Slides by Marco Bonola

Table of Contents

Firewall Overview

Source: William Stallings and Lawrie Brown, "Computer Security: Principles and Practice", chapter 9

The Need For Firewalls

- Internet connectivity is essential
  - However it creates a threat
- Effective means of protecting LANs
- Inserted between the premises network and the Internet to establish a controlled link
  - Can be a single computer system or a set of two or more systems working together
- Used as a perimeter defense
  - Single choke point to impose security and auditing
  - Insulates the internal systems from external network

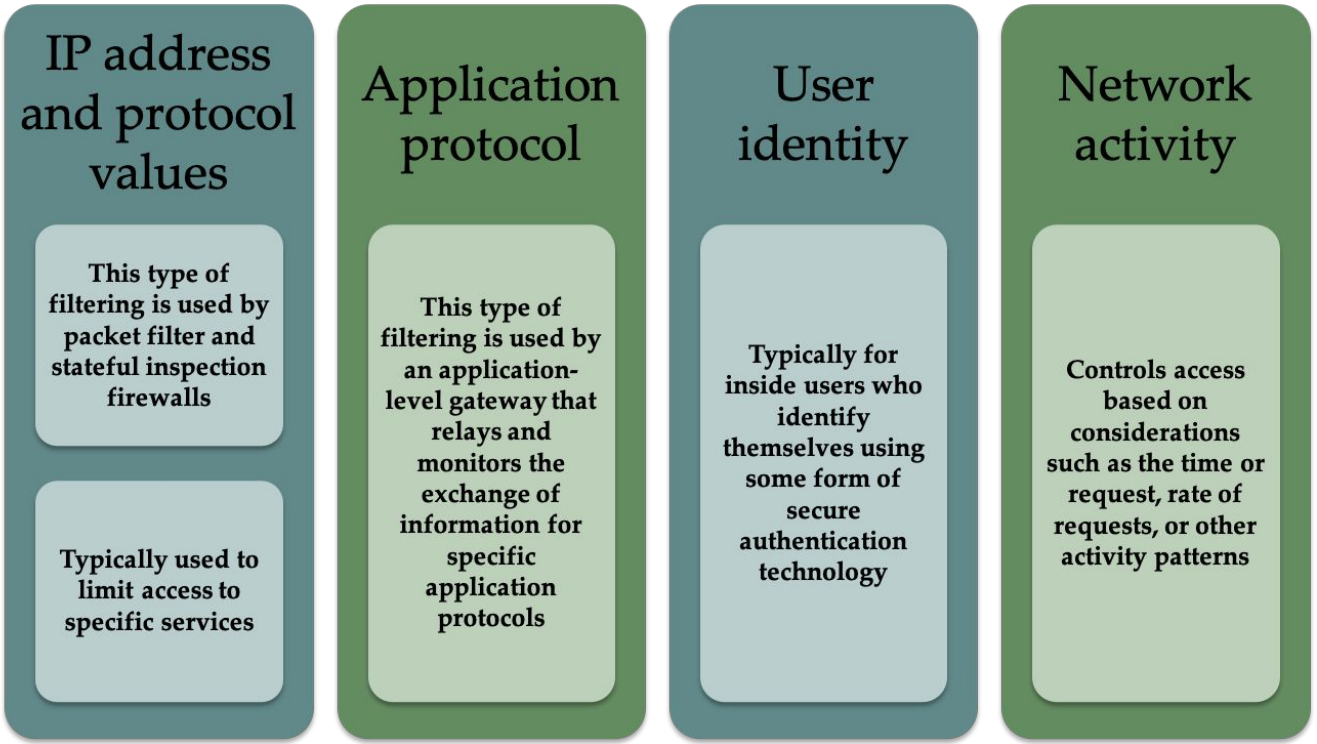
Firewall Characteristics: Design goals

- All traffic from inside to outside, and vice versa, must pass through the firewall
- Only authorized traffic as defined by the local security policy will be allowed to pass
- The firewall itself is immune to penetration

Firewall Access Policy

- A critical component in the planning and implementation of a firewall is specifying a suitable access policy
  - This lists the types of traffic authorized to pass through the firewall
  - Includes address ranges, protocols, applications and content types
- This policy should be developed from the organization's information security risk assessment and policy
- Should be developed from a broad specification of which traffic types the organization needs to support
  - Then refined to detail the filter elements which can then be implemented within an appropriate firewall topology

Firewall Filter Characteristics



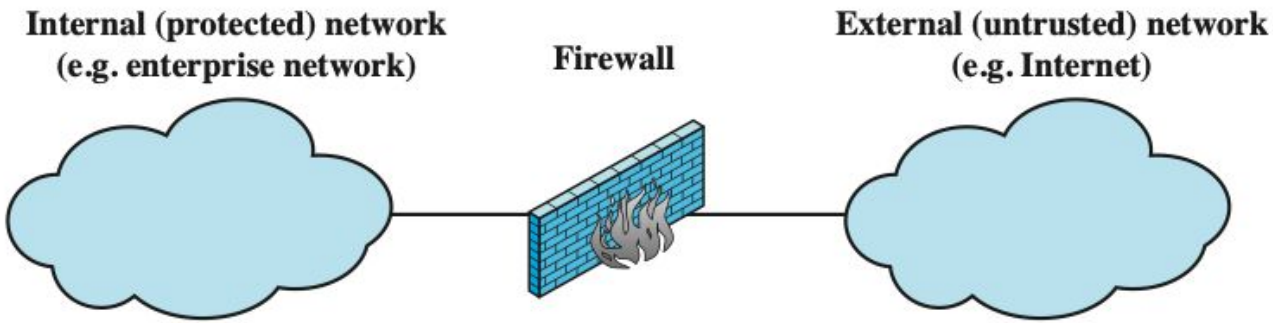
Firewall Capabilities And Limits

Capabilities

- ❏ Defines a single choke point
- ❏ Provides a location for monitoring security events
- ❏ Convenient platform for several Internet functions that are not security related
- ❏ Can serve as the platform for IPSec

Limitations

- ❏ Cannot protect against attacks bypassing firewall
- ❏ May not protect fully against internal threats
- ❏ Improperly secured wireless LAN can be accessed from outside the organization
- ❏ Laptop or portable storage device may be infected outside the corporate network then used internally



(a) General model

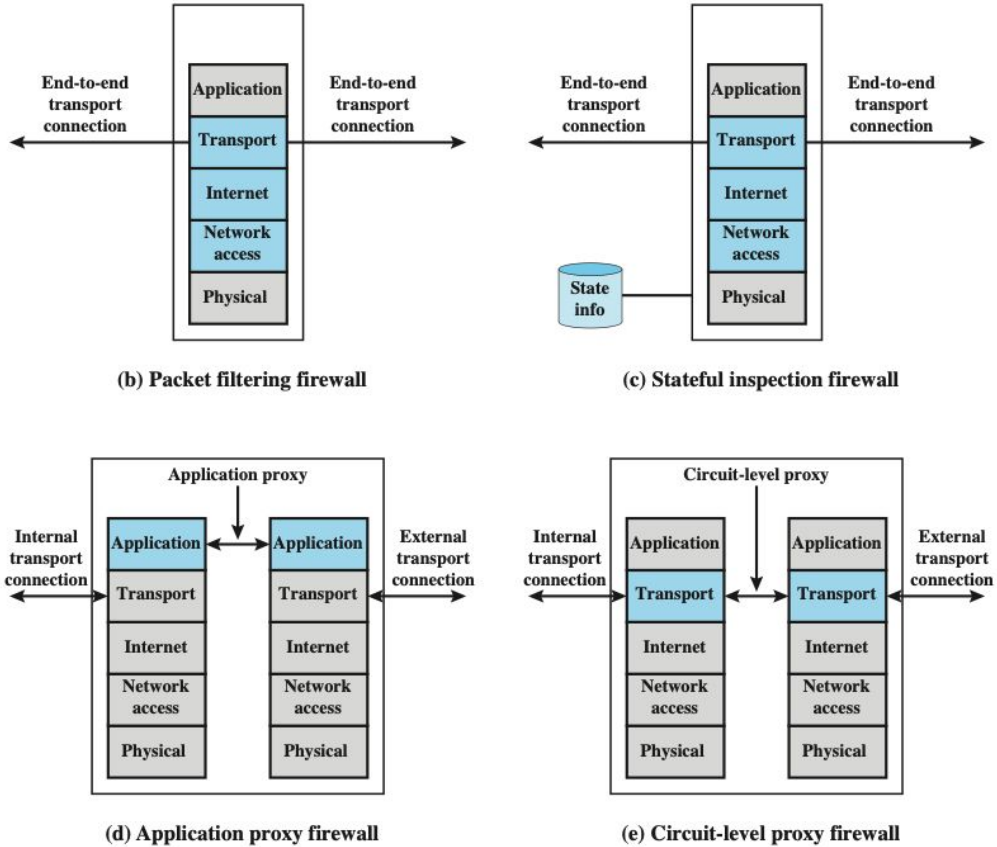


Figure 9.1 Types of Firewalls

Packet Filtering Firewall

- Applies rules to ***each incoming and outgoing IP packet***
  - Typically a list of rules based on matches in the IP or transport header
  - Forwards or discards the packet based on rules match:
    - Source IP address, Destination IP address, Source and destination transport-level address, IP protocol field, Interface
- Two default policies:
  - ***Discard*** - prohibit unless expressly permitted
    - More conservative, controlled, visible to users
  - ***Forward*** - permit unless expressly prohibited
    - Easier to manage and use but less secure

Packet-Filtering Examples

Rule	Direction	Src address	Dest addresss	Protocol	Dest port	Action
1	In	External	Internal	TCP	25	Permit
2	Out	Internal	External	TCP	>1023	Permit
3	Out	Internal	External	TCP	25	Permit
4	In	External	Internal	TCP	>1023	Permit
5	Either	Any	Any	Any	Any	Deny

Packet Filter: Advantages And Weaknesses

- ***Advantages***
  - Simplicity
  - Typically transparent to users and are very fast
- ***Weaknesses***
  - Cannot prevent attacks that employ application specific vulnerabilities or functions
  - Limited logging functionality
  - Do not support advanced user authentication
  - Vulnerable to attacks on TCP/IP protocol bugs
  - Improper configuration can lead to breaches

Stateful Inspection Firewall

Tightens rules for TCP traffic by creating a ***directory of outbound TCP connections***

- ***There is an entry for each currently established connection***
- Packet filter allows incoming traffic to high numbered ports only for those packets that fit the profile of one of the entries in this directory

Reviews packet information but also records information about TCP connections

- **Keeps track of TCP sequence numbers** to prevent attacks that depend on the sequence number
- ***Inspects data for higher protocols*** like FTP, IM and SIPS commands

Stateful firewalls are applied also to ***connectionless protocols (e.g. UDP)***

Connection State Table

Source Address	Source Port	Destination Address	Destination Port	Connection State
192.168.1.100	1030	210.9.88.29	80	Established
192.168.1.102	1031	216.32.42.123	80	Established
192.168.1.101	1033	173.66.32.122	25	Established
192.168.1.106	1035	177.231.32.12	79	Established
223.43.21.231	1990	192.168.1.6	80	Established
219.22.123.32	2112	192.168.1.6	80	Established
210.99.212.18	3321	192.168.1.6	80	Established
24.102.32.23	1025	192.168.1.6	80	Established
223.21.22.12	1046	192.168.1.6	80	Established

Application-Level Gateway

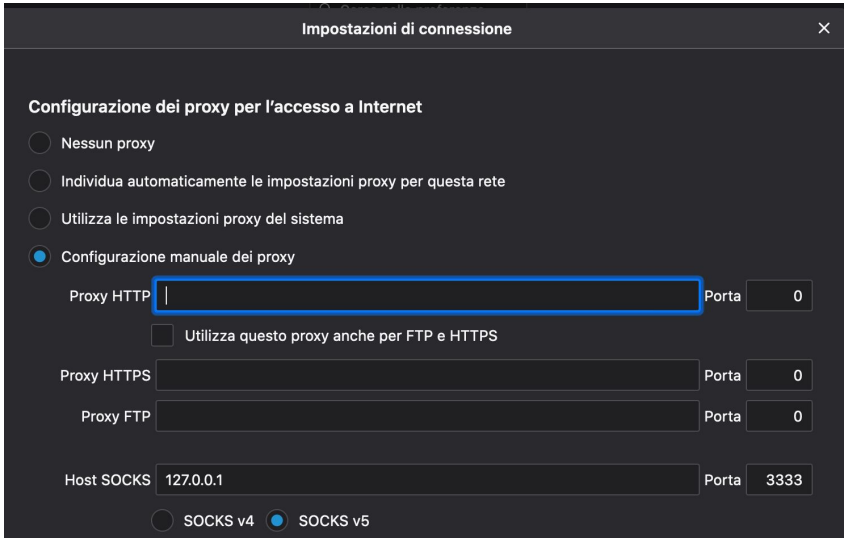
- Also called ***application proxy***
- Acts as a relay of application-level traffic
  - User contacts gateway using a TCP/IP application
  - User is authenticated
  - Gateway contacts application on remote host and relays TCP segments between server and user
- Must have proxy code for each application
  - May restrict application features supported
- Tend to be more secure than packet filters
- Disadvantage is the additional processing overhead on each connection

Circuit-Level Gateway

- AKA ***Circuit level proxy***
  - Sets up two TCP connections, one between itself and a TCP user on an inner host and one on an outside host
  - Relays TCP segments from one connection to the other without examining contents
  - Security function consists of determining which connections will be allowed
- Typically used when inside users are trusted
  - May use application-level gateway inbound and circuit-level gateway outbound
  - Lower overheads

SOCKS Circuit-Level Gateway

- ***SOCKS v5*** defined in RFC1928
- Designed to provide a framework for client-server applications in TCP/UDP domains to conveniently and securely use the services of a network firewall
- Client application contacts SOCKS server, authenticates, sends relay request
- Server evaluates and either establishes or denies the connection



firefox proxy SOCKS configuration

Host-Based Firewalls

- Used to secure ***an individual host***
- Available in operating systems or can be provided as an add-on package
- Filter and restrict packet flows
- Common location is a ***server***
- Advantages
  - Filtering rules can be tailored to the host environment
  - Protection is provided independent of topology
  - Provides an additional layer of protection

Personal Firewall

- Controls traffic between ***a personal computer or workstation and the Internet or enterprise network***
- For both home or corporate use
- Typically is a software module on a personal computer
- Can be housed in a router that connects all of the home computers to a DSL, cable modem, or other Internet interface
- Typically much less complex than server-based or standalone firewalls
- Primary role is to deny unauthorized remote access
- May also monitor outgoing traffic to detect and block ***worms and malware activity***

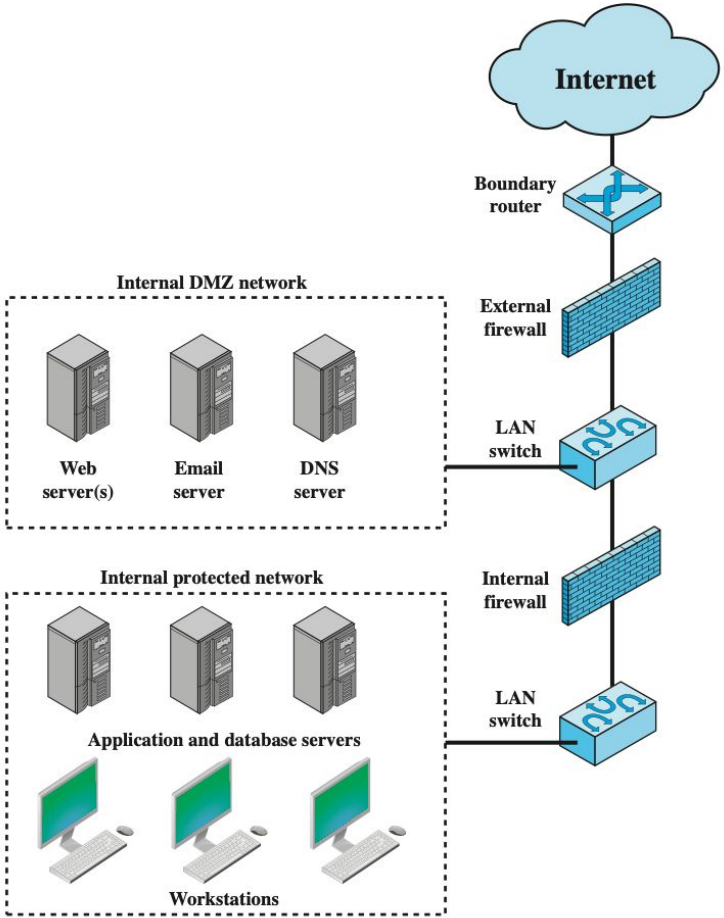


Figure 9.2 Example Firewall Configuration

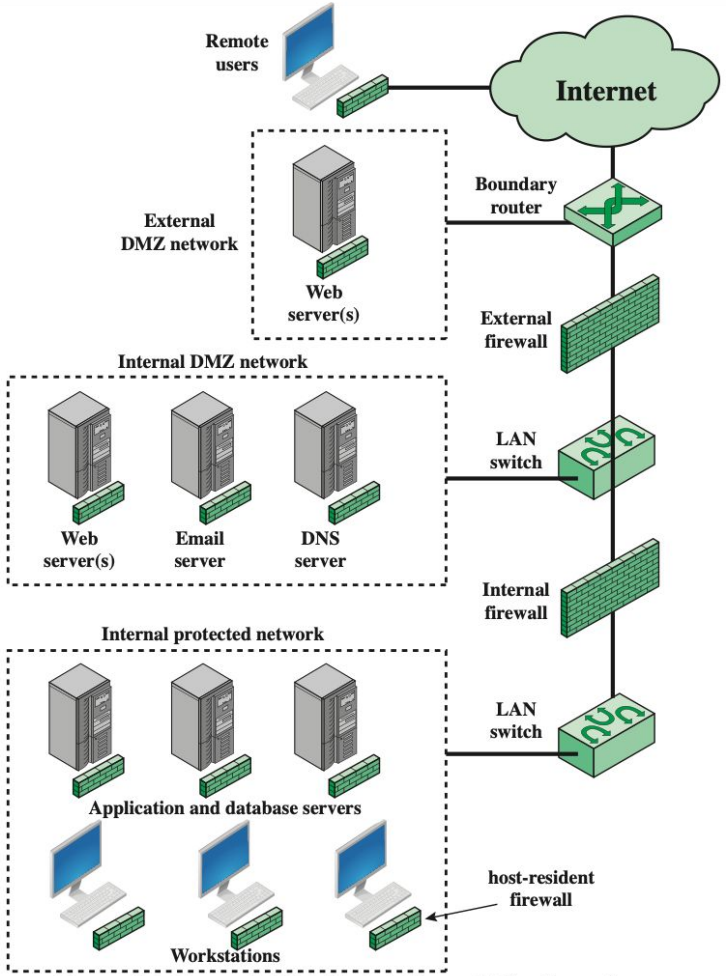


Figure 9.4 Example Distributed Firewall Configuration

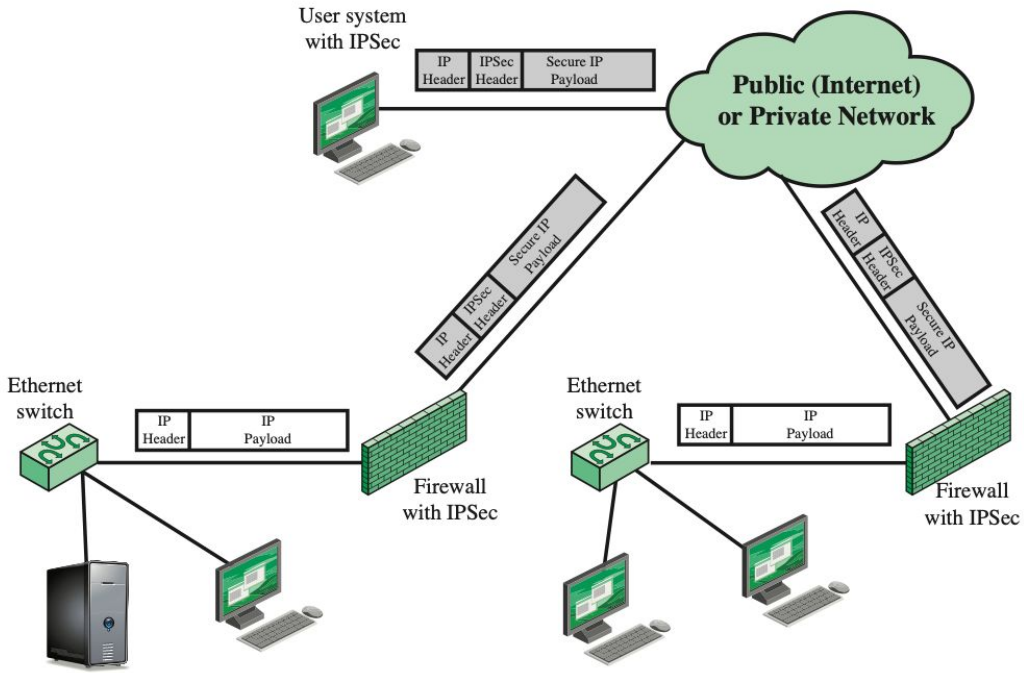


Figure 9.3 A VPN Security Scenario

A “real world” implementation: Linux’s NETFILTER and iptables

NETFILTER

- ***NETFILTER*** is a framework that provides hook handling within the Linux kernel for intercepting and manipulating network packets
- A hook is an “entry point” within the Linux Kernel IP (v4|v6) networking subsystem that allows packet mangling operations
- Packets traversing (incoming/outgoing/forwarded) the IP stack are intercepted by these hooks, verified against a given set of matching rules and processed as described by an action configured by the user
- 5 built-in hooks: ***PRE_ROUTING***, ***LOCAL_INPUT***, ***FORWARD***, ***LOCAL_OUT*** ***POST_ROUTING***

hook invocation from linux/net/ipv4/ip_input.c

```c
/*
 *      Deliver IP Packets to the higher protocol layers.
 */
int ip_local_deliver(struct sk_buff *skb)
{
        /*
         *      Reassemble IP fragments.
         */
        struct net *net = dev_net(skb->dev);

        if (ip_is_fragment(ip_hdr(skb))) {
                if (ip_defrag(net, skb, IP_DEFRAG_LOCAL_DELIVER))
                        return 0;
        }

        return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN,
                       net, NULL, skb, skb->dev, NULL,
                       ip_local_deliver_finish);
}
EXPORT_SYMBOL(ip_local_deliver);
```

LOCAL_INPUT

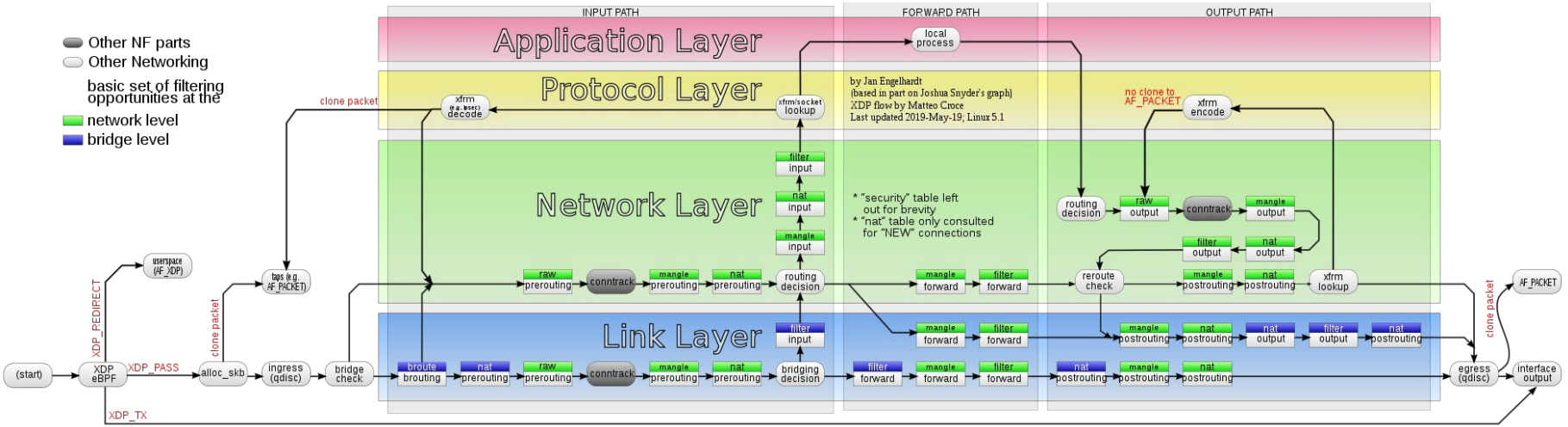
```c
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,
           struct net_device *orig_dev)
{
        struct net *net = dev_net(dev);

        skb = ip_rcv_core(skb, net);
        if (skb == NULL)
                return NET_RX_DROP;

        return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
                       net, NULL, skb, dev, NULL,
                       ip_rcv_finish);
}
```

PRE_ROUTING

NETFILTER

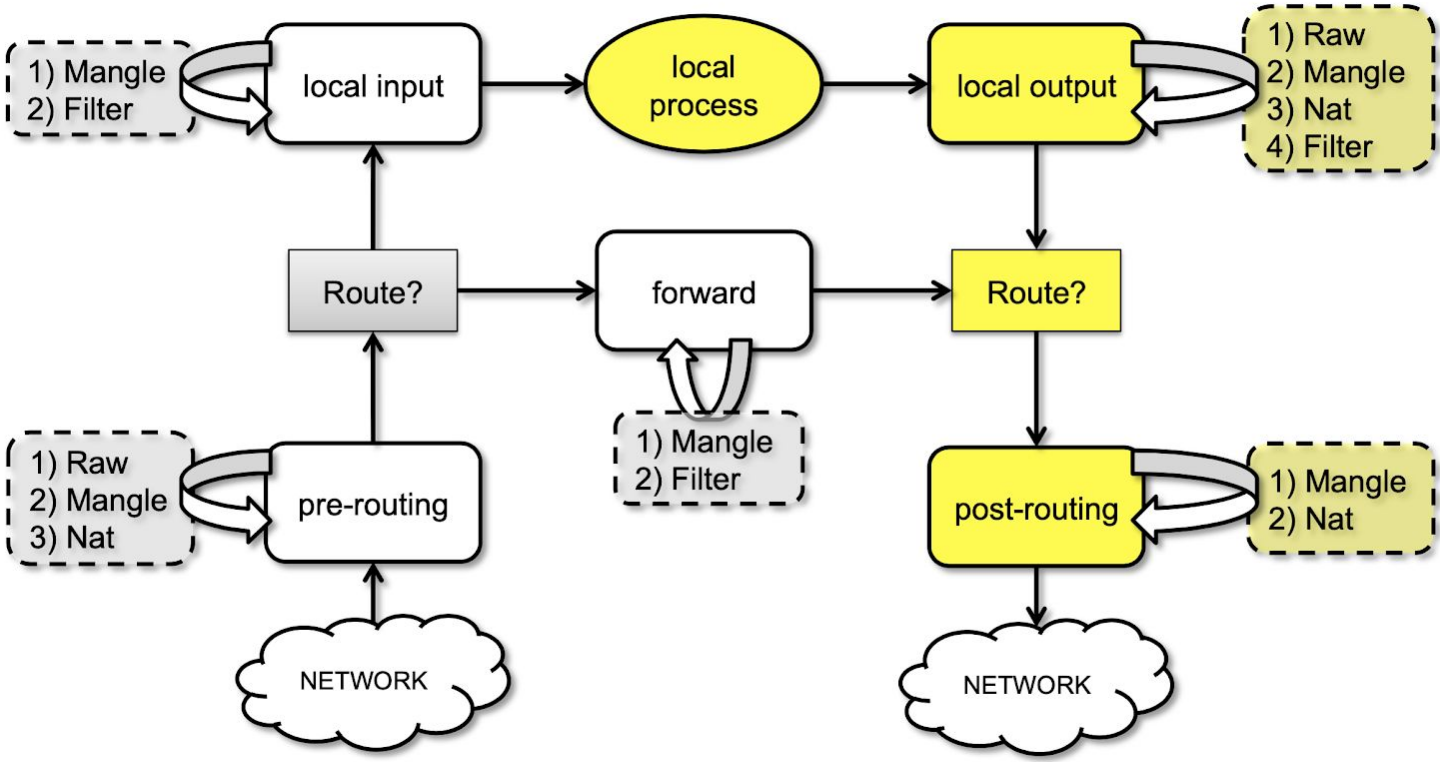
- All packet intercepted by the hooks pass through a sequence of ***built-in tables*** (queues) for processing
- Each of these queues is dedicated to a particular type of packet activity and is controlled by an associated packet transformation/filtering chain
- ***4 built-in tables***
  - **Filter**: packet filtering (accept, drop)
  - **Nat**: network address translation (snat, dnat, masquerade)
  - **Mangle**: modify the packet header (tos, ttl)
  - **Raw**: used mainly for configuring exemptions from connection tracking in combination with the NOTRACK target

Packet Flow in NETFILTER and General Networking

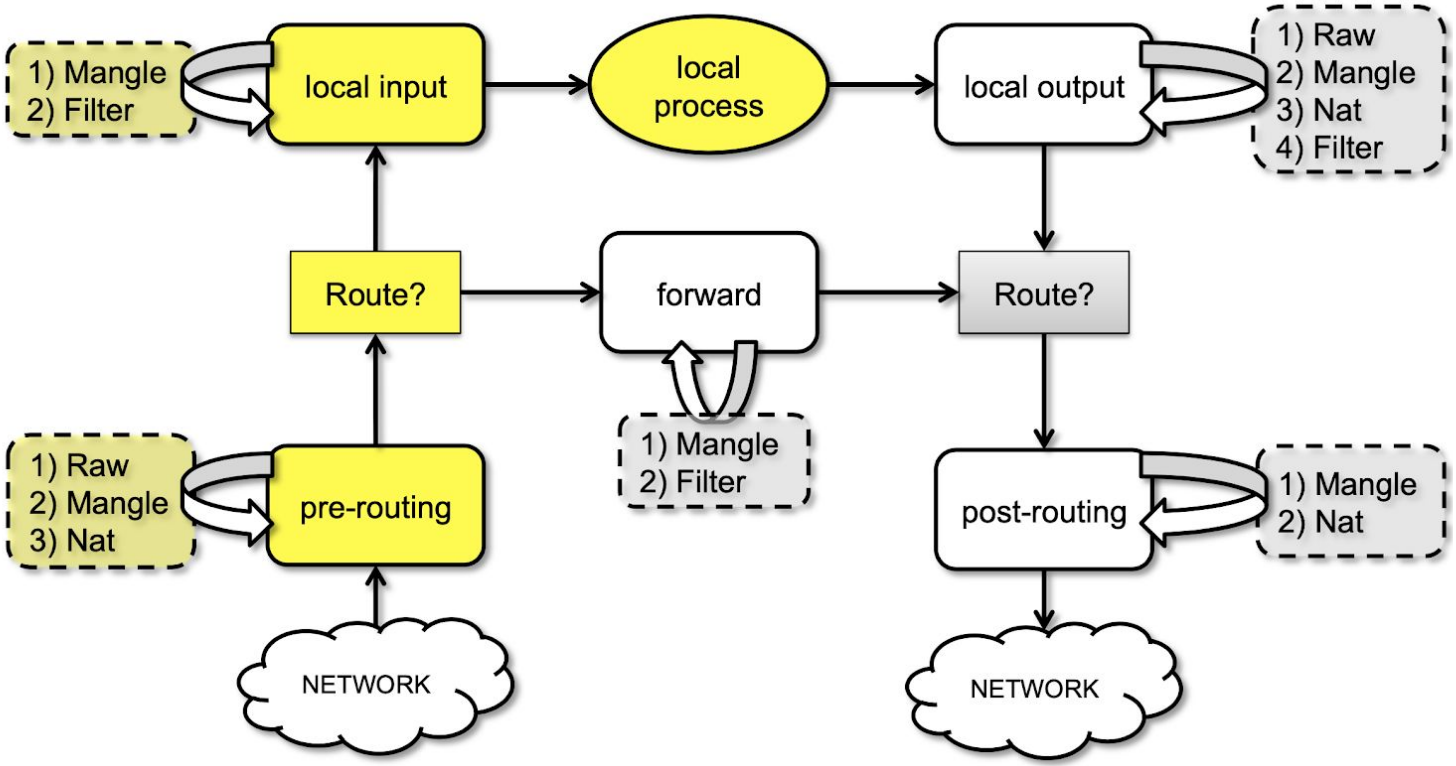


https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg

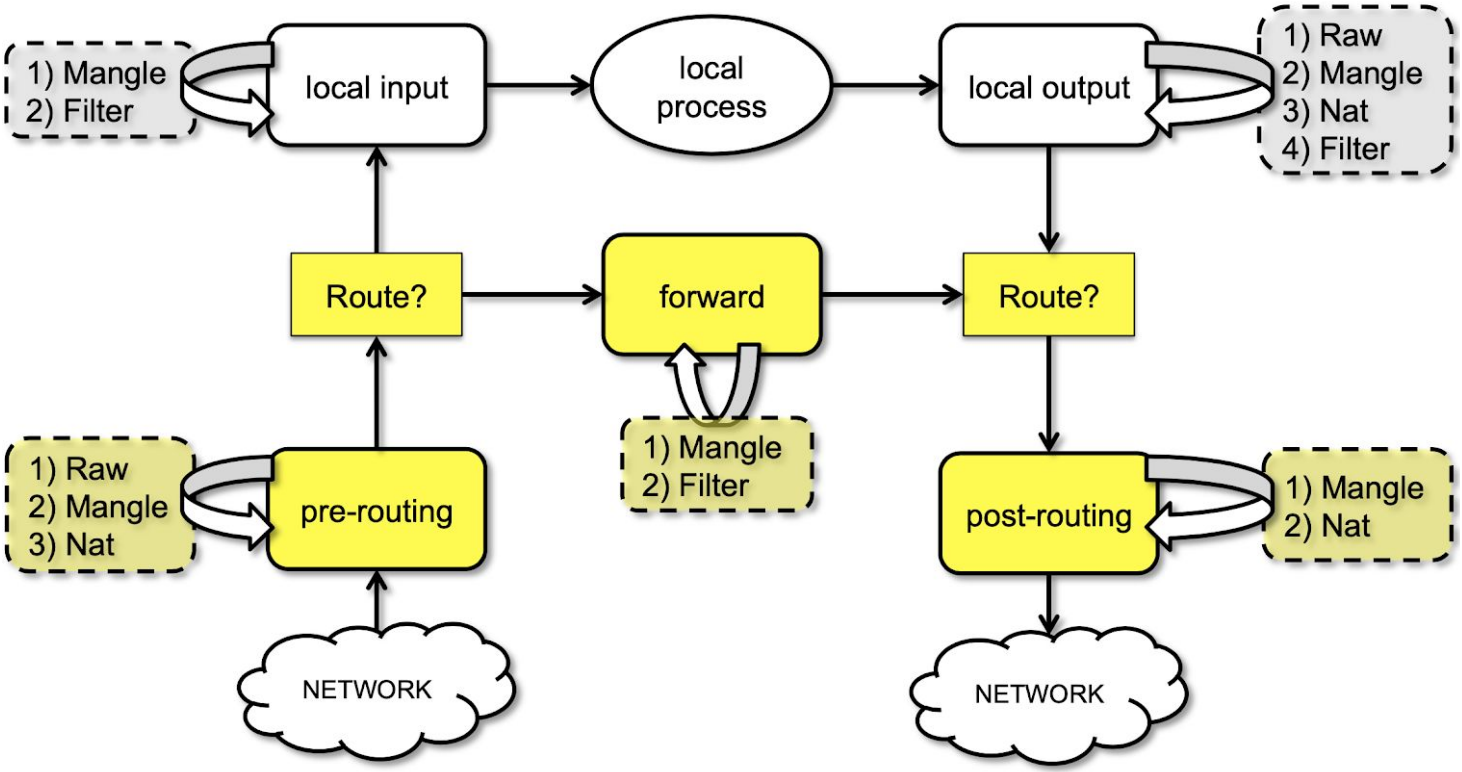
Packets sent from the local host



Packets sent to a local address



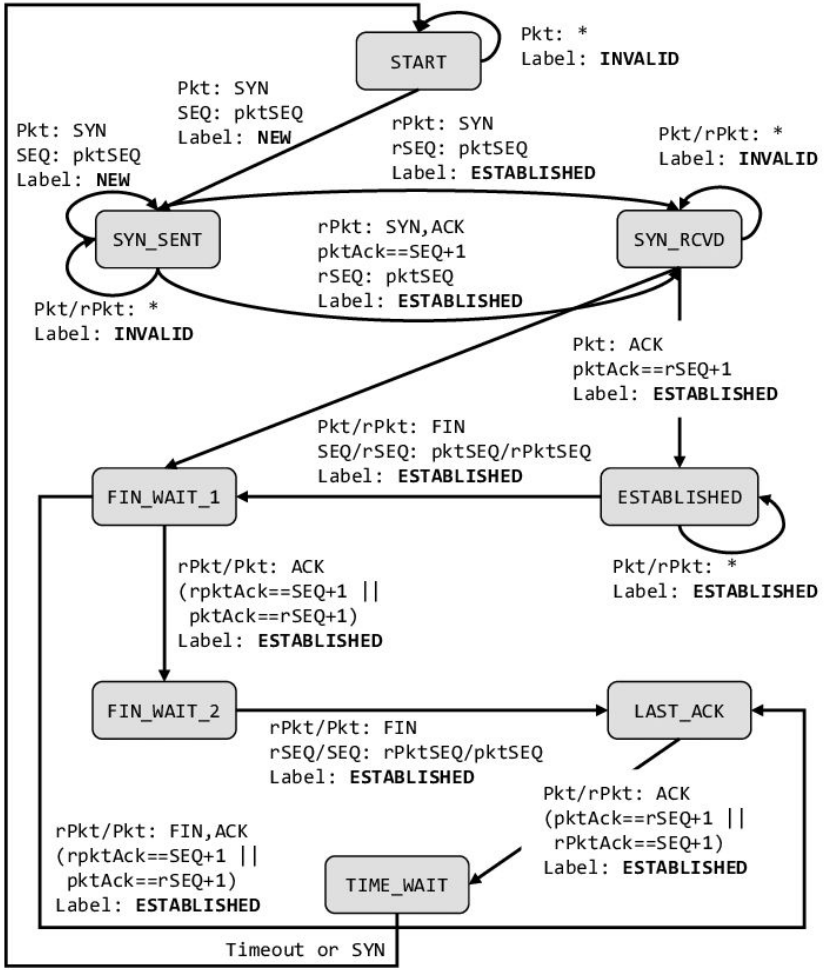
Forwarded Packets



Connection Tracking

- Within ***NETFILTER*** packets can be related to tracked connections in four different *so called* states
  - ***NEW***, ***ESTABLISHED***, ***RELATED***, ***INVALID***
- With the ***"state" match*** we can easily control who or what is allowed to initiate new sessions. More later on...
- To load the conntrack module – `modprobe ip_conntrack`
- /proc/net/ip_conntrack gives a list of all the current entries in your conntrack database
  - newer distributions does not have this file, use `conntrack -L` instead

TCP state machine



Linux conntrack

```
marlon@marlon-vmxbn:~$ sudo cat /proc/net/ip_conntrack
[sudo] password for marlon:

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=43716 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=43716 mark=0 use=2

tcp      6 431951 ESTABLISHED src=172.16.166.156 dst=172.16.166.2 sport=48680 dport=9999
src=172.16.166.2 dst=172.16.166.156 sport=9999 dport=48680 [ASSURED] mark=0 use=2

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=44936 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=44936 mark=0 use=2

udp      17 19 src=172.16.166.156 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.156 sport=5353 dport=5353 mark=0 use=2

tcp      6 431487 ESTABLISHED src=172.16.166.156 dst=172.16.166.1 sport=43733 dport=139
src=172.16.166.1 dst=172.16.166.156 sport=139 dport=43733 [ASSURED] mark=0 use=2

udp      17 28 src=172.16.166.156 dst=172.16.166.2 sport=43581 dport=53 src=172.16.166.2
dst=172.16.166.156 sport=53 dport=43581 mark=0 use=2
```

conntrack events

```
root@marlon-vmxbn:/home/marlon# conntrack --event
[NEW] udp      17 30 src=172.16.166.156 dst=172.16.166.156 sport=47282 dport=4444 [UNREPLIED]
src=172.16.166.156 dst=172.16.166.156 sport=4444 dport=47282

[DESTROY] udp      17 src=172.16.166.2 dst=172.16.166.156 sport=5353 dport=5353 [UNREPLIED]
src=172.16.166.156 dst=172.16.166.2 sport=5353 dport=5353

[DESTROY] udp      17 src=172.16.166.156 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.156 sport=5353 dport=5353

[DESTROY] udp      17 src=172.16.166.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
src=224.0.0.251 dst=172.16.166.1 sport=5353 dport=5353

[NEW] tcp      6 120 SYN_SENT src=172.16.166.156 dst=160.80.103.147 sport=45696 dport=80
[UNREPLIED] src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696

[UPDATE] tcp      6 60 SYN_RECV src=172.16.166.156 dst=160.80.103.147 sport=45696 dport=80
src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696

[UPDATE] tcp      6 432000 ESTABLISHED src=172.16.166.156 dst=160.80.103.147 sport=45696
dport=80 src=160.80.103.147 dst=172.16.166.156 sport=80 dport=45696 [ASSURED]
```

Application Level Gateway (ALG) related state

- protocols like FTP, IRC, and others carry information within the actual data payload of the packets, and hence requires special connection tracking helpers to enable it to function correctly
- For example, FTP first opens up a single connection that is called the FTP control session and negotiate the opening of the data session over a different socket
- When a connection is done actively, the FTP client sends the server a port and IP address to connect to. After this, the FTP client opens up the port and the server connects to that specified port from a random unprivileged port (>1024) and sends the data over it
- A special NETFILTER conntrack helper can read the FTP control payload and read the “data port”
- The new data socket will be considered as RELATED

iptables

- ***iptables is the front end of NETFILTER***
- In other words, iptables is the userspace application used to configure the NETFILTER tables
- It is mainly used to add/remove rules to a chain (mapping of NETFILTER hooks) within a table
- General structure for adding a rule:

```
iptables <command> <chain> <table> <match> <target>
iptables -A POSTROUTING -t nat -o eth0 -j MASQUERADE
```

iptables

- **`iptables`** is used to set up, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel
- Several different ***chains*** may be defined. Each chain contains a number of built-in ***tables***
  - a chain could be also custom, i.e. defined by the user
- In each table, there is a list of rules which can ***match*** a set of packets
- Each rule specifies what to do with a packet that matches. This is called a ***target***,
  - which may be a jump to a user-defined chain in the same table
  - or a standard *action* (DROP, ACCEPT…)

iptables commands

Append, delete, insert, replace rules

```
iptables [-t table] {-A|-D} chain rule-specification
iptables [-t table] -D chain rulenum
iptables [-t table] -I chain [rulenum] rule-specification
iptables [-t table] -R chain rulenum rule-specification
```

List, flush rules

```
iptables [-t table] -S [chain [rulenum]]
iptables [-t table] -{F|L} [chain [rulenum]] [options...]
```

Create, delete, rename chains and set policy to a chain

```
iptables [-t table] -N chain
iptables [-t table] -X [chain]
iptables [-t table] -E old-chain-name new-chain-name
iptables [-t table] -P chain target
```

Where:

```
rule-specification = [matches...] [target]
match = -m matchname [per-match-options]
target = -j targetname [per-target-options]
```

Queue Type	Queue Function	Packet Transformation Chain in Queue	Chain Function
Filter	Packet filtering	`FORWARD`	Filters packets to servers accessible by another NIC on the firewall.
		`INPUT`	Filters packets destined to the firewall.
		`OUTPUT`	Filters packets originating from the firewall
Nat	Network Address Translation	`PREROUTING`	Address translation occurs before routing. Facilitates the transformation of the destination IP address to be compatible with the firewall's routing table. Used with NAT of the destination IP address, also known as **destination NAT** or **DNAT**.
		`POSTROUTING`	Address translation occurs after routing. This implies that there was no need to modify the destination IP address of the packet as in pre-routing. Used with NAT of the source IP address using either one-to-one or many-to-one NAT. This is known as **source NAT**, or **SNAT**.
		`OUTPUT`	Network address translation for packets generated by the firewall. (Rarely used in SOHO environments)
Mangle	TCP header modification	`PREROUTING` `POSTROUTING` `OUTPUT` `INPUT` `FORWARD`	Modification of the TCP packet quality of service bits before routing occurs. (Rarely used in SOHO environments)

iptables TARGETS

- A firewall rule specifies criteria for a packet and a target. If the packet does not match, the next rule in the chain is the examined;
- If the packet does match, then the next rule is specified by the value of the target (option -j), which can be the name of a user-defined chain or one of the special (standard) values
  - ***ACCEPT*** means to let the packet through (no other rules will be checked)
  - ***DROP*** means to drop the packet on the floor
  - ***QUEUE*** means to pass the packet to userspace
  - ***RETURN*** means stop traversing this chain and resume at the next rule in the previous (calling) chain. If the end of a built-in chain is reached or a rule in a built-in chain with target RETURN is matched, the target specified by the chain policy determines the fate of the packet

- More targets with target extensions. More later on...

iptables matches

- in this class we are going to use the following iptables matches
  - IP source address (also net supported): **`-s $address`**
  - IP destination address (also net supported): **`-d $address`**
  - IP protocol: **`-p tcp|udp [--sport $sp] [--dport $dp]`**
  - input interface: **`-i $iif`**
  - output interface: **`-o $oif`**
  - state: **`-m state --state NEW|ESTABLISHED`**

Example (not all matches are mandatory. matches can be arbitrary combined. all matches in the same rule are in logical AND)

```
iptables -A INPUT -s 10.0.0.100 -d 10.0.0.1 -p tcp --dport
80 --sport 54400 -j ACCEPT
```

that's all with the theory

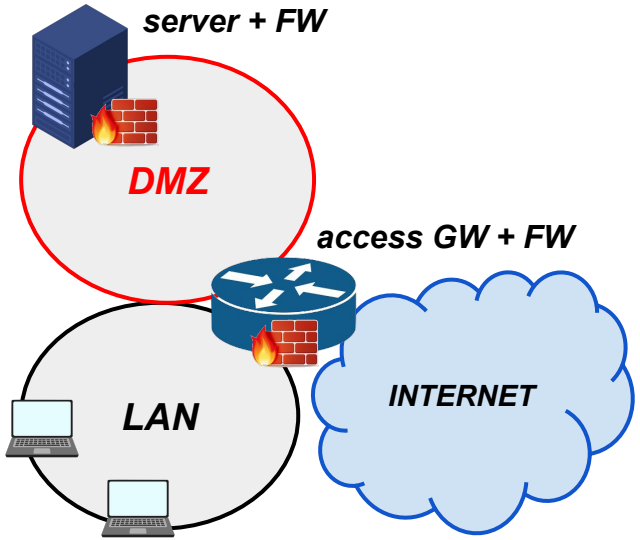
- a lot of further details should be discussed…
- ***… better learn by doing!***

Access GW security policy plan

- **Permit** traffic between DMZ and the INTERNET
- **Permit** traffic between LAN and DMZ only if initiated from LAN
- **Permit** SSH, HTTP, HTTPS and DNS traffic between LAN and INTERNET only if initiated from LAN
- **Deny** all traffic to GW except ssh and expect reply packets for locally initiated flows
- **Deny** all traffic initiated from DMZ to GW
- **Permit** traffic from GW to anywhere
- **Permit** all ICMP traffic (e.g. echo request/reply, port unreach, etc..)

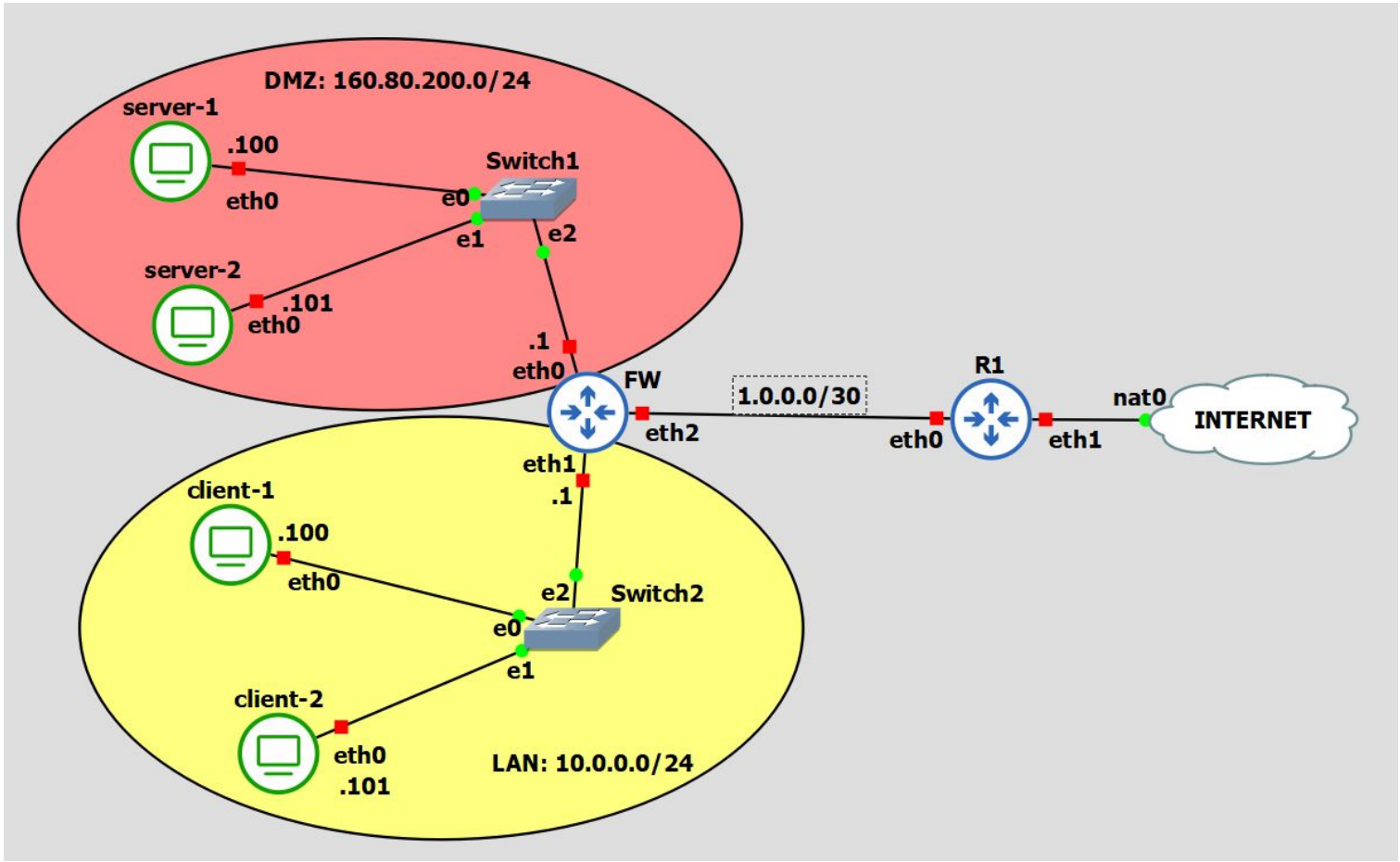
Server security policy plan

- **Permit** incoming traffic only for HTTP, HTTPS, SSH and all traffic related to connections locally initiated
- **Permit** all outgoing traffic

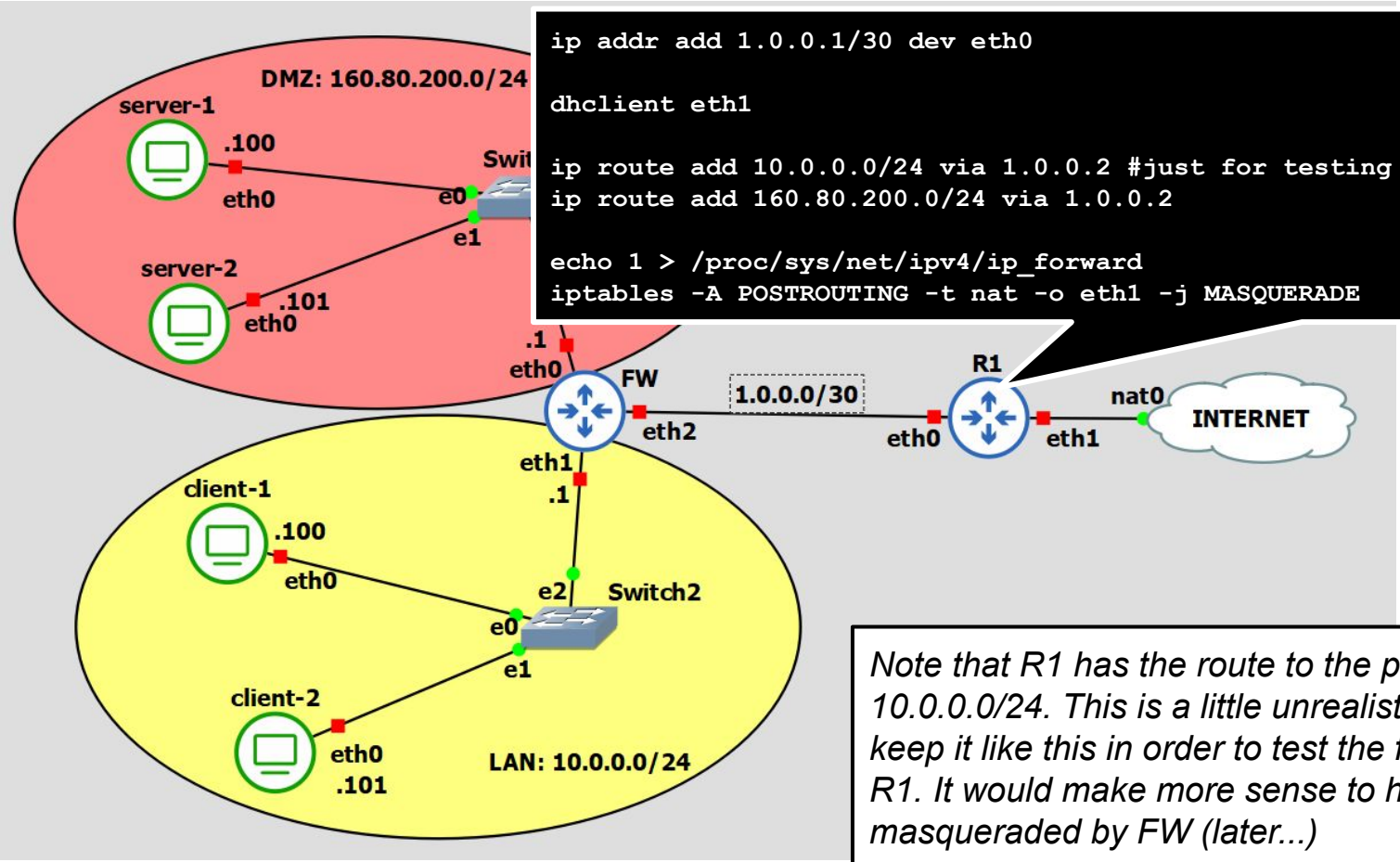


high-level topology

Detailed Topology

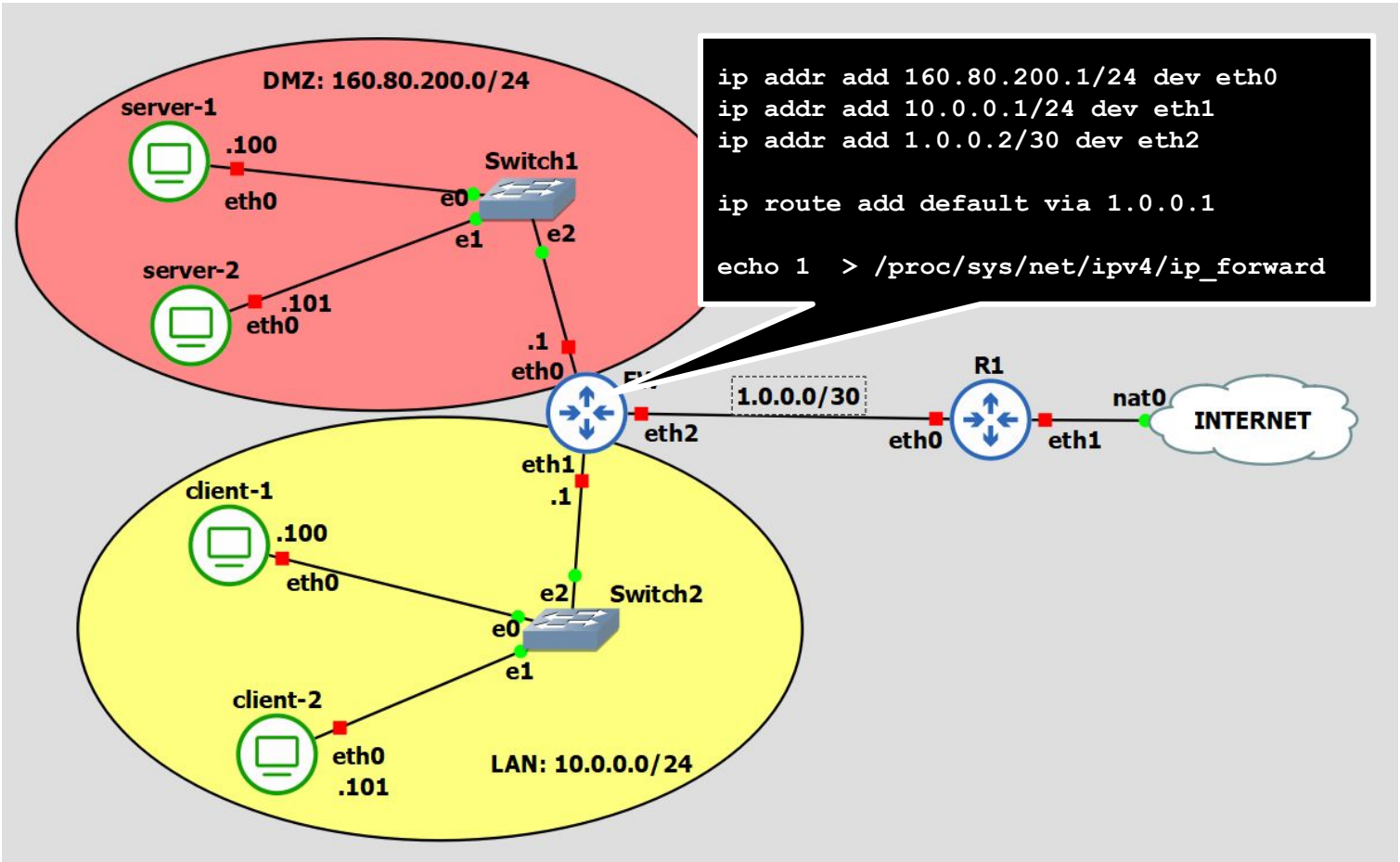


Detailed Topology

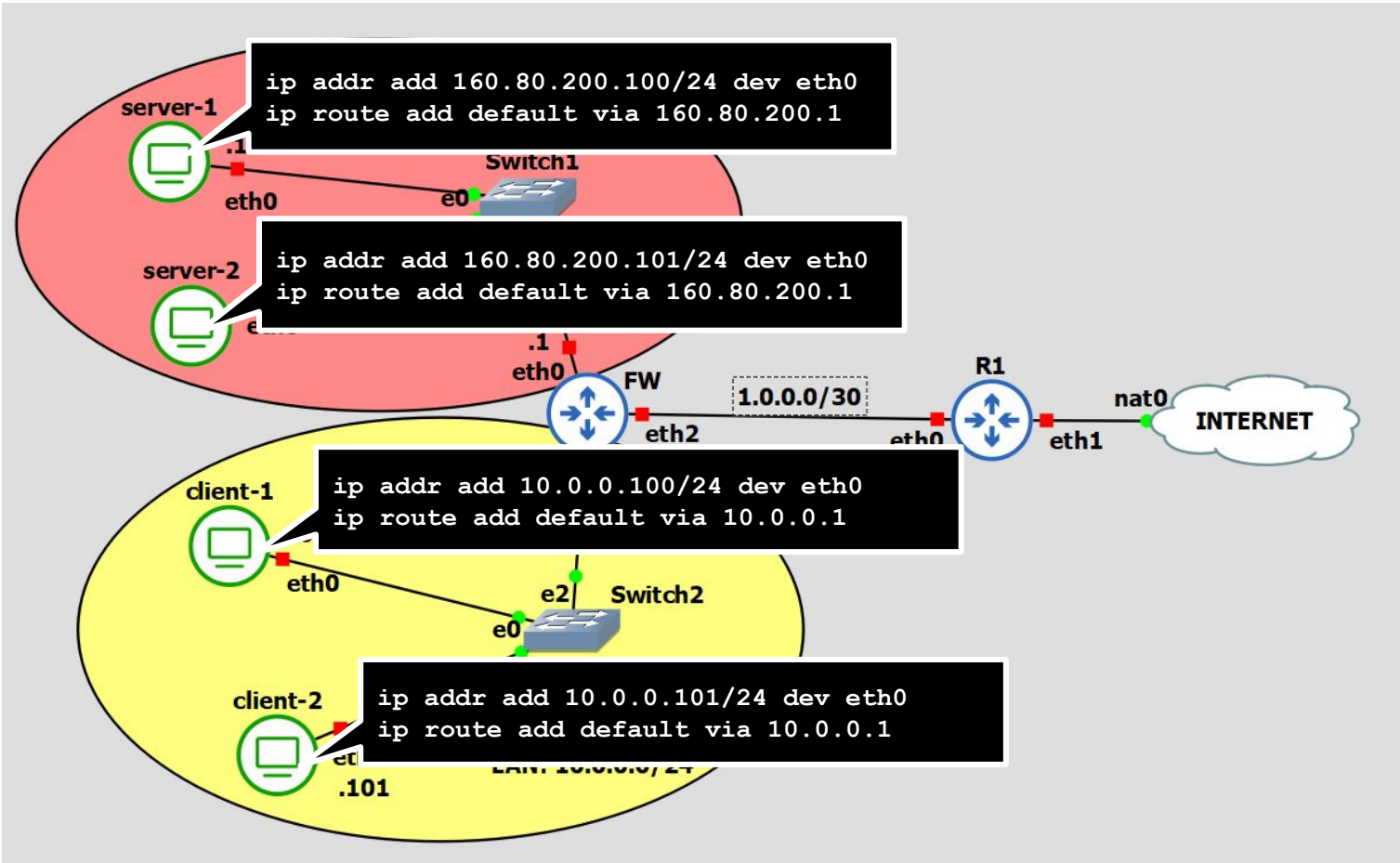


Note that R1 has the route to the private net 10.0.0.0/24. This is a little unrealistic but let's keep it like this in order to test the firewall from R1. It would make more sense to have this LAN masqueraded by FW (later...)

Detailed Topology



Detailed Topology



```
iptables -F  # flush already present entries
iptables -P FORWARD DROP
iptables -P INPUT DROP
iptables -P OUTPUT ACCEPT

iptables -A FORWARD -m state --state ESTABLISHED -j ACCEPT

iptables -A FORWARD -i $LAN -p tcp --dport 80 -j ACCEPT
iptables -A FORWARD -i $LAN -p tcp --dport 443 -j ACCEPT
iptables -A FORWARD -i $LAN -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i $LAN -p udp --dport 53 -j ACCEPT

iptables -A FORWARD -i $LAN -o $DMZ -j ACCEPT
iptables -A FORWARD -i $WAN -o $DMZ -j ACCEPT
iptables -A FORWARD -i $DMZ -o $WAN -j ACCEPT


iptables -A INPUT -m state --state ESTABLISHED -j ACCEPT
iptables -A INPUT -i $LAN -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -i $WAN -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p icmp -j ACCEPT

iptables -A FORWARD -p icmp -j ACCEPT
```

GW config

```
#interfaces

export LAN=eth1
export DMZ=eth0
export WAN=eth2
```

```
iptables -F
iptables -P INPUT DROP
iptables -P OUTPUT ACCEPT

iptables -A INPUT -m state --state ESTABLISHED -j ACCEPT
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

server config

How to test the firewall

- the simplest way is tot use the network tool **`nc`** (**`netcat`**)
- **`nc`** allows to open client and server socket (TCP and UDP) with arbitrary port
- syntax for clients
  - **`nc [-uv] $addr $port`**
    - -v for verbose, -u for UDP (default TCP)
- syntax for servers
  - **`nc [-uv] -l -p $port`**
    - -v for verbose, -u for UDP (default TCP)
- For incoming connections we can use R1 as a client (that's why we have the route to 10.0.0.0/24)
- For outgoing connections we may also connect to real servers on the internet

Network Address Translation

- NETFILTER also support network address translations
- Dynamic source address translation:
  - `-j MASQUERADE`
- Static destination address translation (port forwarding)
  - `-j DNAT --to-destination $addr:[$port]`
- Local redirect (remember the DNS spoofing LAB??)
  - `-j REDIRECT`

Packet Classification

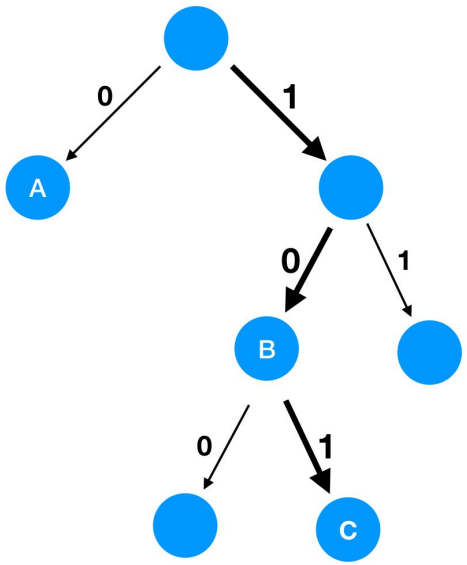
Source: George Varghese, "Network Algorithmics. An Interdisciplinary Approach to Designing Fast Networked Devices", Chapters 11 and 12

Why packet classification?

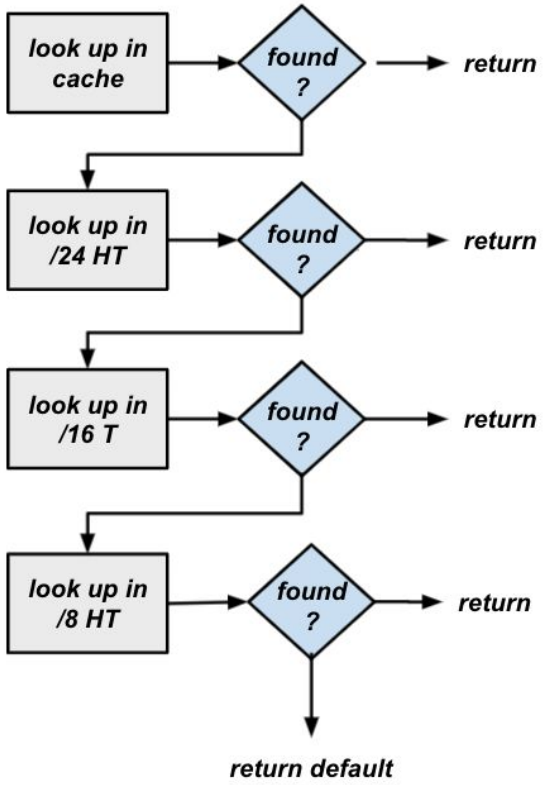
- ***Critical operation not only for firewalls***
  - Resource reservations, QoS routing, unicast routing, multicast routing, IDS, NAT, monitoring, traffic engineering, etc..
- Packet forwarding based on a longest-matching-prefix lookup of destination IP addresses is fairly well understood
  - LPM can be seen as a specific packet classification primitive
  - Efficiently solved with algorithmic solutions (basic variants of tries) and CAM pipelines
- Unfortunately, the Internet is becoming more complex because of its use for mission critical functions executed by organizations
  - Both QoS and security guarantees require a finer discrimination of packets, based on fields other than the destination
  - Examples of other fields a router may need to examine include source addresses (to forbid or provide different service to some source networks), port fields (to discriminate between traffic types, such as Napster and E-mail), and even TCP flags (to distinguish between externally and internally initiated connections)

Algorithmic LPM Approaches

Prefix	Router
0*	A
10*	B
101*	C



LPM Trie



LPM with multiple HTs (basic idea)

The Packet Classification Problem

- The matching rules for classifying a message are called matching **rules** and the packet-classification problem is to determine the ***lowest-cost matching rule for each incoming message***
- Information relevant to a lookup is contained in ***K*** distinct ***header fields***
- Header fields are denoted ***H[1], H[2], . . . , H[K]***
- The ***classifier*** consists of a finite set of rules, $R_1, R_2, \ldots, R_N$
- Each field in a rule is allowed three kinds of matches: ***(i) exact match***, ***(ii) prefix match***, and ***(iii) range match***
- Each rule R_i has an associated directive $disp_i$, which specifies how to process the packet matching this rule
  - in a (basic) firewall ***ACCEPT***, ***DROP***

The Packet Classification Problem

- A packet ***P*** is said to match a rule ***R*** if each field of P matches the corresponding field of R
- The match type is implicit in the specification of the field
  - if the destination field is specified as 1010∗, then it requires a prefix match
  - if the protocol field is UDP, then it requires an exact match
  - if the port field is a range, such as 1024–1100, then it requires a range match
- For instance, let ***R = (1010∗, ∗, TCP, 1024:1080, ∗)*** be a rule with ***disp=DROP***
- a packet with header ***(10101 . . . 111, 11110 . . . 000, TCP, 1050, 3)*** matches R and is therefore blocked
- The packet ***(10110 . . . 000, 11110 . . . 000, TCP, 80, 3)*** doesn't match R

The Packet Classification Problem

- Since a packet may match multiple rules in the database, each rule R in the database is associated with a nonnegative number, ***cost(R)***.
- Ambiguity is avoided by returning the ***least-cost rule matching the packet's header***.
- The cost function generalizes the implicit precedence rules that are used in practice to choose between multiple matching rules.
- In firewall applications or Cisco ACLs, for instance, rules are placed in the database in a specific linear order, where each rule takes precedence over a subsequent rule.
- Thus, the goal there is to find the first matching rule.
- Of course, the same effect can be achieved by making cost(R) equal to the position of rule R in the database.

Firewall example

Destination	Source	Destination Port	Source Port	Flags	Comments
M	*	25	*	*	Allow inbound mail
M	*	53	*	UDP	Allow DNS access
M	S	53	*	*	Secondary access
M	*	23	*	*	Incoming telnet
TI	*TO*	123	123	UDP	NTP time info
*	Net	*	*	*	Outgoing packets
Net	*	*	*	TCP ack	Return ACKs OK
*	*	*	*	*	Block everything!

Requirements and Metrics

- The requirements for rule matching are similar to those for IP lookups
- We wish to do packet classification at wire speed for minimum-size packets, and thus ***speed is the dominant metric***
- To allow the database to fit in high-speed memory it is useful to ***reduce the amount of memory needed***
- For most firewall databases, insertion speed is not an issue because rules are rarely changed
  - However, this is not always true for dynamic or stateful packet rules. E.g.: ***UDP "connection" tracking***
  - In some products the solution is to have the outgoing request packet dynamically trigger the insertion of a rule (which has addresses and ports that match the request) that allows the inbound response to be passed
  - This requires very ***fast update times*** (a third metric)

netfilter/iptables is not really scalable...

- **`iptables`** has been the primary tool to implement firewalls and packet filters on Linux for many years
- Over the years, **`iptables`** has been a blessing and a curse
  - A blessing for its flexibility and quick fixes
  - A curse during times debugging a 5K rules iptables setup in an environment where multiple system components are fighting over who gets to install what iptables rules
- Back then, network speeds were slow (in ‘95 the most common router would provide 56 kbps throughput). ***Today: 802.11ax (theoretically) up to 10 gbps***
- The standard practice of implementing access control lists (ACLs) as implemented by iptables was to use ***sequential list of rules***
  - i.e. every packet received or transmitted is matched against a list of rules, one by one
- ***However, linear processing has an obvious massive disadvantage, the cost of filtering a packet can increase linearly with the number of rules added***

nice reading: https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables

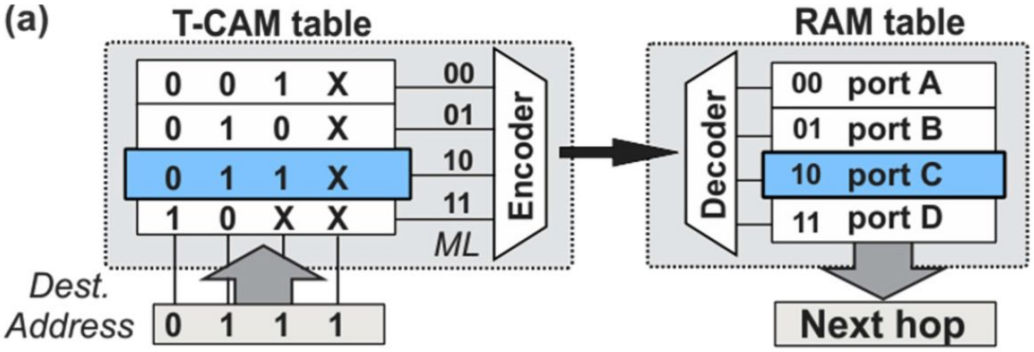
A simple solution: Caching

- Some implementations cache the result of the search keyed against the whole header
  - usually the socket 5-tuple is used as cache key
- The ***cache hit rate of caching full IP addresses*** in the backbones is typically ***at most 80–90%*** [Par96][NMH97]
  - Web accesses and other flows that send only a small number of packets
  - Caching full headers takes a lot more memory
  - In reality an eviction policy (e.g. LRU) is most likely enforced
  - ***So the cache hit rate is even worse***
- Even if the hit rate was 90%, the 10% in the slow match path still inexorably degrades the performance
  - suppose that a search of the cache costs 100 nsec (one memory access) and that a linear search of 10,000 rules costs 1,000,000 nsec = 1 msec (one memory access per rule). Then the average search time with a cache hit rate of 90% is still 0.1 msec, which is rather slow
- ***Even with caching, we need a fast matching algorithms***

A better solution: Content-Addressable Memories (CAM)

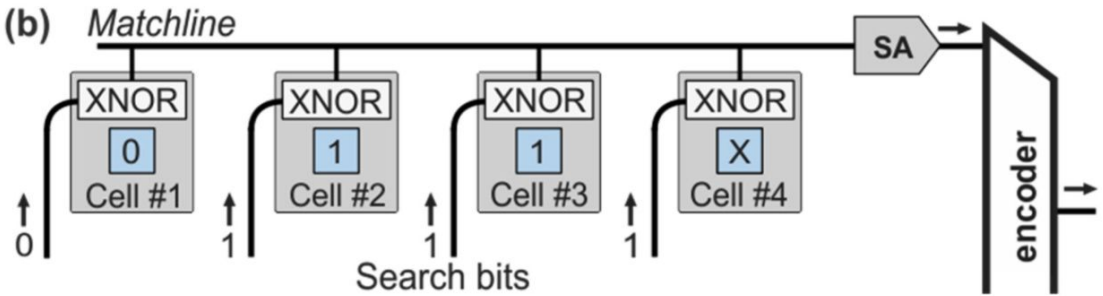
- A ***CAM*** is a ***content-addressable memory***, where the first cell that matches a data item will be returned using a parallel lookup in hardware
- A ***ternary CAM*** (TCAM) allows each bit of data to be either a 0, a 1, or a wildcard
  - Clearly, ternary CAMs can be used for rule matching as well as for prefix matching
  - However, the CAMs must provide wide lengths
    - ipv4 → 32 ip.src + 32 ip.dst + 16 sport + 16 dport + 8 ip.proto = **104 bits**
    - ipv6 → 128 ip.src + 128 ip.dst + 16 sport + 16 dport + 8 ip.proto = **296 bits**

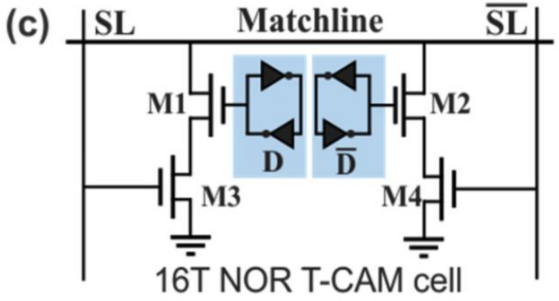
TCAMs for routing look-up



Router RIB Table

RIB entry	Destination address	Next hop port
1	001X	Port A
2	010X	Port B
3	011X	Port C
4	10XX	Port D





CAMs: not always a viable solution

- Several reasons to consider ***algorithmic alternatives to ternary CAMs***
  - Smaller density and larger power of CAMs versus SRAMs
  - Difficulty of integrating forwarding logic with the CAM
  - Rule multiplication caused by ranges
  - Several CAM vendors were also considering algorithmic solutions
  - No CAMs in SW packet processing
    - we still need fast SW implementations (especially with new NFV programming models)
- **Examples of algorithmic approaches:**
  - Tries: Multi-dimensional schemes
  - Binary search

An alternative approach: divide-and-conquer

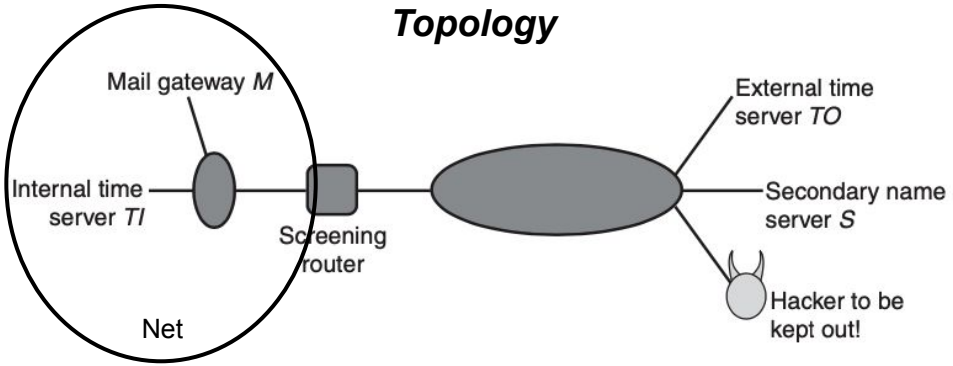
- ***Divide-and-conquer*** refers to dividing a problem into simpler pieces and then efficiently combining the answers from the pieces
- Start by slicing the rule database into columns, with the ***ith*** column storing all distinct prefixes (or ranges) in field ***i***
- Given a packet ***P***, determine the matching prefixes for each of its fields separately
- Finally, combine the results of the best-matching-prefix lookups on individual fields
- 3 methods that differ from each other in how the lookup of individual fields is combined into a single compound lookup
  - ***(i) Bit Vector Linear Search***; ***(ii) Cross-producting***; ***(iii) Decision Tree Approach***

Reference example

Firewall rule database

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	*M*	*	25	*	*	Allow inbound mail
2	*M*	*	53	*	UDP	Allow DNS access
3	*M*	S	53	*	*	Secondary access
4	*M*	*	23	*	*	Incoming telnet
5	*TI*	*TO*	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Topology

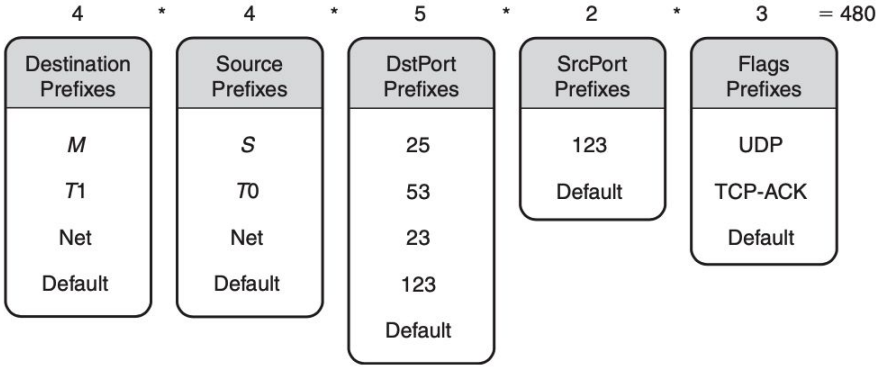


Reference example

Firewall rule database

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	*M*	*	25	*	*	Allow inbound mail
2	*M*	*	53	*	UDP	Allow DNS access
3	*M*	S	53	*	*	Secondary access
4	*M*	*	23	*	*	Incoming telnet
5	*TI*	*TO*	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

"Sliced" database



Bit Vector Linear Search

- Any match X in a given single field DB excludes the entries that don't match X in the other single field DBs
  - e.g.: a packet with destination address in T1 matches rules 5, 6, 7, 8 in DB 1
  - rules 1, 2, 3 and 4 can be "ignored" also in the other DBs
- The search algorithm needs to search only the intersection of the remaining sets obtained by each field lookup
- This would clearly be a good heuristic for optimizing the average case if the remaining sets are typically small
- one can guarantee performance even in the worst case
  - for each single DB we store a bitmask indicating which rules match the input packet
  - the intersection is obtained by simply ***ANDing*** the k bitmasks (k = number of match fields)
  - ***if the rules are ordered by descending priority, the best rule is the one associated to the first bit from the left set to 1***

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	*M*	*	25	*	*	Allow inbound mail
2	*M*	*	53	*	UDP	Allow DNS access
3	*M*	S	53	*	*	Secondary access
4	*M*	*	23	*	*	Incoming telnet
5	*TI*	*TO*	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M								
T1								
Net								
*								

4 different values, 8 bit bitmask

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53	*	*	Secondary access
4	M	*	23	*	*	Incoming telnet
5	T1	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M								
T1								
Net								
*								

if destination matched M, rules 5 never matches

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53	*	*	Secondary access
4	M	*	23	*	*	Incoming telnet
5	T1	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M	1	1	1	1	0	1	1	1
T1								
Net								
*								

resulting bitmask = 11110111

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53			Secondary access
4	M	*	23	*	*	Incoming telnet
5	TI	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M	1	1	1	1	0	1	1	1
T1								
Net								
*								

if destination matches T1, rules 1,2, 3 and 4 never match

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53			Secondary access
4	M	*	23	*	*	Incoming telnet
5	TI	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M	1	1	1	1	0	1	1	1
T1	0	0	0	0	1	1	1	1
Net								
*								

resulting bitmask = 00001111

Destination field DB extraction from full DB

	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53			Secondary access
4	M	*	23	*	*	Incoming telnet
5	TI	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net	*	*	*	TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

field	b0	b1	b2	b3	b4	b5	b6	b7
M	1	1	1	1	0	1	1	1
T1	0	0	0	0	1	1	1	1
Net	0	0	0	0	0	1	1	1
*								

and so on...

Destination field DB extraction from full DB

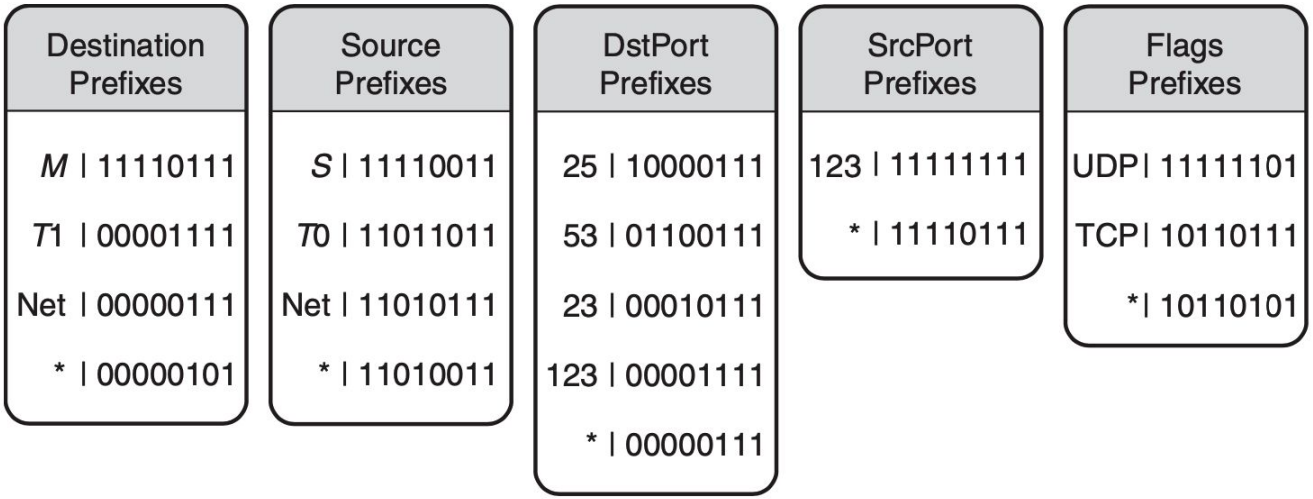
	Destination	Source	Destination Port	Source Port	Flags	Comments
1	M	*	25	*	*	Allow inbound mail
2	M	*	53	*	UDP	Allow DNS access
3	M	S	53			Secondary access
4	M	*	23	*	*	Incoming telnet
5	TI	TO	123	123	UDP	NTP time info
6	*	Net	*	*	*	Outgoing packets
7	Net				TCP ack	Return ACKs OK
8	*	*	*	*	*	Block everything!

Field1 DB

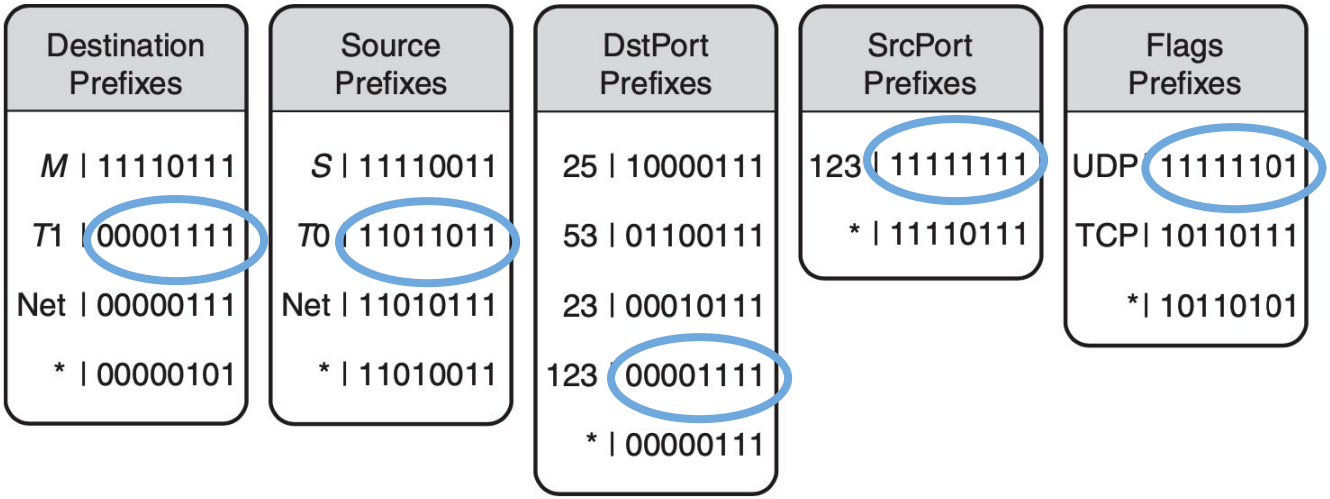
field	b0	b1	b2	b3	b4	b5	b6	b7
M	1	1	1	1	0	1	1	1
T1	0	0	0	0	1	1	1	1
Net	0	0	0	0	0	1	1	1
*	0	0	0	0	0	1	0	1

and so on...

Bit Vector Linear Search example ("disjoint" prefix ranges)



Bit Vector Linear Search example ("disjoint" prefix ranges)



look up keys: T1, T0, 123, 123, UDP

Bit Vector Linear Search example (“disjoint” prefix ranges)

```
0 0 0 0 1 1 1 1
      AND
1 1 0 1 1 0 1 1
      AND
0 0 0 0 1 1 1 1
      AND
1 1 1 1 1 1 1 1
      AND
1 1 1 1 1 1 0 1
```

Bit Vector Linear Search example (“disjoint” prefix ranges)

```
0 0 0 0 1 1 1 1
      AND
1 1 0 1 1 0 1 1
      AND
0 0 0 0 1 1 1 1
      AND
1 1 1 1 1 1 1 1
      AND
1 1 1 1 1 1 0 1
       =
0 0 0 0 1 0 0 1
```

Bit Vector Linear Search example ("disjoint" prefix ranges)

```
0 0 0 0 1 1 1 1
      AND
1 1 0 1 1 0 1 1
      AND
0 0 0 0 1 1 1 1
      AND
1 1 1 1 1 1 1 1
      AND
1 1 1 1 1 1 0 1
        =
0 0 0 0 1 0 0 1
```

matched rule: #5

Destination	Source	Destination Port	Source Port	Flags	Comments
M	*	25	*	*	Allow inbound mail
M	*	53	*	UDP	Allow DNS access
M	S	53	*	*	Secondary access
M	*	23	*	*	Incoming telnet
TI	*TO*	123	123	UDP	NTP time info
*	Net				Outgoing packets
Net	*	*	*	TCP ack	Return ACKs OK
*	*	*	*	*	Block everything!

But what if I found multiples matching rules?

- In case of a simple DROP/ACCEPT firewall, we only need to verify if the final bitvector result is != 0
- But in more complex cases, we need to understand what is the lowest cost rule (or the highest priority matching rule)
- Assuming a descending order, the highest priority rule is the one related to the leftmost bit set to 1 in the final bitvector result
- Searching for this has a linear complexity in a naive approach
- ***We need something more efficient!***
- ***Possible Solution:*** Leiserson et al. "Using de Bruijn sequences to index a 1 in a computer word"
  - https://www.academia.edu/download/41176510/0fcfd5107691fdedff000000.pdf20160115-19908-1ocbbzi.pdf
  - inspired by Miano, Sebastiano, et al. "Securing Linux with a faster and scalable iptables." ACM SIGCOMM Computer Communication Review 49.3 (2019). ***(Further details in the next laboratory)***

How do I search for the leftmost bit set to 1?

- Naive approach: linear search
  - linear complexity
- Better approach:
  - indexing all possible combination of the substrings for each index n
  - lookup time constant
- Shortcoming
  - exponential memory explosion

word	index
000	0
100	1
110	1
101	1
111	1
010	2
011	2
001	3

Optimization

Look for the position of the first bit to 1 in the bitvector using the de Bruijn sequences to find the index of the first bit set in a single word [Mia19]

Algorithm at a glance (input word x, len = n bits)

1. isolate the 1 (*)

   ```
   y = x & (-x)
   ```

2. define a hash function for indexing the n different “one-1 words”

   ```
   h(z)=z*DeBruijn>>(n-log2(n))
   ```

3. find the index of the lowest order 1 in x

   ```
   index = h(x & (-x))
   ```

Using de Bruijn Sequences to Index a 1 in a Computer Word

Charles E. Leiserson
Harald Prokop
Keith H. Randall
MIT Laboratory for Computer Science, Cambridge, MA 02139, USA
{cel,prokop,randall}@lcs.mit.edu

July 7, 1998

(*) with this operation, we isolate the rightmost bit (i.e. in reverse order w.r.t. previous examples)

32-bit C implementation of the DeBruijn strategy

```
#define debruijn32 0x077CB531UL
/* debruijn32 = 0000 0111 0111 1100 1011 0101 0011 0001 */

/* table to convert debruijn index to standard index */
int index32[32];

/* routine to initialize index32 */
void setup( void )
{
  int i;
  for(i=0; i<32; i++)
    index32[ (debruijn32 << i) >> 27 ] = i;
}

/* compute index of rightmost 1 */
int rightmost_index( unsigned long b )
{
  b &= -b;
  b *= debruijn32;
  b >>= 27;
  return index32[b];
}
```

Bit Vector Linear Search: performance

- N rules, the intersected bitmaps are N bits long.
  - Hence, computing the AND ***requires O(N) operations***
  - ***Why not do simple linear search instead?***
- Computing the AND of K bit vectors N-bit long is ***O(N*K)***
  - but general purpose 64 bit CPUs do 64 bit AND and memory look up in 1 clock cycle
  - and specialized HW can do even better...
- Thus, in reality, we have ***O(N*K/W)*** where ***W*** is the is the ***memory width***
- For example, using W = 1000 and k = 5 fields, the number of memory accesses for 5000 rules is 5000 ∗ 6/1000 = 30. Using 10-nsec SRAM, this allows a rule lookup in 300 nsec, which is sufficient to process minimum-size (40-byte) packets at wire speed on a gigabit link.
  - but specialized HW can do even better → ***k-fold parallelism***

Laboratory: a simple Linear Bit Vector Firewall with eBPF/XDP

inspired by: [Mia19] Miano, Sebastiano, et al. "Securing Linux with a faster and scalable iptables." ACM SIGCOMM Computer Communication Review 49.3 (2019)

eBPF introduction (https://ebpf.io/what-is-ebpf/)

- eBPF is a framework (originally developed for Linux) that allows programmers
  - to write small network programs in ***high level languages*** (C, python, rust, etc...)
  - to ***inject the program byte code into the kernel*** via several entry points called ***hooks***
  - to ***execute*** the program within the kernel ***in a sandbox environment***
  - to ***interact*** with the in-kernel program execution at runtime via different interfaces
  - to ***store/read data*** from both the kernel and the user space in specific data structures called ***maps***
- By allowing to run sandboxed programs within the operating system, application developers can run eBPF programs to ***add additional capabilities to the operating system at runtime***
  - without having to deal with low level kernel programming and security details
- The ***operating system guarantees safety and execution efficiency*** as if natively compiled with the aid of a Just-In-Time (JIT) compiler and verification engine
- Direct access to kernel resources **eliminates the resource-intensive context switching** typical in regular applications.

is eBPF really used?

August 19, 2020 ANNOUNCEMENTS

Google announces Cilium & eBPF as the new networking dataplane for GKE

Today marks an exciting day for the Cilium community and all Cilium contributors, Google just announced that Cilium has been selected and made available as the new datapath for GKE and Anthos:

> *"Today, we're introducing GKE Dataplane V2, an opinionated dataplane that harnesses the power of eBPF and Cilium, an open source project that makes the Linux kernel Kubernetes-aware using eBPF.*

You can read all the details in the official announcement. In this post, we will take a look behind the scenes that lead up to this.

Facebook, Google, Isovalent, Microsoft and Netflix Launch eBPF Foundation as Part of the Linux Foundation

Industry leaders come together to drive the growth of eBPF as a transformational technology to redefine networking, security, tracing and observability

Katran

A high performance layer 4 load balancer

Website | GitHub

Katran is a C++ library and eBPF program to build a high-performance layer 4 load ba
forwarding plane. Katran leverages the XDP infrastructure from the Linux kernel to pr
an in-kernel facility for fast packet processing. Its performance scales linearly with th
number of NIC's receive queues and it uses RSS friendly encapsulation for forwarding to L7
load balancers.

CLOUDFLARE

L4Drop: XDP DDoS Mitigations

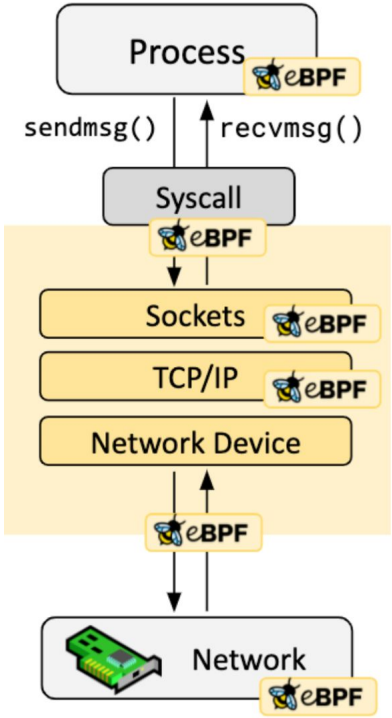
28/11/2018

Arthur Fabre

Efficient packet dropping is a key part of Cloudflare's distributed denial of service (DDoS) attack mitigations. In this post, we introduce a new tool in our packet dropping arsenal: L4Drop.

eBPF hooks

- eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point.
- Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.
- In this Lab we focus on ***XDP***, the lowest possible hook which is implemented directly in the NIC driver
  - eBPF programs injected into the XDP hooks have almost zero dependency on other kernel modules and data structure
    - very simple programming interface
    - good performance
    - less flexibility than “full” eBPF programs

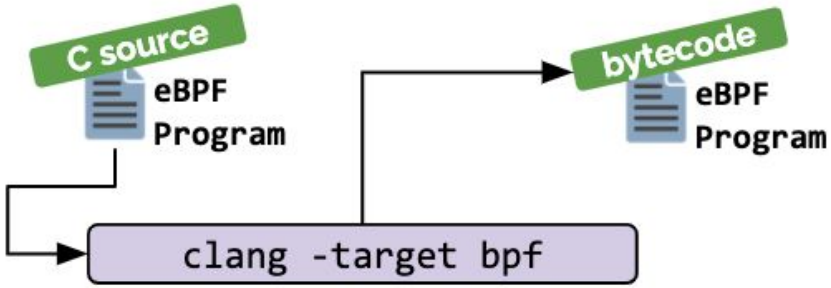


How are eBPF programs written?

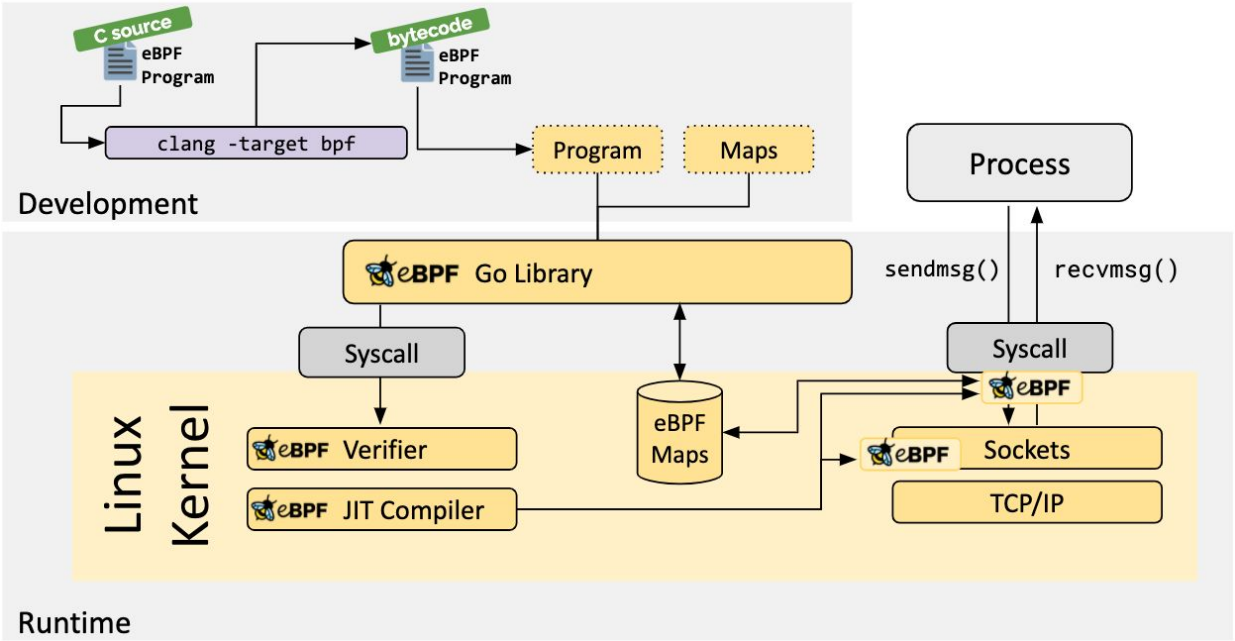
The Linux kernel expects eBPF programs to be loaded in the form of bytecode.

The more common development practice ***is to leverage a compiler suite like LLVM to compile pseudo-C code into eBPF bytecode***.

eBPF target refers to a RISC register virtual machine with a total of 11 64-bit registers, a program counter and a 512 byte fixed-size stack.



Loading and eBPF program

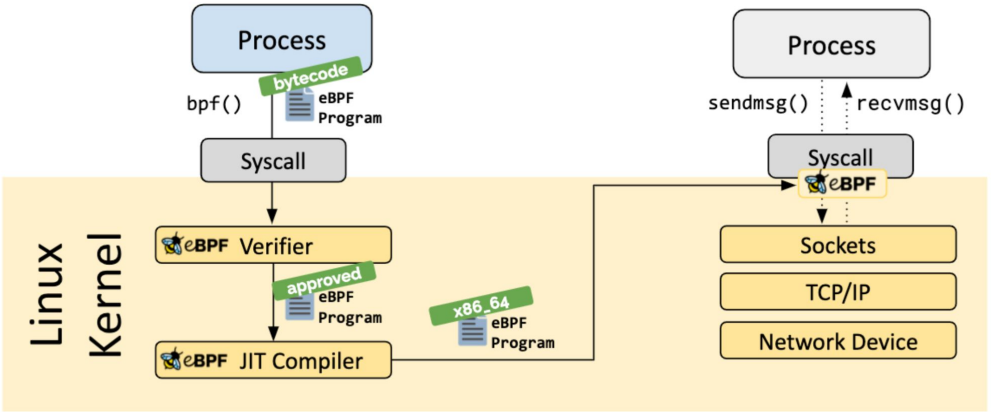


The eBPF program can be loaded into the Linux kernel using the ***bpf system call***.

Verification and Jit Compilation

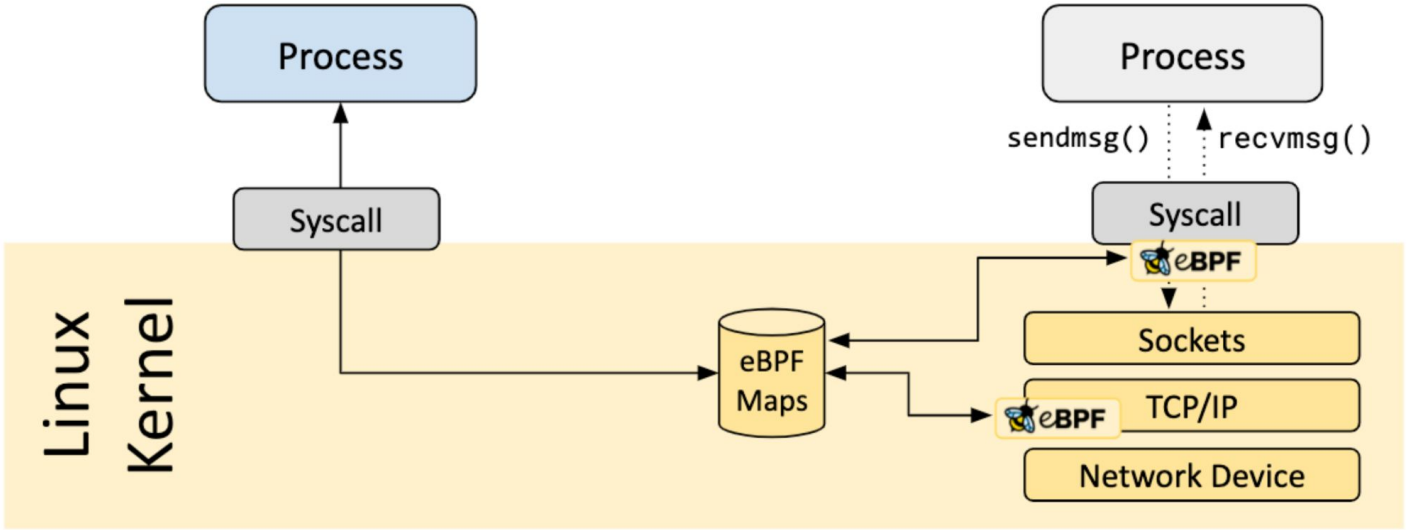
The ***verification*** step ensures that the eBPF program is safe to run: no un-bounded loops, no invalid memory access, no crashes.

The ***Just-in-Time*** (JIT) compilation step translates the generic bytecode of the program into the machine specific instruction set.



eBPF Maps

eBPF programs can leverage the concept of ***eBPF maps*** to store and retrieve data in a wide set of data structures.



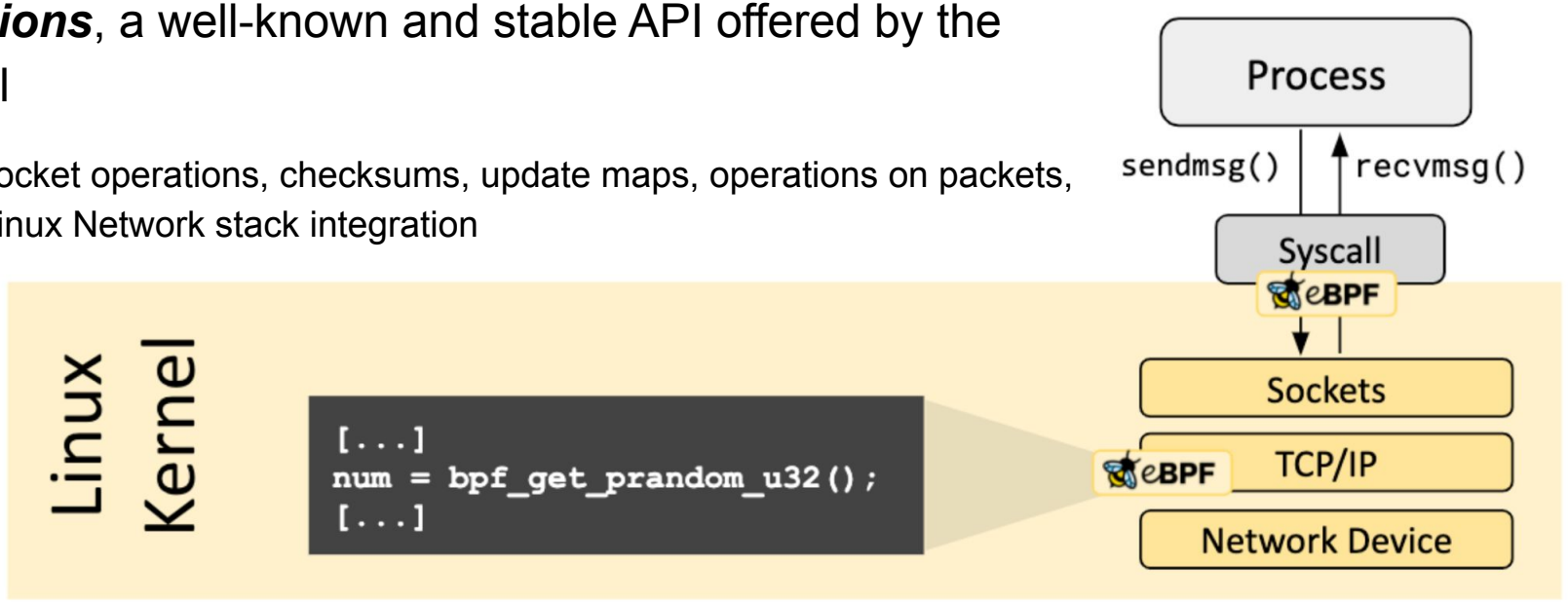
eBPF Map Types

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
};
```

eBPF helper functions

eBPF programs can make function calls into ***helper functions***, a well-known and stable API offered by the kernel

- socket operations, checksums, update maps, operations on packets, Linux Network stack integration



How do we proceed?

- (eBPF/)XDP programming would require a few introductive hours
  - BTW, a nice tutorial: https://github.com/xdp-project/xdp-tutorial
- We don't have time for this...

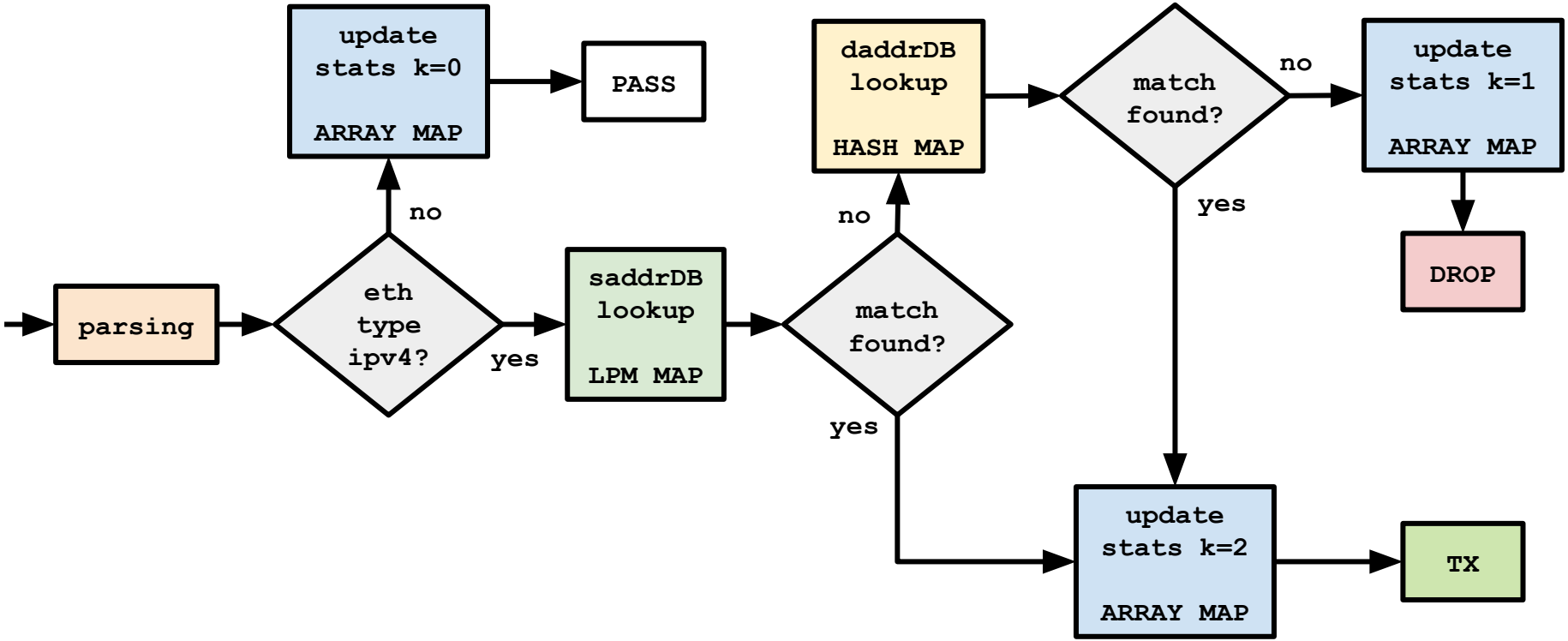
Let's learn by doing!!

Development environment

Simplest thing to do: use xdp-tutorial as our development environment

1. xdp-tutorial current version *natively* supported by *ubuntu-20.04
2. clone GIT repository:
   `git clone https://github.com/xdp-project/xdp-tutorial.git`
3. setup dependencies (read `setup_dependencies.org`)
4. create a lab dir
   `mkdir DIRNAME`
5. modify Makefile to include the new dirs
6. set up the new dir by copying another one

A simple (and perhaps meaningless) ACL example



Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

A MAP can be allocated at runtime with libbpf like in this case, or statically allocated from the kernel program (when it is inserted). This is an LPM map named “saddr_db”

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

eBPF does not care about what is the structure of the map keys and values, but only about their size. In this stupid example the value is a __u8 dummy value (not used, but values can not be of size 0) v

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

v4_lpm_key is defined in this struct

```
struct v4_lpm_key {
    __u32 prefixlen;
    __be32 addr;
};
```

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

```
#define RULE_NUM 32
```

this is the maximum number of map entries

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(key_size, sizeof(__be32));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
} daddr_db SEC(".maps");
```

This is an hash map named daddr_db. The key is an ipv4 address (4 byte). The value is a dummy __u8 value (not used, like for saddr_db)

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, sizeof(__u8));
    __uint(max_entries, RULE_NUM);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} saddr_db SEC(".maps");
```

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(key_size, sizeof(__u32));
    __uint(value_size, sizeof(__u32));
    __uint(max_entries, 3);
} stats_db SEC(".maps");
```

This is an array map used to count the number of packets according to their specific classification. The key is an integer among the ones defined in the enum. The value is an __u32 (overflow not handled)

```
enum stats_enum {
        STATS_PASS_KEY = 0,
        STATS_DROP_KEY,
        STATS_TX_KEY
};
```

Implementing the XDP program: program section and variables

```
SEC("xdp_nsd_test")
int  xdp_parser_func(struct xdp
{
        void *data_end = (void
        void *data = (void *)(long)ctx->data;
        struct ethhdr *eth =(struct ethhdr *)data;
        struct iphdr *ip = NULL;
        __u32 offset = 0;
        __be32 daddr_key;
        struct v4_lpm_key saddr_key;
        int stats_key;
        int xdp_verdict;
        __u8 proto;
        __u8 *val;
        __u32 *stats_value;
```

the program section can be named arbitrarily. When a program is loaded this must be specified.

Implementing the XDP program: program section and variables

```
SEC("xdp_nsd_test")
int  xdp_parser_func(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_en
        void *data = (void *)(long)ctx->data;
        struct ethhdr *eth =(struct ethhdr *)data;
        struct iphdr *ip = NULL;
        __u32 offset = 0;
        __be32 daddr_key;
        struct v4_lpm_key saddr_key;
        int stats_key;
        int xdp_verdict;
        __u8 proto;
        __u8 *val;
        __u32 *stats_value;
```

this function is like the "main" program function invoked by the eBPF executor when a packet passess the XDP hook of the interfaces on which the program is loaded. An XDP program gets as input only a pointer to the XDP context defined in "linux/bpf.h"

```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
struct xdp_md {
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        /* Below access go through struct xdp_rxq_info */
        __u32 ingress_ifindex; /* rxq->dev->ifindex */
        __u32 rx_queue_index;  /* rxq->queue_index  */

        __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

Implementing the XDP program: program section and variables

```c
SEC("xdp_nsd_test")
int  xdp_parser_func(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
        void *data = (void *)(long)ctx->data;
        struct ethhdr *eth =(struct ethhdr *)data;
        struct iphdr *ip = NULL;
        __u32 offset = 0;
        __be32 daddr_key;
        struct v4_lpm_key saddr_key;
        int stats_key;
        int xdp_verdict;
        __u8 proto;
        __u8 *val;
        __u32 *stats_value;
```

these 2 pointers are used to parse the ethernet and IP headers. The two structs are defined in 2 "external" kernel headers: linux/if_ether.h and linux/ip.h

```c
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    ihl:4,
                version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
        __u8    version:4,
                ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8    tos;
        __be16  tot_len;
        __be16  id;
        __be16  frag_off;
        __u8    ttl;
        __u8    protocol;
        __sum16 check;
        __be32  saddr;
        __be32  daddr;
        /*The options start here. */
};
```

```c
struct ethhdr {
        unsigned char   h_dest[ETH_ALEN];
        unsigned char   h_source[ETH_ALEN];
        __be16          h_proto;
} __attribute__((packed));
#endif
```

Implementing the XDP program: parsing and hardcoded classification

```
offset = sizeof(struct ethhdr);
if (data + offset > data_end) {
        return XDP_ABORTED;
}

if (eth->h_proto == __constant_htons(ETH_P_IP)
        ip = (struct iphdr *)(data + offset);
        offset += sizeof(struct iphdr);
        if (data + offset > data_end) {
                return XDP_ABORTED;
        }

        saddr_key.prefixlen = 32;
        saddr_key.addr = ip->saddr;
        daddr_key = ip->daddr;
}
else {
        stats_key = STATS_PASS_KEY;
        xdp_verdict = XDP_PASS;
        goto udpate_stats_and_return;
}
```

this is a so-called "memory boundary check". Every time the memory containing the packet is read, the verifier requires to check if the memory address is within the packet boundaries. without this check the XDP program can not be loaded.

Implementing the XDP program: parsing and hardcoded classification

```
offset = sizeof(struct ethhdr);
if (data + offset > data_end) {
        return XDP_ABORTED;
}

if (eth->h_proto == __constant_htons(ETH_P_IP)) {
        ip = (struct iphdr *)(data + offset);
        offset += sizeof(struct iphdr);
        if (data + offset > data_end) {
                return XDP_ABORTED;
        }

        saddr_key.prefixlen = 32;
        saddr_key.addr = ip->saddr;
        daddr_key = ip->daddr;
}
else {
        stats_key = STATS_PASS_KEY;
        xdp_verdict = XDP_PASS;
        goto udpate_stats_and_return;
}
```

here we check for the ethernet type field. ipv4 packets are further processed.

all other protocols are "passed" to the user space. This is a first classification. It is hard coded because it can not be modified at run time

Implementing the XDP program: parsing and hardcoded classification

```
offset = sizeof(struct ethhdr);
if (data + offset > data_end) {
        return XDP_ABORTED;
}

if (eth->h_proto == __constant_htons(ETH_P_IP))
        ip = (struct iphdr *)(data + offset);
        offset += sizeof(struct iphdr);
        if (data + offset > data_end) {
                return XDP_ABORTED;
        }

        saddr_key.prefixlen = 32;
        saddr_key.addr = ip->saddr;
        daddr_key = ip->daddr;
}
else {
        stats_key = STATS_PASS_KEY;
        xdp_verdict = XDP_PASS;
        goto udpate_stats_and_return;
}
```

ipv4 packets are further processed in order to extract the keys for the map lookups. In this case we have another memory boundary check and 2 keys are constructed:

1. an LPM key with the source address extracted from the packet and prefix len = 32. This key will be used for querying the map saddr_db

2. a 32 bit key containing the IP destination address in the packet and used to query the daddr_db map.

Implementing the XDP program: map lookups and classification

```
//look up in source prefix DB. XXX LPM MATCH
val = (__u8 *)bpf_map_lookup_elem(&saddr_db, &saddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}

//look up in destination prefix DB. XXX EXACT HASH MATCH
val = (__u8 *)bpf_map_lookup_elem(&daddr_db, &daddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}
else {
        stats_key = STATS_DROP_KEY;
        xdp_verdict = XDP_DROP;
}
```

this code portion represents the configurable classification. If a packet is matched against either of the two maps saddr_db and daddr_db, the packets is transmitted.

Implementing the XDP program: map lookups and classification

```
//look up in source prefix DB. XXX LPM MATCH
val = (__u8 *)bpf_map_lookup_elem(&saddr_db, &saddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}

//look up in destination prefix DB. XXX
val = (__u8 *)bpf_map_lookup_elem(&dadd
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}
else {
        stats_key = STATS_DROP_KEY;
        xdp_verdict = XDP_DROP;
}
```

this is the lookup in the LPM table saddr_db. val is a pointer to the memory area representing the value associated to the a key looked up in the map. If a match is found, val is not null. The returned pointer must be always checked, otherwise the in-kernel verification fails and the XDP program cannot be loaded

Implementing the XDP program: map lookups and classification

```
//look up in source prefix DB. XXX LPM MATCH
val = (__u8 *)bpf_map_lookup_elem(&saddr_db, &saddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}

//look up in destination prefix DB. XXX EXACT HASH MATCH
val = (__u8 *)bpf_map_lookup_elem(&daddr_db, &daddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}
else {
        stats_key = STATS_DROP_KEY;
        xdp_verdict = XDP_DROP;
}
```

this is the lookup in the hashmap daddr_db.

Implementing the XDP program: map lookups and classification

```
//look up in source prefix DB. XXX LPM MATCH
val = (__u8 *)bpf_map_lookup_elem(&saddr_db, &saddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}

//look up in destination prefix DB. XXX EXACT HASH MATCH
val = (__u8 *)bpf_map_lookup_elem(&daddr_db, &daddr_key);
if (val) {
        stats_key = STATS_TX_KEY;
        xdp_verdict = XDP_TX;
        goto udpate_stats_and_return;
}
else {
        stats_key = STATS_DROP_KEY;
        xdp_verdict = XDP_DROP;
}
```

if both lookups return NULL (i.e. the packet does not match), the packets is classified to be DROPed. In other words, this XDP program works as a whitelist ACL for IPv4 packets.

Implementing the XDP program: stats update and packet verdict

```
udpate_stats_and_return:
        stats_value = bpf_map_lookup_elem(&stats_db, &stats_key);
        if (!stats_value) {
                return XDP_ABORTE
        }
        lock_xadd(stats_value, 1);
        return xdp_verdict;
}
```

to update a value in a map, the entry pointer must be first retrieved.

Implementing the XDP program: stats update and packet verdict

```
udpate_stats_and_return:
        stats_value = bpf_map_lookup_elem(&stat
        if (!stats_value) {
                return XDP_ABORTED;
        }
        lock_xadd(stats_value, 1);
        return xdp_verdict;
}
```

eBPF is intrinsically multi-core, i.e. the same XDP program runs in parallel on different CPU processing the packets. The maps used in this program are shared among different CPUs (per-cpu maps are available...). In this case we need a "safe" increment to avoid possible race conditions

```
/* LLVM maps __sync_fetch_and_add() as a built-in function to the BPF atomic add
 * instruction (that is BPF_STX | BPF_XADD | BPF_W for word sizes)
 */
#ifndef lock_xadd
#define lock_xadd(ptr, val)     ((void) __sync_fetch_and_add(ptr, val))
#endif
```

Implementing the XDP program: stats update and packet verdict

```
udpate_stats_and_return:
        stats_value = bpf_map_lookup_elem(&stats_db, &stats_key);
        if (!stats_value) {
                return XDP_ABORTED;
        }
        lock_xadd(stats_value, 1);
        return xdp_verdict;
}
```

this return specifies how the packet must be handled by the kernel. Possible verdicts are defined in linux/bpf.h

```
enum xdp_action {
        XDP_ABORTED = 0,
        XDP_DROP,
        XDP_PASS,
        XDP_TX,
        XDP_REDIRECT,
};
```

Configuring the classifier from the user space

```
#configure saddr_db map (key: prefix|addr)
#insert match field 10.0.0.0/24 - key 0x180000000a000000 (0x18 = 24)
bpftool map update name saddr_db key hex 18 00 00 00 0a 00 00 00 value hex 00
#insert match field 192.168.0.0/16 - key  0x100000000c0a80000 (0x10 = 16)
bpftool map update name saddr_db key hex 10 00 00 00 c0 a8 00 00 value hex 00

#configure daddr_db map
#insert match field 8.8.8.8
bpftool map update name daddr_db key hex 08 08 08 08 value hex 00
```

maps can be accessed at run time from the user space in two ways. From a program linking the library libbpf, or from the command shell with the binary bpftool. For this class I have defined a simple configuration with bpftool. Packets that have source address in the ranges 10.0.0.0/24 or 192.168.0.0/16 or packets with destination address exaclty equal to 8.8.8.8 are classified to be TXed.

How to test the program

```
#PASS (key 0)
#dummy ARP request for ip address 0.0.0.0
p=Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(op=1)
#dummy ipv6 packet
p=Ether(dst="00:0c:29:76:28:bf")/IPv6()

#TX (key 2)
#exact match
#dummy IPv4 packet to 8.8.8.8
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="7.7.7.7",dst="8.8.8.8")
#LPM match
#dummy IPv4 packet from 10.0.0.1
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="10.0.0.1",dst="1.1.1.1")
#dummy IPv4 packet from 192.168.0.1
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="192.168.0.1",dst="1.1.1.1")

#DROP (key 1)
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="1.1.1.1",dst="2.2.2.2")

#try and send a ping. arp is passed (response received)
#ping is dropped (no response is received)
#ping 172.16.115.2

#to send packet through proper iface
#my VM is attached to a host only net. On the host iface=bridge100
#sendp(p, iface="bridge100")
```

several strategies are possible. I have chosen to use python/scapy to generate packets and check for the actual classification by dumping the stats map.

```
root@ubuntu:/# bpftool map dump name
stats_db

key: 00 00 00 00  value: 08 00 00 00
key: 01 00 00 00  value: 16 00 00 00
key: 02 00 00 00  value: 08 00 00 00
Found 3 elements
```

Load and unload the program

Programs can be loaded with the **`xdp_loader`** or **`ip link:`**

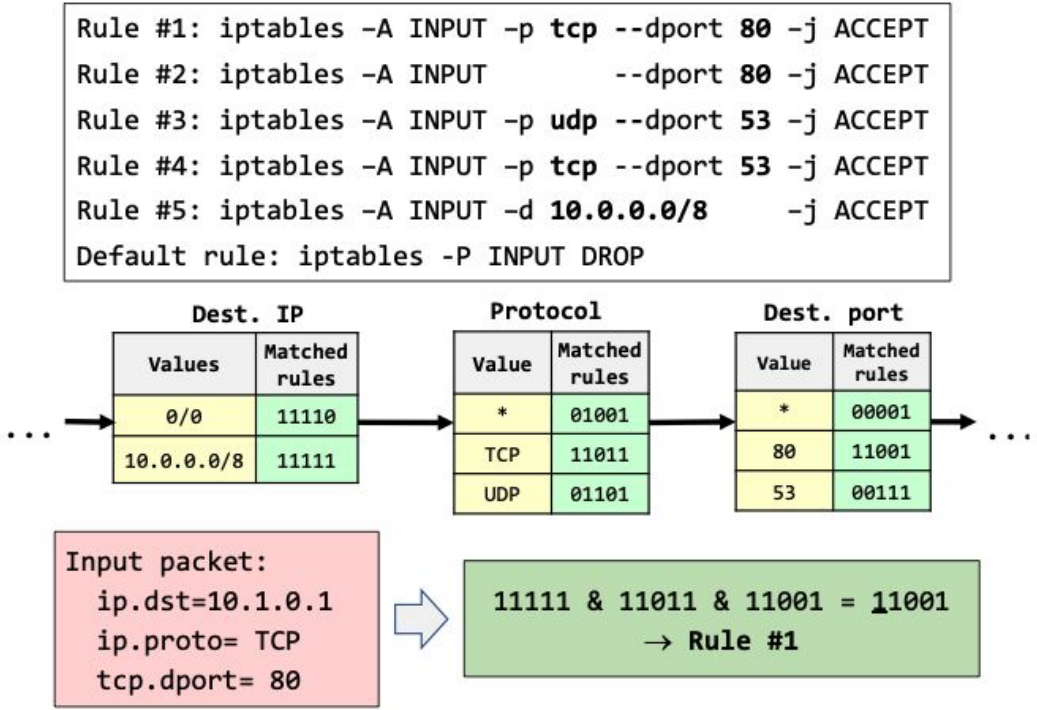
```
sudo ip link set dev ens33 xdp obj xdp_prog_kern.o sec
xdp_nsd_test
```

Programs unloaded again with the **`xdp_loader`** or with **`ip link`**. To delete pinned maps we must delete the files in the sys fs bpf directory related to the interface into which the program was loaded. For example, in case on iface = ens33

```
root@ubuntu:/ ip link set dev ens33 xdp off
```

```
root@ubuntu:# cd /sys/fs/bpf/ens33 && rm *
```

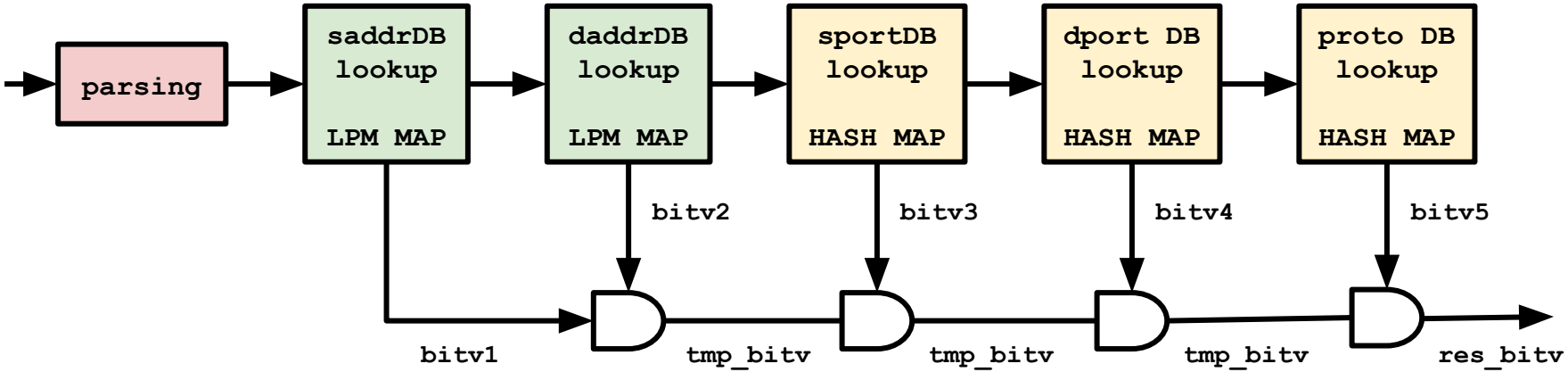
Bit Vector Linear Search: high level approach



FW Specification

- ***host-based firewall:*** no forwarding
  - XDP intercepts only incoming packets
  - all accepted packets are passed to userspace
  - this FW can be seen as low level white list for incoming packets forwarded to the user space
- ***default policy***: DROP
- ***possible action***: ACCEPT
  - we simply need to check if the final bit vector is != 0
- ***5 match fields***: source IP, destination IP, source port, destination port, protocol
- ***Match types:***
  - IP addresses: ***LPM***
  - ports and protocol: ***exact match (no ranges)***
- No automatic rules adaptation to the bit vector linear search
- No connection tracking
- We are going to write our Bit Vector Linear Search Firewall with the ***Linux's eBPF/XDP framework***
  - in-kernel execution
  - besides other positive consequences, we don't have to deal with efficient packet capture drivers:

High level design of our eBPF/XDP firewall




```
if res_bitv != 0:
   accept_packet()
```

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, FW_BITVECTOR_LEN);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(max_entries, FW_RULE_NUM);
} saddr_db SEC(".maps");
```

```
struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(key_size, sizeof(struct v4_lpm_key));
    __uint(value_size, FW_BITVECTOR_LEN);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(max_entries, FW_RULE_NUM);
} daddr_db SEC(".maps");
```

Implementing the XDP program: maps

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(key_size, sizeof(__u8));
    __uint(value_size, FW_BITVECTOR_LEN);
    __uint(max_entries, FW_RULE_NUM);
} proto_db SEC(".maps");
```

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(key_size, sizeof(__u16));
    __uint(value_size, FW_BITVECTOR_LEN);
    __uint(max_entries, FW_RULE_NUM);
} dport_db SEC(".maps");
```

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(key_size, sizeof(__u16));
    __uint(value_size, FW_BITVECTOR_LEN);
    __uint(max_entries, FW_RULE_NUM);
} sport_db SEC(".maps");
```

Implementing the XDP program: structs and defines

```
#to read the bpf trace pipe:
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

```
#define FW_RULE_NUM 32
#define FW_BITVECTOR_LEN 4
```

```
struct v4_lpm_key {
    __u32 prefixlen;
    __be32 addr;
};
```

```
struct fw_key {
  __be32 saddr;
  __be32 daddr;
  __u16 sport;
  __u16 dport;
  __u8 proto;
};
```

packet parsing (1)

```
SEC("xdp_fw")
int  xdp_parser_func(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
        void *data = (void *)(long)ctx->data;
        struct fw_key key;
        struct iphdr *ip = NULL;
        struct udphdr *l4 = NULL;
        __u32 *bitvector = NULL, res_bitvector = 0xffffffff, offset = 0;
        struct v4_lpm_key lpm_key;

        struct ethhdr *eth =(struct ethhdr *)data;
        offset = sizeof(struct ethhdr);
        if (data + offset > data_end) {
                return XDP_DROP;
        }

        if (eth->h_proto == __constant_htons(ETH_P_IP)) {
                ip = (struct iphdr *)(data + offset);
                offset += sizeof(struct iphdr);
                if (data + offset > data_end) {
                        return XDP_DROP;
                }

                if (ip->ihl != 5) {
                        return XDP_DROP;
                }
```

packet parsing (2)

```
        key.saddr = ip->saddr;
        key.daddr = ip->daddr;
        key.proto = ip->protocol;

        l4 = (struct udphdr *)(data + offset);
        if (ip->protocol == IPPROTO_TCP) {
                offset += sizeof(struct tcphdr);
        }
        else if (ip->protocol == IPPROTO_UDP) {
                offset += sizeof(struct udphdr);
        }
        else {
                return XDP_PASS;
        }

        if (data + offset > data_end) {
                return XDP_DROP;
        }

        key.sport = l4->source;
        key.dport = l4->dest;

}
else {
        return XDP_PASS;
}
```

LPM lookup

```
//look up in source prefix DB
lpm_key.prefixlen = 32;
lpm_key.addr = (__be32)key.saddr;
bitvector = bpf_map_lookup_elem(&saddr_db, &lpm_key);
if (bitvector) {
        res_bitvector = res_bitvector & *bitvector;
        bpf_printk("saddr lookup %u %u", *bitvector, res_bitvector);
} else {
        bpf_printk("no saddr in db. DROP\n");
        return XDP_DROP;
}

//look up in destination prefix DB
lpm_key.prefixlen = 32;
lpm_key.addr = (__be32)key.daddr;
bitvector = bpf_map_lookup_elem(&daddr_db, &lpm_key);
if (bitvector) {
        res_bitvector = res_bitvector & *bitvector;
        bpf_printk("daddr lookup %u %u", *bitvector, res_bitvector);
} else {
        bpf_printk("no daddr in db. DROP\n");
        return XDP_DROP;
}
```

Port lookup

```
//lookup in source port DB
bitvector = bpf_map_lookup_elem(&sport_db, &key.sport);
if (!bitvector) {
        bitvector = bpf_map_lookup_elem(&sport_db, &port_default);
}
if (bitvector) {
        res_bitvector = res_bitvector & *bitvector;
        bpf_printk("sport lookup %u %u", *bitvector, res_bitvector);
} else {
        bpf_printk("no sport in db. DROP\n");
        return XDP_DROP;
}


//lookup in destination port DB
bitvector = bpf_map_lookup_elem(&dport_db, &key.dport);
if (!bitvector) {
        bitvector = bpf_map_lookup_elem(&dport_db, &port_default);
}
if (bitvector) {
        res_bitvector = res_bitvector & *bitvector;
        bpf_printk("dport lookup %u %u", *bitvector, res_bitvector);
} else {
        bpf_printk("no dport in db. DROP\n");
        return XDP_DROP;
}
```

Proto lookup

```
//lookup in protocol DB
bitvector = bpf_map_lookup_elem(&proto_db, &key.proto);
if (!bitvector) {
        bitvector = bpf_map_lookup_elem(&proto_db, &proto_default);
}
if (bitvector) {
        res_bitvector = res_bitvector & *bitvector;
        bpf_printk("proto lookup %u %u", *bitvector, res_bitvector);

} else {
        bpf_printk("no proto in cb. DROP\n");
        return XDP_DROP;
}
```

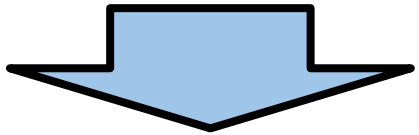
Verdict + debug

```
        if (res_bitvector == 0) {
#if DEBUG
                bpf_printk("NULL rule bitvector interception. DROP\n");
#endif
                return XDP_DROP;
        }
#ifdef PRINT_RULE_ID
{
        int pos=0, k=0, j=0;
        if (res_bitvector) {
                __u8 *p = (__u8 *) &res_bitvector;
                for (k=0; k<4; k++) {
                        if (p[k] != 0) {
                                for (j=0; j<8; j++) {
                                        if ((0xff & (p[k]>>j))==1) {
                                                pos = k*8 + 8-j-1;
                                                break;
                                        }
                                }
                                if (pos) break;
                        }
                }

                bpf_printk("res bitvector: %d matched rule: %d\n", res_bitvector, pos);
        }
        else {
                bpf_printk("no rule matched\n");
        }
}
#endif
        return XDP_PASS;
}
```

Simple Firewall Configuration (meaningless rules)

default DROP

#	saddr	daddr	sport	dport	proto
0	10.0.0.0/24	*	*	22	*
1	10.0.0.0/8	10.0.0.0/8	*	80	TCP
2	*	10.0.1.0/24	10000	80	TCP
3	*	8.8.8.8/32	*	53	UDP



adaptation

saddr	bitv
10.0.0.0/24	1111
10.0.0.0/8	0111
0/0	0011

saddr	bitv
10.0.0.0/8	1100
10.0.1.0/24	1110
8.8.8.8/32	1001
0/0	1000

sport	bitv
10000	1111
*	1101

dport	bitv
22	1000
80	0110
53	0001

proto	bitv
TCP	1110
UDP	1001
*	1000

Configuration Script

```
#configure saddr_db map (XXX key: prefix|addr)
#insert match field 10.0.0.0/24 - key 0x180000000a000000 (0x18 = 24), bitv 11110000 = 0xf0
bpftool map update name saddr_db key hex 18 00 00 00 0a 00 00 00 value hex f0 00 00 00
#insert match field 10.0.0.0/8 - key  0x080000000a000000 (0x08 = 8), bitv 01110000 = 0x70
bpftool map update name saddr_db key hex 08 00 00 00 0a 00 00 00 value hex 70 00 00 00
#insert match fiel 0/0 - key  0x0000000000000000, bitv 00110000 = 0x30
bpftool map update name saddr_db key hex 00 00 00 00 00 00 00 00 value hex 30 00 00 00

#configure daddr_db map (XXX key: prefix|addr)
#insert match field 10.0.0.0/8 - key 0x080000000a000100 (0x08 = 8), bitv 11000000 = 0xc0
bpftool map update name daddr_db key hex 08 00 00 00 0a 00 00 00 value hex c0 00 00 00
#insert match field 10.0.1.0/24 - key  0x180000000a0000101 (0x18 = 24), bitv 11100000 = 0xe0
bpftool map update name daddr_db key hex 18 00 00 00 0a 00 01 00 value hex e0 00 00 00
#insert match field 8.8.8.8/32 - key  0x2000000008080808 (0x20 = 32), bitv 10010000 = 0x90
bpftool map update name daddr_db key hex 20 00 00 00 08 08 08 08 value hex 90 00 00 00
#insert match fiel 0/0 - key  0x0000000000000000, bitv 10000000 = 0x80
bpftool map update name daddr_db key hex 00 00 00 00 00 00 00 00 value hex 80 00 00 00
```

Configuration Script

```
#configure sport_db map
#insert match field 10000 - key 0x2710 (0x2710 in network order = 10000), bitv 11110000 = 0xf0
bpftool map update name sport_db key hex 27 10 value hex f0 00 00 00
#insert match field 0 (don't care) - key  0x0000000000000000, bitv 11010000 = 0xd0
bpftool map update name sport_db key hex 00 00 value hex d0 00 00 00

#configure dport_db map
#insert match field 22 - key 0x0016 (0x0016 in network order = 22), bitv 10000000 = 0x80
bpftool map update name dport_db key hex 00 16 value hex 80 00 00 00
#insert match field 80 - key 0x0050 (0x050 in network order = 80), bitv 01100000 = 0x60
bpftool map update name dport_db key hex 00 50 value hex 60 00 00 00
#insert match field 53 - key 0x0035 (0x0035 in network order = 35), bitv 00010000 = 0x10
bpftool map update name dport_db key hex 00 35 value hex 10 00 00 00

#configure proto_db map
#insert match field TCP - key 0x06, bitv 11100000 = 0xe0
bpftool map update name proto_db key hex 06 value hex e0 00 00 00
#insert match field UDP - key 0x11, bitv 10010000 = 0x90
bpftool map update name proto_db key hex 11 value hex 90 00 00 00
#insert match field 0 (don't care) - key  0x0000000000000000, bitv 10000000 = 0x80
bpftool map update name proto_db key hex 00 value hex 80 00 00 00
```

a few test cases

```
#Rule 0
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="10.0.0.1",dst="1.1.1.1")/TC
P(dport=22, sport=12345)

#Rule 1
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="10.0.0.1",dst="10.0.0.2")/T
CP(dport=80, sport=12345)

#Rule 1 (but also 2)
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="10.0.0.1",dst="10.0.1.2")/T
CP(dport=80, sport=10000)

#Rule 2
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="1.0.0.1",dst="10.0.1.2")/TC
P(dport=80, sport=10000)

#Rule 3
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="10.0.0.1",dst="8.8.8.8")/UD
P(dport=53, sport=12345)

#default DROP
p=Ether(dst="00:0c:29:76:28:bf")/IP(src="100.0.0.1",dst="80.8.8.8")/
UDP(dport=53, sport=12345)

#sendp(p, iface="bridge100")
```

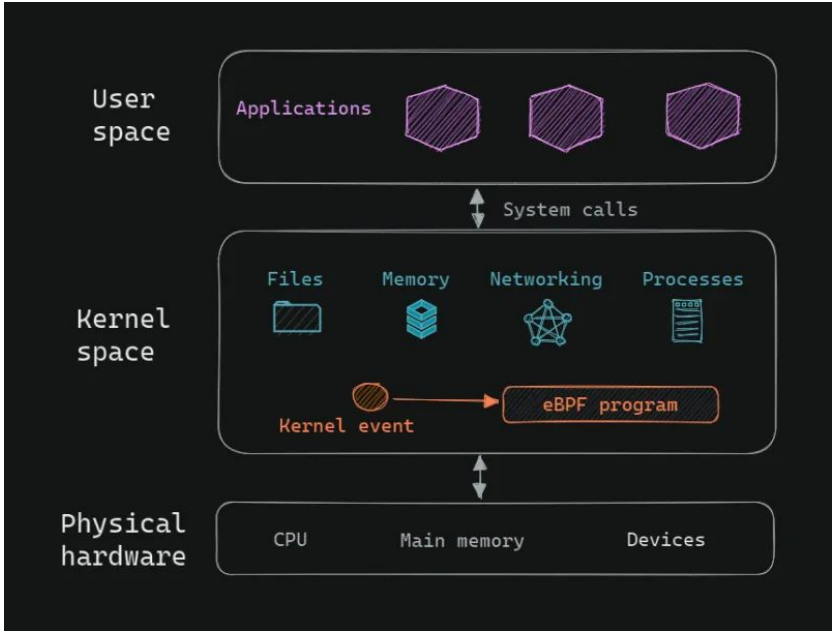
Limitations and possible extensions

- Small DB (32 rules → bitvector of 32 bits)
- Linear search for the leftmost bit set to 1
- Static rule adaptation
- Develop a cache to accelerate the classification
  - flows already classified are inserted in the cache so we don’t need to process the entire map pipeline

eBPF: Revolutionizing Security and Observability in the Cloud-Native Era

Beyond Networking

- eBPF programs can observe everything happening at kernel level, not only networking
- **Security**: can check for unexpected behaviour raising alerts
- **Observability**: everything passes through the kernel, eBPF allows to observe what happens in a highly performant and secure way



eBPF hooks

	Static	Dynamic
Kernel	Kernel tracepoints	Kprobes (Kernel Probes)
	Hooks into events pre-defined by kernel developers (with TRACE_EVENT macros)	Dynamically hooks into any part of the kernel code at runtime
	Monitoring system calls, hardware events	Debugging kernel code, monitoring dynamic kernel events
User	USDT (User-Level Statically Defined Tracing)	Uprobes (User Probes)
	Hooks into predefined tracepoints set by developers in application code	Dynamically hooks into any part of a user-space application at runtime
	Monitoring application-specific events	Debugging application code, monitoring dynamic application events

Kernel Tracepoints

- Pre-defined hooks in kernel for custom tracing
- Stable across kernel versions
- Used for debugging, performance analysis, real-time monitoring
- No performance impact if not enabled
    - Small time penalty for checking condition

Mini-Lab: Exploring Kernel Tracepoints

- List available tracepoints
  - `sudo ls /sys/kernel/debug/tracing/events`
- Enable 'sched/sched_switch' tracepoint
  - `echo 1 | sudo tee /sys/kernel/debug/tracing/events/sched/sched_switch/enable`
- Only trace when next process PID is 1000
  - `echo 'next_pid == 1000' | sudo tee /sys/kernel/debug/tracing/events/sched/sched_switch/filter`

Mini-Lab: eBPF with Kernel Tracepoints

- Suppose we want to detect a fork bomb attack
- sched_process_fork triggered whenever a process is forked
- real-time monitoring of kernel events is crucial for timely detection and response to security threats
- We use the libbpf-bootstrap repository for compiling

libbpf-bootstrap

- Repository with examples for starting up experimenting with different eBPF hooks
- Provides both user-space and kernel-space examples
- We copy examples inside this repository for compiling them
- Dependencies install (Ubuntu)
  - `sudo apt-get update -y`
  - `sudo apt-get install -y libz-dev libelf-dev llvm clang`
- Clone the repository and submodules
  - `git clone --recurse-submodules https://github.com/libbpf/libbpf-bootstrap.git`
- Copy the content of eBPF_tracing_examples in libbpf-bootstrap/examples/c

Mini-Lab: eBPF with Kernel Tracepoints

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
    __uint(max_entries, 1);
} fork_count SEC(".maps");
```

Mini-Lab: eBPF with Kernel Tracepoints

```
SEC("tp/sched/sched_process_fork")
int bpf_prog1(struct bpf_perf_event_data *ctx) {
    int key = 0;
    int zero = 0;
    int *val;

    val = bpf_map_lookup_elem(&fork_count, &key);
    if (!val) {
        bpf_map_update_elem(&fork_count, &key, &zero, BPF_ANY);
        val = bpf_map_lookup_elem(&fork_count, &key);
        if (!val)
            return 0;
    }
    (*val)++;
    return 0;
}

char _license[] SEC("license") = "GPL";
```

Mini-Lab: eBPF with Kernel Tracepoints

- Compile the program with make
  - `make nsd01_tracepoint`
- Load the eBPF program using loader
  - `sudo ./nsd01_tracepoint`
- In another terminal read the fork_count map (update every 1s)
  - `sudo watch -n 1 bpftool map dump name fork_count`
- In another terminal generate 100 processes
  - `sudo for i in {1..100}; do echo spawned $i ; done`
- Counter in map should increase!

User Statically-Defined Tracing (USDT) probe

- Custom tracepoints in user-space applications
  - Provides a mechanism for applications to define their own tracepoints, making them observable from the outside
- Track custom events, errors, or other significant occurrences
  - Enable monitoring without code modifications or recompilation
- USDT can be hooked by eBPF programs for deeper analysis and action.
- Applications like DTrace and SystemTap utilize USDT for dynamic tracing
- Libraries and applications can define their own tracepoints for customized monitoring (MySQL, Java, Node.js, …)

Mini-Lab: eBPF with USDT

- Sample login application (could be any application like MySQL)
- Trace failed login using USDT

Mini-Lab: eBPF with USDT

```
// Sample login application login_app.c
#include <stdio.h>
#include <sys/sdt.h>
#include <string.h>

int main() {
    char password[100];
    while (1) {
        printf("Enter password: ");
        scanf("%s", password);
        if (strcmp(password, "securepassword") != 0) {
            // DTRACE_PROBEX, where X is # arguments
            // domain=binary name, probename=failed_login, arg1
            DTRACE_PROBE1(login_app, failed_login, password);
            printf("Incorrect password, try again.\n");
        } else {
            printf("Access granted.\n");
            break;
        }
    }
    return 0;
}
```

Mini-Lab: eBPF with USDT

```
#include <vmlinux.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/usdt.bpf.h>

// usdt/path:domain:probe_name
SEC("usdt/./login_app:login_app:failed_login")
int BPF_USDT(trace_failed_login, char *password) {
        bpf_printk("Failed login attempt with password: %s\n", password);
        return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Mini-Lab: eBPF with USDT

- Install sdt library for instrumenting the application
  - `sudo apt-get install -y systemtap-sdt-dev`
- Compile the application
  - `gcc -o login_app login_app.c`
- Run the login programcd
  - `./login_app`
- Insert right password (securepassword)
- See trace pipe
  - `sudo cat /sys/kernel/debug/tracing/trace_pipe`
- Insert wrong password
- See the wrong password in trace pipe

Kernel Probes (Kprobes)

- Allow dynamic breakpoints in the kernel code for inspection or modification of kernel behavior at specified points during runtime
- Ability to insert probes on almost any kernel symbol or instruction at runtime
  - the symbol has to be exported by the kernel
- Triggers a specified function at the entry/exit of kernel functions whenever execution reaches the probe point
  - Pre-handler executes before the probed symbol and a post handler can execute after
  - Handlers can gather/modify function data
- eBPF programs can be attached to Kprobes

Kprobes vs Tracepoints

- Flexibility
  - Kprobes are highly flexible, allowing probes on nearly any part of the kernel code
  - Tracepoints are fixed observability points defined by kernel developers at compile time
- Performance Impact
  - Potential performance impact with Kprobes, especially if misused or used extensively
  - Tracepoints are less flexible but have less impact on performance
- Skill & Stability
  - Kprobes require a deeper understanding of kernel internals compared to tracepoints
  - Less structured and not stable between kernel releases, whereas tracepoints are more stable and structured

Mini-Lab: Practical Usage of Kprobes with eBPF

- Monitor open() system call to detect abnormal file access indicating potential security threats
- Develop an eBPF program to increment a counter each time open() system call is executed
- Load the eBPF program and attach it to a Kprobe on do_sys_open
- Extract the counter value to user space to monitor the frequency of open() system calls

Mini-Lab: Practical Usage of Kprobes with eBPF

```
#include "vmlinux.h"

#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
    __uint(max_entries, 1);
} do_sys_open_count SEC(".maps");

SEC("kprobe/do_sys_openat2")
int BPF_KPROBE(do_sys_open, int dfd, char *filename) {
    int key = 0;
    int zero = 0;
    int *val;

    val = bpf_map_lookup_elem(&do_sys_open_count, &key);
    if (!val) {
        bpf_map_update_elem(&do_sys_open_count, &key, &zero, BPF_ANY);
        val = bpf_map_lookup_elem(&do_sys_open_count, &key);
        if (!val)
            return 0;
    }
    (*val)++;
    return 0;
}
char _license[] SEC("license") = "GPL";
```

Mini-Lab: Practical Usage of Kprobes with eBPF

- struct pt_regs is a structure containing processor registers
- It’s architecture dependent
- e.g., for x86 (arch/x86/include/asm/ptrace.h)

```
struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entry
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long bp;
        unsigned long bx;
```

Mini-Lab: Practical Usage of Kprobes with eBPF

```
#!/bin/bash

# This script will create a spike in open() system calls by repeatedly opening a file in a loop.

# Create a temporary file
temp_file=$(mktemp)

# Loop to open the temporary file
while true; do
    # Open the file and immediately close it
    exec 3<"$temp_file"
    exec 3<&-
done

# Cleanup
rm "$temp_file"
```

Mini-Lab: Practical Usage of Kprobes with eBPF

- Compile with make
  - `make nsd03_kprobe`
- Load
  - `./nsd03_kprobe`
- Looking up map id
  - `sudo bpftool map show`
- Monitor map
  - `sudo watch -n 1 bpftool map dump id <map_id>`
- Generate open with the bash script
  - `./open_generator.sh`

User-level Probes (Uprobes)

- Feature in the Linux kernel that allows you to dynamically instrument user-space functions at runtime
- You can attach a probe to a user-space function which triggers whenever the function is executed
- Extend the capabilities of uprobes by writing eBPF programs to inspect or modify the behavior and data
- Uprobes are dynamic and don't require modifications to the source code, allowing to set probes on almost any instruction in user-space at runtime
    - vs USDT which requires pre-defined probes in the source code

Mini-Lab: Uprobes

- Monitor and log all command inputs entered in all running Bash shells using the readline function as a hook point

Mini-Lab: Uprobes

```
#include <vmlinux.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

#define MAX_LINE_SIZE 80

struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);
    __type(key, char[MAX_LINE_SIZE]);
    __type(value, int);
    __uint(max_entries, 100);
} cmd_count SEC(".maps");
```

Remember! eBPF is multi-core, we need to use a separate map for each CPU or we will have a race condition on the cmd_count map

Mini-Lab: Uprobes

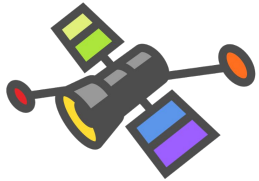
```
// uretprobe since we need the value being read from stdin
// which is available as return value at the end of readline
// uretprobe/path:function_to_be_probed
SEC("uretprobe//bin/bash:readline")
int BPF_URETPROBE(printret, const void *ret)
{
    char cmd[MAX_LINE_SIZE];
    int *count;
    int one = 1;

    if (!ret)
        return 0;

    bpf_probe_read_user_str(cmd, sizeof(cmd), ret);

    count = bpf_map_lookup_elem(&cmd_count, cmd);
    if (count) {
        int new_count = *count += 1;
        bpf_map_update_elem(&cmd_count, cmd, &new_count, BPF_EXIST);
    } else
            bpf_map_update_elem(&cmd_count, cmd, &one, BPF_NOEXIST);

    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Mini-Lab: Uprobes

- Compile
  - `make nsd04-uprobe`
- Load
  - `sudo ./nds04-uprobe`
- Use bpftool to read the cmd_count map and observe the count of different commands entered across all running Bash shells
  - `sudo watch -n 1 bpftool map dump name cmd_count`

Some real world examples

- Cilium
  - CNI (Container Network Interface) for Kubernetes
  - Focus: Networking, Security, and Load Balancing
  - Description: Cilium leverages eBPF to provide API-aware network security, load balancing, and visibility. It enables efficient service routing and visibility into application protocols.
  - Source: https://github.com/cilium/cilium



- Hubble
  - Networking and security observability platform built on top of Cilium
  - Focus: Network Observability
  - Description: Hubble uses eBPF to achieve real-time network observability into security, application behaviors, and dependencies within microservices architectures.
  - Source: https://github.com/cilium/hubble

Some real world examples

- Tetragon
  - Cilium's component dedicated to real-time Security Observability and Runtime Enforcement
  - Focus: Performance Monitoring
  - Description: Tetragon utilizes eBPF for performance troubleshooting and monitoring in cloud-native environments, gathering metrics directly from the kernel (react to significative events such as Process execution, syscalls, I/O activity)
  - Source: https://github.com/cilium/tetragon
- Skyfall
  - Monitoring agents for the LinkedIn infrastructure
  - Focus: Security and Performance Analysis
  - Description: LinkedIn employs Skyfall to trace and monitor network performance, and analyze the security aspects of their infrastructure using eBPF.
  - Source: https://engineering.linkedin.com/blog/2022/skyfall--ebpf-agent-for-infrastructure-observability

Some real world examples

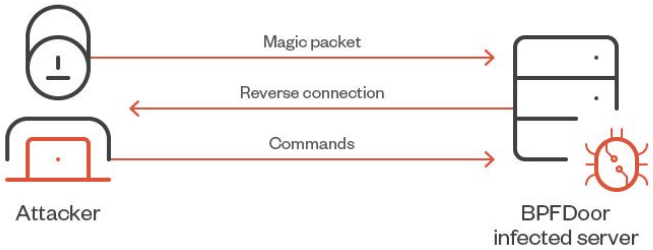
- Falco
  - Monitoring solution for security events, used by Shopify
  - Focus: Security Monitoring
  - Description: Falco helps Shopify detect potential security threats in their Kubernetes setup by monitoring the communication between applications and the underlying kernel
  - Source: https://falco.org/
- Many other projects and big players using it
  - Projects: https://ebpf.io/applications/
  - Users: https://ebpf.io/case-studies/

Abusing BPF Filters: BPFDoor

- eBPF can be used to write sophisticated malware
- BPFDoor is a backdoor designed to exploit the eBPF capabilities of the Linux kernel to install a backdoor on the target system
- Malware used by the APT group Red Menshen
- Emerged in 2018 but Red Menshen continues developing it
    - 6x increase in # instructions from 2022
- BPF filters used by BPFDoor allow the actors to activate the backdoor with a single “magic” network packet
- Since BPF intercepts packets before the Kernel stack, magic packet triggers the backdoor even when the packet is blocked by a firewall

Abusing BPF Filters: BPFDoor

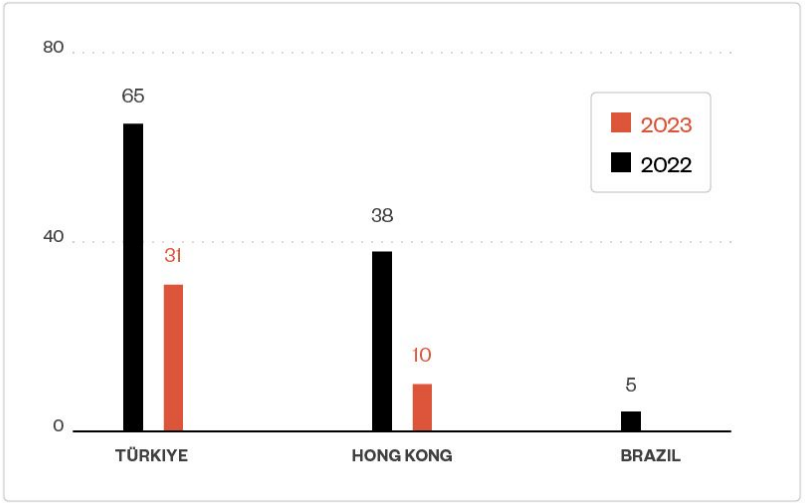
- Attached with the O_ATTACH_FILTER option from setsockopt() syscall
- Filters expect packets containing a magic number and, when it arrives, BPFDoor connects back to the source IP address of whoever sent the matching packet
- Reverse connection is then used to send commands to the infected machine's shell via a pipe
    - The reverse shell opened is privileged

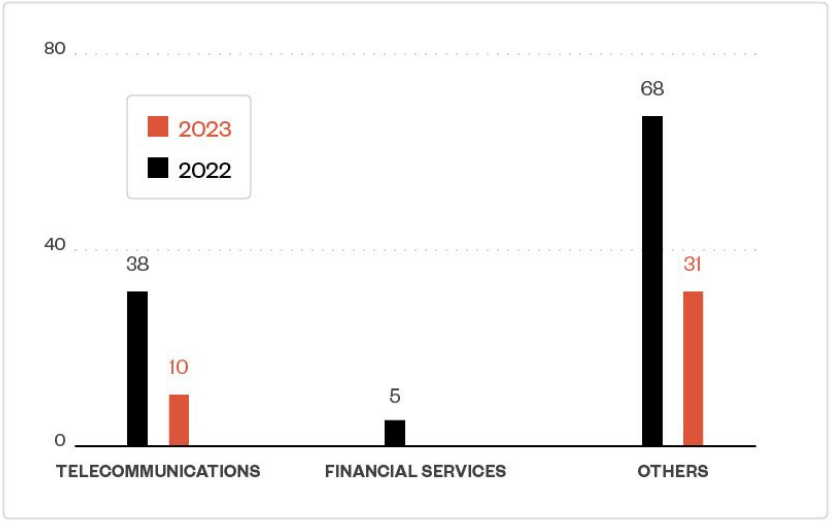


Abusing BPF Filters: BPFDoor

- Samples from 2018 to 2022 contained a filter of ~30 BPF instructions (BPF assembly)
  - Specific magic numbers for TCP, UDP and ICMP
  - `udp[8:2]=0x7255 or (icmp[8:2]=0x7255 and icmp[icmptype] == icmp-echo) or tcp[((tcp[12]&0xf0)>>2):2]=0x5293`
- Samples from 2023 contains a filter of ~39 BPF instructions (BPF assembly)
  - support an additional 4-byte magic number for TCP packets
  - `udp[8:2]=0x7255 or (icmp[8:2]=0x7255 and icmp[icmptype] == icmp-echo) or tcp[((tcp[12]&0xf0)>>2):2]=0x5293 or tcp[((tcp[12]&0xf0)>>2)+26:4]=0x39393939`

Abusing BPF Filters: BPFDoor





Abusing BPF Filters: BPFDoor

- Can be detected using ss comand

```
user@mwdebian64:~/bpfdoor$ sudo ss -0pb
Netid         Recv-Q         Send-Q                  Local Address:Port                  Peer Address:Port
Process
p_raw         0              0                             *:ens33                            *
 users:(("dhclient",pid=1893,fd=6))
        bpf filter (11):  0x28 0 0 12, 0x15 0 8 2048, 0x30 0 0 23, 0x15 0 6 17, 0x28 0 0 20, 0x45 4 0 8191, 0xb1 0 0 14
, 0x48 0 0 16, 0x15 0 1 68, 0x06 0 0 4294967295, 0x06 0 0 0,
p_raw         0              0                            ip:*                               *
 users:(("hald-addon-acpi",pid=2629,fd=3))
        bpf filter (30):  0x28 0 0 12, 0x15 0 27 2048, 0x30 0 0 23, 0x15 0 5 17, 0x28 0 0 20, 0x45 23 0 8191, 0xb1 0 0
14, 0x48 0 0 22, 0x15 19 20 29269, 0x15 0 7 1, 0x28 0 0 20, 0x45 17 0 8191, 0xb1 0 0 14, 0x48 0 0 22, 0x15 0 14 29269,
0x50 0 0 14, 0x15 11 12 8, 0x15 0 11 6, 0x28 0 0 20, 0x45 9 0 8191, 0xb1 0 0 14, 0x50 0 0 26, 0x54 0 0 240, 0x74 0 0 2,
 0x0c 0 0 0, 0x07 0 0 0, 0x48 0 0 14, 0x15 0 1 21139, 0x06 0 0 65535, 0x06 0 0 0,
```

29269 == 0x7255
2139 == 0x5293

Abusing BPF Filters: other examples

- dewdrop
  - From the ShadowBrokers leaks
  - Utilizes a port knocking technique to maintain stealth while enabling unauthorized access (similarly to BPFDoor)
  - More here https://reverse.put.as/2021/12/17/knock-knock-whos-there/
- Bvp47
  - Attributed to The Equation Group
  - Implement covert communication technique using TCP SYN packets, aided by BPF, to establish a stealthy communication channel between compromised servers and the attacker's machine, enabling clandestine data exfiltration and command execution on targeted systems (similarly to BPFDoor)
  - More here https://www.pangulab.cn/files/The_Bvp47_a_top-tier_backdoor_of_us_nsa_equation_group.en.pdf
- Symbiote
  - Uses BPF to conceal malicious network traffic on the compromised machine. When an admin initiates any packet capture tool on the affected machine, Symbiote injects BPF bytecode into the kernel to dictate which packets should be captured, thereby filtering out network traffic it wishes to hide.
  - More here https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossible-to-detect-linux-threat

References (eBPF for monitoring and security)

https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

https://blogs.oracle.com/linux/post/taming-tracepoints-in-the-linux-kernel

https://lwn.net/Articles/753601/

https://lwn.net/Articles/132196/

https://www.kernel.org/doc/Documentation/kprobes.txt

https://elinux.org/images/d/dc/Kernel-Analysis-Using-eBPF-Daniel-Thompson-Linaro.pdf

https://docs.kernel.org/trace/uprobetracer.html

https://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html

https://isovalent.com/blog/post/next-generation-observability-with-ebpf/

https://medium.com/@samyukktha/decoding-ebpf-observability-how-ebpf-transforms-observability-as-we-know-it-c8680a4d6f0e

https://www.trendmicro.com/it_it/research/23/g/detecting-bpfdoor-backdoor-variants-abusing-bpf-filters.html