《openEuler内核编程》

课程讲稿

第七章 第3讲

BPF、Netfilter和eBPF的原理

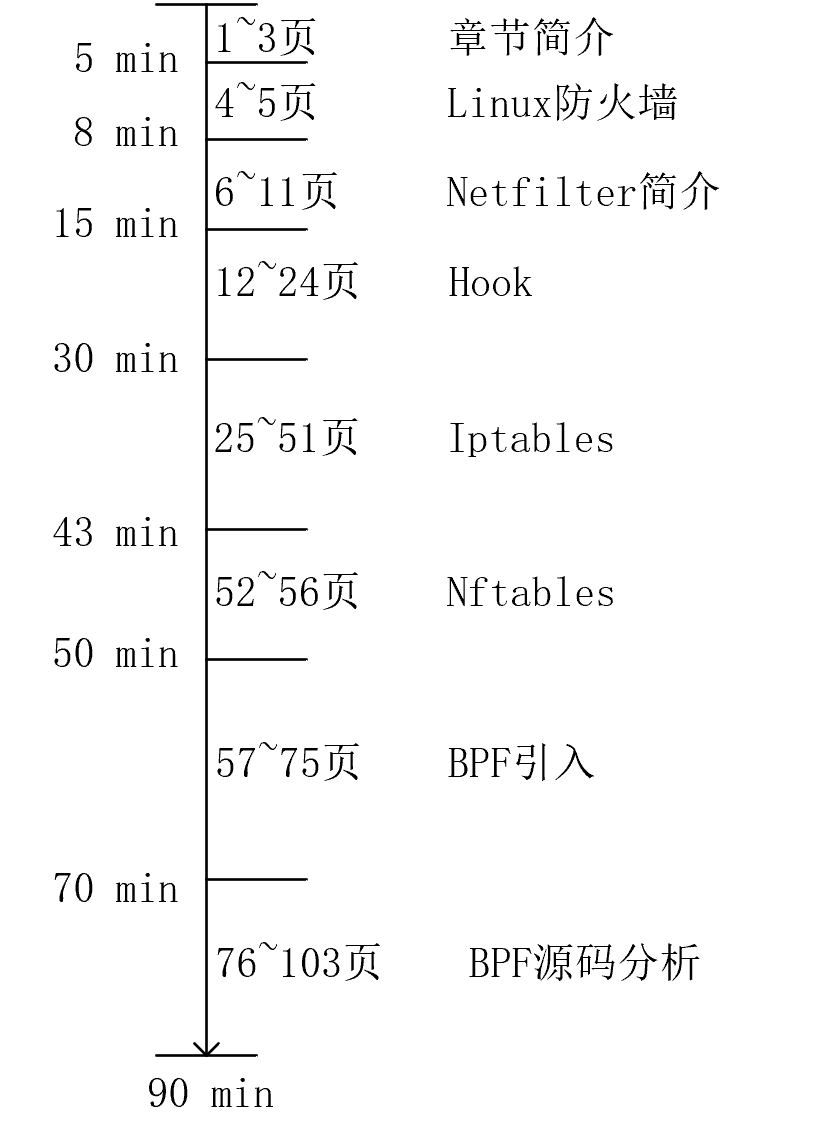
软件所制

第七章 第3讲 BPF、Netfilter和eBPF的原理

**学时：**2学时

**教学目的：**系统学习Netfilter和BPF，深入了解Netfilter和BPF原理

**课程时间线：**



**课外参考读物：**

<https://www.cnblogs.com/sparkdev/p/9328713.html>

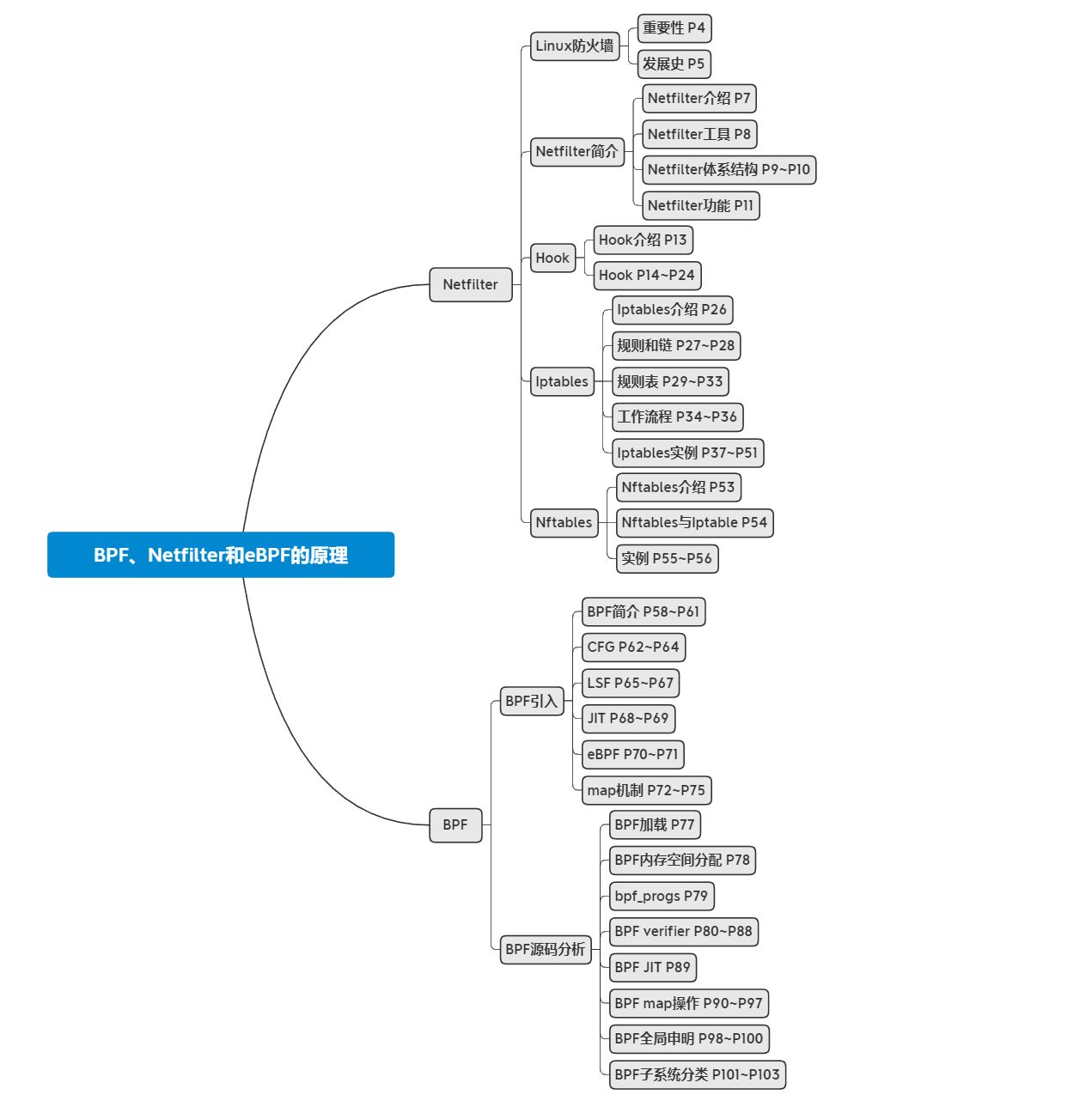
<https://blog.csdn.net/armlinuxww/article/details/76736934>

<https://www.ibm.com/developerworks/cn/linux/l-lo-eBPF-history/index.html>

<https://blog.csdn.net/pwl999/article/details/82884882>

<https://blog.csdn.net/F8qG7f9YD02Pe/article/details/79815702>

**知识框图：**



**PPT讲稿：**

1. 我们继续第七章网络知识的学习。
2. 这一节我们主要学习Netfilter的原理和实现。
3. 我们将从Linux防火墙的发展，netfilter简介，hook钩子，iptable和nftables五个方面学习。下面是第一个内容Linux防火墙的发展。
4. Linux防火墙有多重要呢？开放的、国际化的Internet的发展给政府机构、企事业单位的工作带来了革命性的变革和开放，使得他们能够利用internet提高办事效率、市场反应能力和竞争力。通过Internet，他们可以从异地传递重要的数据，同时也面临Internet开放所带来的数据安全的挑战与危险，举个例子，2000年2月上旬，Yahoo、eBay、CNN.com、Amazon等著名商业网站连续遭到黑客攻击，造成数十亿美元的损失。，因此如何保护企业的机密信息不受黑客和工业间谍的入侵已成为政府机构，企事业单位信息化建设健康发展所要考虑的重要因素之一。
5. 防火墙的发展史是从墙到链再到表，也是从简单到复杂的过程。这一过程对应的是ipfirewall--->ipchains--->iptables-->nftables(正在推广)。在Linux2.0内核中包过滤机制为ipfw，管理工具是ipfwadm，其中ipfw是ipfirewall的简称；Linux 2.2版内核中：包过滤机制为ipchain，管理工具是ipchains，对应的是ipchains；Linux 2.4，2.6,3.0+版内核中：包过滤机制为netfilter，管理工具是iptables，对应的是iptables；Linux 3.1（3.13+）版内核中：包过滤机制为netfilter，中间采取daemon动态管理防火墙，管理工具是firewalld，对应的是nftables。在Linux 2.4之前所有版本的内核级代码和用户级代码都是非模块化的，在2.4中增加的nefilter/iptables打破了这种局面。它在内核中实现了一个模块化，通用化的防火墙框架netfilter。用户可以自定义功能模块，例如用户可以编写一个可加载的内核模块来创建内核级代码，通过用户级代码的实现控制过滤器的行为等。
6. 目前Linux主流使用的防火墙机制就是netfilter。
7. 那么Netfilter是什么？Netfilter是Linux内核中的一个数据包处理框架，是Linux防火墙构建的基础，主要功能是数据包过滤和修改，NAT（网络地址转换）和端口转换等。它的实现原理是提供一系列钩子供其他模块控制包的流动，包括NF\_IP\_PRE\_ROUTING、NF\_IP\_LOCAL\_IN、NF\_IP\_FORWARD、NF\_IP\_LOCAL\_OUT和NF\_IP\_POST\_ROUTING。那么什么是钩子呢？这五个钩子就是在五个位置放置挂钩，并在这些hook点注册处理函数,这样当数据包经过这些hook点时,其上注册的处理函数就被依次调用注册处理函数时必须提供优先级，以便 hook 触发时能按照优先级高低调用处理函数。这使得多个模块（或者同一内核模块的多个实例）可以在同一 hook 点注册，并且有确定的处理顺序。内核模块会依次被调用，每次返回一个结果给 netfilter 框架，提示该对这个包做什么操作。
8. 目前很多工具都是基于netfilter的，IPVS即IP虚拟服务器是一个传输层负载均衡解决方案。在很早期的内核中就有对IPv4 IPVS的支持，在2.6.28内核之后增加了对IPv6的支持。IP sets 设置IP的管理工具。iptables可以说是最流行的Linux防火墙，它是netfilter的前端，为netfilter提供了一个管理层，比如可以增加和删除netfilter规则，列出统计信息，增加一个表，对表中计数器清零等。nftables 是下一代 iptables，比 iptables 更简洁易用，更易读，更容易理解，扩展性和也更好。但是目前各个发行版中对 nftables 的支持还比较参差不齐，导致 nftables 很多功能比 iptables 还是有所缺失，所以个人感觉短期内还是替代不了 iptables。
9. Netfilter的体系结构主要是通过表、链实现规则，可以这么说，Netfilter是表的容器，表是链的容器，链是规则的容器，最终形成对数据报处理规则的实现。例如，如果规则是处理 网络地址转换的，那会放到 nat table；如果是判断是否允许包继续向前，那可能会放到 filter table。详细地说，Netfilter/IPTables的体系结构可以分为三大部分：1. Netfilter的HOOK机制：HOOK提供了一种方便的机制，在数据报通过linux内核的不同位置上截获和操作处理数据报；IPTables基础模块：实现了三个表（filter表，nat表，mangle表）来筛选各种数据报。表是通过规则链实现的；具体功能模块包括数据报过滤模块、连接跟踪模块、网络地址转换模块、数据报修改模块 、其他高级功能模块。
10. 这张图展示了netfilter的整体架构，从图中我们可以看到，Netfilter 框架采用模块化设计理念，并且贯穿了 Linux 系统的内核态和用户态。在用户态层面，根据不同的协议类型，为上层用户提供了不同的系统调用工具，比如我们常用的针对 IPv4 协议 iptables，IPv6 协议的 ip6tables，针对 ARP 协议的 arptables，针对网桥控制的 ebtables，针对网络连接追踪的 conntrack 等等。不同的用户态工具在内核中有对应的模块进行实现，而底层都需要调用 Netfilter hook API 接口进行实现。关于体系结构总结：Netfilter的体系结构说明了从功能到实现的流程，以及各个层次的关系，就像建好的房子内地基与墙以及房梁加上房梁柱之间的系统关联一样。
11. netfilter框架提供了下面这些功能 包过滤 网络地址转换（NAT）包标记（在路由之前或之后修改包头的内容）连接跟踪 收集网络统计信息
12. hook是netfilter实现的关键。
13. 在Linux中定义了六种hook，但在网络协议栈中设置的只有5个钩子点，定义的最后一个用于记录钩子数。netfilter的基本实现框架就是根据网络报文的流向，在五个钩子点插入处理过程，利用钩子点查阅用户注册的回调函数，根据用户定义的回调函数来监视进出的网络数据包。NF\_INET\_PRE\_ROUTING 是刚刚通过数据链路层解包进入网络层的数据包通过此钩子，它在路由之前处理；NF\_INET\_LOCAL\_IN 是经过路由查找后，送往本机（目的地址在本地）的包会通过此钩子；NF\_INET\_FORWARD 是不是本地产生的并且目的地不是本地的包（即转发的包）会通过此钩子；NF\_INET\_POST\_ROUTING 是在包就要离开本机之前会通过该钩子，它在路由之后处理；NF\_INET\_LOCAL\_OUT 是所有本地生成的发往其他机器的包会通过该钩子。
14. 首先是NF\_INET\_PRE\_ROUTING，从图中可以看出刚刚通过数据链路层解包进入网络层的数据包通过此钩子，它在路由之前处理，它的入口函数是ip\_rcv，ip\_rcv是当报文从以太网进入网络层的必经之路，当判断为普通的ipv4报文则会交给ip\_rcv，在ip\_rcv将报文交给路由之前，会先处理写在NF\_INET\_PRE\_ROUTING挂钩点的函数。
15. 经过了NF\_INET\_PRE\_ROUTING后就会进行路由选择，经过路由查找后，送往本机（目的地址在本地）的包会通过此钩子NF\_INET\_LOCAL\_IN。NF\_INET\_LOCAL\_IN的入口函数是ip\_local\_deliver，在ip\_local\_deliver()中，如果发现数据报有被分片，则进行组装。然后调用NF\_INET\_LOCAL\_IN处的钩子函数，如果数据包被钩子函数放行，则调用ip\_local\_deliver\_finish()继续处理。
16. 同样是经过路由选择，但是不是目的地址不是本地即是转发的包，则会经过NF\_INET\_FORWARD挂钩，NF\_INET\_FORWARD挂钩的入口函数是ip\_forward，一般的出于安全考虑，Linux系统默认是禁止数据包转发的，所以要让Linux系统更具有路由转发的功能，需要通过ip\_forward。
17. 如果是在本机发出的数据包，在查询路由成功之后，会调用\_\_ip\_local\_out函数，函数首先进行必要字段设置和校验和计算，然后经过NF\_INET\_LOCAL\_OUT钩子点，之后会调用dst\_output继续完成数据包输出的其他工作。
18. ipv4的路由输出函数实际上就是ip\_output函数，在包就要离开本机之前会通过NF\_INET\_POST\_ROUTING钩子，转发的数据包或者是本地输出的数据包都会经过这个挂钩，最后都会经过ip\_output进行输出，函数设置设备和协议之后，经过NF\_INET\_POST\_ROUTING钩子点，之后调用ip\_finish\_output进行后续输出操作，其中包括了分片等。
19. 最后总结一下，当以太网的数据包要进入上层时会首先调用ip\_rcv，并在函数中经过NF\_IP\_PRE\_ROUTING挂钩，之后再进行路由判断，如果是目的地址是本机则会调用ip\_local\_deliver并经过NF\_IP\_LOCAL\_IN钩子。如果目的地址不是本机则进行转发，转发过程需要调用ip\_forward并经过NF\_IP\_FORWARD。如果是从本机来的数据想要进入下层则会调用\_\_ip\_local\_out并经过NF\_INET\_LOCAL\_OUT钩子点，最后不管是转发的数据包还是本地需要输出的数据包都要调用ip\_output并经过NF\_IP\_POST\_ROUTING钩子点。
20. 看了上面的代码都是调用的NF\_HOOK。NF\_HHOOK是nf\_hook宏，在数据包流过网络协议栈的过程中在多个不同的点被调用。它的参数包括pf协议族，hook钩子点，skb正在处理的包，in输入网络设备，out输出网络设备，okfn钩子结束后调用的函数。
21. nf\_hook函数首先找到钩子点函数入口，如果有钩子函数，则进一步初始化nf\_hook\_state结构，然后调用nf\_hook\_slow进入钩子函数调用流程；nf\_hook\_slow会遍历当前钩子点上的钩子函数，通过函数nf\_hook\_entry\_hookfn调用钩子函数，并根据返回值判断如何进行下一步处理。
22. 如果返回值是nf\_drop丢包释放数据包内存结束遍历直接返回；如果返回值是nf\_accept则接受数据包继续下一个钩子函数；如果是NF\_STOLEN则表示数据包已经被接管，回调函数处理该包，NF不再处理；如果返回NF\_QUEUE则将数据包加入到缓存队列中，交给用户空间的进程处理，结束遍历直接返回。如果是NF\_REPEAT会再次调用钩子函数。NF\_STOP不推荐使用。
23. 注册一个hook函数是围绕nf\_hook\_ops数据结构的一个非常简单的操作，nf\_hook\_ops数据结构在linux/netfilter.h中定义，该数据结构的定义如下，其中hook成员是一个指向nf\_hookfn类型的函数的指针，该函数是这个hook被调用时执行的函数。nf\_hookfn同样在linux/netfilter.h中定义。pf这个成员用于指定协议族。有效的协议族在linux/socket.h中列出，但对于IPv4我们希望使用协议族PF\_INET。hooknum这个成员用于指定安装的这个函数对应的具体的hook类型，其值为表1中列出的值之一。最后，priority这个成员用于指定在执行的顺序中，这个hook函数应当在被放在什么地方。
24. 有两种方法可以用来注册hook函数，一种是nf\_register\_net\_hook，用来注册单个nf\_hook\_ops，与它相对应的是注销单个nf\_hook\_ops对象的nf\_unregister\_net\_hook函数。另一种是nf\_register\_net\_hooks用于注册包含n个nf\_hook\_ops对象的数组，与之对应的是注销包含n个nf\_hook\_ops对象的数组nf\_unregister\_net\_hooks。
25. nftables 是下一代 iptables，比 iptables 更简洁易用，更易读，更容易理解，扩展性和也更好。但是目前各个发行版中对 nftables 的支持还比较参差不齐，导致 nftables 很多功能比 iptables 还是有所缺失，所以个人感觉短期内还是替代不了 iptables。
26. 虽然netfilter/iptables包过滤系统被称为单个实体，但它实际上由两个组件netfilter 和 iptables 组成。netfilter 组件也称为内核空间（kernelspace），是内核的一部分，由一些信息包过滤表组成，这些表包含内核用来控制信息包过滤处理的规则集。iptables 组件是一种工具，也称为用户空间（userspace），它使插入、修改和除去信息包过滤表中的规则变得容易。不同的内核模块和程序分别用于不同的协议，比如iptables用于IPv4、ip6tables用于IPv6、arptables用于ARP、ebtables用于以太网。
27. 由于iptables 组件是一种工具，可以插入、修改和除去信息包过滤表中的规则。而我们配置防火墙的主要工作就是添加、修改和删除这些规则。规则（rules）是网络管理员预定义的条件，规则一般的定义为“如果数据包头符合这样的条件，就这样处理这个数据包”。 规则存储在内核空间的信息包过滤表中，这些规则分别指定了源地址、目的地址、传输协议（如TCP、UDP、ICMP）和服务类型（如HTTP、FTP和SMTP）等。数据包与规则匹配时，iptables就根据规则所定义的方法来处理这些数据包，如放行（accept）、拒绝（reject）和丢弃（drop）等。除了规则还有链，链（chains）是数据包传播的路径，每一条链其实就是众多规则中的一个检查清单，每一条链中可以有一条或数条规则。当一个数据包到达一个链时，iptables就会从链中第一条规则开始检查，看该数据包是否满足规则所定义的条件。如果满足，系统就会根据该条规则所定义的方法处理该数据包；否则iptables将继续检查下一条规则，如果该数据包不符合链中任一条规则，iptables就会根据该链预先定义的默认策略来处理数据包。
28. 共有五种规则链INPUT——进来的数据包应用此规则链中的策略；OUTPUT——外出的数据包应用此规则链中的策略；FORWARD——转发数据包时应用此规则链中的策略；PREROUTING——对数据包作路由选择前应用此链中的规则；POSTROUTING——对数据包作路由选择后应用此链中的规则。而这五种规则链的先后顺序是入站顺序:PREROUTING→INPUT；出站顺序:OUTPUT→POSTROUTING；转发顺序:PREROUTING→FORWARD→POSTROUTING。
29. iptables的主要构成是表，iptables的操作是对iptables上的表的操作。iptables内置了四个表：即raw表、filter表、nat表和mangle表，分别用于实现包过滤，网络地址转换和包重构的功能。每个表中又对应着各个链。四个表的优先级由高到低的顺序为:raw-->mangle-->nat-->filter。
30. 首先是raw表，RAW表只使用在PREROUTING链和OUTPUT链上,因为优先级最高，从而可以对收到的数据包在连接跟踪前进行处理。一但用户使用了RAW表,在某个链上,RAW表处理完后,将跳过NAT表和 ip\_conntrack处理,即不再做地址转换和数据包的链接跟踪处理了.RAW表可以应用在那些不需要做nat的情况下，以提高性能。如大量访问的web服务器，可以让80端口不再让iptables做数据包的链接跟踪处理，以提高用户的访问速度。
31. nat表用于网络地址转换,iptables可以进行以下的nat:：DNAT:主要用于改变数据包的目的地址.以使数据包能重新路由到某台主机；SNAT:主要用于改变数据包的源地址,以帮助内部网络能连接到internet；MASQUERADE:和SNAT完全一样,只是MASQUERADE会查找可用的IP地址,而不像SNAT要有一个固定的IP,所以MASQUERADE一般用于ADSL/PPP等拔号共享上网的方式。Nat表包含三条连，PREROUTING链是可以在数据包到达防火墙的时候改变包的目标地址；OUTPUT链是可以改变本地产生的数据包的目标地址；POSTROUTING链:在数据包就要离开防火墙的时候改变数据包的源地址。网络数据包通过NAT操作后，数据包的地址发生了改变，这种改变是根据所定义的规则进行的。一个网络数据包只经过一次nat表，如果一个网络数据流的第一个数据包经过nat表，则剩余的数据包也会经过这个表。即其他的数据包不会一个一个地通过nat表，而是自动地完成这种转换操作。
32. mangle表主要用于修改数据包,通过mangle可以根据需要改变包头中的内容(如TTL,TOS,MARK),mangle表主要有以下几种操作：TOS操作:主要用于设置或改变数据包的服务类型域.该操作并不完善,无法在internet上使用；TTL操作:主要用于改变数据包的生存时间域,可以让所有的数据包只有一个特殊的TTL,这样可以欺骗一些ISP,比如ISP不希望看到共享上网的情况；MARK:主要用于给数据包设置特殊的标记,通过这些标记可以配置带宽限制和基于请求的分类,不过MARK并没有真正改动数据包,它只是在内核空间中为包设置了标记.防火墙通过其标记对包进行过滤和高级路由。Mangle表包含5个链：PREROUTING、POSTROUTING、OUTPUT、INPUT和FORWARD。PREROUTING在包进入防火墙之后、路由判断之前改变包；POSTROUTING是在所有路由判断之后；OUTPUT在确定包的目的之前更改数据包；INPUT在包被路由到本地之后，但在用户空间的程序看到它之前改变包；FORWARD在最初的路由判断之后、最后一次更改包的目的之前对包进行标记。
33. filter表主要用于过滤数据包，该表根据系统管理员预定义的一组规则过滤符合条件的数据包。对于防火墙而言，主要利用在filter表中指定的规则来实现对数据包的过滤。Filter表是默认的表，如果没有指定哪个表，iptables 就默认使用filter表来执行所有命令，filter表包含了INPUT链（处理进入的数据包），RORWARD链（处理转发的数据包），OUTPUT链（处理本地生成的数据包）在filter表中只能允许对数据包进行接受，丢弃的操作，而无法对数据包进行更改。
34. 当数据进入网络有三种可能情况，一是数据包的目的地址是本机，二是数据包的源地址是本机，三是经过本地转发的数据包。一般的数据包通过防火墙，来自外界的数据包到达防火墙后，首先被PREROUTING规则链处理这里需要验证是否修改数据包地址等，之后会进行路由选择来判断该数据包应该发往何处，如果数据包的目标地址是防火墙本机，如Internet用户访问防火墙主机中Web服务的数据包，那么内核将其传递给INPUT链进行决定是否允许通过，通过以后再交给系统上层的应用程序（如httpd服务器）进行响应。如果需要转发，来自外界的数据包到达防火墙后，首先被PREROUTING规则链处理，之后会进行路由选择，如果数据包的目标地址是其他外部地址（如局域网用户通过网关访问QQ站点的数据包），则内核将其传递给FORWARD链进行决定是否转发或拦截，然后再交给POSTROUTING规则链（是否修改数据包的地址等）进行处理。防火墙本机向外部地址发送的数据包，首先被OUTPUT规则链处理，之后进行路由选择，然后传递给POSTTOUTING规则链（是否修改数据包的地址等）进行处理。具体来说如果目的地址是本机为例：第一步:数据包进入网络接口；第二步:进入mangle表的PREROUTING链,在这里可以根据需要改变数据包头内容(比如数据包的TTL值)；第三步:进入nat表的PREROUTING链,在这里可以根据需要做DNAT(目标地址转换)；第四步:进行路由判断(进入本地还是要转发)；第五步:进入mangle表的INPUT链,在路由之后到达本地程序之前修改数据包头内容；第六步:进入filter表的INPUT链,所有目标地址是本机的数据包都会经过这里,可以在这里对数据包的过滤条件进行设置；第七步:到达本地应用程序处理。
35. 当数据包的源地址是本机时，第一步:本地应用程序产生数据包；第二步:路由判断；第三步:进入mangle表的OUTPUT链,在这里可以根据需要改变包头内容；第四步:进入nat表的OUTPUT链,在这里可以根据需要对防火墙产生的数据包做DNAT；第五步:进入filter表的OUTPUT链,在这里可以对数据包的过滤条件进行设置；第六步:进入mangle表的POSTROUTING链,这里主要对数据包做DNAT操作,数据包离开本机之前修改数据包头内容；第七步:进入nat表的POSTROUTING链,在这里对数据包做SNAT(源地址转换)；第八步:离开本地。
36. 当经过本机转发的数据包(源地址,目标地址都不是本机)时，第一步:数据包进入网络接口；第二步:进入mangle表的PREROUTING链,在这里可以根据需要修改数据包头内容(如TTL值)；第三步:进入nat表的PREROUTING链,在这里可以根据需要对数据包做DNAT；第四步:进入路由判断(进入本地还要转发)；第五步:进入mangle表的FORWARD链,在这里数据包头内容被修改.这次mangle发生在最初的路由判断之后,在最后一次更改数据包的目标之前；第六步:进入filter表的FROWARD链,只有需要转发的数据包才会到达这里,并且针对这些数据包的所有过滤也在这里进行,即所有转发的数据都要经过这里；第七步:进入mangle表的POSTROUTING链,这个链也是针对一些特殊类型的数据包,这一步修改数据包内容是在所有包的目的地址的操作完成之后才进行；第八步:进入nat表的POSTROUTING链,在这里可以根据需要对数据包做SNAT,当然也包括Masquerade(伪装),但不进行过滤；第九步:离开网络接口
37. 为了理解iptables是如何工作的，我们用一个包含filter表的现实例子来看看，为了简单起见，我们假设只构建了一个filter表，并且支持log操作，使用的规则也仅用于日志管理。
38. 首先来看看filter表的定义，table通过 ipt\_register\_table向系统进行注册，在filter\_valid\_hooks中可以找到对应的三个钩子点。Filter的信息存储在一个xt\_table的结构中，并调用iptable\_filter\_table\_init进行初始化。
39. Filter表的初始化会依次调用iptable\_filter\_init->iptable\_filter\_net\_ops->iptable\_filter\_net\_init->iptable\_filter\_table\_init最终由iptable\_filter\_table\_init完成注册和保存，注册filter表，注册后，ipv4.iptable\_filter保存了注册后的新表。
40. 然后我们利用iptables命令向filter表添加一条规则，记为规则A：iptables -A INPUT -p udp --dport=5001 -j LOG --log-level 1，此规则的含义是，将目标端口号是5001的UDP数据包记录到系统日志中，log-level是标准系统log级别，范围是0-7，0为紧急事件，7为调试。
41. 这是iptable规则，包括四个基本元素：表、命令、匹配和目标。一般的设置规则是这样的格式。其中-t table是指定设置的表（filter nat mangle）不指定默认为filter；COMMAND是定义如何对规则进行管理；chain是指定你接下来的规则到底是在哪个链上操作的，当定义策略的时候，可以省略；CRITERIA指定匹配标准；-j ACTION 指定如何进行处理。首先是command命令，下表展示的是各种命令以及对应的功能和他对应的语法。
42. CRITERIA是指定匹配标准，由通用匹配和扩展匹配两种。通用匹配时源地址目标地址的匹配，扩展匹配又分隐含扩展和显示扩展，其中隐含扩展是对协议的扩展。
43. -j ACTION :指定如何进行处理，可选的选项有DROP表示丢弃、REJECT表示拒绝、ACCEPT表示接受等待。
44. 现在来看我们的规则A。当一个目标端口号是5001的UDP数据包到达网络驱动并走到网络层（L3）时，遇到的第一个钩子是NF\_INET\_PRE\_ROUTING，但是filter并没有注册这个钩子。
45. 因此程序会继续调用ip\_rcv\_finish函数并进行路由选择，现在有两种可能的情况，一种是数据包传递到本机，另一种是转发。这里我们忽略丢包的情况。
46. 如果数据包传递到本机则会调用ip\_local\_deliver函数，在函数中进行分片重组然后经过LOCAL\_IN钩子点。
47. 我们可以看到，ip\_local\_deliver()函数中有NF\_INET\_LOCAL\_IN钩子，该钩子是filter表的钩子之一，所以NF\_HOOK宏会调用iptable\_filter\_hook()函数。
48. ipt\_do\_table()调用LOG操作函数ipt\_log\_packet()，该函数将包头写入syslog中。如果有更多的规则，则在这里被调用。在我们的例子中没有其他规则，则继续执行ip\_local\_deliver\_finish()函数，并将包传递给传输层（L4）。
49. 第二种情况是转发包。在路由子系统中查找之后，您发现要转发数据包，因此调用了ip\_forward方法。Ip\_forward函数中会经过FORWARD钩子点。
50. 由于filter表在NF\_INET\_FORWARD钩子点注册了回调函数，所以会调用iptable\_filter\_hook()函数，所以，与前面的传送到本地的情况类似，会调用ipt\_do\_tables()，ipt\_log\_packet()，及ip\_forward\_finish()函数。
51. 最后调用ip\_output()函数，由于filter表没有NF\_INET\_POST\_ROUTING钩子，所以会直接调用ip\_finish\_output()函数。
52. Nftables可以看作是下一代的iptables。
53. nftables在Linux 3.13中正式发布，该项目为Linux提供包过滤和包分类的功能，致力于替代已有的iptables，ip6tables，arptables和ebtables框架。为iptables和ip6tables提供一个新的包过滤框架、一个新的用户空间实用程序（nft）和一个兼容层。它使用现有的钩子、链接跟踪系统、用户空间排队组件和 netfilter 日志子系统。nftables 主要由三个组件组成：内核实现、libnl netlink 通信和 nftables 用户空间。其中，内核提供了一个 netlink 配置接口以及运行时规则集评估；libnl 包含了与内核通信的基本函数；用户空间可以通过 nft 和用户进行交互。
54. nftables 和 iptables 一样，由表（table）、链（chain）和规则（rule）组成，其中表包含链，链包含规则，规则是真正的 action。与 iptables 相比，nftables 主要有以下几个变化：1. iptables 规则的布局是基于连续的大块内存的，即数组式布局；而 nftables 的规则采用链式布局；2. iptables 大部分工作在内核态完成，如果要添加新功能，只能重新编译内核；而 nftables 的大部分工作是在用户态完成的，添加新功能很简便，不需要改内核；3. iptables 有内置的链，即使你只需要一条链，其他的链也会跟着注册；而 nftables 不存在内置的链，你可以按需注册。由于 iptables 内置了一个数据包计数器，所以即使这些内置的链是空的，也会带来性能损耗。4.nftables简化了了 IPv4/IPv6 双栈管理；5.nftables支持集合、字典和映射。
55. 举几个例子来比较nftables和iptables。假如想用iptables记录并丢弃一个包，你必须写两条规则，一条记录，一条丢弃，就像这样。而使用nftables，你可以把两个目标合并到一起，将log和drop写在一条命令里。
56. 又假如你想针对不同的端口允许包并且允许不同的icmpv6类型，使用iptables需要写一下五条。而使用nftables，集合可以用在规则中任一元素。这样更容易编写，而且对于过滤方更加高效，因为对每一种协议只需要添加一个规则。
57. 我们将从BPF的引入和BPF的源码分析两个角度进行学习。
58. BPF可能我们比较生疏，但是我说起tcpdump、Wireshark等流行的网络抓包和分析工具你一定听说并可能使用过，他们底层的包过滤实现就是用的BPF。那么什么是BPF，BPF又是做什么用的呢？一篇来自92年的论文告诉我们答案。BPF又叫伯克利包过滤器Berkeley Packet Filter，最初构想提出于 1992 年，其目的是为了提供一种过滤包的方法。在 BPF 出现之前，如果你想去做包过滤，你必须拷贝所有的包到用户空间，然后才能去过滤它们，比如sunos的nit、DEC Ultrix的Ultrix包过滤器和SGI IRIX的Snoop。这样做存在两个问题：如果你在用户空间中过滤，意味着你将拷贝所有的包到用户空间，拷贝数据的代价是很昂贵的。还有就是使用的过滤算法很低效。而BPF的出现就是为了解决这两个问题。
59. 这是那篇发表于1992年的介绍BPF的论文，有兴趣的同学可以阅读一下。
60. BPF有两个主要的组件，一个是网络分接头，另一个是数据包过滤器。网络分接头是从网络设备驱动程序收集数据包的副本，并将其传递给侦听应用程序；而数据包过滤器是决定是否应该接受数据包，以及将多少数据包复制到侦听应用程序。
61. 这张图描述了BPF的主要工作流程：当一个数据包到达网络接口时，数据链路层的驱动会把它向系统的协议栈传送。但如果 BPF 监听接口，驱动首先调用 BPF。BPF 首先进行过滤操作，这个用户定义的过滤器决定是否接收一个数据包以及应保存每个数据包多少字节。对于每个接收数据包的过滤器，BPF会将请求的数据量复制到与该过滤器关联的缓冲区中，最后设备驱动再次获得控制。如果该数据包未寻址到本地主机，则驱动程序从中断返回，否则，进行正常的协议处理。
62. BPF采用的报文过滤设计的全称是CFG，是将过滤器构筑于一套基于if-else的控制流之上。一般的，数据过滤模型中数据包过滤器只是数据包上的布尔值函数，如果该函数值为TRUE则内核为应用程序复制数据包，否则忽略该数据包。从历史上看，有两种方法可以实现过滤器抽象，一种是布尔表达式树，另一种是有向非循环控制流程图。从这两张图中，我们可以来比较两种方法。该过滤器可以识别以太网上的IP或ARP数据包。 在树模型中，每个节点表示一个布尔运算，而叶子表示在数据包字段上的测试谓词。 边缘代表算子-操作数关系。 在CFG模型中，每个节点代表一个数据包字段谓词，而边缘则代表控制传输。 如果谓词为true，则遍历右侧分支，如果为false，则遍历左侧分支。 有两个终止叶子，它们代表整个过滤器的是非。
63. CSPF筛选器引擎基于操作数堆栈。 指令将常量或数据包数据压入堆栈，或者对前两个元素执行二进制布尔或按位运算。 过滤程序是顺序执行的指令列表。 在评估程序之后，如果堆栈的顶部具有非零值或堆栈为空，则该数据包将被接受，否则将被拒绝。但是这种表达式树结构在实现上存在几大缺点：1.必须模拟操作数堆栈。 在大多数现代机器上，这意味着生成加法和减法操作以维护模拟的堆栈指针，并实际执行加载和存储到内存以模拟堆栈。由于内存往往是现代体系结构中的主要瓶颈，因此可以使用机器寄存器中的值并避免这种内存流量的筛选器模型将更加有效。2. 布尔表达式树模型会执行不要或多余的计算，就像这张图中，即使已经计算过ether.type == IP 为true但是另一条指令也会再做一遍。虽然可以添加“短路”操作来缓解此问题，但是效率还是比较低3. 无法解析可变长度的数据包报头，例如封装在可变长度IP报头中的TCP报头。 由于CSPF指令集未包含间接运算符，因此只能访问固定偏移量的数据包数据。举个例子，如果你运行 tcpdump host foo，它实际上运行了一个相当复杂的查询，用下图的这个树来描述它：那么他每一个叶子节点都需要计算一遍。
64. 而CFG就试图在保持CSPF优势的前提下，解决其局限性和性能上的缺陷。在CFG模型中，每个节点代表一个数据包字段谓词，而边缘则代表控制传输。 如果谓词为true，则遍历右侧分支，如果为false，则遍历左侧分支。 有两个终止叶子，它们代表整个过滤器的是非。CFG允许将解析信息内置到流程图中，也就是说在图中记录了数据包解析状态。同样的例子，如果是CFG模型，如果你设置 ether.type = IP 和 ip.src = foo，很快就能判断包是 host foo。所以这个模型叫做控制流图。
65. BPF 是在 1997 年首次被引入 Linux 的，当时的内核版本尚为 2.1.75。准确的说，Linux 内核中的报文过滤机制其实是有自己的名字的：Linux Socket Filter，简称 LSF。尽管BSD和Linux kernel的过滤(filter)有一些明显的区别，但是当我们在linux环境中谈到BPF或LSF时，我们指的在Linux kernel中完全相同的过滤机制。LSF中有三个系统调用，其中，SO\_ATTTACH\_FILTER选项发送到内核，将val代码块传入内核；SO\_DETACH\_FILTER选项把filter从socket上移除，这个一般不常使用，因为当你关闭socket的时候如果有filter会被自动移除；SO\_LOCK\_FILTER选项运行锁定附加到socket上的filter，一旦设置，filter不能被移除或者改变。这种允许一个进程设置一个socket、附加一个filter、锁定它们并放弃特权，确保这个filter保持到socket的关闭。
66. 以tcpdump -dd为例，Code数组是由tcpdump –dd生成的，一般的，-d是将匹配信息包的代码以人们能够理解的汇编格式给出；-dd是将匹配信息包的代码以C语言程序段的格式给出。Code中数组遵循BPF指令的固定形式，其中opcode是操作码字段指示指令类型和寻址模式；jt和jf是条件跳转指令，是从下一条指令到true和false目标的偏移量；k是使用于各种目的的通用字段。
67. 在实际运用中，会利用SO\_ATTACH\_FILTER将bpf代码块传入内核。上述实例代码附加一个socket filter到一个AF\_PACKET socket上，为了让所有IPv4/IPv6 port 22的包通过。这个socket上所有其他的包将会被丢弃。
68. BPF 被引入 Linux 之后，除了一些小的性能方面的调整意外，很长一段时间都没有什么动静。直到 3.0 才首次迎来了比较大的革新：在一些特定硬件平台上，BPF 开始有了用于提速的 JIT(Just-In-Time) Compiler。最先实现 JIT 的是x86平台，其后包括arm、ppc、S390、mips等一众平台纷纷跟进，到今天 Linux 的主流平台中支持 JIT For BPF 的已经占了绝大多数了。BPF JIT 的接口是bpf\_int\_jit\_compile()，如果 CONFIG\_BPF\_JIT 被打开，则传入的 BPF 伪代码就会被传入该函数加以编译，编译结果被拿来替换掉默认的处理函数 sk\_run\_filter()。打开 BPF 的 JIT 很简单，只要向/proc/sys/net/core/bpf\_jit\_enable 写入 1 即可；对于有调试需求的开发者而言，如果写入 2 的话，还可以在内核 log 中看到载入 BPF 代码时候 JIT 生成的优化代码，内核开发者们还提供了一个更加方便的工具bpf\_jit\_disam，可以将内核 log 中的二进制转换为汇编以便阅读。
69. 对于JIT开发人员做审核等，每一次编译运行都可以通过以下方式将生成的操作码映像输出到内核日志中，但是这样看很难看出问题，所以需要通过bpf\_jit\_disam工具从内核日志的hexdump生成反汇编代码，也可以添加选项“-o”将“注释”操作码到生成的汇编指令，这对JIT开发人员非常有用。
70. 原始的BPF在内核内部运行用户的提供的程序的能力被证明是一个有用的设计决定，但是原始BPF设计的其他方面却表现的不是那么的理想，一方面是随着现代处理器移至64位寄存器并发明了多处理器系统所需要的新指令，BPF对提供少量RISC指令的关注不再符合现代处理器的现实。自 3.15 伊始，一个套源于 BPF 的全新设计开始逐渐进入人们的视野，并最终(3.17)被添置到了 kernel/bpf 下。这一全新设计最终被命名为了 extended BPF(eBPF)：顾名思义，有全面扩充既有 BPF 功能之意；而相对应的，为了后向兼容，传统的 BPF 仍被保留了下来，并被重命名为 classical BPF(cBPF)。相对于 cBPF，eBPF 带来的改变可谓是革命性的：一方面，它已经为内核追踪(Kernel Tracing)、应用性能调优/监控、流控(Traffic Control)等领域带来了激动人心的变革；另一方面，在接口的设计以及易用性上，eBPF 也有了较大的改进。比较eBPF和BPF，做出了以下改进：1. 旧的格式拥有两个寄存器A和X，以及一个隐藏的堆栈指针(frame pointer)。新的布局10个内部寄存器和一个只读的堆栈指针。由于64位cpu通过寄存器将参数传递给函数，因此从eBPF程序到内核函数的args数量限制为5个，一个寄存器用于接受内核函数的返回值。原生的，x86\_64在寄存器中传递前6个参数，aarch64/sparcv9/mips64有7 - 8个寄存器作为参数;x86\_64有6个被调用者保存寄存器，aarch64/sparcv9/mips64有11个或更多被调用者保存寄存器。2.为了对BPF兼容，最初的32位ALU操作的语义仍然通过32位子寄存器保存。所有eBPF寄存器都是64位的，低32位为子寄存器高32位为零扩展。3. 条件jt/jf目标替换为jt/fall-through。4.在调用一个内核函数之前，内部BPF程序需要将函数参数放入R1到R5寄存器以满足调用约定，然后解释器将从寄存器中取出它们并传递给内核函数。给定体系结构上如果R1 - R5寄存器被映射到CPU寄存器用参数于传递，JIT编译器不需要额外的动作。函数参数将在正确的寄存器中，BPF\_CALL指令将被JIT翻译成单个’call’ HW指令。这个调用约定是用来覆盖通用的调用场景而没有性能损失。
71. 在64位体系结构上，所有到HW的寄存器映射都是一对一的。例如，x86\_64 JIT编译器可以将它们映射为如图所示。在内核函数调用之后，R1 - R5被重置为不可读，R0有函数的返回值。因为R6 - R9是被调用保护，它们的状态在整个调用中需要被保护。R0-R5是暂存寄存器，所以eBPF程序需要在两次调用之间保护它们。在函数调用之后，寄存器R1-R5包含垃圾值不能读取。一个内部eBPF校验器会用来验证eBPF程序。
72. 由 eBPF 伊始，BPF 程序开始有分类了，通过 bpf\_load\_program 的参数 bpf\_prog\_type，我们可以看到 eBPF 支持的程序类型。下面罗列了一下常用的类型。
73. 在BPF代码进内核之后，cBPF 和内核通讯的方式是 recv()；而 eBPF 则将 socket 丢到一边，使用一种名为 map 的全新机制和内核通讯，其大致原理下图所示。这套设计本身不复杂：位于用户空间中的应用在内核中辟出一块空间建立起一个数据库用以和 eBPF 程序交互(bpf\_create\_map())；数据库本身以 Key-Value 的形式进行组织，无论是从用户空间还是内核空间都可以对其进行访问，两边有着相似的接口，最终在逻辑上也都殊途同归。“map”是用于在内核和用户空间之间共享不同类型数据的通用存储机制。不难发现，map 带来的最大优势是效率：相对于 cBPF 一言不合就把一个通信报文从内核空间丢出来的豪放，map 机制下的通讯耗费就要小家碧玉的多了。map 机制解决的另一个问题是通信数据的多样性问题。cBPF 所覆盖的功能范围很简单，无外乎是网络监控和 seccomp 两块，数据接口设计相对比较粗放；而 eBPF 的利用范围则要广的多，性能调优、内核监控、流量控制什么的应有尽有，数据接口的多样性设计就显得很必要了。map的存取通过BPF系统调用，它拥有以下命令。1.通过给定的类型和参数创建map，用的调用如下。2.在给定map中查找关键词。3.在给定的map中创建或者更新key/value键值对。4.在给定map中通过关键词查找或者删除元素。5.删除map。
74. 下表中就列出了现有 eBPF 中的 map 机制中常见的数据类型。虽然目前eBPF支持多种map类型，但是从类型名字中可以看出，map类型可以分为hash和array两大类，也就对应了两种不同的实现方式。而每一类中不同map类型的实现又有共同的部分，所以懂得了每一大类map共有部分的实现，就对大类内其他类型的map实现有了整体的了解，然后再结合各自应用场景，就可以比较好地掌握适用于不同场景的map的实现。
75. 总结一下这是eBPF的整体架构。首先是map数组，位于用户空间中的应用在内核中辟出一块空间建立起一个数据库用以和 eBPF 程序交互。然后是BPF程序，在经过内核的验证过后，经由JIT即时编译器编译成内核可识别的语言，最终的实现通过BPF\_PROG\_RUN执行。
76. 了解了BPF的理论知识，下来看看BPF的源码分析。
77. BPF的入口函数在kernel/bpf/syscall.c中，系统调用名称为bpf。其主要构成是一个switch-case结构，根据传入的命令选取合适的操作。一般的程序开始是加载BPF即BPF\_PROG\_LOAD即调用bpf\_prog\_load函数。这个过程可以分为四个步骤：拷贝程序到内核、校验安全性、如果可能则对它进行JIT编译、分配一个文件句柄fd给它。
78. 这是bpf\_prog\_load的代码实现。Bpf\_prog\_load的第一步检查的ebpf license是否为GPL证书的一种，然后检查指令条数是否超过4096。第二步根据bpf指令数分配bpf\_prog空间并将bpf代码从用户空间地址attr->insns拷贝到内核空间地址prog->insns。第三步是使用verifer对bpf重新进行合法性扫描。第四步是尝试对bpf程序进行jit转换。最后一步给bpf程序分配一个文件句柄fd。
79. 这是第二步新建的bpf\_progs数据结构。其中len是程序包含bpf指令的数目；type是当前bpf程序的类型；aux主要用来辅助verifer校验和转换的数据；bpf\_func是运行时BPF程序的入口。如果JIT转换成功，这里指向的就是BPF程序JIT转换后的映像；否则这里指向内核解析器(interpreter)的通用入口\_\_bpf\_prog\_run()；联合体中insns是从用户拷贝来的ebpf源程序，而bpf\_insnsi是从用户拷贝来的bpf程序原始指令的存放空间。
80. Bpf进行verifier验证的过程是一个类似于深度优先搜索的过程。他的整个过程分为两个步骤，第一步是深度优先搜索，检查程序是否为DAG即有向无环图，当程序中出现指令数大于4096或者出现循环或者不可达指令或者越界和畸形跳跃这些程序都将被拒绝。第二步是从第一步找到的所有可能路径中展开，限制分析的最大擦汗高难度为32k个指令，分支的分析数量最大为1k。
81. 下面列举一些验证过程中的规则有正例也有反例。rule 1、在程序的开始R1包含指向context的指针类型为PTR\_TO\_CTX。如果verifier看到的指令为R2=R1，那么R2现有也有了类型PTR\_TO\_CTX，并且可以在表达式右侧使用。如果是 R1=PTR\_TO\_CTX and R2=R1+R1, 那么R2=SCALAR\_VALUE，因为两个指针相加导致无效指针。rule 2、如果寄存器从来没被写过，那它是不可读的，将被拒绝，因为R2在程序开始时不可读。rule 3、在内核函数调用以后，R1-R5重置成不可读状态，R0拥有一个函数的返回类型。而R6-R9是对被调用者保护的，所以它们的状态在函数调用后不变。这里R0=R1将会被拒绝。rule 4、load/store指令只有在寄存器类型有效时才被运行，包含PTR\_TO\_CTX, PTR\_TO\_MAP, PTR\_TO\_STACK类型。它们还有边界和对齐检查。将会被拒绝，因为在执行指令bpf\_xadd时R1没有有效的指针类型。
82. Rule5 打算从地址R6 + 8加载一个word并将其存储到R0中。如果R6=PTR\_TO\_CTX，通过is\_valid\_access()回调，验证者将知道偏移量为8的4字节成员是否可读，否则验证者将拒绝程序。如果R6=PTR\_TO\_STACK，那么访问应该对齐并在堆栈边界内，即[-MAX\_BPF\_STACK, 0] 。在这个例子中偏移量是8，因此验证失败，因为它超出了界限。Rule6 尽管R0是正确PTR\_TO\_STACK类型的只读寄存器，并且R10 - 4也在堆栈范围内，但是那个位置没有被存储过。rule 7指针寄存器spill/fill也被被追踪(spill/fill就是寄存器push/pop堆栈)，因为4个被调用者保护寄存器(R6-R9)对某些程序来说不够用。rule 8使用pf\_verifier\_ops->get\_func\_proto()来决定是否允许函数调用，eBPF verifier将检查寄存器是否匹配参数约束。在函数调用以后寄存器R0将被设置为函数返回类型。rule 9函数调用是扩展eBPF程序功能的主要机制。rule 10如果一个函数可以被eBPF程序访问，它需要从安全的角度考虑。verifier将确保使用有效的参数调用函数。cBPF中seccomp和socket filters拥有不同的安全限制，而对于eBPF，所有情况共享一个可配置的verifier。
83. BPF verifier具体实现首先是调用replace\_map\_fd\_with\_map\_ptr函数，根据\_\_section(“maps”)中定义的map调用bpf()创建map，再加载其他的程序section，这一步的代码将在后续讲；然后将符合条件的指令为map指令的加载指令；把原始的立即数作为fd找到对应的map指针；把64bit的map指针拆分成两个32bit的立即数，存储到insn[0].imm、insn[1].imm中。
84. replace\_map\_fd\_with\_map\_ptr返回后会运行下一步操作check\_cfg()，作用是检查有没有环路，通过SAG的DFS来遍历BPF程序的代码路径，确保没有环路发生。这里有四个概念需要介绍。Tree edges是通过dfs后在树中存在的边；back edges是从子节点连向祖先节点的边，也就是形成了环，如图中的13和465，这是我们要找到需要拒绝的；forward edges是直接连向孙子节点的边，即跨越了子节点的连接，如图中的0跳过4直接连7；cross edges是交叉边，叶子之间的连接如23或者是树之间的连接如1到0。
85. 这是一个以栈为基础实现的深度优先搜索的代码，他的跳转是通过操作码和goto跳转实现的。首先进入分支指令的程序是一个if语句if (BPF\_CLASS(insns[t].code) == BPF\_JMP)，insns数组中存储的是ebpf指令集，以数组的形式存储。循环每次取出一条指令的操作码，当碰到BPF\_EXIT指令，路径终结，开始回溯确认，这里的mark\_explored是遇到函数末尾，对执行过的指令进行explored标记，并退栈操作，即深度搜索到一个分支的叶节点，需要回溯了。如果碰到BPF\_CALL指令，继续探索，其中ret = push\_insn(t, t + 1, FALLTHROUGH, env);中因为初始时e=fallthrough但insn\_state[0]=discovered，所以前几个if都不会进入，这里到insn\_state[1]==0成立，然后赋值，这里discovered表示已经发现，而discovered|fallthrough表示已经发现且已经走过了，这里将前一条指令赋值discovered|fallthrough后一条指令赋值discovered，将下一条指令入栈，并返回1表示继续探索，ret一般的为1表示入栈成功继续跳转peek\_stack从新的节点继续探索，为0表示已经是叶子节点了需要跳转到mark\_explored确认并回溯，而小于0表示探测到back-edge环路了，当一个指令的状态是discovered|fallthrough时即为0x10|1，此时，(insn\_state[w] & 0xF0) == DISCOVERED）成立。
86. 跳转指令在ret = push\_insn(t, t + insns[t].off + 1,FALLTHROUGH, env);中通过off偏移确定下一个指令。最后检查如果在结束时还有没有执行过的指令，说明有环则verifier失败。
87. do\_check这段代码的具体算法就是把刚才的路径重新走一遍，并且跟踪寄存器和堆栈的变化，判断最坏情况下是否有违反规则的情况出现。在碰到指令对应explored\_states[]被设置成STATE\_LIST\_MARK，需要给当前指令独立分配一个bpf\_verifier\_state\_list链表，来存储这个指令在多个分支上的不同状况。这里也有一个快速分析的优化方法：修剪(Pruning)。如果当前指令的当前分支的状态cur\_state，和当前指令另一个已分析分支的状态(当前指令explored\_states[]链表中的一个bpf\_verifier\_state\_list成员)相等或者是它的一个子集，那么当前指令的当前分支就不需要分析了，因为它肯定是符合规则的。
88. fixup\_bpf\_calls用于修复BPF指令中对内核helper function函数的调用，把函数编号替换成实际的函数指针，对于符合条件的指令，就是那些调用helper function的指令进行处理，处理过程是根据insn->imm指定的编号找打对应的函数指针，然后再把函数指针和\_\_bpf\_call\_base之间的offset，赋值到insn->imm中。
89. 在verifier验证通过以后，内核通过JIT(Just-In-Time)将BPF目编码转换成本地指令码；如果当前架构不支持JIT转换内核则会使用一个解析器(interpreter)来模拟运行，这种运行效率较低；有些架构(64 bit x86\_64, arm64, ppc64, s390x, mips64, sparc64 and 32 bit arm)已经支持BPF的JIT，它可以高效的几乎一比一的把BPF代码转换成本机代码(因为eBPF的指令集已经做了优化，非常类似最新的arm/x86架构，ABI也类似)。JIT的核心转换部分分为三个不放呢prologue+body+epilogue。prologue：新增的指令，负责BPF运行堆栈的构建和运行现场的保护；body：BPF主体部分；epilogue：负责BPF运行完现场的恢复和清理。
90. BPF map的操作，map的应用场景有一下四种：1. BPF程序和用户态态的交互：BPF程序运行完，得到的结果存储到map中，供用户态访问；2. BPF程序内部交互：如果BPF程序内部需要用全局变量来交互，但是由于安全原因BPF程序不允许访问全局变量，可以使用map来充当全局变量；3. BPF Tail call：Tail call是一个BPF程序跳转到另一BPF程序，BPF程序首先通过BPF\_MAP\_TYPE\_PROG\_ARRAY类型的map来知道另一个BPF程序的指针，然后调用tail\_call()的helper function来执行Tail call；4. BPF程序和内核态的交互：和BPF程序以外的内核程序交互，也可以使用map作为中介；
91. 首先map的创建时在bpf的系统调用中通过BPF\_MAP\_CREATE调用map\_create函数实现的，在map\_create中调用find\_and\_alloc\_map函数根据map的类型分配空间。
92. Map\_create中传入的参数bpf\_attr结构的代码如右上图，map\_type是map的类型，key\_size是key成员的大小，value\_size是值value成员的大小，max\_entries是需要存储多少个条目即键值对。Map\_alloc将根据map的类型调用相应的分配函数，由下图展示的是bpf map的一些操作函数集包括对map的增删查改操作。另外除了用户态空间需要通过bpf()系统调用来查找key对应的value值。BPF程序中也需要根据key查找到value的地址，然后在BPF程序中使用。BPF程序时通过调用BPF\_FUNC\_map\_lookup\_elem helper function来实现的。
93. map\_alloc将根据map类型调用相应的分配函数，首先来看第一类BPF\_MAP\_TYPE\_ARRAY。入口函数是array\_map\_alloc函数，先计算value的大小，因为key的值直接就是index所以key的大小就不用计算了。然后计算数组的总大小bpf\_array+value，bpf\_array中包含了map的通用结构bpf\_map。得到总大小就可以根据这个大小给bpf\_array分配空间了。最后返回array的map。右上角的图是array\_map对应的操作函数，除了我们讲的这个map\_alloc还有map\_free释放map，map\_lookup/update/delete\_elem查找/更新/删除元素等操作。
94. BPF\_MAP\_TYPE\_HASH的map分配的入口函数是htab\_map\_alloc函数。因为hash是用链表来存储的所以bpf\_htab结构是固定的，会优先分配，bpf\_htab记录了hash表的所有信息，右上图中的代码对对象中每个字段的含义都进行了注释。Buckets是指向hash表中的所有桶组成的数组，这里统计链表头buckets的个数，等于和最大条目值最接近的2的n次方。Elem\_size是每一个element的大小，它的大小是htab\_elem+key\_size+value\_size。总占用内存大小是bucket\_size\*max\_entries+elem\_size\*max\_entries+extra\_element\_size，其中extra\_element\_size=elem\_size\*num\_possible\_cpus()。
95. Map的查找过程通过函数map\_lookup\_elem实现，查找就是利用key来找到对应的value值。函数会根据不同类型的map和key值进行查找，最后将结果拷贝到value中，并拷贝给用户空间。
96. 以BPF\_MAP\_TYPE\_ARRAY这种类型为例，会接着调用bpf\_fd\_array\_map\_lookup\_elem和array\_map\_lookup\_elem，分两步走，第一步先将key值赋给index，第二部根据index，找到找到array->value[]数组中的value指针。
97. BPF\_MAP\_TYPE\_HASH类型的map最终调用到htab\_map\_lookup\_elem()，之后有调用\_\_htab\_map\_lookup\_elem。函数分三步走：1.根据key值计算出hash值。2.根据hash值找到链表头bucket。3.在bucket链表中搜索key值相等的htab\_elem如果找不到则返回NULL。
98. BPF map 和程序作为内核资源只能通过文件描述符访问，其背后是内核中的匿名 inode。这带来了很多优点，但同时也有很多缺点。优点包括：用户空间应用能够使用大部分文件描述符相关的 API，传递给 Unix socket 的文 件描述符是透明工作的等等。但同时，文件描述符受限于进程的生命周期，使得 map 共享之类的操作非常笨重。为了解决这个问题，内核实现了一个最小内核空间 BPF 文件系统，BPF map 和 BPF 程序 都可以钉到（pin）这个文件系统内，这个过程称为 object pinning（钉住对象）。相应 地，BPF 系统调用进行了扩展，添加了两个新命令，分别用于钉住（BPF\_OBJ\_PIN）一个 对象和获取（BPF\_OBJ\_GET）一个被钉住的对象（pinned objects）
99. bpf\_obj\_pin会继续调用bpf\_obj\_pin\_user，根据pathname获取路径和根据fd获取bpf\_map/bpf\_prog对象，最后创建文件节点将路径和bpf对象连结起来。
100. bpf\_obj\_get会继续调用bpf\_obj\_get\_user，同样的先根据pathname获取路径，根据路径在对应的inode中找到bpf\_obj\_pin中连结起来的bpf对象的raw指针和type。最后根据对象type在本进程中给bpf对象分配一个fd，这样就可以像访问文件一样访问了。
101. 现在bpf程序和map已经加载到内核当中了，那么什么时候bpf程序才能发挥它的作用呢？这就需要bpf的应用系统把其挂载到适当的钩子上，当钩子所在点的路径被执行，钩子被触发，BPF程序得以执行。目前应用bpf的子系统分为两大类，tracing和filter。Tracing中比较典型的有kprobe可以在内核的任意地方注入代码, 不只包括函数的entry/exit, 也可以是函数内部的某个地址；tracepoint内核在固定位置也预埋了一些tracepoint, 这些tracepoint往往是精心挑选的；perf\_event，perf是在ftrace基础上出来的，ftrace只管抓trace数据并没有分析而perf在trace数据分析方面做出了很多成果，在trace数据采集方面，perf复用了ftrace的所有插桩点，并且加入了采样法(硬件PMU)，perf对每一个event分配一个对应的perf\_event结构。围绕perf\_event来展开，Perf Event是面向事件的观察工具，而针对如何如何利用Perf\_Events进行测量，一共有三种方法：1. 直接利用Counting Event计数，可以利用perf工具直接对发生的次数进行计数。这种方法不会生成perf.data文件，直接利用perf stat命令即可。2. 在指定的时间进行取样，使用这种方法会将Perf\_Events数据写到内核缓存里面，然后再由Perf隔一段时间写入perf.data文件中。最后利用perf report或者perf script读取。但是利用这种方法进行采样，report文件的大小overhead比较高（文件很大）。3. 利用BPF触发用户自己写的程序，这种方法最灵活。但是同时这种方法也比较复杂，需要自己写触发程序。
102. Filter中有sk\_filter、sched\_cls、sched\_act、xdp、cg\_skb。以XDP为例,数据包在 XDP 中的表示形式是 xdp\_buff，这也是传递给 BPF 程序的结构体。XDP hook位于网络驱动的快速路径上，XDP 程序直接从接收缓冲区中将包拿下来，无需执行任何耗时的操作。因此xdp提供了一个内核态、高性能、可编程 BPF 包处理框架。
103. 整个过程是内核中运行着ebpf模块，但这个模块必须通过编译编译成ebpf字节码才能使用。Ebpf与应用程序的交互通过map进行，当数据包进入网卡中XDP hook响应并运行相关ebpf字节码，此时用户态响应pref\_event事件。数据包将有三种去向：1. XDP\_DROP：指示驱动程序丢弃数据包，使用XDP进行丢弃的速度很快，缓冲区只是回收到rx环形队列；2. XDP\_PASS：通过，可能在修改后继续向上层网络协议栈传递；3. XDP\_REDIRECT：使用XDP程序中的XDP\_REDIRECT操作，该程序可以将入口帧重定向到其他启用XDP的netdev。