The BSD Packet Filter
BSD 数据包过滤器
用户级数据包捕获的新架构

Steven McCanne 和 Van Jacobson
劳伦斯伯克利实验室
One Cyclotron Road 伯克利，CA 94720
mccanne@ee.lbl.gov，van@ee.lbl.gov

1992年12月19日

摘要

许多版本的 Unix 都提供了用户级数据包捕获功能，使得使用通用工作站进行网络监控成为可能。由于网络监控器作为用户级进程运行，因此必须跨内核/用户空间保护边界复制数据包。可以通过部署称为数据包过滤器的内核代理来尽量减少这种复制，该代理会尽早丢弃不需要的数据包。原始Unix数据包过滤器是围绕基于堆栈的过滤器评估器设计的，该评估器在当前 RISC CPU 上的性能不佳。BSD 数据包过滤器 (BPF) 使用一种新的基于寄存器的过滤器评估器，其速度比原始设计快 20 倍。BPF 还使用了一种简单的缓冲策略，使其整体性能比在相同硬件上运行的 Sun 的 NIT 快 100 倍。

1 简介

Unix已成为高质量网络的代名词，而当今的 Unix 用户依赖于可靠、响应迅速的网络访问。不幸的是，这种依赖意味着网络故障可能导致无法完成

有用的工作，越来越多的用户和系统管理员发现，他们大部分时间都花在隔离和修复网络问题上。解决问题需要适当的诊断和分析工具，理想情况下，这些工具应该在问题所在的地方（Unix 工作站）可用。为了允许构建此类工具，内核必须包含一些功能，使用户级程序能够访问原始的、未处理的网络流量。[7] 当今的大多数工作站操作系统都包含这样的功能，例如 NIT[10] SunOS、DEC 的 Ultrix 中的 UltrixPacketFilter[2] 和 SGI 的 IRIX 中的 Snoop。这些内核工具源自卡内基梅隆大学和斯坦福大学为将 Xerox Alto“数据包过滤器”适配到 Unix 内核而开展的开创性工作[8]。1980 年完成的卡内基梅隆大学/斯坦福大学数据包过滤器 CSPF 提供了大量所需且广泛使用的工具。然而，在当今的机器上，它的性能及其后代的性能却不尽如人意——完全适合 64KB PDP–11 的设计与 16MB Sparcstation 2 完全不匹配。本文介绍了 BSD 数据包过滤器 BPF，一种用于数据包捕获的新内核架构。BPF 的性能显著优于现有的数据包捕获工具——在相同的硬件和流量组合下，比 Sun 的 NIT 快 10 到 150 倍，比 CSPF 快 1.5 到 20 倍。性能提升是两项架构改进的结果：

BPF使用重新设计的基于寄存器的“过滤机”，该机器可以在当今基于寄存器的 RISC CPU 上有效实现。CSPF 使用基于内存堆栈的过滤机，该机器在 PDP–11 上运行良好，但与内存瓶颈的现代 CPU 不匹配。

BPF使用简单的非共享缓冲区模型，这得益于当今更大的地址空间。该模型对于数据包捕获的“常见情况”非常有效。

在本文中，我们介绍了 BPF 的设计，概述了它如何与系统的其余部分交互，并描述了过滤机制的新方法。最后，我们介绍了 BPF、NIT 和 CSPF 的性能测量，这些测量说明了 BPF 比其他方法表现更好的原因。

2 网络分接头

BPF 有两个主要组件：网络分接头和数据包过滤器。网络分接头从网络设备驱动程序收集数据包副本，并将它们传送到侦听应用程序。过滤器决定是否应接受数据包，如果接受，则将多少数据包复制到侦听应用程序。

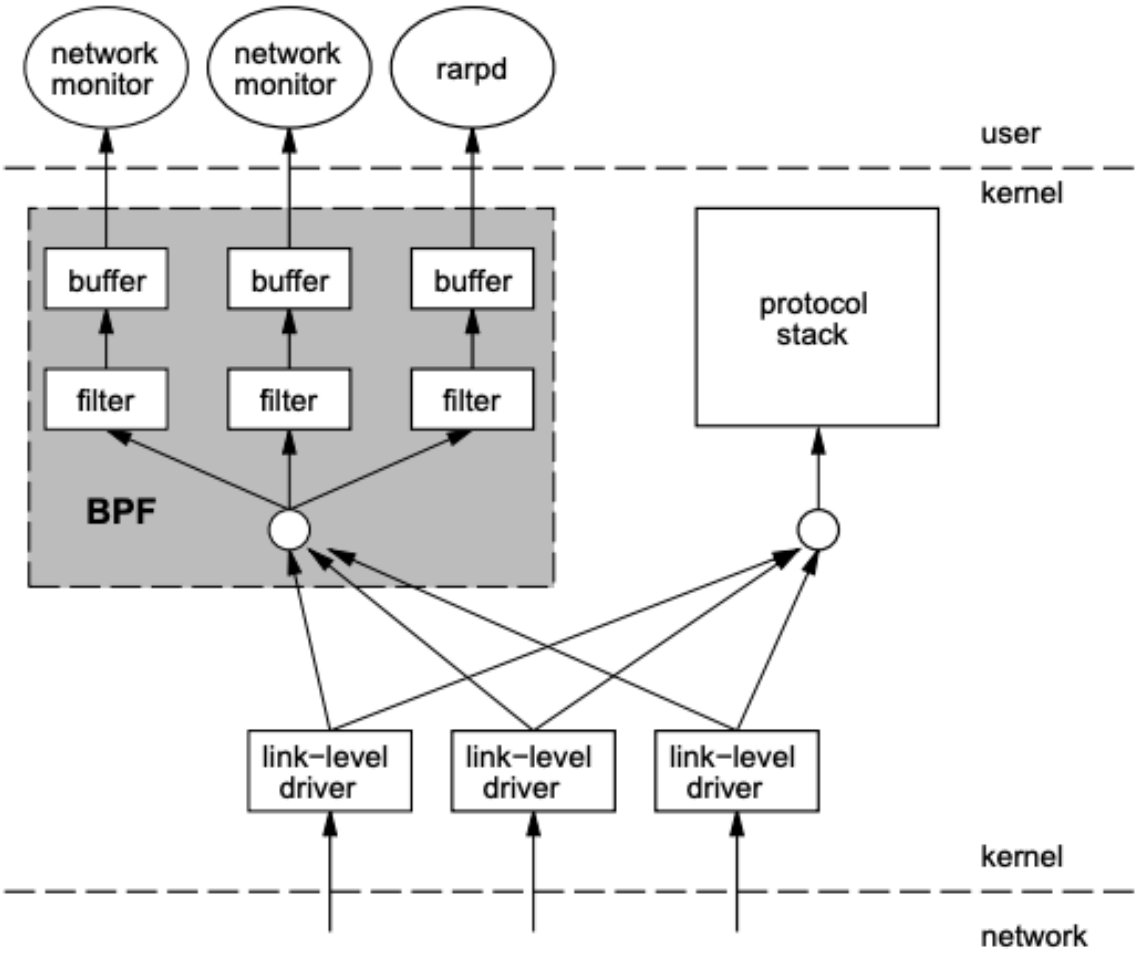


Figure 1: BPF Overview

图 1 说明了 BPF 与系统其余部分的接口。当数据包到达网络接口时，链路级设备驱动程序通常会将其发送到系统协议栈。但是当 BPF 侦听此接口时，驱动程序首先调用 BPF。BPF 将数据包馈送到每个参与进程的过滤器。这个用户定义的过滤器决定是否要接受数据包以及应保存每个数据包的多少字节。对于接受数据包的每个过滤器，BPF 将请求的数据量复制到与该过滤器关联的缓冲区。然后，设备驱动程序重新获得控制权。如果数据包不是发往本地主机的，则驱动程序从中断返回。否则，正常的协议处理将继续进行。

由于进程可能想要查看网络上的每个数据包，而数据包之间的时间间隔可能只有几微秒，因此不可能对每个数据包执行一次读取系统调用，而 BPF 必须从多个数据包中收集数据，并在监控应用程序执行读取时将其作为一个单元返回。为了维护数据包边界，BPF 将从每个数据包中捕获的数据封装到一个标头中，该标头包含时间戳、长度和用于数据对齐的偏移量。

2.1 数据包过滤

由于网络监视器通常只需要一小部分网络流量，因此通过在中断上下文中过滤掉不需要的数据包可以实现显着的性能提升。为了最大限度地减少内存流量（大多数现代工作站的主要瓶颈），应该“就地”过滤数据包（例如，网络接口 DMA 引擎放置它的地方），而不是在过滤之前复制到其他内核缓

冲区。因此，如果数据包不被接受，主机只会引用过滤过程所需的那些字节。

相比之下，SunOS 的 STREAMS NIT 在过滤之前会复制数据包，因此性能会下降。STREAMS 数据包过滤器模块 (nit pf(4M)) 位于数据包捕获模块 (nit if(4M)) 的顶部。收到的每个数据包都会复制到 mbuf，然后传递给 NIT，然后 NIT 会分配一个 STREAMS 消息缓冲区并复制数据包。然后，消息缓冲区会向上游发送到数据包过滤器，数据包过滤器可能会决定丢弃该数据包。因此，每个数据包都会被复制，复制不需要的数据包会浪费许多 CPU 周期。

2.2 Tap 性能测量

在讨论数据包过滤器的细节之前，我们先介绍一些测量方法，比较 BPF 和 SunOS STREAMS 缓冲模型的相对成本。此性能与数据包过滤机制无关。

我们将 BPF 和 NIT 配置到同一个 SunOS 4.1.1 内核中，并在 Sparcstation 2 上进行了测量。

测量结果反映了中断处理期间产生的开销 — 即每个系统将数据包存储到缓冲区所需的时间。对于 BPF，我们只是使用 Sparcstation 的微秒时钟测量了 tap 调用 bpf tap() 的前后时间。对于 NIT，我们测量了 tap 调用 snit intr() 的时间以及将混杂数据包复制到 mbufs 的额外开销。 （混杂数据包是指那些未发送到本地主机的数据包，它们仅因为数据包过滤器正在运行而存

在。）换句话说，我们考虑了 NIT 因未过滤数据包而导致的性能损失。为了获得准确的时间，在检测的代码段期间锁定了中断。

数据集被看作处理时间与数据包长度的直方图。我们绘制了两种配置的每个数据包的平均处理时间与数据包大小的关系图：“接受所有”过滤器和“拒绝所有”过滤器。

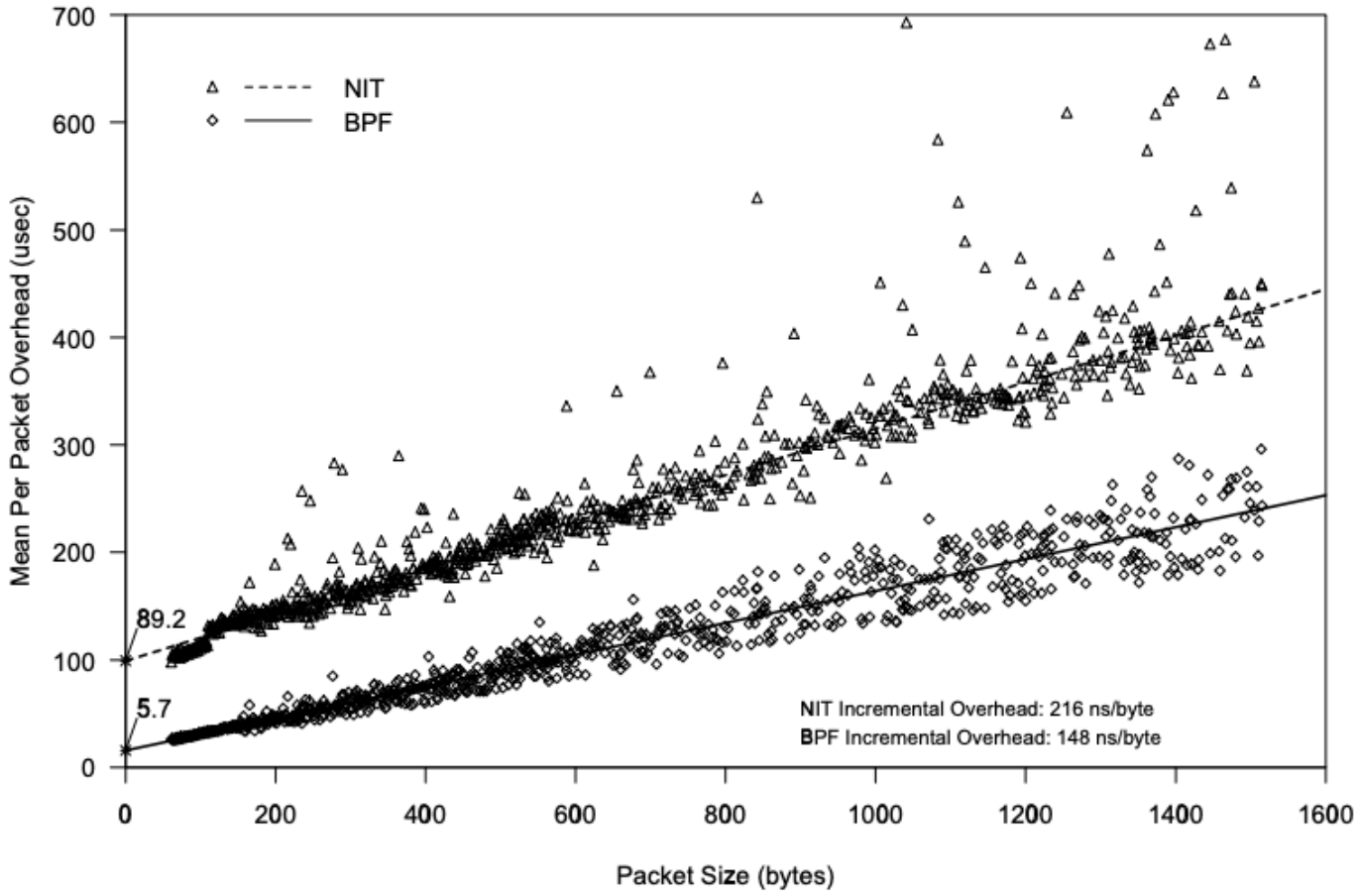


Figure 2: NIT versus BPF: “accept all”

在第一种情况下，STREAMS NIT 缓冲模块 (nit buf(4M)) 被推送到 NIT 流上，其块大小参数设置为 16K 字节。同样，BPF 被配置为使用 16K 缓冲区。通常位于 NIT 接口和 NIT 缓冲模块之间的数据包过滤模块被省略，以实

现“接受所有”语义。在这两种情况下，都没有指定截断限制。该数据如图 2 所示。BPF 和 NIT 都显示线性增长，捕获的数据包大小反映了数据包到过滤器缓冲区副本的成本。然而，BPF 和 NIT 线的不同斜率表明 BPF 以内存速度（148ns/字节）进行复制，而 NIT 运行速度慢 45%（216ns/字节）。 y 轴截距给出了固定的每数据包处理开销：

BPF 调用的开销约为 6 秒，而 NIT 的开销是 BPF 调用的 15 倍，为每数据包 89 秒。这种巨大差异的大部分原因似乎是在非常复杂的 AT&T STREAM I/O 系统下分配和初始化缓冲区的成本。

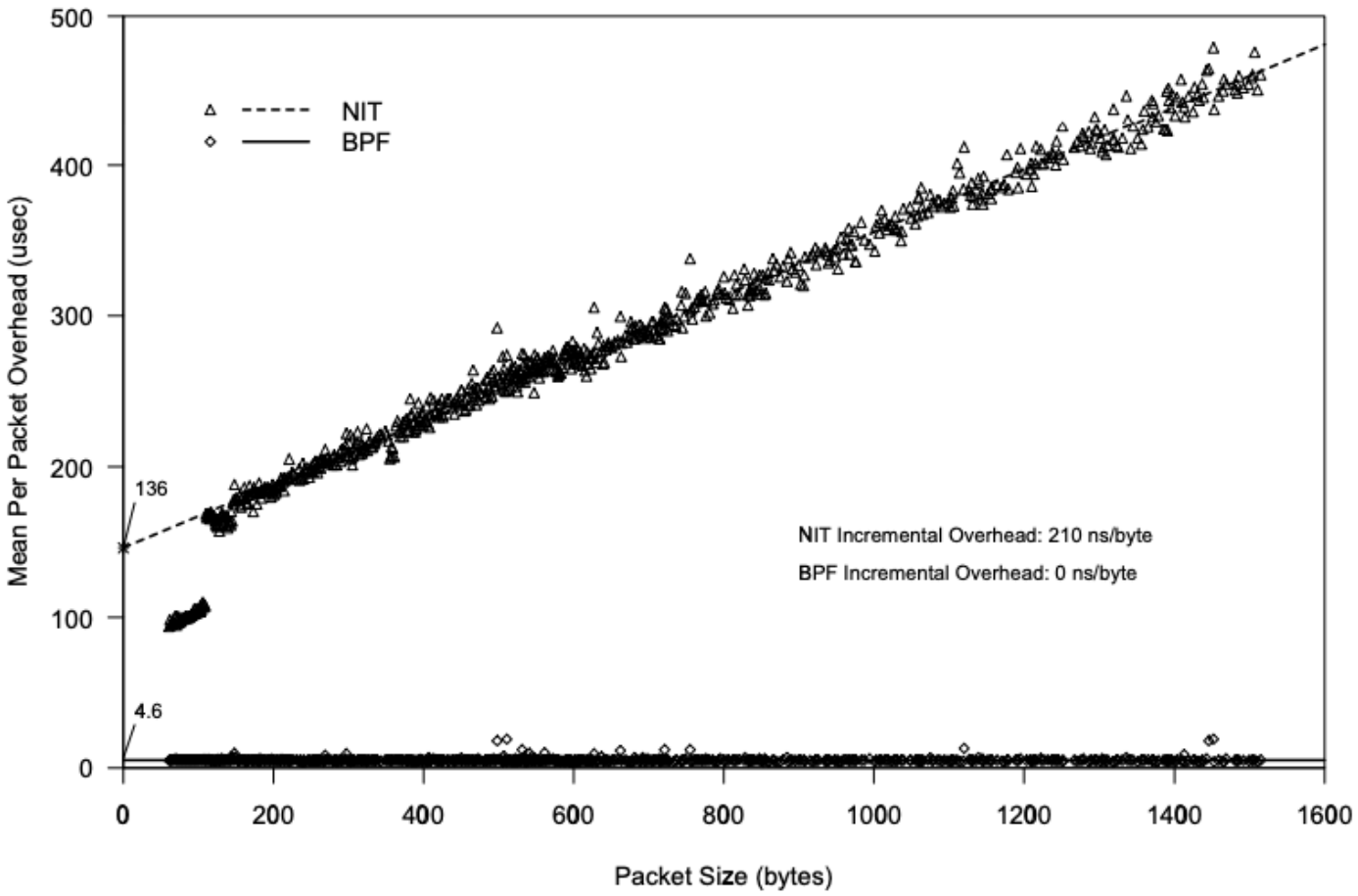


Figure 3: NIT versus BPF: “reject all”

3 过滤模型

假设在设计缓冲模型时谨慎小心，它将是你接受的数据包的主要成本，而数据包过滤器计算将是你拒绝的数据包的主要成本。数据包捕获工具的大多数应用程序拒绝的数据包比它们接受的数据包多得多，因此，数据包过滤器的良好性能对于良好的整体性能至关重要。

数据包过滤器只是数据包上的布尔值函数。如果函数值为真，则内核将为应用程序复制数据包；如果为假，则忽略数据包。

历史上，过滤器抽象有两种方法：布尔表达式树（由 CSPF 使用）和有向无环控制流图或 CFG（首先由 NNStat 使用，并由 BPF 使用）。例如，图 4 说明了这两个模型，其中一个过滤器可识别以太网上的 IP 或 ARP 数据包。在树模型中，每个节点代表布尔运算，而叶子代表数据包字段上的测试谓词。边表示运算符–操作数关系。在 CFG 模型中，每个节点代表数据包字段谓词，而边表示控制传输。如果谓词为真，则遍历右侧分支，如果为假，则遍历左侧分支。有两个终止叶子，分别代表整个过滤器的真和假。

这两种过滤模型在计算上是等效的。也就是说，任何可以用一种模型表达的过滤器都可以用另一种模型表达。然而，在实现上，它们非常不同：树模型自然地映射到堆栈机的代码中，而 CFG 模型自然地映射到寄存器机的代码中。由于大多数现代机器都是基于寄存器的，我们认为 CFG 方法更适合于更高效的实现。

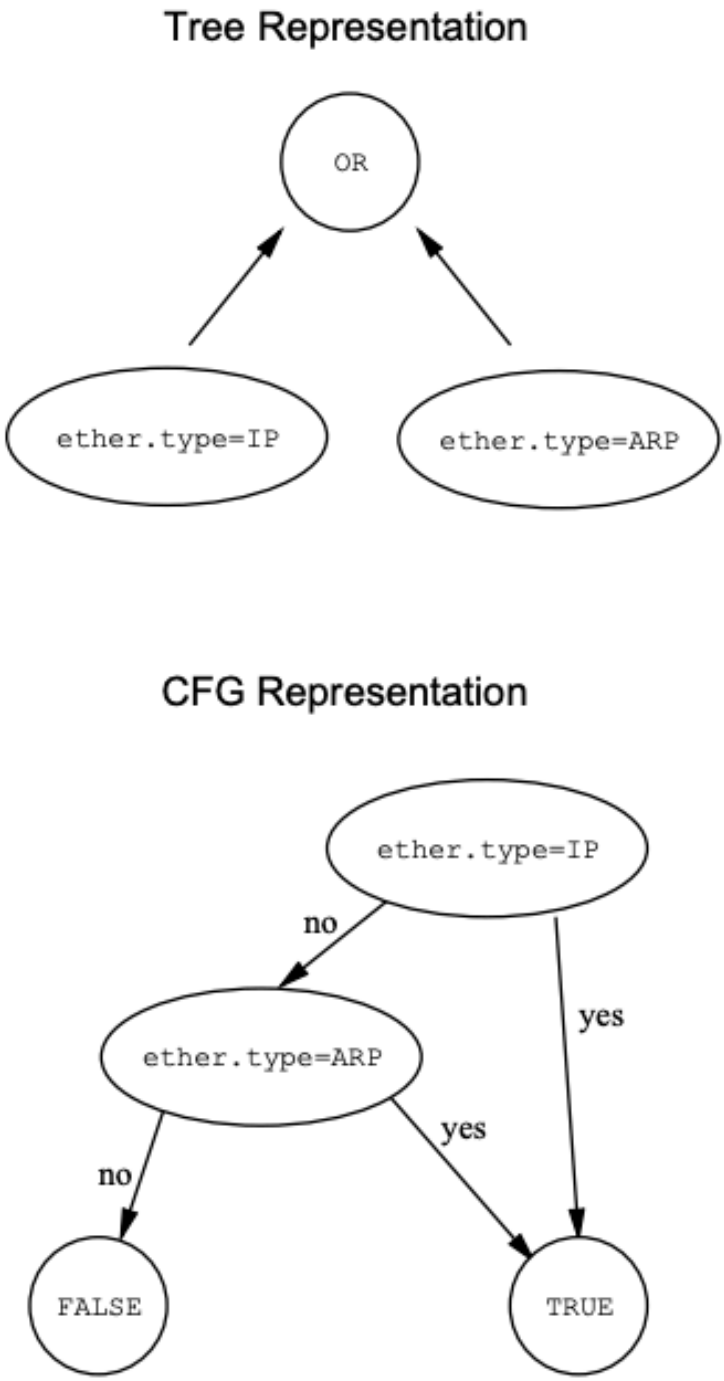


Figure 4: Filter Function Representations.

3.1 CSPF（树）模型

CSPF 过滤引擎基于操作数堆栈。指令要么将常量或数据包数据推送到堆栈上，要么对顶部两个元素执行二进制布尔或按位运算。过滤程序是按顺序执行的指令列表。在评估程序后，如果堆栈顶部具有非零值或堆栈为空，则数据包被接受，否则被拒绝。

表达式树方法有两个实现缺点：

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运算以维护模拟堆栈指针，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架构的主要瓶颈，因此可以使用机器寄存器中的值并避免这种内存流量的过滤模型将更有效率。

2) 树模型通常会进行不必要或多余的计算。

例如，图 4 中的树将计算“ether.type == ARP”的值，即使 IP 的测试为真。虽然可以通过向过滤器机器添加“短路”运算符来缓解此问题，但某些低效率是固有的：由于网络协议的分层设计，必须解析数据包头才能到达连续的封装层。由于表达式树的每个叶子都代表一个数据包字段，独立于其他叶子，因此可能会执行冗余解析来评估整个树。在 CFG 表示中，始终可以重新排序图，以便对任何层最多进行一次解析。

设计人员认识到 CSPF 的另一个问题是无法解析可变长度的数据包头，例如封装在可变长度 IP 头中的 TCP 头。由于 CSPF 指令集不包含间接运算符，因此只能访问固定偏移量的数据包数据。此外，CSPF 模型仅限于单个 16 位数据类型，这导致处理 32 位数据（例如 Internet 地址或 TCP 序列号）的操作数加倍。最后，该设计不允许访问奇数长度数据包的最后一个字节。

尽管 CSPF 模型存在缺点，但它提供了一种新颖的数据包过滤泛化：将伪机器语言解释器放入内核的想法为描述和实现过滤机制提供了一个很好的抽象。而且，由于 CSPF 将数据包视为一个简单的字节数组，因此过滤模型完全独立于协议。（指定过滤器的应用程序负责为底层网络媒体和协议适当地编码过滤器。）

下一节中描述的 BPF 模型试图在解决其局限性和基于堆栈的过滤器机器的性能缺点的同时保持 CSPF 的优势。

3.2 BPF 模型

3.2.1 CFG vs. Trees

BPF使用CFG 过滤器模型，因为它比表达式树模型具有显著的性能优势。虽然树模型可能需要多次冗余地解析数据包，但 CFG 模型允许将解析信息“内置”到流图中。也就是说，数据包解析状态被“记住”在图中，因为您知道必须遍历哪些路径才能到达特定节点，并且一旦评估了子表达式，就不需

要重新计算，因为控制流图总是可以（重新）组织，因此该值仅在遵循原始计算的节点上使用。

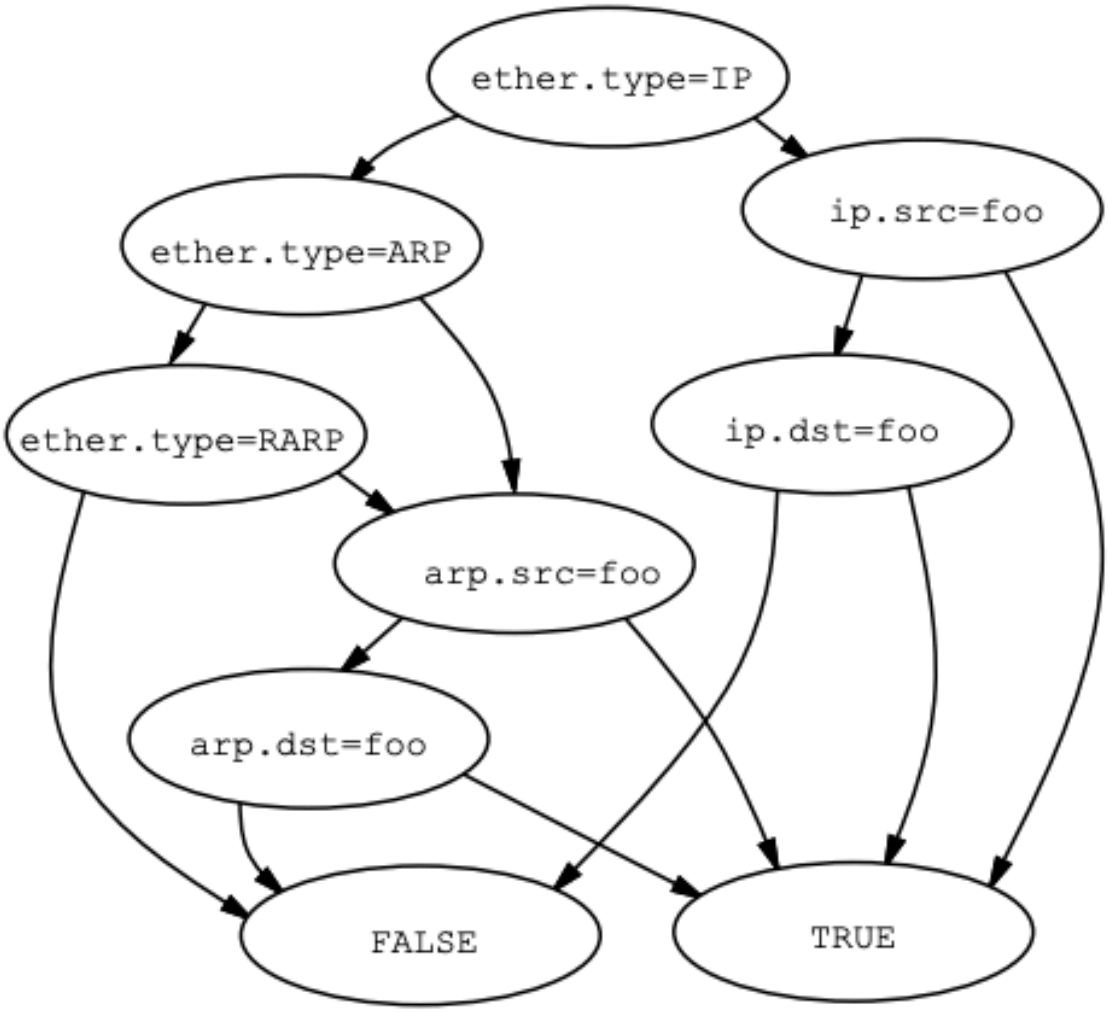


Figure 5: CFG Filter Function for “host foo”.

例如，图 5 显示了 CFG 过滤器函数，它接受所有具有 Internet 地址 foo 的数据包。我们考虑一种情况，其中网络层协议是 IP、ARP 和反向 ARP，它们都包含源和目标 Internet 地址。过滤器应该捕获所有情况。因此，首先测试链路层类型字段。对于 IP 数据包，查询 IP 主机地址字段，而

对于 ARP 数据包，则使用 ARP 地址字段。请注意，一旦我们得知数据包是 IP，我们就不需要检查它可能是 ARP 还是 RARP。在图 6 所示的表达式树模型中，需要七个比较谓词和六个布尔运算来遍历整个树。通过 CFG 的最长路径有五个比较操作，平均比较次数为三次。

3.2.2 过滤器伪机的设计

使用控制流图而不是表达式树作为过滤器伪机的理论基础是实现高效实现的必要步骤，但还不够。即使在利用了 CSPF 和 NNStat 的经验和伪机模型之后，BPF 模型也经历了几代（和几年）的设计和测试。我们相信当前的模型提供了足够的通用性，而不会牺牲性能。它的演变受到以下设计约束的指导：

1. 它必须独立于协议。内核不应该被修改来添加新的协议支持。
2. 它必须是通用的。指令集应该足够丰富，以处理不可预见的用途。
3. 应尽量减少数据包数据引用。
4. 解码指令应由单个 C switch 语句组成。
5. 抽象机器寄存器应驻留在物理寄存器中。

与 CSPF 一样，约束 1 的遵守方式很简单，就是在模型中不提及任何协议。数据包被视为简单的字节数组。

约束 2 意味着我们必须提供一个相当通用的计算模型，具有控制流、足够的 ALU 操作和常规寻址模式。

约束 3 要求我们只能接触一次给定的数据包字。过滤器通常会将给定的数据包字段与一组值进行比较，然后将另一个字段与另一组值进行比较，依此类推。例如，过滤器可能会匹配发往一组机器或一组 TCP 端口的数据包。理想情况下，我们希望将数据包字段缓存在寄存器中，并在该组值之间进行比较。如果字段封装在可变长度的标头中，则我们必须解析外部标头才能到达数据。此外，在对齐受限的机器上，访问多字节数据可能涉及昂贵的逐字节加载。此外，对于 mbuf 中的数据包，字段访问可能涉及遍历 mbuf 链。完成一次这项工作后，我们不应该再做一次。

约束 4 意味着我们将拥有一个高效的指令解码步骤，但它排除了正交寻址模式设计，除非我们愿意适应 switch 案例的组合爆炸。例如，虽然三个地址指令对于实际处理器（其中许多工作是并行完成的）是有意义的，但解释器的顺序执行模型意味着每个地址描述符都必须串行解码。单地址指令格式最小化解码，同时保持足够的通用性。

最后，约束 5 是一个简单的性能考虑因素。与约束 4 一起，它强化了伪机器寄存器集应该很小的概念。

这些约束促使采用累加器机器模型。在此模型下，流程图中的每个节点通过将值计算到累加器中并基于该值进行分支来计算其相应的谓词。图 7 显示了使用 BPF 指令集的图 5 的过滤函数。

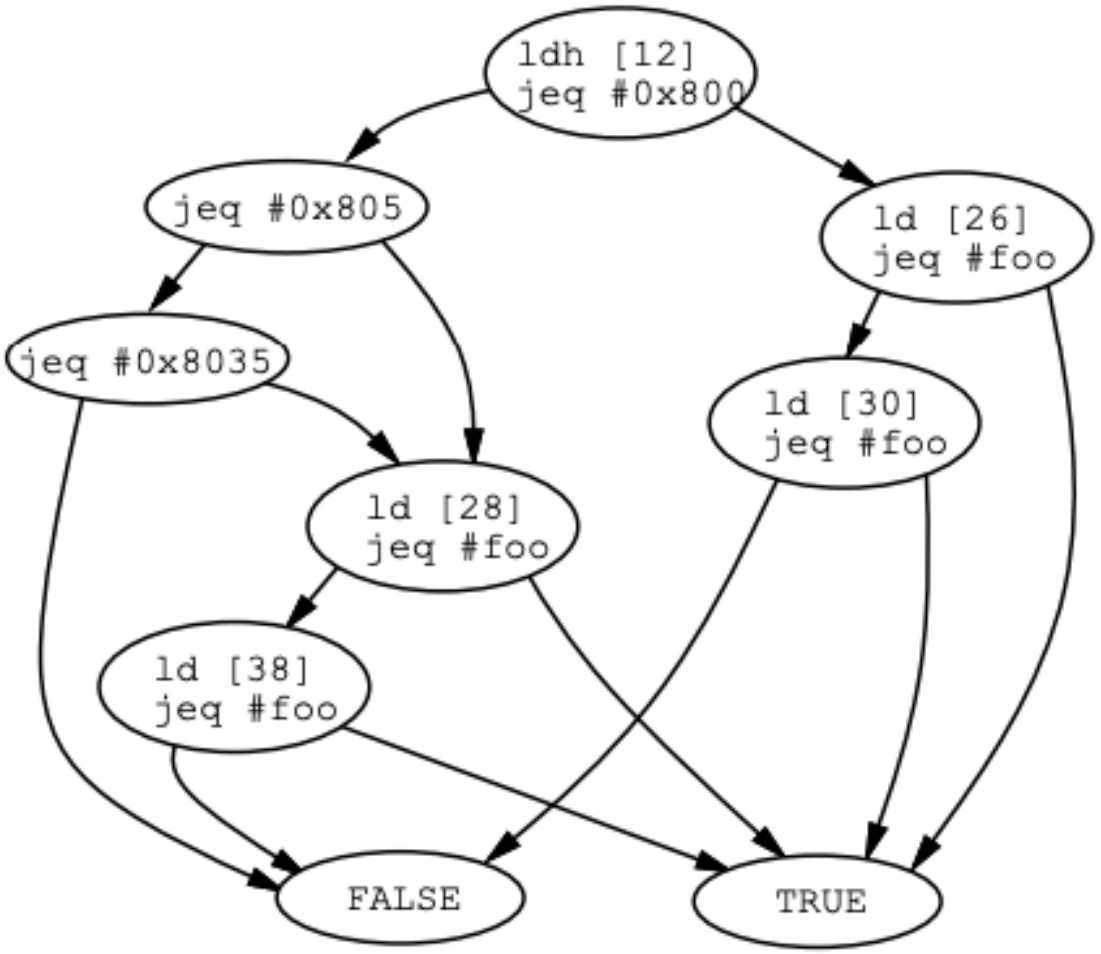


Figure 7: BPF Program for “host foo”.

3.3 BPF 伪机

BPF 机器抽象由一个累加器、一个索引寄存器 (x)、一个临时存储器和一个隐式程序计数器组成。对这些元素的操作可以分为以下几类：

1. LOAD IN STRUCTIONS 将值复制到累加器或索引寄存器中。源可以是立即值、固定偏移处的数据包数据、可变偏移处的数据包数据、数据包长

度或临时存储器。

2. 存储指令将累加器或索引寄存器复制到暂存存储器中。

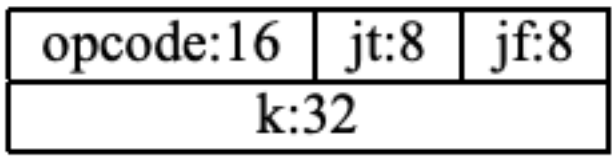
3. ALU 指令使用索引寄存器或常量作为操作数对累加器执行算术或逻辑运算。

4. 分支指令根据常数或 x 寄存器与累加器之间的比较测试改变控制流。

5. 返回指令终止过滤器并指示要保存数据包的哪一部分。如果过滤器返回 0，则数据包将被完全丢弃。

6. 杂项指令包括其他所有内容 — 目前为寄存器传输指令。

固定长度指令格式定义如下：



opcode 字段指示指令类型和寻址模式。jt 和 jf 字段由条件跳转指令使用，是从下一条指令到真和假目标的偏移量。k 字段是用于各种目的的通用字段。

opcodes	*addr modes*				
ldb	[k]			[x+k]	
ldh	[k]			[x+k]	
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k			x	
sub	#k			x	
mul	#k			x	
div	#k			x	
and	#k			x	
or	#k			x	
lsh	#k			x	
rsh	#k			x	
ret	#k			a	
tax					
txa					

Table 1: BPF Instruction Set

表 1 显示了整个 BPF 指令集。我们采用了这种“汇编语法”来说明 BPF 过滤器并用于调试输出。实际编码是用 C 宏定义的，我们在此省略其细节（有关完整详细信息，请参阅 [6]）。标记为 addr modes 的列列出了 opcode 列中列出的每条指令允许的寻址模式。寻址模式的语义列于表 2 中。

加载指令只是将指示的值复制到累加器 (ld、ldh、ldb) 或索引寄存器 (ldx) 中。索引寄存器不能使用数据包寻址模式。相反，必须将数据包值加载到累加器中并通过 tax 传输到索引寄存器。这种情况并不常见，因为索引寄存器主要用于解析可变长度 IP 标头，可以通过 4*([k]&0xf) 寻址模式直接加载。所有值都是 32 位字，但数据包数据可以作为无符号字节 (ldb) 或无符号半字 (ldh) 加载到累加器中。类似地，暂存存储器被寻址为 32 位字数组。指令字段全部按主机字节顺序排列，加载指令将数据包数据从网络顺序转换为主机顺序。对数据包末尾以外的数据的任何引用都会终止过滤器，返回值为零（即数据包被丢弃）。 ALU 操作（加、减等）使用累加器和操作数执行指示的操作，并将结果存储回累加器。除以零将终止过滤器。

跳转指令将累加器中的值与常数进行比较（jset 执行“按位与”——适用于条件位测试）。如果结果为真（或非零），则执行真分支，否则执行假分支。任意比较不太常见，可以通过减去并与 0 进行比较来完成。请注意，没有 jlt、jle 或 jne 操作码，因为这些操作码可以通过反转分支从上面的代码构建。由于跳转偏移量以八位编码，因此最长的跳转是 256 条指令。可以想象比这更长的跳转，因此提供了始终跳转操作码 (jmp)，它使用 32 位操作数字段作为偏移量。

返回指令终止程序并指示要接受数据包的字节数。如果该数量为 0，则数据包将被完全拒绝。实际接受的数量将是数据包长度和过滤器指示的数量中的最小值。

`#k`	the literal value stored in *k*
`#len`	the length of the packet
`M[k]`	the word at offset *k* in the scratch memory store
`[k]`	the byte, halfword, or word at byte offset *k* in the packet
`[x+k]`	the byte, halfword, or word at offset *x*+*k* in the packet
`L`	an offset from the current instruction to L
`#k, Lt, Lf`	the offset to `Lt` if the predicate is true, otherwise the offset to `Lf`
`x`	the index register
`4*([k]&0xf)`	four times the value of the low four bits of the byte at offset *k* in the packet

Table 2: BPF Addressing Modes

3.4 示例

我们现在提供一些示例来说明如何使用 BPF 指令集来表达数据包过滤器。（在以下所有示例中，我们假设链路级标头采用以太网格式。）

此过滤器接受所有 IP 数据包：

```
      ldh     [12]
      jeq     #ETHERTYPE_IP, L1, L2
L1:   ret     #TRUE
L2:   ret     #0
```

```
        ldh     [12]
        jeq     #ETHERTYPE_IP, L1, L4
L1:     ld      [26]
        and     #0xffffff00
        jeq     #0x80037000, L4, L2
L2:     jeq     #0x8003fe00, L4, L3
L3:     ret     #TRUE
L4:     ret     #0
```

第一条指令加载以太网类型字段。我们将其与 IP 类型进行比较。如果比较失败，则返回零并拒绝数据包。如果比较成功，则返回 TRUE 并接受数据包。（TRUE 是某个非零值，表示要保存的字节数。）

下一个过滤器接受所有不是来自两个特定 IP 网络 128.3.112 或 128.3.254 的 IP 数据包。如果以太网类型为 IP，则加载 IP 源地址并屏蔽高 24 位。将此值与两个网络地址进行比较。

3.5 解析数据包头

前面的示例假设感兴趣的数据位于数据包中的固定偏移处。但事实并非如此，例如，TCP 数据包被封装在可变长度的 IP 头中。TCP 头的开头必须根据 IP 头中给出的长度计算出来。

IP 报头长度由 IP 部分中第一个字节（以太网上的字节 14）的低四位给出。此值是字偏移量，必须按四倍缩放才能获得相应的字节偏移量。以下指令将把此偏移量加载到累加器中：

```
        ldb     [14]
        and     #0xf
        lsh     #2
```

一旦计算出 IP 报头长度，就可以使用间接负载访问 TCP 部分中的数据。请注意，有效偏移量有三个组成部分：

IP 报头长度、链路级报头长度和相对于 TCP 报头的数据偏移量。例如，以太网报头为 14 个字节，而 TCP 数据包中的目标端口位于第二个字节。因此，将 IP 报头长度加上 16 即可得出 TCP 目标端口的偏移量。前面的代码段如下所示，经过扩充以根据某个值 N 测试 TCP 目标端口：

```
        ldb     [14]
        and     #0xf
        lsh     #2
        tax
        ldh     [x+16]
        jeq     #N, L1, L2
L1:     ret     #TRUE
L2:     ret     #0
```

由于 IPheader 长度计算是常用操作，因此引入了 4*([k]&0xf) 寻址模式。代入 ldx 指令可将过滤器简化为：

```
        ldx     4*([14]&0xf)
        ldh     [x+16]
        jeq     #N, L1, L2
L1:     ret     #TRUE
L2:     ret     #0
```

但是，上述过滤器仅在我们查看的数据确实是 TCP/IP 标头时才有效。因此，过滤器还必须检查链路层类型是否为 IP，以及 IP 协议类型是否为 TCP。此外，IP 层可能会对 TCP 数据包进行分段，在这种情况下，TCP 标头仅存在于第一个片段中。因此，任何具有非零片段偏移量的数据包都应被拒绝。最终的过滤器如下所示：

```
        ldh     [12]
        jeq     #ETHERPROTO_IP, L1, L5
L1:     ldb     [23]
        jeq     #IPPROTO_TCP, L2, L5
L2:     ldh     [20]
        jset    #0x1fff, L5, L3
L3:     ldx     4*([14]&0xf)
        ldh     [x+16]
        jeq     #N, L4, L5
L4:     ret     #TRUE
L5:     ret     #0
```

3.6 过滤器性能测量

我们使用 iprof [9]（一种指令计数分析器）在内核外分析了 BPF 和 CSPF 过滤模型。为了全面比较这两个模型，在 CSPF 中添加了一个间接运算符，以便它可以解析 IP 标头。这个变化很小，不会对原始过滤性能产生不利影响。

测试是在从繁忙的加州大学伯克利分校校园网络收集的大型数据包跟踪文件上运行的。图 8 显示了四个相当典型的过滤器的结果。

过滤器 1 很简单。它测试数据包中的一个 16 位字是否是给定值。这两个模型相当具有可比性，BPF 快了大约 50%。

过滤器 2 查找特定的 IP 主机（源或目标），并显示出更大的差异——性能差距为 240%。这里更大的差异主要是因为 CSPF 只对 16 位字进行操作，并且需要两个比较操作来确定 32 位 Internet 地址是否相等。

过滤器 3 是数据包解析的一个示例（需要定位 TCP 目标端口字段），并说明了更大的性能差距。BPF 过滤器解析数据包一次，将端口字段加载到累加器中，然后简单地对感兴趣的端口进行比较级联。CSPF 过滤器必须为每个要测试的端口重新进行解析并重新定位 TCP 标头。最后，过滤器 5 演示了 CSPF 对类似于图 6 和图 5 中描述的过滤器所做的不必要计算的影响。

4 应用

BPF 已经问世两年了，并已在多个应用中投入使用。最广泛使用的就是 tcpdump [4]，它是一种网络监控和数据采集工具。Tcpdump 执行三个主要任务：过滤器转换、数据包采集和数据包显示。这里感兴趣的是过滤器转换机制。过滤器使用用户友好的高级描述语言来指定。Tcpdump 有一个内置编译器（和优化器），可将高级过滤器转换为 BPF 程序。当然，这个转换过程对用户来说是透明的。

Arpwatch [5] 是一个被动监控程序，用于跟踪以太网到 IP 地址的映射。当建立新的映射或发现异常行为时，它会通过电子邮件通知系统管理

员。一个常见的管理麻烦是多个物理主机使用一个 IP 地址，arpwatch 会尽职尽责地检测并报告这种情况。

BPF 的一个非常不同的应用是将其纳入 Icon 编程语言的一个变体 [3]。Icon 解释器中内置了两种新的数据类型，即数据包和数据包生成器。数据包作为第一类记录对象出现，允许方便地使用“点运算符”访问数据包头。数据包生成器可以直接从网络外实例化，也可以从先前收集的跟踪数据文件中实例化。Icon 是一种解释型、动态类型语言，具有高级字符串扫描原语和丰富的数据结构。借助 BPF 扩展，它非常适合快速构建网络分析工具的原型。

Netload 和 histo 是两个网络可视化工具，它们在 X 显示器上生成实时网络统计信息。Netload 使用 tcpdump 样式过滤器规范实时绘制利用率数据图表。Histo 生成带时间戳的多媒体网络数据包的动态到达间隔时间直方图。

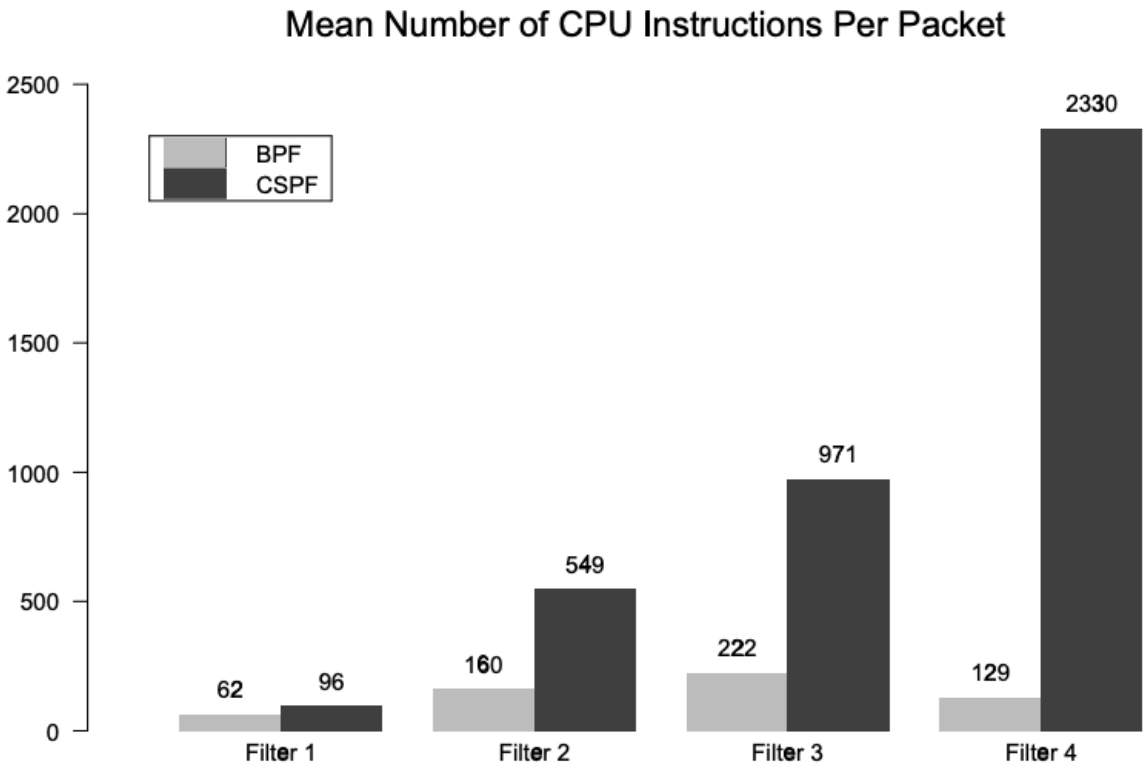
反向 ARP 守护进程使用 BPF 接口直接读取和写入反向 ARP 请求并回复本地网络。（我们开发此程序是为了让我们能够在我们的 SunOS4 系统中用 BPF 完全取代 NIT。每个基于 Sun NIT 的应用程序（etherfind、traffic 和 rarpd）现在都有一个 BPF 模拟。）

最后，NNStat[1] 和 nfswatch 的最新版本可以配置为在 BPF 上运行（除了在 NIT 上运行）。

5 结论

BPF 已被证明是一种高效、可扩展且可移植的网络监控接口。我们的比较研究表明，它在缓冲区管理方面优于 NIT，在过滤机制方面优于 CSPF。其可编程伪机器模型已展示出出色的通用性和可扩展性（所有特定协议的知识

都从内核中分离出来）。最后，该系统是可移植的，可在大多数 BSD 和 BSD 衍生系统上运行，并且可以与各种数据链路层交互。



Filter 1	IP packets
Filter 2	IP packets with src or dst “horse”
Filter 3	TCP packets with src or dst port of finger, domain, login, or shell
Filter 4	IP, ARP or RARP packets between hosts “horse” and “gauguin”

Figure 8: BPF/CSPF Filter Performance

6 可用性

BPF 可通过匿名 ftp 从主机 ftp.ee.lbl.gov 获取，作为 tcpdump 发行版的一部分，目前位于文件 tcpdump–2.2.1.tar.Z 中。最终我们计划将 BPF 纳

入其自己的发行版，因此将来请查找 bpf–*.tar.Z。Arpwatch 和 netload 也可从此站点获取。

7 致谢

如果没有 Jeffrey Mogul 的鼓励，本文将永远不会发表。Jeff 将 tcpdump 移植到 Ultrix 并添加了 little–endian 支持，从而发现了数十个字节排序错误。他还通过迫使我们考虑解析 DECNET 数据包头的艰巨任务，启发了 jset 指令。Mike Karels 建议过滤器不仅应决定是否接受数据包，还应决定保留多少数据包。CraigLeres 是 BPF/tcpdump 的第一个主要用户，负责查找和修复两者中的许多错误。Chris Torek 帮助进行了数据包处理性能测量，并提供了有关各种 BSD 特性的见解。最后，我们感谢互联网上众多 BPF/tcpdump 用户和扩展者，感谢他们的建议、错误修复、源代码和许多问题，这些年来，它们极大地拓宽了我们对网络世界和 BPF 在其中的地位的视野。

最后，我们要感谢 Vern Paxson、Craig Leres、Jeff Mogul、Sugih Jamin 和审稿人对本文草稿的有益评论。

参考文献

[1] BRADEN, R. T. 用于数据包监控和统计的伪机器。在 SIGCOMM ’88 会议论文集（斯坦福，加利福尼亚州，1988 年 8 月），ACM。

[2] DIG ITAL EQUIPMENT CORPORATION packetfilter(4)，Ultrix V4.1 手册。

[3] GRISWOLD, R. E.，和 GRISWOLD, M. T. Icon 编程语言。Prentice Hall, Inc.，新泽西州恩格尔伍德克利夫斯，1983 年。

[4] JACOBSON，V.，LERES，C.，和 MCCANNE，S. Tcpdump 手册页。劳伦斯伯克利实验室，加利福尼亚州伯克利，1989 年 6 月。

[5] LERES，C. Arpwatch 手册页。劳伦斯伯克利实验室，加州伯克利，1992 年 9 月。

[6] MCCANNE，S。BPF 手册页。劳伦斯伯克利实验室，加州伯克利，1991 年 5 月。

[7] MOG UL，J. C. 高效使用工作站进行局域网被动监控。在 SIGCOMM '90 论文集（宾夕法尼亚州费城，1990 年 9 月），ACM。

[8] MOG UL，J. C.、RASHID，R. F. 和 ACCETTA，M. J. 数据包过滤器：一种有效的用户级网络代码机制。在 第 11 届操作系统原理研讨会论文集（德克萨斯州奥斯汀，1987 年 11 月），ACM，第 39–51 页。

[9] RICE，S. P. iprof 源代码，1991 年 5 月。布朗大学。

[10] SUN MICROSYSTEMS INC. NIT(4P)；SunOS 4.1.1 参考手册。加利福尼亚州山景城，1990 年 10 月。部件号：800–5480–10。