Yaksha-Prashna：理解扩展伯克利包过滤器字节码的网络功能行为

Animesh Singh 印度理工学院海得拉巴分校

K Shiv Kumar 印度理工学院海得拉巴分校

S. VenkataKeerthy 印度理工学院海得拉巴

Pragna Mamidipaka 卡内基梅隆大学

R V B R N Aaseesh 印度理工学院海得拉巴分校

Sayandeep Sen IBM印度研究院

Palanivel Kodeswaran 沃尔玛全球技术实验室

Theophilus A. Benson 卡内基梅隆大学

Ramakrishna Upadrasta 印度理工学院海得拉巴

Praveen Tammana 印度理工学院海得拉巴分校

arXiv:2602.11232v1 [cs.CR] 11 Feb 2026

摘要

许多云基础设施组织越来越依赖第三方基于 eBPF 的网络功能来实现安全、可观测性和负载均衡等应用场景，这样就不必让每个组织都配备一支由高技能 eBPF 专家组成的团队。然而，第三方（例如 F5、Palo Alto）提供给云运营者的网络功能采用字节码格式，运营者几乎无法了解其功能正确性及其与链中其他网络功能的交互情况。此外，eBPF 开发者希望在不向操作员披露源代码的情况下，为其开发的网络功能提供功能正确性证明。

我们设计了 Yaksha-Prashna，这是一个允许运维人员/开发者断言并查询字节码是否符合其规范，以及其对其他字节码的依赖关系的系统。我们的工作构建了领域专用模型，使我们能够采用可扩展的程序分析来提取并建模 eBPF 程序。借助 Yaksha-Prashna语言，我们针对标准和非标准的基于 eBPF 的网络功能表达了 24 项属性，速度比最先进的研究工作快 200 至 1000 倍。

1 引言

如今，许多大型知名组织的云基础设施 [17, 34, 41, 65] 依赖基于扩展伯克利包过滤器的网络功能（网络功能），以提供关键的管理功能（例如安全 [22], 可观测性 [18], 和负载均衡器 [64]）。这些扩展伯克利包过滤器网络功能曾导致多起引人注目的服务中断，例如 Datadog 的五百万美元服务中断 [37, 69] ，以及 Meta 的 NetEdit 中提及的服务中断 [16]。扩展伯克利包过滤器凭借其可编程性、高性能和安全性保证而得到广泛采用，但扩展伯克利包过滤器开发需要一支技术精湛的专家团队 [13], ，这迫使许多组织依赖第三方提供的扩展伯克利包过滤器程序（例如 F5 的 DoS 防护 [7], Palo Alto 的防火墙 [10], Datadog [29], Cilium [18], 和 Meta 的负载均衡器（Katran） [64]）。由于此类第三方网络功能近期导致了大规模服务中断，人们越来越担忧这些扩展伯克利包过滤器程序的功能正确性 [47, 55, 61]。

不幸的是，理解功能正确性及其与其他网络功能的交互并非易事，因为它既需要理解因内核而异的程序行为 [16], ，也需要分析包含许多 BPF 特定结构的程序代码，而且 eBPF 语言及程序的文档质量较差。即便是高级开发者也感到困难，因为理解网络功能行为不仅需要了解该网络功能与并行部署的其他网络功能之间的交互，还需要了解其与内核的交互。更重要的是，要检查网络功能与其规范的符合性，关键在于了解网络上下文，例如对数据包内容的读/写/复制操作、所处理的协议以及所调用的 eBPF 映射。字节码的网络上下文还有助于理解其与其他网络功能的交互（例如，在数据包头部上先写后读）。因此，尽管理解基于扩展伯克利包过滤器的功能能够让运维人员有信心在部署到生产网络之前进行调试或避免服务中断 [24, 36],，但他们缺乏此类知识，可能不得不等待更长时间 [31] ，以获得第三方 eBPF 程序开发者的支持。

在本文中，我们采用经典诊断方法来解决这一问题：我们旨在通过引入一种查询语言，隐藏与内核版本、eBPF 规范、程序设计的详细知识及交互相关的细微差异。该查询语言使网络运营商和开发者能够根据基于 *eBPF* 的网络功能的字节码对其进行断言和查询，并理解其行为。我们的查询语言支持广泛的

通过简单的语言结构（例如前置/后置条件 [42]）和语言扩展（例如编写断言/契约 [80]）进行分析。为了支持这种查询语言，我们引入了一个基于程序分析（例如程序验证 [33, 44] 和抽象解释 [27, 28]）的框架，用于构建能够推断程序行为的模型。我们的工作不同于以往的诊断研究，因为我们需要处理 eBPF 特有的结构，例如辅助函数和映射，这使得 eBPF 字节码与传统程序有很大不同。为此，我们需要应对两个全新的挑战：以 eBPF 为中心的静态分析模型，以及支持用户交互的以 eBPF 为中心的查询系统。

用于分析的行为模型。 现有的网络功能验证工作依赖源代码的形式化模型，以验证由 C 和 P4 等高级语言实现的网络功能的属性。例如，eBPF-SE [47] 等工具会为 eBPF 程序建模，并利用 Klee [23] 等通用符号执行引擎分析这些模型。然而，由于缺乏针对 eBPF 字节码的行为模型，此类方法在 eBPF 字节码上的适用性有限。另一方面，也有针对字节码运行的 eBPF 验证器 [40, 82], 因此无需源代码；但它们主要使用静态分析 [82], 、抽象解释 [40]和符号执行 [70]技术，对语义属性建模，以验证程序终止、内存安全和资源有界性。这些验证器并非为从字节码中提取网络上下文而设计。

从字节码中提取网络上下文（例如，处理的协议）比标准源代码分析面临两项关键挑战：首先，扩展伯克利包过滤器字节码指令处于类似汇编语言的低级层次。每条高级语句都可能转换为多条字节码指令，因此难以解读和理解这些指令与执行流中其他指令之间的关系。其次，由于寄存器和栈内存数量有限，大型且复杂的扩展伯克利包过滤器程序会重复使用寄存器和内存位置，导致频繁重新定义，进一步增加了分析难度。

为应对这些挑战，我们设计了**Yaksha-Prashna 分析器**，基于经典的数据流分析 [43],并由控制流分析提供指导（§4.1），对字节码进行分析。分析器通过仔细跟踪寄存器和内存位置之间的数据流，捕获每个程序点的细粒度信息（§4.2）。我们定义了数据流规则，以有效提取扩展伯克利包过滤器字节码的网络上下文（§4.3）。值得注意的是，扩展伯克利包过滤器的特性，例如有限的状态空间、没有循环以及固定数量的寄存器等，使我们的分析器兼具高效性和可扩展性。

行为领域特定语言（DSL）。另一项要求是，用户能够通过对 eBPF 字节码提出行为查询来进行交互。现有的网络功能验证工具采用标准的程序验证方法，并主要侧重于通过添加断言来扩展（源）语言 [12, 38, 61, 67]。这些扩展后的断言语言适用范围有限；断言适合验证某些属性是否成立（例如：“网络功能是否向映射写入数据？”），但无法检索网络上下文（例如：“网络功能向映射写入了哪个数据包头字段？”）。此外，每条断言通常都需要单独的分析遍次；因此，每增加一条断言，都必须从头分析程序，以验证这些额外属性，从而产生显著的计算开销（包括分析时间和内存消耗）。

因此，我们需要一种自定义查询语言，将网络上下文的检索与对所提取网络上下文的查询分离开来。设计这种查询语言面临两大挑战：首先，该语言应能有效地将网络功能操作员/开发者的高层意图映射到字节码中的底层构造，同时屏蔽字节码中的底层细节。其次，该语言应能表达复杂查询，用于检查符合性以及字节码与其他网络功能之间的交互。

为应对这些挑战，我们设计了**Yaksha-Prashna语言**，这是一种简单而富有表现力的 DSL，用于抽象分析中的复杂性。Yaksha-Prashna 让网络运营商/开发者能够表达有关网络功能行为的高层查询，而无需涉及基本块标识符、寄存器映射等底层字节码细节。Yaksha-Prashna 支持用户针对单个字节码以及多个字节码之间的跨函数交互表达两种类型的查询：断言和检索。它具有可组合性，允许用户组合简单查询来构造复杂的现实场景查询；同时易于采用，借助 Prolog 实现基于逻辑的查询。

为了确保系统的可扩展性，我们将字节码分析与查询阶段解耦，使字节码只需分析一次，提取出的网络上下文即可用于回答后续任意查询。此方法避免了重复的分析传递，降低了开销，并支持高效的大规模部署。

本文的主要贡献如下：

- 我们提出了 Yaksha-Prashna语言，这是一种面向网络功能操作员和网络功能开发者的领域专用查询语言，可用于声明和检索 eBPF 字节码的网络上下文，并抽象低级字节码中指令之间复杂的依赖关系（第5节）。
- 我们构建了一个由控制流分析指导的形式化数据流分析模型，用于分析扩展伯克利包过滤器字节码，并设计了 Yaksha-Prashna 分析器来提取网络上下文（第4节）。
- 我们介绍 Yaksha-Prashna（第3节），这是一个集成了分析器（第4节）和查询语言（第5节）的综合系统。该系统具有可扩展性，因为它将字节码分析与查询执行阶段解耦。

• 我们实现了 Yaksha-Prashna 原型，并通过对来自 Katran [64] 和 Suricata [35]等项目的 16 个 XDP 程序执行查询进行了测试。

分析器在 300 ms 中提取包含 4K 条路径的字节码网络上下文，同时占用的内存不到 50 MB。Yaksha-Prashna 的查询系统在不到 50 μs 的时间内，对 16 个 XDP 程序执行 15 项查询，同时占用的内存不到 15 MB。

• 我们根据语言的表达能力和验证时间，将 Yaksha-Prashna 与最先进的工具 Klint [70] 和 DRACO [61]，进行了比较。Yaksha-Prashna 能够表达用于验证的属性，同时将验证时间缩短 200-1000×。

2 背景与动机

2.1 第三方 eBPF 网络功能

许多组织依赖第三方 eBPF 程序。虽然其中许多是开源的（例如，Data-dog 的 NPM、Cilium 的 CNI、Meta 的 Katran），但越来越多的可观测性和安全程序是专有的，其字节码会直接分发给运维人员 [9, 10]，或通过 L3AF 等市场进行分发 [55]。一些闭源且专有的基于 eBPF 的网络功能包括 F5 Networks 的 NGINX App protect DoS [7]，以及 Palo Alto 的 CN 系列防火墙 [10]。我们认为，这与虚拟网络功能（VNF）的市场模式一致，例如 equinox network edge [6]，它提供来自 Cisco、Juniper、Palo Alto 和 Fortinet 等供应商的 VNF。

为什么理解字节码很重要？ 通常，基于扩展伯克利包过滤器的网络功能使用高级语言开发 [32, 46,51, 63, 74, 79]，并通过 Clang/LLVM 工具链编译为字节码 [56]。随后，借助系统调用将字节码加载到宿主机内核中 [76]。在此过程中，扩展伯克利包过滤器验证器会对字节码进行静态检查，以确保字节码在内核中安全运行。然而，验证器不会对字节码的功能正确性进行建模或推理。更具体地说，第三方提供的基于扩展伯克利包过滤器的网络功能可能没有错误，但也可能无法完全覆盖目标行为 [61],，也就是无法保证功能正确性。这可能由多种原因造成，例如缺少语句（如路由器中缺少 TTL 递减 [81]）、出现意料之外的数据包转发行为（如路由器转发 TTL 为 = 0的数据包），或偏离标准网络功能行为（如防火墙意外放行未知流量）等。此外，网络功能与系统中其他网络功能的交互也可能导致意外行为。如果这些程序来自 eBPF 市场 [55],，则可能包含未知或恶意代码 [61]。因此，在将其部署到网络之前，有必要了解字节码的数据包处理行为及其与其他网络功能的交互。

Yaksha-Prashna 的目标用户。网络运营商和网络开发者是该工具的两类主要用户。**（1）网络运营商：**文档和

```
Query                ::=  Q-谓词_list ';'
Q − Predicate_list   ::=  Q-谓词 [操作员 Q-谓词]*
Q − Predicate        ::=  '!'('Q-谓词')'
                          | 字段谓词'(' nf_id ',' 字段参数')'
                          | pkt_act_pred'(' nf_id ',' 钩子参数 ',' 对列表')'
                          | 映射操作谓词'(nf_id',''映射标识',''字段参数')'
                          | 相关映射谓词'(nf_id',''映射列表')'
                          | 顺序谓词'(' nf_id ',' 变量')'
                          | 协议谓词'(' nf_id ',''字段参数 ',''值')'
                          | helper_pred'(' nf_id ',' 辅助函数')'
operator             ::=  ',' |';'
list_of_pairs        ::=  '['对 [操作员 对]*']'
list_of_maps         ::=  '['('映射标识 ',' 映射标识 ')''']'
pair                 ::=  '('字段参数 ',' 值 ')'
field_pred           ::=  更新字段 |读取字段
pkt_act_pred         ::=  丢弃 | 传递 | 中止 | 重定向 | 发送 | 全部
map_operation_pred   ::=  映射查找 |映射写入
correlated_maps_pred ::=  相关映射
order_pred           ::=  后继网络功能 |前驱NF
protocol_pred        ::=  访问协议
helper_pred          ::=  调用辅助函数
fld_arg              ::=  头部字段 | 缓冲区字段 | 变量 | '*'
hook_arg             ::=  XDP | TC | 变量 | ...                    //钩子点
header_field         ::=  以太网类型 | 以太网目的地址 | 以太网源地址 | ...
                          | IPv4 源地址 | IPv4 目的地址 | ...
                          | ...                                   //标准头字段
buffer_field         ::=  xdp_md.data | xdp_md 数据结束位置 | ...
                          | sk_buff 标记 | ...
                          | ...                                   //内核缓冲区字段
helpers              ::=  BPF 映射查找元素
                          | bpf_map_update_elem
                          | bpf_probe_write_user| 变量 | ...
                          | ...                                   //eBPF辅助函数
nf_id                ::=  字符串 |变量 |'*'
map_id               ::=  字符串 | 变量 | '*'
value                ::=  常量 |变量 |'*'
var                  ::=  [A-Z_][A-Za-z0-9_]*                    //变量
const                ::=  !常量 | [0-9]+ | 字符串 | IP 地址
```

图1. Yaksha-Prashna语言的语法。

针对封闭字节码的规范几乎无法揭示其内部运作方式 [70]。网络功能的规范说明了它能做什么，却没有规定应如何实现。如果网络功能符合相应规范，即使其源代码访问受限，运维人员也能放心部署。此外，运维人员可能希望验证，集成新的网络功能不会导致现有网络功能链中断或产生意外交互。即便提供了源代码（例如 Cilium），要理解此类行为也需要耗费大量时间和精力。Yaksha-Prashna 能帮助运维人员验证网络功能的功能正确性，从而确保其安全部署到现有网络功能链中。

（2）网络开发者：开发者希望在不向运维人员披露源代码的情况下，确保自己开发的网络功能具备功能正确性。此外，字节码符合规范使开发者能够使用任意语言和工具链，尤其是处于实验阶段的语言和工具链，从而确保编译后的二进制文件符合目标行为 [70]。Yaksha-Prashna 能帮助开发者实现这些目标。

2.2 动机用例

一个实用的验证系统应满足用户的三个要求[81]，使其能够：(1) 轻松表达正确的网络功能行为（规范）；(2) 在合理的时间内，以较低的内存开销检查网络功能/NF 链的符合性；(3) 推理分析规范违规的根本原因。Yaksha-Prashna语言（图1）

该语言旨在满足这三项需求，使网络功能操作员/网络功能开发人员能够使用断言验证字节码，并使用检索查询了解网络功能行为。语言的详情见第 §5 节。

我们将 Yaksha-Prashna 的应用场景分为三类。**类别 1：单个网络功能符合规范**。假设用户（网络功能操作员/网络功能开发人员）想要验证某个网络功能实现（字节码）是否符合其规范。例如，有状态防火墙规范规定：（1）不得修改数据包内容；（2）仅允许 TCP 和 UDP 流量，并丢弃其他流量（例如，ICMP [61]）。清单 1 展示了一个不符合规范的有状态防火墙代码片段；它更新了 TCP 流量的源端口（第 10 行），并处理 ICMP 流量（第 6 行）。他们必须知道，该 eBPF 程序生成的字节码不符合规范。为此，他们可以编写以下断言，检查网络功能字节码是否符合规范。

下面的断言断定，防火墙字节码（清单 1）不会修改（更新）数据包内容。

A1： !更新字段(xdp_fw, *)。

它会获取字节码更新的数据包字段列表，并在列表非空时返回 false。参数 ∗ 表示这是一个断言。由于防火墙会修改 TCP 源端口，因此该断言会失败。

下面的断言检查 XDP 挂钩点处的防火墙字节码是否会阻止 ICMP 流量。

A2： 传递(xdp_fw, xdp, [(变量, 变量)]), !accessesProtocol(xdp_fw, “ipv4.proto”, “icmp”)。

*passes*以（*var*， *var*）为参数的构造会返回所有会传递该数据包的程序路径的网络上下文。接下来，它使用 ! *accessesProtocol* 构造检查这些路径是否不应访问 ICMP 协议头。根据清单 1 生成的字节码未能通过断言，因为它会处理 ICMP 流量（第 6 行）并传递该数据包。**类别 2：非预期的网络功能交互**。多个扩展伯克利包过滤器程序附加到同一个挂钩点（例如，XDP、TC）。这使得开发模块化程序来构建复杂系统成为可能。不过，如果链中的另一个程序依赖某个程序更新的状态，那么对该状态的更新可能会导致意外行为。具体而言，操作员可能希望了解的依赖关系包括写后读（RAW）、读后写（WAR）和写后写（WAW）[61]。为进一步说明这一点，请考虑一个实践中观察到的、类似于读后写依赖关系的问题。Cilium 与 AWS 弹性网络接口 [24, 36] 之间的不良交互，导致部分客户的流量被非预期地丢弃。清单 2 中的代码片段展示了这一交互。这里，Cilium 的 *identity_manager*（NF1）会将安全 ID *id_prefix* 存储在 *sk_buff->mark* 字段中，以表示来自特定集群中特定 Pod 的流量。

```
1 int xdp_fw(struct xdp_md *ctx) { 2 ... 3 switch(ip ->protocol){ 4
```

清单 1. 防火墙代码片段

```
1 ********** Cilium 网络功能（NF1）********** 2 int identity_
```

清单 2。 AWS 的 NF2 在 Cilium 的 NF3 读取前发生意外写入，导致流量丢弃。

沿着链路往下，另一个来自 Cilium 的网络功能 *enforce_policy* (NF3) 会在 *cluster_id* in *sk_buff->mark* 中进行查找（第 26-32 行），并允许流量通过。然而，刚引入数据包路径、位于 AWS 弹性网络接口上的 *pbr_routing* (NF2) 会更新 *sk_buff->mark*（第 19 行），以捕获连接类型（默认值）。因此， *cluster_id* 被

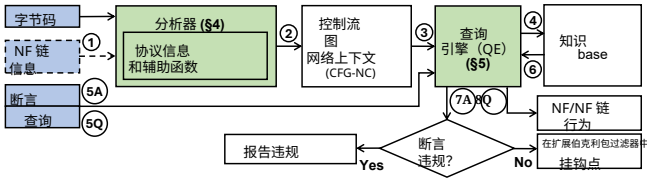

图 2. Yaksha-Prashna 系统工作流程

破坏，导致部分客户的流量被丢弃。为避免此类不良交互在发生之前出现，在将字节码引入链中之前，了解依赖关系至关重要。

如下方清单 2 所示，以下断言用于检查链中 NF2 与任意后继网络功能（NF3）之间是否存在写后读（RAW）依赖关系。在部署 NF2 的字节码之前，可以对新链执行此断言。

A3： !更新字段(NF2, *), 后继网络功能(NF2, SNf), 读取字段(SNf, *).

updatesField 使用 ∗参数可获取 NF2 字节码更新的字段列表（写入列表），其中包括套接字缓冲区和数据包头。 *successorNF*获取 NF2 的所有后继网络功能，而 *readsField* 构造则获取这些后继网络功能读取的字段列表（读取列表）。如果写入列表与 SNf 中任一 NF2 后继网络功能的读取列表存在重叠，则断言失败。Yaksha-Prashna 语言还支持用于检查其他依赖关系（WAR、WAW）的断言。

类别 3：理解网络功能链行为。一旦确定网络功能字节码不符合其规范，或网络功能链中存在依赖关系，找出原因便能帮助我们精准、可靠地报告缺陷。例如，可以准确记录并报告：清单 1 中的规范违规是由对 TCP 源端口的修改造成的，而*sk_buff->mark* 字段则是清单 2 中依赖关系的成因。

在第一种情况下，检索查询

RQ1: `updatesField(xdp_fw, Fld).`

检索由防火墙更新的数据包字段列表（例如，*TCP* 源端口），将其存储在 *Fld* 中，并将其作为查询结果返回。

在第二种情况下，可以使用以下检索查询来调试断言 A2 的失败。

RQ2： 更新字段(NF2, Fld)，后继网络功能(NF2, SNf)，读取字段(SNf, Fld)。

它检索由 NF2 更新并由 NF2 的任一后继读取的套接字缓冲区和数据包头字段列表，将其存储在 *Fld* 中，并将其作为查询结果返回（即，*sk_buff->mark* 在清单 2 中）。总之，检索查询可以深入了解数据包处理行为，例如所访问的协议、所读取/更新的头部/缓冲区字段、网络功能之间的共享映射、所调用的辅助函数等。

3 Yaksha-Prashna 概述

Yaksha-Prashna 工作流如图2所示。它由两个主要组件组成：**分析器**（第4节）和**查询引擎（QE）**（第5节）。**（1）**对于给定的字节码（如果用户希望在部署新的网络功能之前检查是否存在非预期交互，也可以附带 NF 链信息），分析器在控制流分析的指导下构建形式化的数据流分析模型。该模型表示为控制流图（CFG），并使用从字节码中提取的网络上下文（NC）进行注释，从而得到**（2）**CFG-NC。我们将字节码的网络上下文定义为：对数据包内容（报头和元数据）执行的读/写/复制操作、所处理的网络协议、所访问的 eBPF 映射，以及字节码调用的辅助函数。**（3）**查询引擎（QE）将 CFG-NC 中编码的网络上下文转换为**（4）**知识库（Prolog 事实），同时保持 CFG-NC 的结构完整性。**（5A）**以该语言（图1）编写的断言由 QE 通过**（6）**将其谓词映射到知识库中存储的相应事实来进行评估。**（7A）**如果检测到断言违规，则报告该违规；否则，可以将网络功能部署到 eBPF 挂接点。在发生违规的情况下，**（5Q）**会向 QE 发出检索查询，并**（8Q）**检索网络功能或 NF 链的行为。

4 Yaksha-Prashna 分析器

用于网络上下文提取的数据流分析。 为了提取网络上下文，我们采用广泛使用的经典数据流分析和传播方法 [43, 48, 49, 52] ，因为它具有良好的可扩展性，并能全面覆盖程序。SAT/SMT 求解器 [30] 和抽象解释 [27], 等替代方法虽然擅长处理穷举状态空间，并能为验证程序属性带来各种优势，但也存在明显局限：首先，它们的计算开销很大 [26, 28, 39], ；此外，它们可能不适合提取关于程序行为的精确、详细信息，尤其是关于字节码的信息。

相比之下，数据流分析非常适用于扩展伯克利包过滤器字节码，因为与传统程序相比，它没有循环且寄存器数量固定，因此更为简单。这种简单性缩小了状态空间，使分析更具可扩展性且更易于管理，即使是复杂的网络功能也只需极少的开销即可处理。数据流分析能够深入洞察每个程序点上的网络上下文，提供更高效、更细致的分析方法。

挑战。 如 §1 所述，在 eBPF 字节码层提取网络上下文信息时会遇到两个关键挑战：（1）解释指令之间的语义关系；（2）跟踪涉及重新定义的寄存器和内存位置之间的数据流信息。**概述**。 为应对这些挑战，我们提出了一个形式化流分析框架，将寄存器/内存状态跟踪与上下文状态提取相结合。具体而言，我们定义了形式化的上下文状态（C）规则，将低级字节码指令映射到高级程序语义。这些规则涵盖了读取数据包等重要操作

表1. 传递函数规则。

Rule	指令	条件	状态转换	
			寄存器和内存状态	上下文状态
R1: 算术运算	$dst \leftarrow dst \oplus imm \mid src$	–	R $[dst].val \leftarrow$ R $[dst].val \oplus imm \mid$ R$[src].val$	–
R2: 赋值操作	$dst \leftarrow src \mid imm$	–	$\mathcal{R}$ $[dst].tag \leftarrow \mathcal{R}[src].tag \mid$ **常量**, R $[dst].val \leftarrow$ R$[src].val \mid imm$	–
R3: 写入到数据包缓冲区	$*(dst + off) \leftarrow imm \mid src$	$\mathcal{R}$ $[dst].tag ==$ **数据包缓冲区**	–	C $[write_buff].val$ += [getBuffFldName^{spec}($buff_type, off$) $imm \mid$ R$[src].val$]
R4: 写入数据包数据	$*(dst + off) \leftarrow imm \mid src$	$\mathcal{R}$ $[dst].tag ==$ **数据包数据起始位置**	–	C $[write_hdr].val$ += [getHdrFldName $_{spec}$(C $[curr_proto].val, \ off$) $imm \mid$ R$[src].val$]
R5: 写入到程序栈	$*(dst + off) \leftarrow imm \mid src$	$\mathcal{R}$ $[dst].tag ==$ **栈帧**	$\mathcal{M}[off].tag \leftarrow$ **常量** $\mid \mathcal{R}[src].tag$, M$[off].val \leftarrow imm \mid$ R$[src].val$	–
R6: 正在访问 map	$dst \leftarrow map_by_fd(imm64)$	–	$\mathcal{R}$ $[dst].tag \leftarrow$ **引用**^{map} R $[dst].val \leftarrow imm64$	C $[map_read].val$ +=获取映射名称^{elf}()
R7: 写入映射	$CALL\ 2$	$\mathcal{R}[1].tag ==$**引用**^{map}	–	C $[map_write].val$ += [获取映射名称^{elf}(), getHdrFldName $_{spec}$ （C $[curr_proto].val$, R$[3].val$)]
R8: 读取数据包缓冲区	$dst \leftarrow *(src + off)$	$\mathcal{R}[src].tag ==$**数据包缓冲区**, $off ==$ getBuffOffset^{spec}($buff_type$, **数据** | **数据末尾** | <**其他字段**>)	$\mathcal{R}$ $[dst].tag \leftarrow$ **数据包数据起始位置** |**数据包数据末端** |**数据包缓冲区**, R $[dst].val \leftarrow off$	C $[read_buff].val$+= getBuffFldName^{spec}($buff_type, off$)
R9: 读取数据包数据	$dst \leftarrow *(src + off)$	$\mathcal{R}[src].tag ==$**数据包数据起始位置**	R $[dst].tag \leftarrow$ R$[src].tag$, R $[dst].val \leftarrow$ (R$[src].val + off$)	C $[read_hdr].val$+= getHdrFldName^{spec}(C $[curr_proto].val, \ off$)
R10: 读取程序栈	$dst \leftarrow *(src + off)$	$\mathcal{R}[src].tag ==$**栈帧**	R $[dst].tag \leftarrow$ M$[off].tag$, R $[dst].val \leftarrow$ M$[off].val$	–
R11: 下一步协议检查	$dst \neq imm \mid dst == imm$	$\mathcal{R}$ $[dst].tag ==$ **数据包数据起始位置**, 是尾部字段^{spec}(C $[curr_proto].val$, R $[dst].val$)	–	C $[next_proto].val \leftarrow$ 获取协议名称 $_{spec}$(imm)
R12: 内存边界检查	$dst < src$	$\mathcal{R}$ $[dst].tag ==$ **数据包数据起始位置**, $\mathcal{R}[src].tag ==$**数据包数据末端**	–	C $[curr_proto].val \leftarrow$ C $[next_proto].val$
R13: 辅助函数调用	$CALL\ imm$	–	–	C $[helper].val$ += getHlprFuncName $_{spec}$(imm)
R14：返回操作	$return\ r_0$	–	–	C $[pkt_action].val$ += [$hook$, getActionName^{spec}(R[0].val), getPathAndNwContext()]
R15: 已关联 maps	$CALL\ 1\|2\|3\|51$	$\mathcal{R}[2].tag ==$**引用**^{map}	–	C $[correlated_maps].val$ += [获取映射名称^{elf}(), 获取映射名称^{elf}()]

头部或写入映射，以及协议特定行为，使我们能够在遍历指令时准确跟踪上下文的演变（更多详情见表1和§4.3）。此外，我们还制定了精确的寄存器（R）和内存（M）状态规则，以跟踪整个程序中的数据流和交互。这些规则处理寄存器的（重新）定义和内存更新等复杂情况，确保准确跟踪数据流（更多详情见表1和 §4.2）。

上下文状态 (C) 和寄存器/内存状态 (R, M) 均由统一流分析框架管理。寄存器和内存状态为更新上下文状态提供信息。上下文状态用于推导字节码的网络上下文。

4.1 带网络上下文的控制流图

基本块状态与网络上下文（R、M和C）。我们的数据流分析还借助控制流图来考虑程序的控制流。eBPF 代码中缺乏复杂的控制流，这简化了控制流分析。现在，我们说明如何为给定的字节码构建带网络上下文的控制流图（CFG-NC）。CFG-NC 以基本块（BB）作为节点，边则捕获两个节点之间传播的数据。我们将指令分组成一系列序列，每个序列只有一个入口点和一个出口点，结束于跳转指令，或紧接在跳转目标之前，并将每个这样的序列与一个基本块（BB）关联。处理每个 BB 中的指令会生成两类数据：BB 的程序状态（R 和 M）以及 BB 的网络上下文 C。具体而言，程序状态记录处理完一个 BB 后寄存器（R）和栈内存（M）的状态。网络上下文（C）记录 BB 对数据包的处理行为，例如对数据包报头/缓冲区执行的读/写/复制操作、访问的映射以及调用的辅助函数。关于如何生成 R、M 和 C 的更多详情见后续章节（§4.2 和 §4.3）。

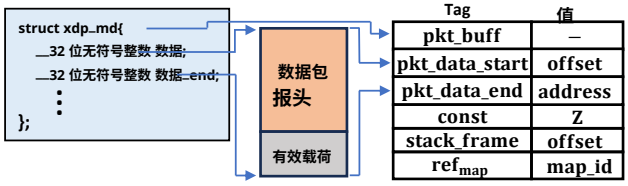

图3. R 和 M <标签、值>对。Z 表示整数 ger.

状态传播与 CFG-NC 生成。我们设计了一种前向分析；BB 的 R、M 和 C[1] 状态（$OUT[BB]$）构成其所有后继节点的 IN状态（$IN[BB]$）。BB 的 OUT状态被计算为一个函数，该函数以 IN状态为输入，对 BB 中的指令应用一组状态转换规则（如表1所示）。关于指令类别以及应用于各类别的规则，详见第4.3节。如果一个 BB 有多个前驱，则使用汇合操作员合并这些前驱的 OUT 状态，并生成 IN 集合。也就是说，

$$IN[BB] = \bigcup_{p \in pred(BB)} OUT[p]$$

CFG 中的基本块按深度优先搜索（DFS）顺序处理，并对从每个基本块提取的网络上下文（C）进行注释。随后，查询引擎模块通过处理 CFG-NC 中的 C 来回答用户查询（第5节）。

4.2 $\mathcal{R}$、$\mathcal{M}$ 和 C 数据结构

我们将介绍 R、M 和 C 数据结构，以及应用于指令的状态转换规则（表1）。

R和M数据结构。eBPF 字节码中的指令最多使用 11 个寄存器，以及一个 512 字节的程序栈。复杂的 eBPF 程序可能会耗尽寄存器，因此一旦出现寄存器溢出，通常就会使用程序栈 [50]。因此，分析某些指令时需要跟踪并更新 R 和 M。$R[n]$ 表示寄存器状态，其中$n \in \{0, 1, 2, \ldots, 10\}$分别对应指令中使用的 11 个寄存器（$r_0$-$r_{11}$）。$M[k]$ 表示程序栈中的 k^{th} 字节，其中 $k \in \{0, 1, 2, \ldots, 511\}$。每个条目 r_n 和字节 k 都有一个（标签、值）对。根据可能的值，我们提出了六种标签，如图3所示：(i) 标签 pkt_buff 表示 r_n 或字节 k 保存了对数据包结构缓冲区（例如 xdp_md、__sk_buff）的引用，其值为 none；(ii) pkt_data_start 包含相对于数据包数据起始位置的偏移量；(iii) pkt_data_end 指向数据包数据的末尾；(iv) const 保存常量值；(v) refmap 指向程序中的一个 eBPF 映射；(vi) stack_frame 指向栈中的某个位置。

网络上下文（C）数据结构。与 R 和 M 类似，C 也有一个由（标签，值）对组成的列表。共有七个标签：**(i) hdr_字段** 和 **buff_字段** 用于记录访问或更新过的数据包头部和数据包缓冲区字段列表；**(ii)辅助函数_函数** 用于记录调用过的辅助函数列表；**(iii) 映射** 用于记录访问过的扩展伯克利包过滤器映射名称；以及(iv) [**下一个_协议、边界_检查、当前_协议**], 这三者都有助于提取字节码处理的协议（§4.3）。

初始化： 按照惯例，在 eBPF 字节码的开头，寄存器 $r1$ 指向数据包缓冲区——数据包缓冲区结构定义在内核头文件 bpf.h 中（见图3）。这些结构为 eBPF 程序提供输入上下文，并保存指向数据包 [84]起始位置和结束位置的指针。因此，寄存器 $r1$ 的状态初始化为（*pkt_buff*, -），其他状态（R、M 和 C）则设为 NULL。

4.3 网络上下文提取

按指令类别划分的状态转换规则。我们应用传递函数规则，以便（i）在基本块内部及跨基本块传播各条指令之间的 R 和 M 状态；（ii）记录每个基本块中的 C。字节码中的一条指令 [1] 包含操作码（8 位）、源寄存器和目标寄存器（各 4 位）、偏移量（16 位）以及立即值（32 位）。如表1所示，我们将指令分为 8 个类别，并应用相应规则。每条规则都有一个可选条件；若该条件满足，R 和 M 状态便会更新，并将关联的网络上下文记录在 C中。变量以斜体表示（例如 R[*src*].*tag*、R [*src*] .*val*、*imm*等），标签以粗体表示（例如，**pkt_buff**、**pkt_data_start**）。

协议信息和辅助函数。字节码指令包含立即值和偏移量，每个值都有特定含义，例如数据包头部或数据包缓冲区中的字段偏移量（例如，sk_buff）、协议编号（例如，ethrType、ipv4.protocol）、辅助函数 ID、特定于钩子点的动作 ID 以及映射 ID。我们将这些 ID 捕获到文件中，并定义一组辅助函数，用于在文件中查找并返回最终用户能够理解的名称（例如，协议名称、辅助函数名称）。该机制可以扩展，以支持新的协议、钩子点和辅助函数。

运行示例。对于简单的算术（R1）和赋值（R2）操作，我们更新 R 和 M 状态，但不记录网络上下文，因为这些指令不会直接更新或读取数据包头部/缓冲区。相比之下，对于写操作（R3-R5）和读操作（R8-R10），我们会记录网络上下文。例如，在图 4 中，我们展示了指令 1、2 和 15 是读操作，并分别应用传递函数规则 R8、R9 和 R9[2]。指令 11 属于写类型，因此应用规则 R4[3]。

R6、R7 和 R13 分别用于映射查找、映射更新和辅助函数。例如，考虑图 4 中的字节码指令 3 和 4。这些指令调用 *cpus_count* 映射上的查找辅助函数。对于这些指令，我们使用 R6 和 R13 提取映射名称以及所调用辅助函数的名称。同样，

[1]我们不会传播完整的网络上下文状态，只传播需要前一基本块网络上下文的那一部分

[2]'getBuffOffset' 支持函数用于获取规范中定义的数据包缓冲区字段的偏移量。[3]'getHdrFldName' 和 'getBuffFldName' 支持函数分别返回头部字段和数据包缓冲区的名称。

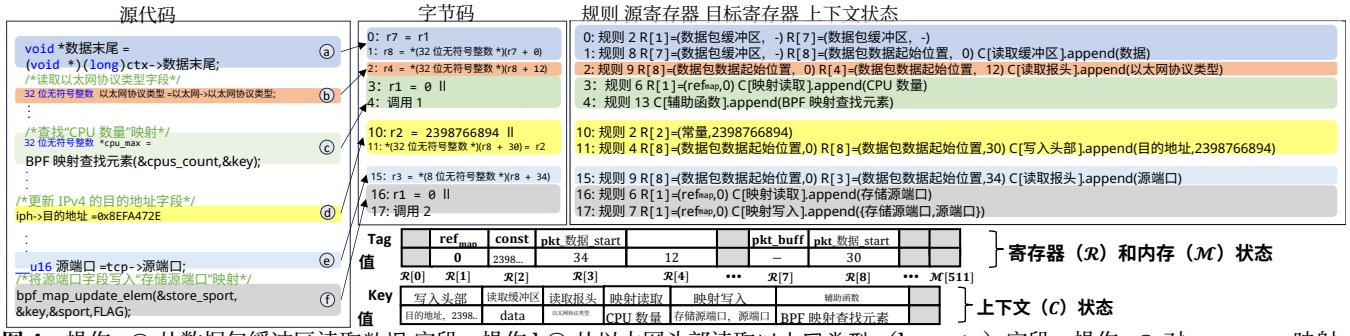

图 4。操作 a◯ 从数据包缓冲区读取数据 字段。操作 b◯ 从以太网头部读取以太网类型 （h_proto）字段。操作 c◯ 对*cpus_count* 映射执行查找。操作 d◯ 更新 IPv4 头部的目标 *IP* 地址 。操作 e◯ 从 TCP 头部读取源端口 。操作 f◯ 将源端口 写入*store_*源端口 映射。

字节指令 16 和 17 调用 *store_sport* 映射上的映射更新辅助函数。对于这些指令，我们记录映射名称、写入映射的字段，以及与立即数值[4]相关联的辅助函数。

应用规则 R11 和 R12 来提取字节码中所处理的协议头部列表。通常，在解析数据包数据时，通过查看先前已识别报头中的特定字段来识别报头。例如，将前 14 个字节视为以太网头部。然后，通过查看以太网类型字段，我们便知道下一个报头是虚拟局域网、IPv4 或 IPv6 头部。同样，通过查看 ipv4.proto 字段，我们便知道下一个报头是 TCP 或 UDP 头部。我们利用数据包头部语义 [11] 并识别报头。我们还利用了这样一个事实：大多数字节码在访问下一个报头之前会执行内存边界检查，以确保后续代码的访问位于数据包内存之内。如果在没有内存边界检查的情况下访问下一个报头，则 eBPF 验证器会抛出错误，并且不允许加载该字节码。附录 A.2 中的图 8 给出了一个带有说明步骤的运行示例字节码。

R13 根据立即数 值获取调用的 eBPF 辅助函数，该值表示与每个辅助函数关联的 ID。辅助函数名称通过支持函数‘getHlprFuncName’获取。R14 根据寄存器 r_0 值捕获返回操作（例如，XDP_PASS、TC_ACT_OK）；该值因不同的钩子点[5]（例如，XDP、TC）而异。它还会将当前路径的网络上下文附加到 C。若映射 A 的查找结果是映射 B 的键，则认为这两个映射相关。规则 R15 根据寄存器 r_2 标签获取相关映射；该标签包含映射操作（例如映射更新（调用 2）和删除（调用 3））所用的键。

5 语言与查询引擎（QE）

Yaksha-Prashna 语言。 如图1所示，其语法遵循声明式（逻辑）编程范式 [75] 语法与广泛使用的 Prolog[6] 语言语法相似 [68, 77, 86]。该语法提供了专门针对扩展伯克利包过滤器的谓词，使该工具的主要目标受众——扩展伯克利包过滤器用户——能够利用已有的扩展伯克利包过滤器知识，轻松表达断言和查询。其两个关键构造如下：（1）**Q-谓词**：Q-谓词（Q-Pred）是查询的基本构件。它们用于断言特定属性或获取网络上下文。Q-Pred 接受一个或多个参数：当 Q-谓词的所有参数都定义了特定值时，该谓词断言该属性成立。当一个或多个参数是变量（即，未定义）时，Q-Pred 会检索该属性适用的所有实例，并将变量填入检索到的值；（2）**查询**：查询由一个或多个通过逻辑 AND(,)/OR(;) 操作符连接的 Q-Pred 组成，并以句点 (.) 结束。

5.1 查询引擎

查询引擎（QE）将以 CFG-NC 编码的网络上下文转换为 Prolog 事实，同时保持 CFG-NC 的结构完整性以及 NF 链中网络功能的顺序。使用 Yaksha-Prashna 语言（图1）编写的用户查询，通过将 Q-Pred 谓词与知识库中的事实进行匹配来回答。表2展示了 Q-Predicate[7]，Prolog 事实，以及 CFG-NC 中对应的网络上下文。事实根据与每个 CFG-NC 节点相关联的网络上下文生成。我们将 NF 链中所有字节码的完整事实集合称为知识库。

为了将 Q-Pred 谓词映射到事实，我们在内部定义了用于匹配的规则。规则的形式为 Q-Pred:- R-Pred$_1$运算符 R-Pred$_2$运算符 R-Pred$_3$运算符 ..., R-Pred$_n$,其中 Q-Pred 是构成规则头的查询谓词，R-Pred 谓词对应于构成规则体的规则谓词，而运算符 是逻辑运算符，例如 AND(,) 和 OR(;)。如图 5 所示，规则头会呈现给操作员（§1），而规则体则构成特定查询谓词的操作

[4]‘getMapName$_{elf}$()’会查找目标文件的 ELF 节，并返回映射名称。[5]‘getActionName’支持函数会获取动作名称。

[6]Prolog 只是 Yaksha-Prashna 使用的一种实现语言；也可以使用其他逻辑编程语言。[7]定义见附录 A.4

表2. 谓词、网络上下文和相关事实。

谓词类别	来自以下内容的网络上下文 CFG-NC	事实
字段谓词	*C*[读取_缓冲区] = [(缓冲区_字段, 字段_名称)], *C*[读取_头部] = [(头部_字段, 字段_名称)], *C*[写入_缓冲区] = [(缓冲区_字段, 字段_名称)], *C*[写入_头部] = [(头部_字段, 字段_名称)]	读取_缓冲区_字段("网络功能_ID", "基本块_ID", 字段_名称), 读取_报头_字段("nf_id", "bb_id", 字段_名称), 写入_缓冲区_字段("nf_id", "bb_id", 字段_名称), 写入_报头_字段("nf_id", "bb_id", 字段_名称)
Map 谓词	*C*[读取_映射] = [(映射, 映射_名称)], *C*[写入_映射] = [(映射, 映射_名称, 字段_名称)] *C*[相关联的_映射] = [(映射, 映射_A，映射_B)]	读取_自_映射("nf_id", "bb_id", 映射_名称), 写入_到_映射("nf_id", "bb_id", 映射_名称, 字段_名称) 关联的_映射("nf_id", "bb_id", 映射_A，映射_B)
辅助谓词	*C*[辅助函数] = [(辅助函数, 函数_名称)]	调用_辅助函数("网络功能_标识 ", "基本块_id_ ", 函数_名称)
协议谓词	*C*[当前_协议] = [(当前_协议, 协议_名称)]	协议_accessed("nf_id", "bb_id", 协议_名称)
动作谓词	*C*[*pkt*_动作] = [(钩子, 动作，*P*)]	返回_动作("nf_id", 钩子, 动作， *P*)
顺序谓词	–	边("bb_id", "bb_id"), 网络功能_边("网络功能_id", "网络功能_id")

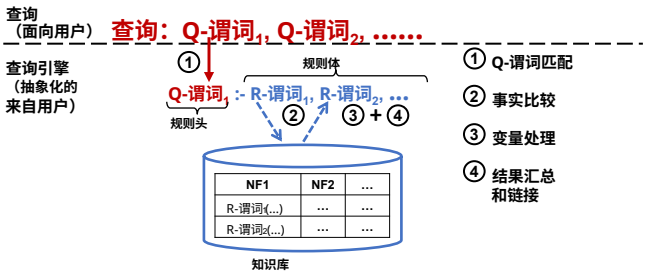

图 5。查询执行工作流。

语义。这种方法抽象了底层事实的细节，同时允许用户编写查询，而无需了解底层事实的详细结构。附录 A.5 中的表 5 总结了所有 Q-Pred 谓词的操作语义。

以下步骤用于从知识库中回答用户查询。**(1) Q-谓词匹配：**将查询中的每个 Q-谓词与已定义规则的头部进行比较。**(2) 事实比较：**将 R-Pred 谓词与知识库中存储的事实进行匹配。**(3)变量处理：**如果查询包含变量，引擎会从知识库中检索所有满足规则的值。**(4) 结果汇总与链接：**匹配过程结束后，通过收集所有满足查询谓词的值来汇总结果。附录 A.6 提供了一个贯穿示例，用于解释这些步骤。

Yaksha-Prashna 断言执行。 在附录 A.1 中，我们使用第 §2.2 节定义的三个断言——**A1**、**A2**和**A3** ——来说明分析器和查询引擎如何处理第 §2.2 节中的两段字节码（清单 1 和清单 2）。

扩展。 Yaksha-Prashna 可以扩展以支持新的用户定义的 Q-Predicate 谓词，从而匹配事实。如前所述，新的 Q-谓词应具有与相应 R-Pred 谓词相匹配的语义，并且可以将其纳入一个库，以支持针对新应用场景的更多查询。

6 评估

我们对 Yaksha-Prashna 进行评估，以研究：1）Yaksha-Prashna 语言的表达能力；2）Yaksha-Prashna的使用情况（§6.2）；3）网络上下文提取的准确率（§6.3）；4）生成 CFG-NC（§6.3）和执行查询（§6.4）所需的时间；以及 5）内存开销（§A.12）。

实现。 Yaksha-Prashna 分析器模块包含 1.8K 行 C 代码++。在内部，QE 模块会调用 SWI-Prolog（版本 9.3.3）[86] 环境，该环境接收事实和规则，并提供用于编写查询的命令行界面。此模块包含 600 行 C 代码++。我们在一台配备 AMD EPYC 7262、3.2 GHz、8 核 CPU 和 32GB 内存的服务器上进行了实验。

6.1 表达能力

我们通过对涵盖三类内容（§2.2）的属性和查询进行表达，评估语言的表达能力；这些内容涉及六种以上的标准网络功能、四种非标准但实际存在的基于 eBPF 的网络功能，以及网络功能链。表 3 列出了 Yaksha-Prashna 及其密切相关的工作 Klint [70] 和 DRACO [61]所支持的属性。每项属性均使用 Yaksha-Prashna 的语言（表 7）进行表达，并在附录 §A.10 中作了详细说明。本节总结了主要观察结果。

第一类包含 21 项属性（P1-P21），如表 3 所示，这些属性源自针对单个网络功能的标准文档（RFC 文档和 IEEE 标准）。Yaksha-Prashna 和 Klint 支持大多数属性，而 DRACO 的支持情况尚不明确（–）；但 P5 和 P16–P21 属性除外，因为它没有公开规范语言。Yaksha-Prashna 仅部分支持 P4 和 P11 属性，因为该语言缺少针对映射类型和广播数据包的谓词。第二类属性针对网络功能链，用于检测非预期交互（§2.2）。目前，Klint 仅支持针对单个网络功能的断言，而 Yaksha-Prashna 和 DRACO 支持针对网络功能链中依赖关系的断言。

第三类包含检索查询（Q23 和 Q24），展示了该语言不仅能够支持断言，还能检索导致违规的具体细节。具体来说，为了找出防火墙修改了哪个字段（Q23），从而导致断言违规，Klint 或 DRACO 必须为每个头部字段编写一条保留断言，例如 “源 *MAC* 地址是否被修改？”或“源 *IP* 是否被修改？”等。这既繁琐又可能无法实现，因为需要检查每个可能的报头，而且对于大型程序而言会耗费很长时间。相比之下，Yaksha-Prashna 可以立即从知识库中存储的事实中获取这些信息。同样，Yaksha-Prashna 也可以检索导致读后写依赖的缓冲区字段（Q24）（清单 2）。

表 3. Yaksha-Prashna、Klint 和 DRACO 支持的属性。(✗)：部分支持。属性的上标标明网络功能名称——RT：路由器，FW：防火墙，LB：负载均衡器，NAT：网络地址转换，BR：网桥，POL：流量监管器，KTRN：Katran 负载均衡器，CRAB：CRAB 负载均衡器，FLU：Fluvia。∗ 表示所有网络功能

类别	属性（P1-P22）/检索查询（Q23-Q24）	Yaksha-Prashna	Klint	DRACO
类别 1：单个网络功能符合规范	P1. 预期的报头$^{(*)}$	✓	✓	-
	P2. IPv4 首部有效性$^{(RT)}$	✓	✓	-
	P3. NF 预期的首部字段更新$^{(RT)}$	✓	✓	-
	P4. 最长前缀匹配$^{(RT)}$	✗	✓	-
	P5. 数据包内容保留$^{(FW)(LB)}$	✓	✓	✓
	P6. 外部流过滤$^{(FW)(NAT)}$	✓	✓	-
	P7. 记住内部流$^{(FW)(NAT)}$	✓	✓	-
	P8. 更新内部流的源 IP 和端口 内部流$^{(NAT)}$	✓	✓	-
	P9. 恢复外部流的目的 IP 地址和端口 外部流$^{(NAT)}$	✓	✓	-
	P10. 所选传输端口不得为 入端口$^{(BR)}$	✓	✓	-
	P11. 未知时广播$^{(BR)}$	✗	✓	-
	P12. 学习过程$^{(BR)}$	✓	✓	-
	P13. 外部流量监管$^{(POL)}$	✓	✓	-
	P14. 后端存活性$^{(LB)}$	✓	✓	-
	P15. 外部流量负载均衡$^{(LB)}$	✓	✓	-
	P16. 丢弃所有分片数据包$^{(KTRN)}$	✓	✓	✓
	P17. 转发所有 ICMP 回显请求 数据包$^{(KTRN)}$	✓	✓	✓
	P18. 仅处理 TCP SYN 数据包$^{(CRAB)}$	✓	✓	✓
	P19. 添加自定义重定向标头$^{(CRAB)}$	✓	✓	✓
	P20. 原样转发所有数据包$^{(FLU)}$	✓	✓	✓
	P21. 相关映射$^{(hXDP-FW)}$	✓	✓	✓
类别 2：非预期的网络功能交互	P22. NF依赖：读取后写（RAW）、读后写（WAR）以及写后写（WAW）	✓	✗	✓
类别 3：理解网络功能链行为	Q23. 找出数据包字段列表 由网络功能更新	✓	✗	✗
	Q24. 找出由某个网络功能更新的缓冲区字段列表 并由后继网络功能读取（依赖 RAW）	✓	✗	✗

6.2 Yaksha-Prashna 的用法

我们在测试平台服务器上演示了操作员如何使用 Yaksha-Prashna 检查网络功能字节码是否符合其规范；如果不符合，还能查明其偏离预期网络功能行为的原因（类别 1）。此外，它还能捕获非预期的网络功能交互（类别 2），并帮助理解网络功能/网络功能链的行为（类别 3），正如 §2.2 中所述。

类别 1 用例。 假设某网络功能操作员希望通过 bpfman 在服务器的 XDP 挂载点部署一个有状态防火墙（清单 1），但该防火墙存在两个错误（参见 §A.8 中的图 9）。这两个错误是：(1) 更新 TCP 源端口（第 10 行）；(2) 允许 ICMP 流量（第 6 行）。部署前，操作员使用 Yaksha-Prashna 针对断言 A1 和 A2 进行验证。A1 断言防火墙不应修改数据包内容。由于防火墙修改了 TCP 源端口，A1 未通过，Yaksha-Prashna 发出了警报。为了解原因，操作员运行了检索查询 RQ1（类别 3），该查询正确地将违规源识别为 TCP 源端口更新。同样，A2 断言应丢弃 ICMP 流量。由于防火墙允许 ICMP 数据包通过，A2 未通过，Yaksha-Prashna 因此发出了另一条警报。通过这些断言，Yaksha-Prashna 成功检测并阻止了非标准网络功能的部署。

类别 2 用例。我们模拟 Cilium 与 AWS 弹性网络接口之间真实场景中的 RAW 依赖关系。我们考虑了由三个基于 XDP 的网络功能组成的 NF 链（清单 2），其中 NF1 和 NF3 按顺序部署（NF1→ NF3）。插入 NF2 之前，我们使用断言 A3 验证了预期顺序（$NF_1 \rightarrow NF_2 \rightarrow NF3$）的字节码；该断言检查 NF2 的写集合是否与 NF3 的读集合重叠

表4. 微基准网络功能。(*) 来自开源项目的网络功能，(+) 手写网络功能，NI：指令数，NAI：分析的指令数，NP：路径数。

ID	网络功能	项目	NI	NAI	NP
NF1	xdp_pktcntr	Katran [64]	22	28	4
NF2	XDP_映射_访问	K2 [88]	42	64	6
NF3	在映射中存储 IP+	–	39	49	7
NF4	XDP_防火墙	K2 [88]	73	496	38
NF5	速率限制_内核_新	ebpf-ratelimiter [14]	136	292	19
NF6	速率限制_内核	ebpf-ratelimiter [14]	167	278	18
NF7	XDP1_内核	K2 [88]	61	925	99
NF8	解封装_内核	Katran [64]	228	430	33
NF9	XDP_过滤器	Suricata [35]	344	13K	639
NF10	XDP_过滤器_缓冲区_更新*	Suricata [35]	348	13K	639
NF11	mptm_封装_XDP	Mptm [15]	348	25K	1K
NF12	mptm_解封装_XDP	Mptm [15]	149	38K	2K
NF13	XDP_负载均衡	Suricata [35]	419	65K	4K
NF14	XDP_负载均衡_标头_更新*	Suricata [35]	423	65K	4K
NF15	XDP_负载均衡_SYN_将 IP 存储到映射中*	Suricata [35]	440	66K	4K
NF16	xdp_lb_syn_在映射中存储 IP 生存时间*	Suricata [35]	460	68K	4K

。断言 A3 因 *sk_buff->mark* 存在重叠而失败，Yaksha-Prashna 随即发出警报。接下来，我们使用检索查询 RQ3（类别 3）查明违规原因。因此，Yaksha-Prashna 阻止了将新NF部署到现有 NF 链时可能发生的意外交互。

6.3 网络上下文提取

网络功能与查询。 我们使用带有 O2、Oz 和 O3 标志的 *LLVM Clang*（V11.1.0）为包含 16 个基于 XDP 的网络功能的微基准测试套件生成字节码（表4）。所选网络功能代表了从广泛使用的开源系统和生产系统中选取的多种真实世界 XDP 程序。它们涵盖常见的数据包处理任务，包括数据包计数、映射访问、防火墙处理、速率限制和负载均衡。该套件涵盖了复杂度和指令数量范围广泛的网络功能：简单的网络功能（例如 NF1 和 NF2）实现数据包计数等基本功能，包含 22–42 条指令；而更复杂的网络功能（例如 NF4、NF9 和 NF13）则执行防火墙处理、过滤和负载均衡，包含数百条指令，其中最大的超过 400 条。这种多样性使我们能够在代表实际部署场景的轻量级和计算密集型网络功能上评估系统。我们总共设计了 16 个查询（附录A.7中的表6），并评估单个网络功能的结构属性以及 NF 链中多个网络功能之间的交互。这些查询平均分配，其中 8 个侧重于断言，另外 8 个侧重于检索，涵盖了表2和表5中详述的所有谓词和规则类别。

准确率。我们将准确率定义为分析器所提取的网络上下文信息相对于真实情况中信息的比例。为了确立真实情况，我们人工审阅源代码，以提取协议列表、每个协议/缓冲区中访问的特定字段、辅助函数以及映射。分析器对所有网络功能都正确提取了信息，但 NF12 中的某些字段除外。我们还将微基准网络功能（表4）中随机堆叠事实的断言查询结果（即，Q1(A)–Q8(A)）进行了比较，发现答案是正确的。

我们的系统假设，网络功能（NF）在访问协议头字段之前，会先执行下一协议检查，随后进行内存边界检查。这一假设源于现实世界中的实现，并符合常见的编程惯例。下一协议检查使网络功能能够应用特定于协议的数据包处理逻辑，而内存边界检查则确保对报头的访问位于数据包边界内，这是通过验证器检查的必要条件。在实践中，开发者通常会以这种方式组织数据包解析逻辑，Katran 和 Suricata 等生产系统中也可以观察到类似模式。

尽管这一假设适用于大多数网络功能，但我们也观察到一些例外。在 NF12（mptm_decap_xdp）中，我们的系统无法识别 UDP 协议及其相关头部字段。这是因为该网络功能没有对 UDP 执行显式的下一协议检查；相反，它默认将所有非 TCP 流量视为 UDP。

当网络功能被设计为仅处理部分流量（例如 TCP 和 UDP），且未实现全面的协议检查时，我们的系统可能无法准确推断其预期的协议语义。这一行为凸显了当前版本系统的局限性。不过，通过扩展系统，使其能够处理针对受限流量类别而非完整通用数据包处理进行定制的网络功能，可以解决这一局限。

CFG-NC生成时间。我们评估了系统在各种网络功能上生成CFG-NC所需的时间，如§A.9中的图10（左）所示。结果取五次运行的平均值，显示CFG-NC生成时间与网络功能的复杂度之间存在明显关系。我们观察到，生成CFG-NC所需的时间随控制流图中路径数量的增加而增加。对于包含4至 $4K$ 条路径的网络功能，生成时间约为3至300毫秒。对于NF13–NF16这类路径数量相同但指令数量不同的网络功能，随着指令数量增加，所需时间也按比例增加。对于O3优化字节码和采用其他优化标志的字节码，我们观察到提取网络上下文所需的时间没有显著差异。

6.4 查询执行

查询引擎构建于高度优化的 SWI-Prolog 运行时（V7.6.4）[86][8]之上；我们的所有查询执行都只需几微秒。我们分析了以下情况下的查询执行时间：

[8]SWI-Prolog实现了索引和统一等多种优化，以优化查询求解

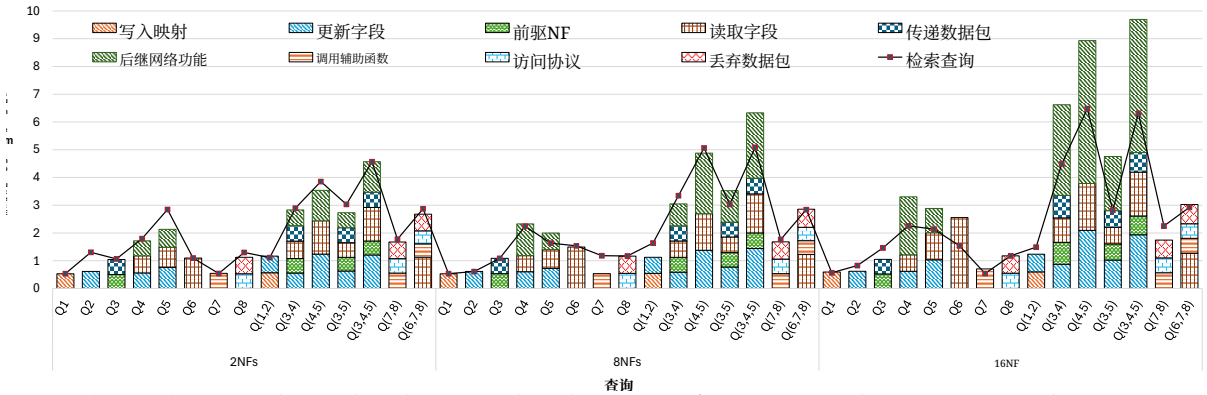

图 6。执行时间：断言　　（柱状图）和检索查询（折线），针对长度不同的网络功能链，其中网络功能为随机选取　　.

涉及 (i) 断言与检索，(ii) 是否存在递归谓词，(iii) Q-谓词数量不同（§A.11），以及 (iv) NF链长度不同（§A.11）。为降低噪声的影响，我们对微基准测试中的每个查询运行 1000 次，每 100 次运行计算一个中位数，然后对这些中位数取平均值。

检索与断言查询。检索查询与断言查询中每个 Q-谓词的执行时间如图 6 所示。我们观察到，各个检索查询 Q1(R)–Q8(R) 和断言查询 Q1(A)–Q8(A) 在对从 16 个网络功能获得的事实执行时耗时 $4\mu s$。检索查询与断言查询的组合变体分别耗时 $1.1\mu s$–$6.4\mu s$ 和 $1.1\mu s$–$9.6\mu s$。

递归谓词。当规则体中包含一个指向同一规则头的谓词时，该规则称为递归规则。例如，表 5 中所示的 successorNF 和 predecessorNF 谓词就是递归谓词。我们发现，由于处理递归调用会产生额外开销，涉及此类递归谓词的查询比不涉及递归谓词的查询执行时间更长。在十六个网络功能的事实上执行这些查询时，耗时为 $1.04\mu s$–$3.3\mu s$；相比之下，不含递归谓词的查询执行时间为 $0.5\mu s$–$2.5\mu s$。

与 Klint 的比较。图 7（左）展示了 Yaksha-Prashna 和 Klint 在六个具有代表性的网络功能上验证 15 项属性（P1–P15，附录 §A.10.1）所需的时间。Yaksha-Prashna 的验证时间包括 CFG-NC 生成和断言执行，而 Klint 的验证时间则包括符号执行、不变量推断和验证。Klint 耗时 2.7 秒至 1.5 分钟，而 Yaksha-Prashna 仅需 13–75 毫秒；与 Klint 的符号执行相比，其轻量级数据流分析带来了 200-1000× 的加速。

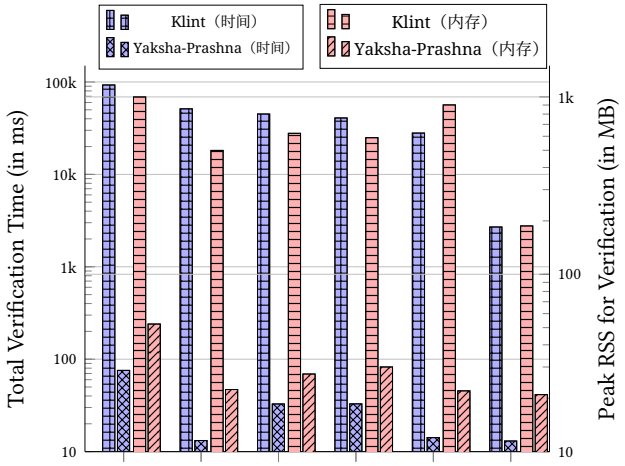

图 7. 验证时间（左）和内存占用（右）。

7 局限性

本节讨论 Yaksha-Prashna 的局限性。附录中的第 §A.3 节进一步讨论了识别数据包处理算法时遇到的复杂性，以及通过随机猜测来识别网络上下文信息的问题，以及对语言和分析器的可能扩展。

运行时查询。网络功能通常根据报头字段值确定数据包动作。例如，负载均衡器利用流信息将后续数据包转发到同一目的地。Yaksha-Prashna 不支持对这类运行时属性进行验证，因为它们需要运行时信息，而这些信息在静态分析阶段不可用。

关于映射规则的合规性断言。映射是 eBPF 程序的重要组成部分，但错误的规则或缺失的动作可能导致意外的数据包行为。运维人员通常希望在插入之前检查映射规则是否合规（例如，不存在通配符规则 [58]）。假设映射规则已给定，Yaksha-Prashna 不支持这些断言。

支持更多挂接点。网络功能可以附加到多个挂接点 [5]，例如 XDP、TC 和套接字过滤器。目前，Yaksha-Prashna 仅支持 XDP 程序。扩展该工具以支持更多挂接点主要是工程工作，而非

我们方法的根本限制，因为不同钩子之间的差异（例如执行上下文结构、字段偏移量和返回码）可以通过额外的实现工作加以纳入。

8 相关工作

用于验证的静态分析。 广泛使用的 eBPF 验证器，如 Linux eBPF 验证器 [82] 和 Prevail [40]，使用静态分析和抽象解释 [27] 来验证程序终止、内存安全和资源有界性，以确保内核安全，但不涉及 eBPF 字节码的网络上下文。Bromberger 等人 [21] 提出使用 Z3 SMT 求解器 [30]验证 eBPF 程序的功能属性。先前的工作也使用程序验证来识别 P4 [60, 78] 中的缺陷。Gravel [91],、Vigor [90],和 VEP [87] 采用符号执行来验证非基于 eBPF 的网络功能。除了发现依赖关系之外，Yaksha-Prashna 还可以扩展用于验证有状态网络的网络策略，如 NetSMC [89]中所示。ASSERT-P4 [38]和 DRACO [61]是基于 Klee 符号执行 [23] 的方法，用于验证 P4/C 程序。

用于优化的静态分析。 Merlin [62],、KFuse [54],、K 2 [88], 和 Bonola 等人 [19]提出了基于静态分析的 eBPF 字节码优化方法。Merlin 在 LLVM-IR 上执行感知 e BPF 的优化，而 KFuse 和 K2 则针对经过验证的 eBPF 字节码设计优化。Morpheus [66] 根据映射访问模式的配置文件，对 eBPF 程序进行运行时优化。另一方面，我们的系统侧重于提取 eBPF 字节码的网络上下文。

领域专用语言。 Halide [71]（用于图像处理）和 NetBlocks [20]（用于临时网络协议设计）等领域专用语言（DSL）推动了转译器系统的发展，使其能够在特定目标领域实现高性能。Yaksha-Prashna语言受到 Datalog [25] 和 NDlog [59] 等其他 DSL 的启发，这些语言基于声明式语言范式。这些语言旨在查询数据库和联网系统，并对其进行推理，以适用于分布式环境。另一方面，我们的语言旨在查询并推理 eBPF 字节码的网络功能行为。

9 结论

我们介绍了 Yaksha-Prashna，这是一个用于理解基于扩展伯克利包过滤器的网络功能字节码及其与其他网络功能交互的系统。我们提出了：（1）分析器，通过数据流分析和传播提取并建模网络上下文；（2）语言，这是一种简单而富有表达力的领域特定语言，供网络功能操作员/开发者断言和查询扩展伯克利包过滤器字节码的行为，同时抽象掉复杂的低级字节码细节。我们通过编写各种查询来检查网络功能属性，评估了（1）Yaksha-Prashna语言的表达能力；（2）Yaksha-Prashna 在 16 个 XDP 个程序上的性能；（3）将 Yaksha-Prashna 与最先进的工具 Klint 和 DRACO 进行了比较。

参考资料

[1] 2024. eBPF 指令集规范，v1.0。https://docs.kernel.org/6.5/bpf/instruction-set.html. [2] 2024. Fluvia 源代码仓库。https://github.com/nttcom/ fluvia.[3] 2024. Linux 手册页。https://www.man7.org/linux/man-pages/ man1/time.1.html.[4] 2025. bpfman：一个 eBPF 管理器。https://bpfman.io/v0.5.6/. [5] 2025. eBPF 钩子与应用场景分类。https://eunomia.dev/others/miscellaneous/ebpf-usecases/. [6] 2025. Equinix Network Edge。https://www.equinix.com/products/digital-infrastructure-services/network-edge. [7] 2025. F5 NGINX App Protect。https://www.f5.com/products/nginx/ nginx-app-protect. [8] 2025. libxdp。https://github.com/xdp-project/xdp-tools/tree/main/lib/libxdp. [9] 2025. NGINX App Protect DoS 3.0。https://docs.nginx.com/nginx- app-protect-dos/releases/about-3.0/#important-notes. [10] 2025. PAN-OS 发行说明。https://docs.paloaltonetworks.com/ content/dam/techdocs/en_US/pdf/pan-os/10-1/pan-os-release- notes/pan-os-release-notes.pdf. [11] PANDA 作者。2023。PANDA（协议与网络数据通路加速）。https://github.com/panda-net/panda/blob/main/documentation/images/Parse-graph.png. [12] Ryan Beckett、Xuan Kelvin Zou、Shuyuan Zhang、Sharad Malik、Jennifer Rexford 和 David Walker。2014。用于调试 SDN 应用的断言语言。发表于 *HotSDN*。[13] Dushyant Behl、Hai Huang、Palani Kodeswaran 和 Sayandeep Sen。2023。关于扩展伯克利包过滤器对 Kubernetes CNI 数据路径的扩展。发表于 *COMSNETS*。[14] Dushyant Behl、Sayandeep Sen 和 Palani Kodeswaran。2023。https: //github.com/ebpf-networking/ebpf-ratelimiter. [15] Dushyant Behl、Sayandeep Sen 和 Palani Kodeswaran。2023。xdp-mptm。https://github.com/dushyantbehl/xdp-mptm. [16] Theophilus A. Benson、Prashanth Kannan、Prankur Gupta、Balasubramanian Madhavan、Kumar Saurabh Arora、Jie Meng、Martin Lau、Abhishek Dhamija、Rajiv Krishnamurthy、Srikanth Sundaresan、Neil Spring 和 Ying Zhang。2024。NetEdit：面向大规模 eBPF 网络功能的编排平台。发表于*ACM SIGCOMM*。[17] Gilberto Bertin。2017。XDP 实践：将 XDP 集成到我们的 DDoS 缓解流水线中。发表于 *Linux* 网络技术会议，*Netdev*。[18] Brenden Blanco、Jakub Kicinski、Salvatore Orlando 和 Tomás Senart。2023。Cilium。https://github.com/cilium/cilium.[19] Marco Bonola、Giacomo Belocchi、Angelo Tulumello、Marco Spaziani Brunella、Giuseppe Siracusano、Giuseppe Bianchi 和 Roberto Bifulco。2022。使用 eBPF 程序扭曲，在 FPGA 网卡上实现更快的软件数据包处理。发表于 *USENIX ATC*。[20] Ajay Brahmakshatriya、Chris Rinard、Manya Ghobadi 和 Saman Amarasinghe。2024。NetBlocks：面向高性能定制主机网络协议栈的暂存布局。（2024）。[21] Martin Bromberger、Simon Schwarz 和 Christoph Weidenbach。2024。eBPF 代码的自动位级与内存精确验证。发表于 第 *25* 届 *Logic for Pro* 会议论文集。[22] Marco Spaziani Brunella、Giacomo Belocchi、Marco Bonola、Salvatore Pontarelli、Giuseppe Siracusano、Giuseppe Bianchi、Aniello Cammarano、Alessandro Palumbo、Luca Petrucci 和 Roberto Bifulco。2020。hXDP：在 FPGA 网卡上实现高效的软件数据包处理。发表于 *USENIXOSDI*。[23] Cristian Cadar、Daniel Dunbar 和 Dawson Engler。2008。KLEE：无需辅助的高覆盖率测试自动生成，适用于

复杂系统程序。载于 USENIX OSDI 08。[24] Carlos Castro。2024。身份未找到，流量被丢弃。https://github.com/cilium/cilium/issues/20797/#issuecomment-1238841882。[25] S. Ceri、G. Gottlob 和 L. Tanca。1989。关于 Datalog，你一直想知道却不敢问的事。IEEE 知识与数据工程汇刊（1989）。[26] Yean-Ru Chen、Si-Han Chen 和 Shang-Wei Lin。2023。带硬件加速的 SMT 求解器。计算机辅助设计与集成电路系统汇刊（2023）。[27] Patrick Cousot 和 Radhia Cousot。1977。抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型。载于 POPL。[28] Patrick Cousot 和 Nicolas Halbwachs。1978。自动发现程序变量之间的线性约束。载于 POPL。[29] Datadog。2024。网络性能监控。https://www.datadoghq.com/product/network-monitoring/network-performance-monitoring/。[30] Leonardo De Moura 和 Nikolaj Bjørner。2008。Z3：一种高效的 SMT 求解器。载于 TACAS。[31] Mugdha Deokar、Jingyang Men、Lucas Castanheira、Ayush Bhardwaj 和 Theophilus A. Benson。2024。关于 eBPF 应用开发挑战的实证研究。载于 ACM SIGCOMM eBPF 与内核扩展研讨会。[32] Alan AA Donovan 和 Brian W Kernighan。2015。Go 编程语言。Addison-Wesley Professional。[33] Robert W Floyd。1993。为程序赋予含义。载于《程序验证：计算机科学中的基础问题》。Springer。[34] Stanislav Fomichev、Eric Dumazet、Willem de Bruijn、Vlad Dumitrescu、Bill Sommerfeld 和 Peter Oskolkov。2024。用 EDT 和 BPF 替代 HTB。https://netdevconf.info//0x14/pub/papers/55/0x14-paper55-talk-paper.pdf。[35] 开放信息安全基金会。2023。Suricata。https://github. com/OISF/suricata。[36] Guillaume Fournier 和 Hemanth Malla。2023。eBPF 程序谋杀谜案。https://www.youtube.com/watch?v= YK7GyEJdJGo。[37] Guillaume Fournier 和 Hemanth Malla。2023。Datadog 的 eBPF 之旅：问题与点滴见闻。https: //lpc.events/event/17/contributions/1582/attachments/1328/2673/Troubles%20and%20Tidbits%20from%20Datadog%E2%80%99s%20eBPF%20Journey%20v2.pdf。[38] Lucas Freire、Miguel Neves、Lucas Leal、Kirill Levchenko、Alberto Schaeffer-Filho 和 Marinho Barcellos。2018。通过基于断言的验证发现 P4 程序中的错误。载于 SOSR。[39] Michael R Garey 和 David S Johnson。1979。计算机与难解问题。第 174 卷。旧金山：Freeman。[40] Elazar Gershuni、Nadav Amit、Arie Gurfinkel、Nina Narodytska、Jorge A Navas、Noam Rinetzky、Leonid Ryzhyk 和 Mooly Sagiv。2019。对不可信 Linux 内核扩展进行简单而精确的静态分析。载于 PLDI。[41] Rachael Graham。2024。IBM Cloud Kubernetes Service 现提供多组 Calico 网络策略，以在公共网络和专用网络上隔离集群。https://www.ibm.com/blog/isolating-clusters-with-calico-policies-in-ibm-cloud-kubernetes-service/。[42] D. Gries。1981。《编程科学》。Springer New York。https://books.google.co.in/books?id=5fUYAQAAIAAJ [43] Matthew S. Hecht。1977。《计算机程序流分析》。Elsevier Science Inc。[44] C. A. R. Hoare。1969。计算机编程的公理基础。ACM 通讯（1969）。[45] Alfred Horn。1951。关于在代数直和中为真的语句。符号逻辑杂志（1951）。[46] Roberto Ierusalimschy。2006。Lua 编程。Roberto Ierusalim- schy。[47] Rishabh Iyer、Katerina Argyraki 和 George Candea。2022。《网络功能的性能接口》。载于 USENIX NSDI。[48] John B. Kam 和 Jeffrey D. Ullman。1976。《全局数据流分析与迭代算法》。发表于《ACM学报》（1976）。[49] John B Kam 和 Jeffrey D Ullman。1977。《单调数据流分析框架》。发表于《信息学学报》（1977）。[50] Linux 内核作者。2024。《经典 BPF 与 eBPF》。https://docs.kernel.org/bpf/classic_vs_extended.html。[51] Brian W Kernighan 和 Dennis M Ritchie。2002。《C 程序设计语言》（2002）。[52] Gary A. Kildall。1973。《一种统一的全局程序优化方法》。载于 POPL。[53] Marios Kogias、Rishabh Iyer 和 Edouard Bugnion。2020。《无悔绕过负载均衡器》。载于第 11 届 ACM 云计算研讨会论文集。[54] Hsuan-Chi Kuo、Kai-Hsun Chen、Yicheng Lu、Dan Williams、Sibin Mohan 和 Tianyin Xu。2022。《经过验证的程序也能玩转优化：通过验证后合并来优化内核扩展》。载于 EuroSys。[55] l3af 项目。2023。eBPF 软件包仓库。https://github.com/l3af-project/eBPF-Package-Repository。[56] Chris Lattner 和 Vikram Adve。2004。《LLVM：用于终身程序分析与转换的编译框架》。载于 IEEE CGO。[57] Guyue Liu、Hugo Sadok、Anne Kohlbrenner、Bryan Parno、Vyas Sekar 和 Justine Sherry。2021。《别拆我的链：可审计的网络功能服务链》。载于 USENIX NSDI。[58] Jed Liu、William Hallahan、Cole Schlesinger、Milad Sharif、Jeongkeun Lee、Robert Soulé、Han Wang、Călin Caşcaval、Nick McKeown 和 Nate Foster。2018。《 p4v：面向可编程数据平面的实用验证》。载于 2018 年 ACM 数据通信特别兴趣小组会议论文集。[59] Boon Thau Loo、Tyson Condie、Minos Garofalakis、David E. Gay、Joseph M. Hellerstein、Petros Maniatis、Raghu Ramakrishnan、Timothy Roscoe 和 Ion Stoica。2009。《声明式网络》。发表于 ACM 通讯（2009）。[60] Nuno Lopes、Nikolaj Bjørner、Nick McKeown、Andrey Rybalchenko、Dan Talayco 和 George Varghese。2016。《在 P4 网络中自动验证可达性与良构性》。技术报告（2016）。[61] Dana Lu、Boxuan Tang、Michael Paper 和 Marios Kogias。2024。《面向 eBPF 程序的功能验证》。载于 ACM SIGCOMM eBPF 与内核扩展研讨会。[62] Jinsong Mao、Hailun Ding、Juan Zhai 和 Shiqing Ma。2024。《Merlin：面向性能与紧凑性的 eBPF 代码多层优化》。载于 ASPLOS。[63] Nicholas D Matsakis 和 Felix S Klock。2014。《 Rust 语言》。发表于 ACM SIGAda《Ada 通讯》（2014）。[64] Meta。2023。Katran。https://github.com/facebookincubator/katran。[65] Meta。2024。《开源 Katran：可扩展的网络负载均衡器》。https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/。[66] Sebastiano Miano、Alireza Sanaee、Fulvio Risso、Gábor Rétvári 和 Gianni Antichi。2022。《软件数据平面的领域专用运行时优化》。载于 ASPLOS。[67] Miguel Neves、Lucas Freire、Alberto Schaeffer-Filho 和 Marinho Barcellos。2018。《使用断言在可行时间内验证 P4 程序》。载于 ACM CoNEXT。[68] Richard O'Keefe。2009。《Prolog 编程技艺》。MIT出版社。[69] Gergely Orosz。2023。《 Datadog 五百万美元故障内幕（真实世界工程挑战 #9）》。https://newsletter.pragmaticengineer.com/p/inside-the-datadog-outage。[70] Solal Pirelli、Akvil˙e Valentukonyt˙e、Katerina Argyraki 和 George Candea。2022。《网络功能二进制文件的自动验证》。载于 USENIX NSDI。

[71] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Amarasinghe。2013。《Halide：用于优化图像处理流水线中的并行性、局部性和重新计算的语言与编译器》。载于 PLDI。[72] H. G. Rice。1953。《递归可枚举集及其判定问题的分类》。发表于《美国数学会汇刊》（1953）。[73] J. A. Robinson。1965。《基于归结原理的机器导向逻辑》。发表于《ACM学报》（1965）。[74] Michel F Sanner 等。1999。《Python：一种用于软件集成与开发的编程语言》。发表于《分子图形与模型学报》（1999）。[75] Michael Scott。2000。《编程语言实践》。Morgan Kaufmann。[76] Alexei Starovoitov。2024。eBPF系统调用。https://docs.kernel.org/userspace-api/ebpf/syscall.html。[77] Leon Sterling 和 Ehud Y Shapiro。1994。《Prolog 的艺术：高级编程技术》。MIT出版社。[78] Radu Stoenescu、Dragos Dumitrescu、Matei Popovici、Lorina Negreanu 和 Costin Raiciu。2018。《使用 Vera 调试 P4 程序》。载于 ACM SIGCOMM。[79] Bjarne Stroustrup。1994。《C++ 的设计与演化》。培生印度教育。[80] Bjarne Stroustrup。2022。《C++ 导览》。Addison-Wesley Professional。[81] Bingchuan Tian、Jiaqi Gao、Mengqi Liu、Ennan Zhai、Yanqing Chen、Yu Zhou、Li Dai、Feng Yan、Mengjing Ma、Ming Tang、Jie Lu、Xionglie Wei、Hongqiang Harry Liu、Ming Zhang、Chen Tian 和 Minlan Yu。2021。《Aquila：一种可实际用于生产规模可编程数据平面的验证系统》。载于 ACM SIGCOMM。[82] Linus Trovalds。2024。验证器。https://github.com/torvalds/linux/blob/de927f6c0b07d9e698416c5b287c521b07694cac/Documentation/bpf/verifier.rst。[83] S VenkataKeerthy、Yashas Andaluri、Sayan Dey、Rinku Shah、Praveen Tammana 和 Ramakrishna Upadrasta。2022。《使用程序嵌入识别数据包处理算法》。载于亚太网络研讨会。[84] Marcos AM Vieira、Matheus S Castanho、Racyus DG Pacífico、Elerson RS Santos、Eduardo PM Câmara Júnior 和 Luiz FM Vieira。2020。《使用 eBPF 和 XDP 快速处理数据包：概念、代码、挑战与应用》。发表于 ACM计算调查（CSUR）（2020）。[85] Jan Wielemaker。2009。《面向知识密集型交互式应用的逻辑编程》。博士论文。阿姆斯特丹大学。http://dare.uva.nl/en/record/300739。[86] Jan Wielemaker、Tom Schrijvers、Markus Triska 和 Torbjörn Lager。2012。SWI-Prolog。发表于《逻辑编程理论与实践》（2012）。[87] Xiwei Wu、Yueyang Feng、Tianyi Huang、Xiaoyang Lu、Shengkai Lin、Lihan Xie、Shizhen Zhao 和 Qinxiang Cao。2025。《VEP：面向完整 eBPF 可编程性的两阶段验证工具链》。载于 USENIX NSDI。[88] Qiongwen Xu、Michael D Wong、Tanvi Wagle、Srinivas Narayana 和 Anirudh Sivaraman。2021。《为数据包处理合成安全高效的内核扩展》。载于 ACM SIGCOMM。[89] Yifei Yuan、Soo-Jin Moon、Sahil Uppal、Limin Jia 和 Vyas Sekar。2020。《NetSMC：用于有状态网络验证的定制符号模型检查器》。载于 NSDI 20。[90] Arseniy Zaostrovnykh、Solal Pirelli、Rishabh Iyer、Matteo Rizzo、Luis Pedrosa、Katerina Argyraki 和 George Candea。2019。《无需验证专业知识即可验证软件网络功能》。载于操作系统原理研讨会。[91] Kaiyuan Zhang、Danyang Zhuo、Aditya Akella、Arvind Krishnamurthy 和 Xi Wang。2020。《使用 Gravel 自动验证可定制的中间盒属性》。载于 USENIX NSDI。

A 附录

A.1 夜叉—问题断言执行

本节将通过三个断言（**A1**、**A2** 和 **A3**，定义于第 §2.2 节）在两个字节码上执行（第 §2.2 节中的清单 1 和 2），说明分析器和查询引擎的工作原理。

考虑针对防火墙字节码（清单 1）的断言 **A1** 和**A2** 。分析器构建字节码的控制流图，并通过应用表**1**中的传递函数规则提取网络上下文。具体而言，规则 **R4** 提取字节码更新的字段，规则 **R11** 和 **R12**提取访问的协议列表，而规则 **R14** 则收集每条路径中所有控制流图节点的网络上下文。在保持字节码控制流结构的同时，这些信息会存储在知识库中。随后，查询引擎（QE）执行这些断言：对于 **A1**，*updatesField* 会检索已更新的字段（即 TCP 源端口），由于列表非空，断言失败；对于 **A2**，*passes* 会收集所有返回操作为 XDP_PASS 的路径上的网络上下文，最后检索这些路径中处理的协议列表（即 TCP、UDP 和 ICMP）。由于列表中包含 ICMP，断言失败。

接下来考虑对一串字节码 NF1→ NF2 → NF3 应用断言**A3**，如清单 2 所示。我们假设操作员会提供新链或旧链中各网络功能的顺序。例如，在 bpfman 工具中， [4], 钩子点上的 NF 优先级决定了它们的执行顺序。传递函数规则（表1）**R3** 和**R4** 提取写入列表，即字节码更新的字段列表；规则**R8** 和**R9** 则提取读取列表，即字节码读取的字段列表。将传递函数规则应用于字节码后，提取出的网络上下文会存储在知识库中。在知识库上执行**A3** 时，QE 发现 NF2 的写入列表（即，skb->mark 字段）与 NF3 的读取列表重叠，这表明 NF2 和 NF3 之间存在依赖关系。了解尚待部署的 NF 链中的依赖关系，有助于运维人员发现意外交互，并在服务中断发生前加以防范。

使用 bpfman 工具 [4],，我们在 XDP 挂钩点部署了 NF1 和 NF3，并在部署 NF2 之前执行了断言**A3**。附录 A.8 中的图 9 展示了在虚拟接口处 NF 链中间部署示例字节码的情况。

A.2 报头识别

我们应用表 1 中的规则 R11 和 R12，分别查找执行报头识别以及在访问报头前进行内存边界检查的指令。具体而言，在 R11 中，如果寄存器指向一个用于识别[9] 的字段

[9]isTailField 是规范中的一个辅助函数，用于检查目标寄存器是否引用了当前协议中用于标识下一个报头的头部字段。

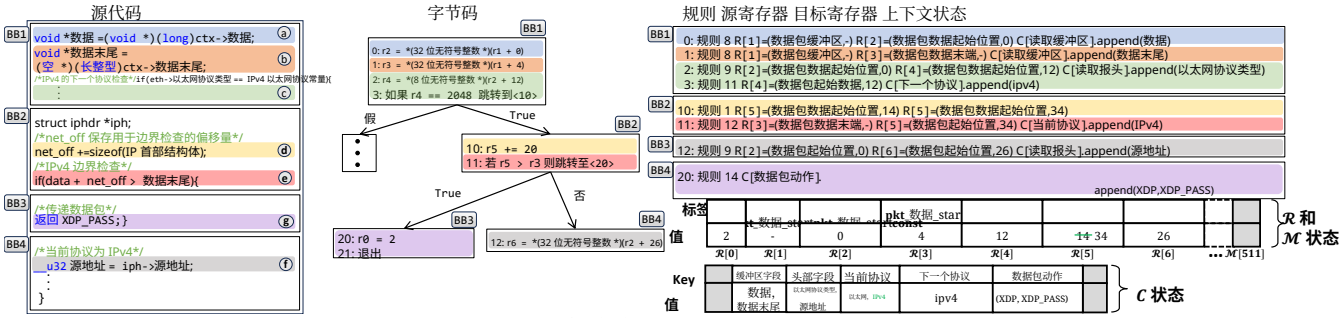

图 8. 使用先前识别的报头中的特定字段进行报头识别。

下一个报头（例如，ipv4.proto），则根据立即值识别下一个报头的协议（例如，TCP 对应 6）。图 8 展示了说明这些步骤的运行示例。这里，我们使用支持函数“获取协议名称spec”检索下一个报头的协议，并将其存储在 C 中，键为 **next_proto**。通过 R12，我们确认字节码会访问下一个报头，方法是确保存在与内存边界检查相关的指令。如果存在，我们就用 next_proto 更新 curr_proto。然而，用于识别报头和执行内存边界检查的字节码指令分散在多个基本块中，因为内存边界检查失败时，数据包处理会停止。因此，需要将协议上下文（即，next_proto 和 curr_proto）转发到后继基本块，这通过前文所述的状态传播来完成。

A.3 讨论

运行时输入。 Yaksha-Prashna 旨在针对表示扩展伯克利包过滤器字节码网络上下文的事实运行查询。Yaksha-Prashna 的分析器和语言可以扩展，以便在运行时理解或断言特定数据包头部及其映射中的一组键值条目的数据包处理行为。例如，“同一流中的所有数据包都应转发到同一目的地”。这有助于 (1) 在运行时调试连接问题和性能问题，以及 (2) 区分意外的数据包处理行为（例如丢弃）究竟是由字节码中的错误导致，还是由映射中的匹配规则不正确导致。

识别数据包处理算法。 目前，Yaksha-Prashna 分析器通过追踪数据包缓冲区偏移量，提取被访问或更新的报头字段或元数据字段。由于静态分析技术的局限性 [72], , 我们只能处理可表示为可判定规则的场景。该方法可以扩展到识别由字节码上的机器学习技术驱动的数据包处理算法（例如哈希、DDoS 检测、负载均衡、加密）。这使得网络功能审计能够确保字节码按预期处理数据包 [57]，或提供将处理卸载到加速器的机会 [83]。

查询引擎。 由于依赖 SWI-Prolog 运行时，Yaksha-Prashna 得以受益于数十年来的研究成果这些研究致力于优化逻辑范式语言中的查询求解 [45, 73, 85]。因此，它拥有高度可扩展且轻量的运行时系统，查询执行仅需几微秒，内存占用也很小。

与随机猜测的比较。 由于涉及的组合空间庞大，随机猜测使识别网络上下文变得愈发复杂。设想一段字节码，其中有 p 种协议、 f 个字段，它们与 m 个映射和 h 个辅助函数交互。用户必须在 $\binom{p_{max}}{p} \times \binom{f_{max}}{f} + \binom{m_{max}}{m} + \binom{h_{max}}{h}$ 种可能的组合中进行猜测。例如，考虑 NF9，其中包含（$p = 6$, $f = 10$, $h = 2$, $m = 9$）和（$p_{max} = 20$, $f_{max} = 15$, $m_{max} = 10$, $h_{max} = 50$），总计 $\binom{20}{6} \times \binom{15}{10} + \binom{10}{2} + \binom{50}{9} \approx 2.6 \times 10^9$ 种组合。如果网络上下文未知，随着搜索空间扩展到（$2^{p_{max}-1} + 1$）×（$2^{f_{max}-1} + 1$）+（$2^{m_{max}-1} + 1$）+（$2^{h_{max}-1} + 1$）$\approx 5.6 \times 10^{14}$种组合，问题将呈指数级增长（扩展伯克利包过滤器有 211 个辅助函数；当（p, f, m, h）分别已知和未知时，组合数分别为 2×10^{15}和 1.64×10^{63}）。这凸显了识别网络上下文的挑战，以及在可能性如此之多的情况下进行随机猜测的困难。借助 Yaksha-Prashna，可以显著减少所需的时间和精力，这是相较于随机猜测的一大优势。

A.4 Yaksha-Prashna 语言谓词

1. **字段谓词** *(字段_谓词)*：检查网络功能是否读取或写入特定的报头字段或缓冲区字段。如果字段参数是变量，则该谓词会返回所有被读取或更新的字段列表。

2. **映射谓词**（映射_操作_谓词，相关_映射-_谓词）：前者谓词检查对映射执行的操作，后者谓词检查映射之间是否存在可能的关联（例如，映射 A 的查找结果是映射 B 的键）。3. **动作谓词**（数据包_动作_谓词）：检查网络功能是否会针对特定协议执行丢弃数据包或重定向数据包等动作。如果没有提及协议详情，则返回程序路径列表（P）及其关联的网络上下文。4. **辅助谓词**（辅助函数_谓词）：检查网络功能是否调用特定的辅助函数。

5. **协议谓词**（协议_谓词）：检查网络功能是否处理特定协议。
6. **顺序谓词**（顺序_谓词）：此谓词返回 NF 链中某个 NF 的前驱或后继。

A.5 Yaksha-Prashna 的操作语义

表 5. Yaksha-Prashna 的操作语义。

规则头	规则体
读取字段(NF, Fld):-	读取_报头_字段(NF, _, Fld); 读取_缓冲区_字段(NF, _, Fld).
更新字段(NF, Fld):-	写入_报头_字段(NF, _, Fld); 写入_缓冲区_字段(NF, _, Fld).
后继网络功能(Nf, SNf):-	边(Nf, SNf);边(Nf, IntNf), 后继网络功能(IntNf, SNf)。
前驱NF(Nf, PNf):-	边(PNf, Nf); 边(PNf, IntNf), 前驱NF(Nf, IntNf)。
传递(Nf, 钩子, [(Fld, Val)]):-	返回_动作(Nf, 钩子,"XDP_PASS", [(Fld, 值)]
丢弃(Nf, 钩子, [(Fld, Val)]):-	返回_动作(网络功能, 钩子,"XDP_丢弃", [(Fld, 值)]
映射写入(网络功能, 映射, Fld):-	写入_到_映射(Nf, _, Fld，映射)。
映射查找(Nf, 映射):-	从_映射(Nf, _, 映射)。
相关映射(Nf, MapA, MapB):-	相关_映射(Nf, _, MapA, MapB).
访问协议(Nf, Fld, Val):-	协议_已访问(Nf, _, Fld, Val).
调用辅助函数(Nf, Helper):-	已调用_辅助函数(Nf, _, Helper).

A.6 查询执行工作流

我们使用 updatesField(Nf, Fld) 查询和知识库中的相关事实，说明回答查询的步骤。

规则： updatesField(Nf, Fld):- write_header_field(Nf,_, Fld).**事实：** write_header_field("firewall_kern_kern/xdp", node_3, IPv4 目的地址).

Q-谓词匹配： 查询中的每个 Q-谓词都会与已定义规则的头部进行比较。查询引擎会评估与匹配项对应的规则体。规则中的所有谓词（R-Pred 谓词）都必须满足，该规则才会返回 true。对于上述查询，Q-谓词 updatesField 将与相应规则匹配，以评估规则体。

事实比较：R-Pred 谓词会与存储在知识库中的事实进行匹配。要使规则成立，规则体中的每个 R-谓词都必须在知识库中找到对应的事实。在上面的示例中，网络功能防火墙_kern/xdp 会更新数据包头部中的 IPv4 目的地址字段这一事实，满足了针对 updatesField 的规则，并将具体节点 node_3抽象化。规则中的下划线（'_'）充当匿名变量，因为节点 ID 的值与匹配无关。这种抽象简化了查询过程，只需关注字段 *Fld* 是否由 *Nf*更新，而不必考虑具体在哪个块中更新。

变量处理： 如果查询包含变量，引擎会从知识库中检索所有满足规则的值（即，替换为检索到的值后，规则体中的 R-谓词将成立）。例如，如果将上述查询修改为询问哪些网络功能会更新 IPv4 目的地址（即，updatesField(Nf," ipv4.dst")），引擎将返回 firewall_kern/xdp，因为它符合网络功能会更新目的 IP 字段这一事实。

结果汇总与链接：匹配过程完成后，系统会收集所有满足查询谓词的值，并汇总结果。如果查询包含多个谓词，则会依次评估每个谓词，并遵循逻辑与/或运算。例如，如果查询要求找出既更新字段（updatesField）又访问特定协议（访问协议）的网络功能，引擎会先找出更新该字段的网络功能，然后根据协议访问情况进行筛选，确保只返回满足所有条件的网络功能。

A.7 检索与断言查询

表 6. 检索与断言查询列表。星号 **(*)** 表示这些查询包含递归谓词（即，successorNF 和 predecessorNF）。

检索查询		断言查询	
Q1(R)	映射写入(**网络功能**, _ ,**Fld**)。	Q1(A)	映射写入(**<网络功能3>**, _ , **IPv4 源地址**)。
Q2(R)	更新字段(**Nf**，**Fld**)。	Q2(A)	更新字段(**<网络功能8>**，**IPv4 目的地址**)。
Q3(R)*	前驱NF(**Nf**，**SNf**), 传递(**SNf**，**钩子**，**[字段, Arg]**)。	Q3(A)*	前驱NF(**<网络功能15>**,**<网络功能10>**), 传递(**<网络功能10>**, **XDP**, [*,*]).
Q4(R)*	更新字段(**网络功能**, **xdp_md.data**）， 后继网络功能(**Nf, SNf**), 读取字段(**SNf,xdp_md.data**)。	Q4(A)*	更新字段(**<网络功能15>, IPv4 源地址**), 后继网络功能(**<网络功能15>**, **<网络功能12>**), 读取字段(**<网络功能12>, IPv4 源地址**)。
Q5(R)*	更新字段(**网络功能**,**Fld**), 后继网络功能(**网络功能**,**SNf**), 读取字段(**SNf,Fld**)。	Q5(A)*	更新字段(**<网络功能10>, xdp_md.data**), 后继网络功能(**<网络功能10>**, **<网络功能12>**), 读取字段(**<网络功能12>,xdp_md.data**).
Q6(R)	读取字段(**网络功能**,**Fld**), 读取字段(**网络功能**,**Fld**).	Q6(A)	读取字段(**<网络功能6>**,**tcp.flags.syn**), 读取字段(**<网络功能6>**,**tcp.flags.ack**).
Q7(R)	调用辅助函数(**网络功能**, **辅助函数**).	Q7(A)	调用辅助函数(**<网络功能3>**, **bpf_map_update_elem**)。
Q8(R)	访问协议(**网络功能**,**Fld**,**协议**), 丢弃(**网络功能**,**钩子**,[**Fld**,**Arg**]).	Q8(A)	访问协议(**<网络功能15>**,**以太网类型**,**IPv4**), 丢弃(**<网络功能15>**, **XDP**,[*,*]).

A.8 使用 bpfman 进行字节码部署



(a) 两个 eBPF 程序分别以优先级 (-p) 10 和 12 挂载。



(b) 一个新的 eBPF 程序以优先级 (-p) 11 挂载。

图 9. bpfman 工具将多个 eBPF 程序挂载到挂钩点（即“eth02”接口）的工作流程。

bpfman 是一款用于管理本地主机和 Kubernetes 上 eBPF 程序的工具。它支持加载、卸载、附加和分离 eBPF 程序。借助 libxdp 库 [8]，它还支持在单个接口上同时运行多个 eBPF 程序。图 9 展示了 bpfman 的工作流程。eBPF 字节码被加载到内核中，并附加到特定的挂钩点。在挂钩点，bpfman 使用一种称为分发器的特殊管理程序，按照操作员为每个字节码设置的优先级（数值越小，优先级越高）以特定顺序执行程序。分发器程序会维护一个配置结构，用于记录已附加字节码的数量、它们的优先级、链调用动作以及其他元数据。

链调用动作决定数据包是继续处理还是终止处理。附加新的字节码时，会创建一个包含更新后程序列表的新分发器。系统会以原子方式将旧分发器替换为更新后的分发器，从而确保更新过程无缝进行，并在不停机的情况下强制执行正确的运行顺序。

A.9 CFG生成的时间和内存开销

A.10 比较概述

我们将 Yaksha-Prashna 与两项最先进的研究工作进行了比较：Klint [70] 和 DRACO [61]。Klint 是一种验证工具，用于检查网络功能二进制文件是否符合以 Python 编写的规范。网络功能可以使用 C 或 Rust 等不同语言编写，并采用 DPDK 或扩展伯克利包过滤器等框架。DRACO 是另一种验证工具，使用规范和断言来验证基于扩展伯克利包过滤器的网络功能，但要求

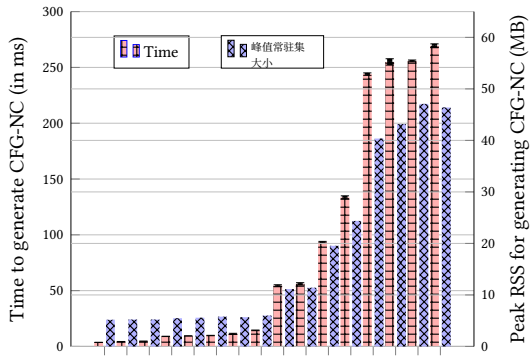

图10。生成 CFG-NC 所消耗的时间（左侧 y 轴）和内存（右侧 y 轴）

提供源代码才能进行验证。比较分为两部分：一部分专门讨论 Klint，另一部分讨论 DRACO，分别涉及相应类别的网络功能及用于验证的属性。

A.10.1 已实现网络功能的详情及为与 Klint 比较而提取的属性。 为进行比较，我们评估了网络功能是否符合其规范，这与 Klint 对基于 C 语言的二进制文件进行验证的方式类似。为进行此次比较，Yaksha-Prashna 需要：（1）网络功能的 eBPF 字节码；（2）用于验证的属性。由于 Klint 的网络功能以 C 语言编写，我们编写了语义等价的、基于 eBPF 的网络功能，并使用其字节码进行比较。它们是六种具有代表性的网络功能：路由器、网络地址转换、防火墙、网桥、负载均衡器和流量监管器。我们验证了原始网络功能与重写后的网络功能在语义上的等价性；验证方法见附录A.13。待验证的属性取自 Klint 规范。所有属性均属于类别 1（§2.2）。下文将详细介绍所开发的网络功能及提取出的属性。用于断言这些属性的 Yaksha-Prashna 断言见表 7。

(1) 路由器： 开发的路由器是一个基本的 IPv4 路由器，根据路由表转发有效的 IP 数据包。路由表通过 eBPF 映射实现，该映射将目标IP地址映射到接口。我们通过以下关键属性验证了路由器：

属性 1：预期的头部顺序。 IPv4 路由器应只允许以太网帧后跟 IPv4 数据包。所有其他数据包（例如，互联网协议第6版数据包）都必须丢弃。此属性有助于执行特定协议的数据包处理操作。为验证此属性，我们断言了一个高层断言，“只应转发以太网帧和 *IPv4* 数据包”，其具体断言见表 7 的 P1 行。

属性 2：IPv4 首部有效性。 路由器必须在处理数据包之前确保 IPv4 头部有效。为此，需要验证以下属性：(1) IP报头长度至少为20字节；(2) IP报头长度

表 7. 使用 Yaksha-Prashna 语言编写的属性断言和检索查询 e.

序号	属性（P1-P22）/检索查询（Q23-24）	Yaksha-Prashna 断言/检索查询
P1	预期的头部顺序	!传递("nf_id", xdp，[(变量, 变量)]), 访问协议("nf_id",_, "以太网"), 访问协议("nf_id","以太网类型","IPv4").
P2	IPv4 首部有效性	!passes("nf_id", xdp，[("IPv4 首部长度", >= 5),("IPv4 首部长度", <= "IPv4 总长度"), (...)]).
P3	NF 预期的首部字段更新	!传递("nf_id", XDP, [(变量, 变量)]), "IPv4 生存时间").
P4	最长前缀匹配	!传递("nf_id", XDP, [(变量, 变量)]), 映射查找("nf_id", "路由器_映射").
P5	数据包内容保留	!更新字段("nf_id", *).
P6	外部流过滤	!传递("nf_id",XDP,[("xdp_md.ingress_ifindex", 外部), (变量，变量)]), 映射查找("nf_id","firewall_map").
P7	记住内部流	!传递("nf_id", xdp，[("xdp_md.ingress_ifindex", !外部),(变量, 变量)]), 映射写入("nf_id", "防火墙_映射", *).
P8	更新内部的源 IP 和端口 flow	!传递("nf_id", xdp，[("xdp_md.ingress_ifindex", !外部), (变量，变量)]), 更新字段("nf_id",IPv4 源地址), 更新字段("nf_id", "TCP 源端口"); 更新字段("nf_id", "UDP 源端口").
P9	恢复目的 IP 地址和端口的 外部流	!传递("nf_id", xdp，[("xdp_md.ingress_ifindex", 外部),(变量, 变量)]), 更新字段("nf_id", "IPv4 目的地址"), 更新字段("nf_id", "tcp.dport"); 更新字段("nf_id", "udp.dport").
P10	所选传输端口不得是 入端口	!passes("nf_id", xdp，[(变量,变量)]), 映射查找("nf_id", "bridge_map"), 读取字段("nf_id", "xdp_md.ingress_ifindex").
P11	未知时广播	!丢弃("nf_id", xdp，[(变量, 变量)]), 映射查找("nf_id", "网桥_映射").
P12	学习过程	!传递("nf_id", xdp，[(变量, 变量)]), 映射写入("nf_id", “网桥_映射”, *).
P13	外部流量监管	!丢弃("nf_id", XDP, [("xdp_md.ingress_ifindex", !外部)]).
P14	后端存活性	!丢弃("nf_id", xdp,[("xdp_md.ingress_ifindex", !外部), (变量，变量)]), 调用辅助函数("nf_id", "bpf_ktime_get_ns"), 映射写入("nf_id","backend_map",*)。
P15	外部流量负载均衡	!传递("网络功能_id", xdp，[("xdp_md.ingress_ifindex", 外部), (变量，变量)]), 映射查找("网络功能_id", "后端_映射").
P16	丢弃所有分片数据包	传递("网络功能_id", xdp，[(变量, 变量)]), 读取字段("网络功能_id", "IPv4 分片").
P17	转发所有 ICMP 回显请求数据包	传递("nf_id", xdp，[(变量, 变量)]), 读取字段("nf_id", "icmp.request"), 更新字段("nf_id", "ICMP 字段"), 更新字段("nf_id", "IPv4 字段"), 更新字段("nf_id", "以太网.*").
P18	仅处理 TCP SYN 数据包	传递("nf_id", XDP, [(变量, 变量)]), 读取字段("nf_id", "tcp.syn").
P19	添加自定义重定向标头	传递("nf_id", xdp，[(变量,变量)]),调用辅助函数("nf_id","bpf_xdp_adjust_head"), 更新字段("nf_id","tcp.dataoffset").
P20	原样转发所有数据包	传递("nf_id", XDP, [(*, *)]). AND !更新字段("nf_id", *).
P21	相关映射	相关映射(["映射_一", "映射_two"]).
P22	NF依赖：写后读（RAW）， 读后写（WAR）和写后 写后写（WAW）	RAW 断言：updatesField("nf_id1", *)，后继网络功能("nf_id1", "nf_id2"), 读取字段("nf_id2", *). WAR 断言：读取字段("nf_id1 "，*)，后继网络功能("nf_id1 "，"nf_id2"), 更新字段("nf_id2", *). 写后写断言：更新字段("nf_id1", *)，后继网络功能("nf_id1", "nf_id2"), updatesField("nf_id2", *).
Q23	查找 NF 更新的数据包字段列表	updatesField("nf_id", Fld)。
Q24	查找一个网络功能 更新且其后继读取的数据包字段列表 （依赖于 RAW）	更新字段(“nf_id1”, Fld), 后继网络功能(“nf_id1”, “nf_id2”), 读取字段(“nf_id2”, Fld)。

不超过数据包的总长度；(3) 校验和正确；(4) TTL 不为零，等等。这些有效性检查对于识别并丢弃格式错误的 IP 数据包至关重要。我们使用诸如“每个发出的数据包都必须具有至少 *20* 字节的 *IP*报头长度”、“每个发出的数据包的 *IP* 首部长度都必须小于总长度”等高级断言来验证这些属性。具体断言见表 7 的 P2 行。

属性 3：NF 预期的首部字段更新。路由器的一项关键属性是递减每个发出数据包的生存时间（TTL）字段。此属性确保数据包不会在网络中无限循环。我们使用高层断言，“每个发出的数据包都必须更新 *IPv4* 的 *TTL*”，验证了此属性。其精确断言见表 7 的 P3 行。

属性 4：最长前缀匹配。IPv4 路由器必须在将目的 IP 地址映射到相应的转发接口时执行最长前缀匹配（LPM）。与精确匹配等替代方法相比，LPM 能够实现更快速、更高效的匹配。扩展伯克利包过滤器提供了一种专用映射类型来执行 LPM 匹配。为了验证这一属性，我们编写了两个高级断言：“所有发出的数据包都必须在路由器映射上执行查找”，以及 “路由器映射的映射类型必须是 *LPM*”。由于 Yaksha-Prashna 语言缺少用于断言映射

类型，我们只能使用第一个断言进行验证。具体断言见表 7 的 P4 行。

（2）防火墙：所开发的防火墙实现了有状态流量过滤，其中状态通过映射来捕获。它主要执行两项任务：（1）阻止未知外部流；（2）在必要时记住内部流，并将其转发。为验证防火墙的正确性，我们使用了以下三个关键属性：-

属性5：数据包内容保留。通常，防火墙不应修改其转发的数据包内容。该属性对于确保转发数据包的完整性非常重要。我们使用了动机部分（§2.2）中讨论的断言 A1 来验证该属性，该断言也见表 7 的 P5 行。

属性6：外部流过滤。防火墙的关键属性之一是阻止未知外部数据包。该属性至关重要，因为它确保了内部网络的安全性。为验证这一点，我们断言："如果在防火墙映射表中找不到条目，则不得转发所有外部数据包"。具体断言见表 7 的 P6 行。**属性7：记住内部流**。有状态防火墙要正常运行，就必须记住内部流的状态。该属性对于允许"已记住"的外部流（属性6）进入网络同样至关重要。为断言此属性，我们使用了一个高级断言："对于所有内部数据包，如果防火墙映射表中不存在流条目，则必须使用流信息更新映射。"具体断言见表 7 的 P7 行。

(3) 网络地址转换：开发的有状态网络地址转换实现了两项主要功能：(1) 对于内部流，将源 IP 地址和端口更新为公共IP地址和掩码端口号；(2) 对于外部流，将目标 IP 地址和端口恢复为原始内部IP地址和端口号。我们还验证了网络地址转换上的**属性 6**和**属性 7**，因为它们对于确保内部网络的安全以及保证网络地址转换的正确有状态行为至关重要。除此之外，我们还验证了另外两项关键属性：-

属性 8：更新内部流的源 IP 和端口。对于任何内部流，网络地址转换都必须将源IP地址更新为公共 IP，并对源端口进行掩码处理。这项属性非常重要，因为它允许多个内部设备由单个公共IP地址表示，同时借助掩码端口号，仍能唯一标识它们的流。为验证这一点，我们使用了一条高层断言："所有内部数据包的源*IP*地址和端口都必须更新"，其确切断言见表 7 中的 P8 行。

属性 9：还原外部流的目的 IP 地址和端口。 对于任何外部流，网络地址转换必须将目的 IP 地址和端口还原为原始的内部私有地址和端口。这对于确保外部到达网络地址转换的数据包能够被转发到正确的内部设备至关重要。为验证这一点，我们使用了一个高级断言，"所有已知外部数据包的目标 *IP* 地址和端口都必须更新"。具体断言见表 7 的 P9 行。

(4) 网桥：开发的网桥是一种 MAC 学习网桥，它根据目的 MAC 地址转发以太网帧。目的 MAC 地址与接口的映射通过映射表存储。我们使用以下三个关键属性验证了该网桥：

属性 10：所选传输端口不得是入端口。对于网桥，所选传输端口不得与接收帧的入端口相同。这项检查对于防止网络中出现不必要的循环至关重要。我们使用高级断言 "出站数据包的传输端口不得与入端口相同"验证了这一属性。具体断言见表 7 中的 P10 行。

属性 11：遇到未知帧时进行广播。如果网桥接收的帧与转发表中的任何条目都不匹配，则不得丢弃该帧，而应将其广播到除入接口之外的所有接口。即使网桥不知道合适的出接口，这项属性也能确保帧仍可到达预期目的地。为验证这一点，我们使用了两个高级断言："如果网桥映射查找失败，则不得丢弃数据包"，以及"数据包必须广播到除入接口之外的所有接口"。由于 Yaksha-Prashna 语言不支持用于断言数据包广播的谓词，我们只能验证第一个断言。具体断言见表 7 中的 P11 行。

属性 12：学习过程。学习过程会观察每个端口收到的帧的源地址，并更新转发表。这一过程非常重要，因为它使网桥能够动态构建转发表并高效转发帧，而不必依赖广播。我们使用以下高级断言验证了此属性："如果收到帧的源 *MAC* 地址不在网桥映射中，则必须相应地更新映射"。具体断言见表 7 中的 P12 行。

(5) 流量监管器：开发的流量监管器是一种有状态网络功能，为外部数据包实现按目的地划分的令牌桶，根据数据包大小有效限制流量。它使用映射跟踪状态，以 IP 地址为键，并将令牌桶详情（如令牌大小）存储为对应的值。我们使用以下属性验证了该流量监管器：-

属性 13：外部流量监管。流量监管器应仅对外部流量进行速率限制，并根据数据包大小更新状态，同时转发所有内部数据包。为了验证前一项属性，我们使用了高级断言 "所有外部数据包都必须执行查找

在限速器映射上并相应地更新它”，而后一个属性通过“所有内部数据包都必须转发”断言进行了验证。具体断言见表 7 中的 P13 行。

（6）负载均衡器：所开发的负载均衡器是谷歌名为“Maglev”的负载均衡器的扩展伯克利包过滤器实现。它基于一致性哈希从后端池中选取一个后端，并使用映射记住流与后端映射。它还使用另一个映射来跟踪存活后端，该映射在接收到来自后端的心跳数据包后进行更新。我们使用**属性5**验证了该负载均衡器，因为负载均衡器在处理数据包时不应修改数据包内容。除此之外，我们还验证了以下两个关键属性：-

属性14：后端存活性。如果数据包来自后端，则必须用当前时间更新存活后端映射，然后将其丢弃。这对于跟踪存活后端并避免将数据包转发到失效后端非常重要。我们使用高层断言验证了该属性，“所有被丢弃的后端数据包都必须用当前时间更新存活后端映射”。确切的断言见 表 7 的 P14 行。

属性15：外部流量负载均衡。外部数据包应转发到所选后端之一；如果后端已失效，则必须丢弃该数据包。我们使用以下高级断言验证了该属性：“如果后端处于运行状态，所有外部数据包都必须转发到该后端”。其精确断言见表 7 第 15 行。

A.10.2 与 DRACO 进行比较时提取的网络功能及属性详情。我们从表达能力方面将 Yaksha-Prashna 与 DRACO 进行了比较。我们提取了 DRACO 用于验证三个真实世界 eBPF 程序的属性：Katran [64], CRAB [53], Fluvia [2], ，以及一篇学术文献中的一个 eBPF 程序 hXDP FW [22]。所有属性均属于类别 1（§2.2）。下文将简要介绍这些网络功能及提取出的属性。Yaksha-Prashna 的断言见表 7。

Katran： Katran 是 Meta 开发的四层负载均衡器，使用 XDP 高效处理数据包。它旨在实现可扩展且低延迟的连接均衡。我们通过以下两个关键属性验证了 Katran：

属性16：丢弃所有分片数据包。Katran 的一项限制是无法处理分片数据包，因此会丢弃所有此类数据包。我们使用高级断言表达了这一属性：“不应转发任何分片数据包”，其确切断言见表 7 中的 P16 行。

属性 17：转发所有 ICMP 回显请求数据包。Katran 会执行许多特定协议的数据包处理操作。其中一项操作用于处理 ICMP 回显请求数据包。遇到此类请求数据包时，Katran 会将其转换为回显应答，同时更新相关的 ICMP、IP 和以太网头部字段，然后将其转发。我们用高层断言 “所有 *ICMP* 回显请求数据包都必须在更新相关的 *ICMP*、*IP* 和以太网头部字段后转发”来表述这一操作。确切的断言见表 7 中的 P17 行。

CRAB： CRAB 是另一种四层负载均衡器，仅在连接建立时参与处理。它处理 SYN 数据包以选择后端服务器，并通过连接重定向让客户端和服务器直接通信，从而降低延迟、避免瓶颈，同时支持复杂的有状态负载均衡。我们通过以下两个关键属性验证了 CRAB：

属性18：仅处理 TCP SYN 数据包。CRAB 仅在连接建立阶段参与处理，在此阶段处理 TCP SYN 数据包并丢弃所有其他数据包。为了表达这一属性，我们使用了高级断言：“所有转发的数据包都必须是 *TCP SYN* 数据包”，其确切断言见表 7 中的 P18 行。

属性19：添加自定义重定向标头。为避免介入客户端与服务器之间的数据通路，CRAB 利用连接重定向，使客户端与服务器之间直接通信，而不经过负载均衡器。所有发出的数据包都会附加自定义重定向标头，以启用此功能。我们使用高级断言“所有发出的数据包都必须附加自定义标头”来表述该属性。具体断言见表 7 的 P19 行。

Fluvia：Fluvia 是一个 IPFIX 导出器，它利用 XDP 收集与传入互联网协议第6版流相关的元数据，并将其导出到用户空间进行分析。我们使用以下属性验证了 Fluvia：-

属性20：原样转发所有数据包。流导出器应透明地收集流的元数据，同时不丢弃任何传入的数据包。此外，它也不应修改数据包内容。我们结合“所有数据包都必须通过”这一高级断言与**属性5**来表达这一要求。具体断言见表 7 的 P20 行。

hXDP 防火墙： hXDP 防火墙是学术文献中的一种简单有状态防火墙[22],，其实现的功能与第 §A.10.1 节讨论的防火墙类似。除验证防火墙的关键属性（属性5-7）外，我们还验证了以下特定属性，该属性是 hXDP 防火墙实现所特有的：-

属性 21：相关映射。相关映射常用于在不同映射之间共享状态。当一个映射操作的返回值被用作另一个映射操作（例如查找或更新）的参数时，这两个映射就被视为相关映射。清单 1 第 13-15 行展示了相关映射的一个实例：从流_*ctx*_表 中进行映射查找所得的值被存储在流_叶节点 中（第 13 行），

随后用作键，通过发送_端口 映射重定向数据包（第 15 行）。我们使用专门的映射谓词相关_映射 来捕获相关映射，如表 7 第 P21 行所示。

除了上述属性之外，DRACO 还断言了一些通用的网络功能属性，例如“以太网源地址在所有数据包中保持不变”、“所有互联网协议第6版数据包都应被丢弃”等。这些属性同样可以用 Yaksha-Prashna 语言表达。

A.11 查询执行

Q-谓词数量。 查询中的谓词越多，就必须遍历知识库中的越多事实来解析查询，因此完成查询所需的时间也越长。例如，在两个非递归查询 Q1 和 Q6 中，前者包含一个 Q-谓词，后者包含两个 Q-谓词；当它们在由十六个网络功能组成的链所生成的事实上执行时，前者完成所需的时间（$0.59\mu s$）少于后者（$2.5\mu s$）。同样，包含两个 Q-谓词的 Q3 完成得比包含三个 Q-谓词的 Q4 和 Q5 更快（分别为 $1.4\mu s$、$1.6\mu s$ 和 $3.3\mu s$）。我们还观察到，与包含三个或更少 Q-谓词的查询 Q(1,2) 和 Q(7,8) 相比，包含三个以上 Q-谓词的组合查询 Q(3,4)、Q(4,5)、Q(3,5)、Q(3,4,5) 和 Q(6,7,8) 完成所需的时间相对更长（$6.6\mu s$、$8.9\mu s$、$4.7\mu s$ 和 $9.6\mu s$；前者所需时间相对较短，分别为 $1.48\mu s$ 和 $2.25\mu s$）；这些查询均在对应于由十六个网络功能组成的链的事实上执行。此外，时间随 Q-谓词数量线性增长。**网络功能链长度**。 我们观察到，随着网络功能链长度增加，需要分析的指令数量也随之增加，因此会为链中的每个网络功能生成大量事实。因而，通常情况下，单个查询和组合查询的查询执行时间都会增加[10]。换言之，网络功能链越长，执行查询所需的时间就越多。在由 2、8 和 16 条网络功能链生成的事实上执行全部 15 个查询，分别需要 $27.9\mu s$、$33.2\mu s$ 和 $48.8\mu s$。图 6 展示了在这些网络功能链各自生成的事实上，不同查询及每个查询中的谓词所需的时间。

A.12 内存占用

我们评估内存占用，以了解系统在生成 CFG-NC 和执行查询时所使用的内存。我们采用峰值常驻集大小（RSS，单位为 MB）作为计算内存占用的指标。峰值 RSS 是进程在整个生命周期中消耗的最大内存量。我们使用 GNU time 命令 [3]来测量峰值 RSS 值。我们在以下情况下计算内存占用：(i) 生成表4中列出的微基准网络功能的 CFG-NC，以及 (ii) 执行表6中列出的各个断言查询（Q1–Q8）及其组合。

生成 CFG-NC 时的峰值常驻集大小。 我们在图10（右）中展示了生成 CFG-NC 时测得的峰值常驻集大小。我们分析了峰值常驻集大小与控制流图（CFG）中的路径数量以及分析的指令数量（NAI）之间的相关性。我们观察到，内存用量会随着路径数量和指令数量的增加而上升。值得注意的是，对于规模最大的微基准网络功能（NF13–NF16），我们的系统占用的内存不到 50 MB。

查询执行时的峰值常驻集大小。 我们观察到，对于给定的一组事实，回答不同查询时的峰值常驻集大小几乎保持不变。具体而言，对于由全部 16 个网络功能生成的事实，查询引擎需要占用 13.25–13.42 MB 内存来回答我们的查询。

与 Klint 的比较。 图7（右）展示了验证期间的峰值常驻集大小。Klint 需要 190–1000 MB，而 Yaksha-Prashna 仅使用 20–50 MB。尽管 Yaksha-Prashna 存储了整个字节码的网络上下文，但其内存占用仍比 Klint 少 90–95%。

A.13 与 Klint 对比的改写 eBPF 程序的语义等价性

为确保与 Klint 对比时的语义等价性，我们采用了三步方法。

保逻辑改写。 我们手动将 Klint 的基于 C 的网络功能改写为兼容 XDP 的 eBPF 程序。在此过程中，我们仔细保留了原始的数据包处理逻辑、控制流和状态更新。随后，我们手动检查了改写后的程序，以确认它们实现了等效的数据包处理行为。

等效简化。 当 eBPF 的限制使某些组件（例如复杂的哈希函数）无法直接转换时，我们对两种实现都进行了等效简化。例如，我们在两个版本中都将复杂的哈希函数替换为较简单的替代方案，以保持计算复杂度相当，同时保留其功能意图。

配置对齐。 我们对齐了不同实现中的配置参数，包括接口数量和表/映射大小，以避免因资源分配不同而产生差异。

我们指出，程序等价性通常是不可判定的 。因此，我们的验证依赖于仔细的人工检查和受控对齐，而非形式化等价性检查。

[10]由于 Prolog 运行时引擎的索引和缓存机制经过优化，可能会出现一些例外情况。