Kgent：内核扩展大型语言模型代理

文章信息

Title	Kgent: Kernel Extensions Large Language Model Agent
Journal	*(10.1145/3672197.3673434)*
Authors	Zheng Yusheng,Yang Yiwei,Chen Maolin,Quinn Andrew
Pub.date	2024-08-04

参考阅读博客：简化内核编程：LLM驱动的eBPF工具 - eunomia

摘要

扩展的伯克利数据包过滤器 (eBPF) 生态系统允许扩展 Linux 和 Windows 内核，但由于需要了解操作系统内部结构以及 eBPF 验证程序强制执行的编程限制，因此编写 eBPF 程序具有挑战性。这些限制确保只有专家内核开发人员才能扩展其内核，从而使初级系统管理员、补丁制作者和 DevOps 人员难以维护扩展。本文提出了 Kgent，这是一种替代框架，它通过允许用自然语言编写内核扩展来减轻编写 eBPF 程序的难度。 Kgent 使用大型语言模型 (LLM) 的最新进展，根据用户的英语提示合成 eBPF 程序。为了确保 LLM 的输出在语义上等同于用户的提示，Kgent 采用了 LLM 授权的程序理解、符号执行和一系列反馈循环的组合。 Kgent 的主要创新之处在于这些技术的结合。特别是，该系统在一种新颖的结构中使用符号执行，使其能够将程序合成和程序理解的结果结合起来，并建立在LLM最近针对每项任务分别取得的成功的基础上。

为了评估 Kgent，我们为 eBPF 程序开发了一个新的自然语言提示语料库。我们表明，Kgent 生成的 eBPF 程序正确率为 80%，与 GPT-4 程序合成基线相比，提高了 2.67 倍。此外，我们发现 Kgent 很少合成“误报”eBPF 程序，即 Kgent 验证正确但手动检查显示输入提示在语义上不正确的 eBPF 程序。 Kgent 的代码可在 https://github.com/eunomiabpf/KEN 上公开访问。

关键词：大语言模型、eBPF、符号执行

📊 研究背景 & 研究目的

介绍

开发人员越来越多地承担修改和扩展操作系统内核以提高性能、安全性和可靠性，或为其系统引入新功能的任务。扩展伯克利数据包过滤器 (eBPF) 已成为扩展操作系统的事实上的方法，最近支持 Linux 和 Windows [11]。 eBPF 程序注入在现有内核逻辑之前或之后执行的新逻辑，以观察或修改内核的行为。 eBPF 程序最初用于跟踪网络流量，但生态系统现在提供了足够的功能来实现各种功能，包括性能监控 [17, 18]、入侵检测 [2, 21] 和特定于应用程序的逻辑 [15, 22, 35、37]。

不幸的是，eBPF 程序很难正确编写。实现 eBPF 程序需要深入了解内核内部结构，以确定在何处注入逻辑 [9]。此外，旨在防止不安全的 eBPF 程序在系统上执行的 eBPF 验证器给 eBPF 程序员带来了许多不幸的编程约束：程序只能使用有限的控制流（例如，循环必须具有常量边界）和有限的数据访问（例如，程序无法访问任意内存）。因此，eBPF 甚至不是一种图灵完备语言 [23]。

在本文中，我们提出了 Kgent，内核扩展 LLM 代理，可以减轻 eBPF 的难度。用户可以通过注入 Kgent 生成的 eBPF 程序来扩展其内核，而无需了解 eBPF 或内核的内部结构。

关键技术背景

少样本情境学习：采用上下文学习的系统通过向 LLM 添加最少的上下文数据来使 LLM 适应新的目标领域（例如，它们使用一小部分特别相关的输入来提示LLM称为“few shot”）并精心设计提示，而不是在大型数据集上训练自定义LLM。与传统替代方案相比，情境学习更具成本效益，并且可以更快地获取新数据。在新的合成领域中采用情境学习的关键挑战包括识别关键示例的语料库并制定指导基础LLM的即时策略[4, 26]。

> **反馈循环**：许多系统采用反馈驱动的方法来根据某些规范验证LLM的输出，并将反馈传递给模型以改进输出。例如，HarmonyOS 开发人员 [33] 和 TPUv4 设计人员 [29] 使用单元测试来评估 LLM 的输出是否正确。开发此类真实单元测试的基本挑战是手动制作一组能够充分指导 LLM 的单元测试，这可能与手动编写理想的合成输出一样困难。上述示例中的开发人员能够使用已经为其用例创建的单元测试。此外，自调试还包括一种通过使用 LLM 来解释合成程序的行为来提供反馈的机制。该系统通过迭代合成程序、解释合成程序检查其单元测试输出，然后迭代重新合成来细化其输出。

自动程序理解：自动程序理解无需开发人员的努力即可确定程序执行的属性。早期的工作 [24] 使用反例驱动的方法，该方法观察被测试的程序并得出适用于所有执行的不变量。最近的研究表明，LLM在程序理解任务中是有效的[1,12,13,25,34,38]，但他们的输出通常是程序的英文描述。

符号执行：自动程序验证的出现可以确保程序正确，而不需要开发人员编写大量测试。现有的解决方案有很多种，包括模型检查 [7, 28]、模糊测试 [14] 和符号执行 (symbex) [3, 19]。Kgent 使用 symbex 是因为符号执行已被证明在查找操作系统 [6] 和用户程序 [3, 19] 中的问题方面非常有效。

符号执行引擎对函数或程序的行为进行推理，以确定程序是否支持特定属性（例如，内存安全）。 symbex 引擎将符号值与程序中的每个变量相关联。当程序执行时，引擎收集符号值的约束（例如，变量 x 小于 3）。在执行期间，引擎使用 SMT 求解器来确定在给定引擎收集的约束的情况下是否可能违反给定属性（例如，“此函数将在此路径条件下执行？”）。因此，符号执行可以推理函数或程序的大类输入，而无需单独执行每个输入。

研究内容

系统设计

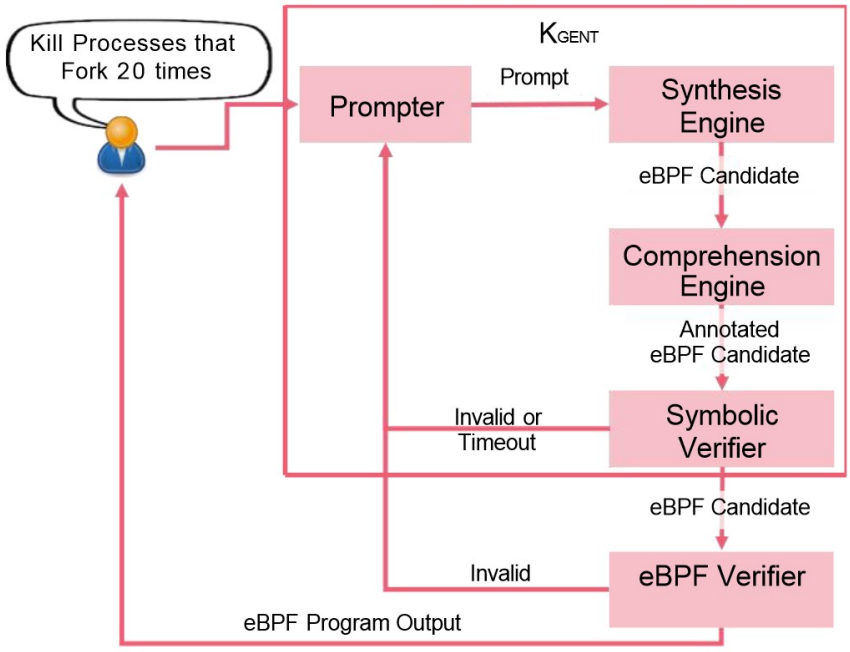


Figure 1: The Workflow of KGENT

(Zheng 等, 2024, p. 31) 图1描述了系统的关键组件以及它们如何交互

Kgent 由四个主要组件组成： **提示器**（第 3.2 节），负责构建生成 eBPF 程序的提示；**合成引擎**（第 3.3 节），负责在给定自然语言提示的情况下合成候选 eBPF 程序；**理解引擎**（第 3.4 节），负责为与候选程序交互的每个内核函数使用准确的霍尔逻辑条件来注释候选 eBPF 程序；**符号验证器**（第 3.5 节），负责确保候选 eBPF 程序支持霍尔逻辑注释。该系统还使用现有的 eBPF 验证器来确保 eBPF 候选者满足基本的安全标准。

Kgent 使用情境学习（第 2 节）来增强现有的LLM，而不需要训练全新的模型。这种设计使得 Kgent 能够保持 LLM 的不可知性。支持情境学习需要两项技术贡献：指导LLM生成正确输出的提示策略以及用于合成和理解引擎的新数据集。

Kgent 使用反馈驱动的方法来迭代地合成正确的 eBPF 程序。 Kgent 包括两个这样的反馈回路。首先，如果符号验证器确定来自合成引擎的候选 eBPF 程序不支持来自理解引擎的霍尔逻辑条件，或者如果符号验证器超时，则验证器会将失败传递回提示器。这个反馈循环允许两个引擎有额外的机会合成正确的输出。第二个反馈是从 eBPF 验证器到提示器的，当 eBPF 验证器未验证合成的 eBPF 程序是否安全时进行。

Kgent 的每个组件都采用了最先进的技术来解决 eBPF 程序合成问题。 Kgent 的主要新颖之处在于它结合了这些技术，从而提供了有用且高效的程序合成工具。特别是，Kgent 使用符号执行来结合程序合成和程序理解的结果，这是一种新颖的结构，使 Kgent 能够利用LLM在两项任务上分别展示的能力。

工作流程

在 Kgent 中合成 eBPF 程序的工作流程如下。用户向提示器发出提示，提示器将用户的提示转发到 Kgent 的合成引擎。合成引擎咨询LLM，根据用户的输入合成候选 eBPF 程序。Kgent 将候选 eBPF 程序传递给合成引擎，合成引擎咨询 LLM，使用候选 eBPF 程序中引用的每个内核函数的霍尔逻辑前置条件和后置条件来注释候选 eBPF 程序。 Kgent 将此带注释的 eBPF 候选传递给其符号验证器，该验证器验证合成的 eBPF 程序是否满足霍尔逻辑属性。如果符号验证器确定 eBPF 程序不支持语义属性，或者符号验证器超时，则符号验证器将其输出传递给提示器以开始 Kgent 的另一次迭代。如果符号验证器成功，它将 eBPF 程序传递给 eBPF 验证器以验证程序的安全属性。如果 eBPF 程序被认为不安全，eBPF 验证程序会将其错误消息传递给提示器以开始 Kgent 的另一次迭代。

如果 Kgent 在经过配置的试验次数（默认为 3）后未能合成经过验证的 eBPF 程序，系统将重新提示用户包含其他信息。有趣的是，我们观察到，包含小的额外语义提示（例如，暗示某些变量的预期大小）通常可以解决 Kgent 的合成问题。

提示器

提示器获取用户的输入提示并对其进行特殊格式化以传递给合成引擎。特别是，提示器添加了样板文本，指示 Kgent 生成为 bpftrace 框架 编写的 eBPF 程序。此外，提示器还会附加

通过所有反馈循环收到的当前合成任务的所有错误消息。例如，如果符号验证器在第一次迭代中失败并显示消息 failure1，而 eBPF 验证器在第二次迭代中失败并显示消息 failure2，则提示器将在将其发送到合成引擎之前将 failure1 和 failure2 附加到其消息中。

合成引擎

合成引擎采用自然语言提示并咨询LLM来生成候选 eBPF 程序。该引擎使用 LangChain [20] 作为与任意 LLM 交互的机制，这使得 Kgent 能够支持各种隐私-成本-性能权衡。

与其他系统 [5] 一样，合成引擎使用 VectorDB（例如 Milvus [31]）来实现上下文学习。引擎将来自 eBPFNLDataset 数据集（见下文）的提示-eBPF 对存储在其 VectorDB 中。在每个查询中，引擎使用 VectorDB 来识别与用户提示最相似的提示-eBPF 对，并将这些对作为正确输入-输出对的示例包含在对大语言模型的查询中。因此，Kgent 类似于少样本学习[32]。Kent 还在每次合成后更新其 VectorDB，这使得系统能够从成功和失败的 eBPF 合成中学习。

```
eBPFNLDataset——eBPF 合成数据集。合成引擎由 eBPFNLDataset 提供支持，这是一个包含 145 个自
```

通过在 LLM 提示符中指定所需的语法，我们发现合成引擎几乎总是合成正确的语法。我们的结果表明，它也经常合成正确的语义（第 5 节）。

理解引擎

理解引擎使用 LLM 提示（包括 eBPF 函数、开发人员提示和来自 KeRnelCompDataset 的条件）通过霍尔逻辑前置条件和后置条件来注释 eBPF 候选程序。这种方法允许引擎利用开发人员的提示，消除自动生成的 KeRnelCompDataset 中的不准确之处，并通过更新内部 VectorDB 从错误和成功中学习。实证评估第 5 节表明，理解引擎不会恶意调整前置/后置条件来同义反复地验证所有候选程序，这可能是由于 VectorDB 的学习速度较慢。

```
KeRnelCompDataset——内核函数的霍尔逻辑契约的数据集。为每个 eBPF 可检测内核函数手动创建霍尔
```

符号验证器

符号验证器使用符号执行来验证理解引擎生成的的带注释的 eBPF 候选程序。如果符号验证器确定程序支持断言语句，它将删除断言/假设语句并将候选 eBPF 程序传递给 eBPF 验证器。如果符号验证器发现断言语句不被支持或超时，则它将其错误消息传递给提示器。

eBPF验证器

Kgent 使用操作系统中现有的 eBPF Verifier。即eBPF验证器验证符号验证器产生的eBPF候选程序不包含无限循环或任意存储器访问。如果 eBPF 验证器无法确保候选程序是安全的，它将其错误消息传递回提示器。否则，eBPF验证器将eBPF候选程序传递给用户作为其最终合成的eBPF程序。

实施

我们在Python中使用4244 LOC实现Kgent、eBPFNLDataset和KeRnelCompDataset，我们向bpftrace编译器添加51 LOC以添加对假设和断言函数的支持。 Kgent 使用 SeaHorn [19] 作为其符号验证器，使用 Z3 [10] 作为其后端 SMT 求解器。 SeaHorn象征性地执行LLMV IR；Kgent 通过将理解引擎的输出编译为 LLVM IR 来使用 SeaHorn。

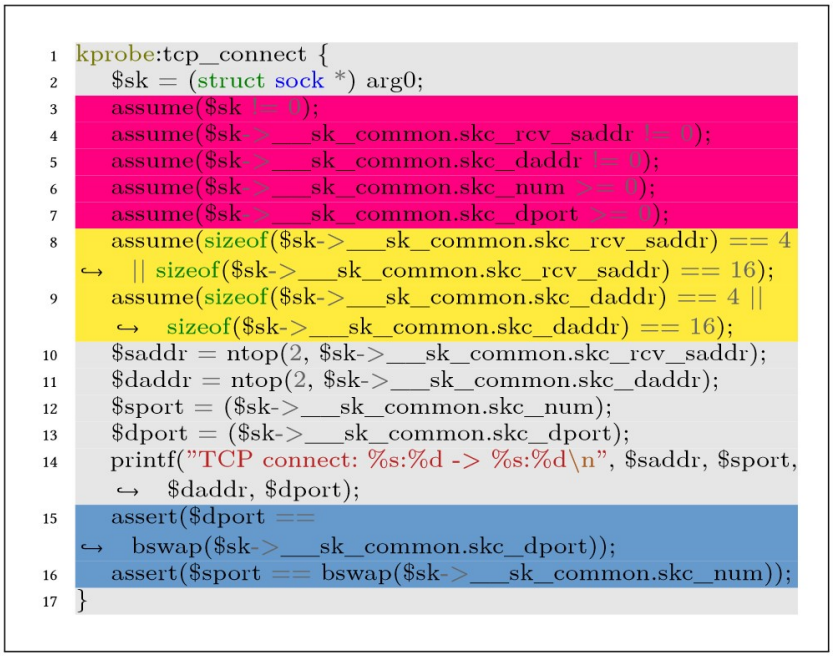
Kgent 支持多种 LLM，但默认使用 GPT-4 API。我们当前的 Kgent 原型仅适用于 Linux。Kgent 可以为 libbpf 和 bpftrace 合成 eBPF 程序，但它默认为 bpftrace，因为我们发现 Kgent 在合成 bpftrace 程序方面更有效（第 5 节）。

案例研究

在此案例研究中，用户希望向其终端显示有关与其 Web 服务器建立的所有 TCP 连接的信息，以便他们可以检测针对其进行的拒绝服务攻击。内核扩展提供了执行完整系统遥测的良好机制。然而，由于开发人员不是 Linux 内核专家，他们求助于 Kgent 来帮助他们构建 eBPF 程序。图2是开发者向Kgent发出的开发者提示。他们要求系统跟踪所有 IPv4 和 IPv6 tcp_connect 事件并指定所需的输出，包括每个端口的正确顺序。

Prompt: Write a bpftrace program to trace tcp_connect events for both IPv4 and IPv6 connection attempts, display the source and destination IP addresses and the source and destination ports in host byte order.

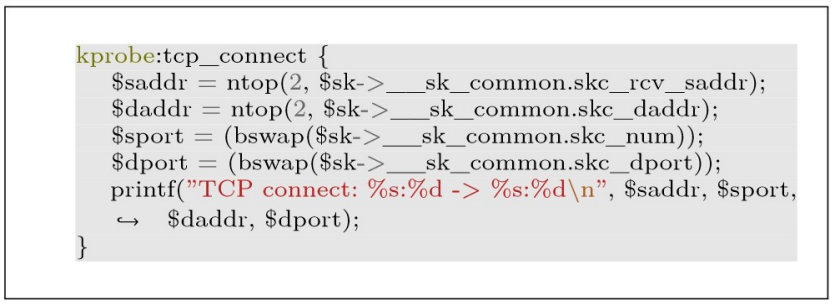
Figure 2: A prompt passed into KGENT instructing it to print basic connection information for all TCP connect attempts.



```
kprobe:tcp_connect {
    $sk = (struct sock *) arg0;
    assume($sk != 0);
    assume($sk->___sk_common.skc_rcv_saddr != 0);
    assume($sk->___sk_common.skc_daddr != 0);
    assume($sk->___sk_common.skc_num >= 0);
    assume($sk->___sk_common.skc_dport >= 0);
    assume(sizeof($sk->___sk_common.skc_rcv_saddr) == 4
↪   || sizeof($sk->___sk_common.skc_rcv_saddr) == 16);
    assume(sizeof($sk->___sk_common.skc_daddr) == 4 ||
    ↪   sizeof($sk->___sk_common.skc_daddr) == 16);
    $saddr = ntop(2, $sk->___sk_common.skc_rcv_saddr);
    $daddr = ntop(2, $sk->___sk_common.skc_daddr);
    $sport = ($sk->___sk_common.skc_num);
    $dport = ($sk->___sk_common.skc_dport);
    printf("TCP connect: %s:%d -> %s:%d\n", $saddr, $sport,
    ↪   $daddr, $dport);
    assert($dport ==
↪   bswap($sk->___sk_common.skc_dport));
    assert($sport == bswap($sk->___sk_common.skc_num));
}
```

Figure 4: Annotated candidate eBPF program produced by the Comprehension Engine for verification. The original synthesized eBPF program is shown in gray. Pre-conditions inferred from KernelCompDataset are shown in pink. Pre-conditions inferred from the user prompt are shown in yellow. Finally, post-conditions inferred from the user's prompt are shown in blue.

(Zheng 等, 2024, p. 33) 由理解引擎生成的带注释的候选 eBPF 程序用于验证。原始合成的 eBPF 程序以灰色显示。从 KeRnelCompDataset 推断的前提条件以粉色显示。从用户提示推断的前提条件显示为黄色。最后，从用户提示推断出的后置条件以蓝色显示。



```
kprobe:tcp_connect {
    $saddr = ntop(2, $sk->___sk_common.skc_rcv_saddr);
    $daddr = ntop(2, $sk->___sk_common.skc_daddr);
    $sport = (bswap($sk->___sk_common.skc_num));
    $dport = (bswap($sk->___sk_common.skc_dport));
    printf("TCP connect: %s:%d -> %s:%d\n", $saddr, $sport,
    ↪   $daddr, $dport);
}
```

Figure 5: The output of the Synthesis Engine on the second iteration for the prompt of fig. 2 and the error message for the symbolic verification failure of fig. 4

(Zheng 等, 2024, p. 33) 合成引擎在第二次迭代时针对图2提示的输出，以及图4符号验证失败的错误信息。

📌 研究结论

Kgent 采用 GPT-4 API 2023 年 9 月版本作为其 LLM。此外，除非另有说明，我们使用 KeRnelCompDataset 和来自流行网络博客 [16] 中废弃的 eBPFNLDataset 中的提示-eBPF 程序对集来训练系统，并使用来自 eBPFNLDataset 中的提示-eBPF 程序对集来测试系统我们重新创造了。我们创建一个基线，使用 GPT-4 的单个提示来合成 eBPF 程序，并使用内置的 eBPF 验证器验证输出的 eBPF 程序。

由于每个提示都可以通过多种方式正确实现，因此我们手动检查 Kgent 的每个提示的合成 eBPF 程序，以确定输出是否正确实现提示。我们计算每个实验的 Kgent 准确度 (A)，即 Kgent 合成正确 eBPF 程序的提示分数。

为了理解 Kgent 错误输出的后果，我们将 Kgent 未能正确合成 eBPF 程序的提示分为两类：假阴性 (FN)，这是 Kgent 未能正确合成经过验证的 eBPF 程序的提示的百分比，假阳性误报 (FP)，这是 Kgent 合成未正确实现提示的经过验证的 eBPF 程序的提示的百分比。从概念上讲，FP 代表了一种安全违规，因为当 Kgent 产生误报时，使用 Kgent 的开发人员可能会错误地扩展其内核。相比之下，FN 代表了活性违规，因为开发人员实际上无法使用 Kgent 来进行此类提示。

对于 eBPFNLDataset 测试集中的每个提示，图 6 显示了描绘 Kgent 和 GPT-4 基线准确、产生假阳性和在提示的 10 次迭代中产生假阴性的时间百分比的条形图。检查每个单独测试用例的结果，我们发现在 40 个测试用例中，有 37 个测试用例中 Kgent 比基线更频繁地生成正确的程序。在 40 个测试用例中的 11 个中，Kgent 将准确率提高了 9 倍以上（即，准确率从基线中的最多 10% 提高到 Kgent 中的至少 90%）。此外，Kgent 在 9 个测试案例中的 6 个中提高了假阳性率，在这 9 个测试案例中，任一系统都观察到假阳性。

消融实验

Kgent 高层设计决策的有效性。表 1 显示了每个高级设计决策如何影响 Kgent 的有效性。表中的每一行代表不同的配置。我们从 GPT-4 少样本基线开始，按照对精度影响最大的顺序应用 Kgent 的设计特征。也就是说，我们首先包含 Kgent 的模型引导反馈，然后包含 Kgent 的理解和符号执行组件 (symbex)，最后使用 eBPFNLDataset 数据集的博客收集部分添加训练。我们注意到，将 Kgent 的理解引擎与其符号执行组件分开是没有意义的。

System	A	FP	FN
GPT-4 few shot	30%	2.5%	67.5%
GPT-4+Feedback	60%	7.5%	32.5%
GPT-4+Feedback+Symbex	77.5%	5%	17.5%
Human Expertise	72.5%	2.5%	25%
KGENT	80%	2.5%	17.5%

Table 1: The Breakdown Accuracy Analysis of KGENT

结果表明，模型引导的反馈在提高 Kgent 的准确性方面发挥着重要作用，因为它将准确性提高了 2 倍（从 30% 到 60%）。然而，准确率的提高伴随着误报率增加了 3 倍（从 2.5% 到 7.5%）。包含理解引擎和符号执行组件也提高了 Kgent 的有效性，准确率提高到 77.5%，而误报率则提高到 5%。在训练中包含 eBPFNLDataset 数据集对 Kgent 的准确性影响相对较小，仅提高了 2.5%。然而，使用 eBPFNLDataset 数据集进行训练确实使 Kgent 的误报率回落至 2.5% 的基线。

Kgent 理解和符号执行引擎的有效性。为了确定 Kgent 自动推理的能力，我们创建了一个基线，用开发人员的专业知识取代了理解和符号执行引擎。我们使用每个提示的正确 kprobe 和 kretprobe 位置来增强 eBPFNLDataset 测试集，并使用此增强数据集作为 GPT-4 + 反馈 + 数据集基线的输入，从而生成人类专业知识基线。尽管没有直接将理解和符号执行引擎的输出传递到其合成引擎中，但与人类专业知识基线相比，Kgent 表现出了更高的准确性，并且没有额外的误报。我们假设 Kgent 的反馈循环在每次符号执行失败时启动，为合成引擎提供足够的知识来取代基线的人类专业知识，从而揭示了 Kgent 的反馈功能与其自动推理功能之间的强大协同作用。

总结

总之，我们提出了 Kgent，这是一个系统，可帮助开发人员使用 eBPF 扩展其内核，方法是使用符号执行来集成基于 LLM 的程序合成和理解的结果。此外，我们还生成对于构建和评估 Kgent 以及未来 eBPF 程序合成工作有价值的数据集。未来的工作包括增强 Kgent 的性能和可用性、扩展其数据集以及探索现实世界的应用程序以进一步推进 eBPF 程序合成。

疑问

[不足之处1]：虽然Kgent是向前迈出的重要一步，但它也有一些限制。目前，我们的实现专注于小于100行的小程序，这是由于LLM的上下文窗口限制。此外，我们的eBPF程序数据集相对较小，这限制了工具处理更复杂和多样任务的能力。目前，Kgent的用例主要限于简单的跟踪程序和网络功能。

[不足之处2]：为LLM编写正确的Hoare合同具有挑战性，当前的验证方法可能无法涵盖生成的eBPF程序的所有行为。为改进这一点，需要更好的背景描述和更强大的Hoare表达式。结合更多的软件工程实践，例如反例生成和测试驱动开发，可以帮助确保全面的验证。