arXiv:2304.02014v1 [cs.SE] 4 Apr 2023

# 大语言模型是边界情况模糊测试器：通过 FuzzGPT 测试深度学习库

Yinlin Deng 伊利诺伊大学厄巴纳-香槟分校
yinlind2@illinois.edu

Chunqiu Steven Xia 伊利诺伊大学厄巴纳-香槟分校
chunqiu2@illinois.edu

Chenyuan Yang 伊利诺伊大学厄巴纳-香槟分校
cy54@illinois.edu

Shizhuo Dylan Zhang 伊利诺伊大学厄巴纳-香槟分校
shizhuo2@illinois.edu

Shujing Yang 伊利诺伊大学厄巴纳-香槟分校
shujing6@illinois.edu

Lingming Zhang 伊利诺伊大学厄巴纳-香槟分校 l
ingming@illinois.edu

## 摘要

深度学习（DL）库中的软件缺陷可能影响几乎所有下游深度学习应用，因此确保这类系统的质量至关重要。为深度学习库进行模糊测试时，生成有效的输入程序颇具挑战，因为输入程序既要符合支持的语言（如 Python）的语法和语义，也要满足构建有效计算图所需的张量/算子约束。近期的 TitanFuzz 工作表明，可以直接利用现代大语言模型（LLM），隐式学习所有语言约束和深度学习计算约束，从而生成用于深度学习库模糊测试的有效程序。然而，大语言模型倾向于生成遵循相似模式和词元的常规程序，这些模式和词元在其海量训练语料（如 GitHub）中的典型程序里十分常见；而模糊测试更需要不寻常的输入，以覆盖边界情况或那些不太可能由人工编写的情形。为弥补这一差距，本文提出 FuzzGPT，这是首个引导大语言模型合成不寻常程序以进行模糊测试的技术。FuzzGPT 建立在一个广为人知的假设之上：历史触发漏洞程序可能包含对漏洞发现至关重要的稀有且有价值的代码要素。与此同时，传统技术在利用此类历史信息时，既需要投入大量人工设计专用生成器，也需要确保生成程序的语法/语义有效性；FuzzGPT 则表明，借助大语言模型的内在能力（包括微调和上下文学习），这一过程可以完全自动化，同时还具有良好的泛化能力，并适用于具有挑战性的领域。FuzzGPT 可与不同的大语言模型搭配使用，但本文聚焦于能力强大的 GPT 风格模型：Codex 和 CodeGen。此外，FuzzGPT 还展示了直接利用近期模型指令遵循能力的潜力。

ChatGPT 助力实现有效的模糊测试。针对两个热门深度学习库（PyTorch 和 TensorFlow）开展的实验研究表明，FuzzGPT 的表现显著优于 TitanFuzz，共检测到 76 个软件缺陷，其中 49 个已确认为此前未知的软件缺陷，包括 11 个高优先级软件缺陷或安全漏洞。

## 1 引言

深度学习（DL）已广泛应用于各类领域，包括科学发现 [54]，医疗保健 [6]，金融 [65]，和交通运输 [58]。此类深度学习应用通常使用深度学习库（例如 PyTorch [51] 和 TensorFlow [64]）构建，开发者利用库的 API 来构建、训练和部署深度学习模型。与其他复杂软件系统一样，

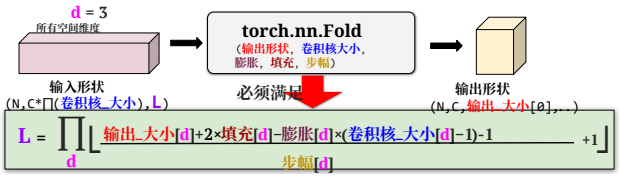



**图 1：使用`torch.nn.Fold`进行张量计算。**

深度学习库也可能存在软件缺陷。此外，深度学习库中的软件缺陷可能造成严重后果，因为它们可能影响几乎所有下游深度学习应用，包括安全关键应用 [44, 56]。因此，确保深度学习库的质量至关重要。

模糊测试 [5, 63, 82]，是一种通过随机输入生成来发现软件缺陷的强大方法，近年来已被广泛用于深度学习库测试。然而，为深度学习库生成任意输入程序极具挑战性，因为这些程序既要满足受支持语言（例如具有动态类型的 Python）的语法和语义要求，也要满足用于构建有效计算图的张量/算子约束。例如，图 1 展示了要正确使用 PyTorch API `torch.nn.Fold`，输入/输出张量形状及其他算子输入（例如膨胀）必须满足的复杂约束：输入中的最后一个形状值$L$ 应等于涵盖所有空间维度的公式乘积 $d$。为简化这一问题，既有的深度学习库模糊测试技术主要采用模型级 [24, 26, 35, 50, 70]或 API 级模糊测试 [15, 71, 75, 77]。模型级模糊测试器会复用/变异现有种子模型 [26, 50, 70]，，或从头生成深度学习模型 [24, 35]。由于张量/算子约束错综复杂，这类模型级模糊测试器要么只关注操作保持形状不变的 API [70]，要么要求为每个 API 手动编写规范 [24, 35]，以保持模型有效。因此，它们只能覆盖有限的深度学习 API 和程序模式。另一方面，API 级模糊测试器通过有效的输入生成 [71, 75]或预言机推断 [15, 77]，专注于测试各个 API。尽管 API 级模糊测试器能够轻松覆盖大量 API，却无法发现由不同深度学习 API 交互引发的软件缺陷。

随着大语言模型（LLM）近期取得巨大进展，TitanFuzz [14] 应运而生，它提出直接利用大语言模型对深度学习库进行模糊测试。其核心洞见在于，大语言模型在来自开源项目、使用不同语言编写的数十亿个代码片段上进行预训练，

其中可能包含大量适用于热门深度学习库的有效深度学习程序；因此，大语言模型能够隐式学习语言的语法/语义，以及有效深度学习计算所需的张量/算子约束。研究表明，TitanFuzz 能够有效生成有效的深度学习输入程序，其表现大幅优于传统的模型级模糊测试器和 API 级模糊测试器。更重要的是，与需要投入大量人工精力来构建生成/变异策略的传统模糊测试技术相比，[17, 29, 32, 33, 62, 78, 83],TitanFuzz 实现了全自动化，且易于推广至不同的应用领域和编程语言。与此同时，TitanFuzz 直接利用大语言模型的生成能力；这种能力基于词元自然度 [27]，旨在模仿模型在训练语料库中见过的内容。由此，TitanFuzz 能够轻松生成类人深度学习程序，这些程序显然可以作为有效程序用于模糊测试。然而，鉴于世界各地的开发者都在使用深度学习库，这类普通输入程序很难帮助覆盖更多深度学习库行为和执行路径，从而导致模糊测试效果有限。

**我们的工作**。 本文提出了 FuzzGPT，这是首个引导大语言模型合成不同寻常的 输入程序以进行有效模糊测试的技术。FuzzGPT 建立在一个广为人知的假设之上：历史上触发漏洞的程序可能包含对漏洞发现至关重要的边缘案例或有价值的代码要素。在已有研究中，研究人员提出了各种技术，用于重新组合此类代码要素，或将其插入新的代码上下文，以发现新的错误/漏洞 [10, 29, 61, 84]。然而，这些技术需要投入大量人力，既要设计专用生成器，也要确保生成程序的语法/语义有效性。例如，根据已有工作 [29, 84], , 即使是解决十分常见的未声明标识符问题，也可能并非易事。此外，这类技术难以推广到不同领域，更不用说具有挑战性的深度学习库模糊测试问题了。相比之下，我们的核心洞见是，近期先进的大语言模型为利用此类历史程序提供了一种自然、可推广且完全自动化的解决方案——只需轻松地提示 [53] 或微调 [52] 模型，使其吸收这些历史程序，随后便能生成更多与历史程序相似的不同寻常程序，并有效利用其中的代码要素。与传统的历史信息利用技术相比，FuzzGPT 能够隐式学习所有生成约束（包括语言语法/语义、深度学习计算约束以及新的非常规性约束），且整个过程完全自动化。此外，尽管本文聚焦于具有挑战性的深度学习库模糊测试问题，FuzzGPT 的思路可直接推广至其他应用领域和编程语言（例如，对各种编译器/解释器、数据库系统、SMT求解器等进行模糊测试）。

为了实现FuzzGPT，我们首先通过挖掘目标深度学习库开源代码仓库中的漏洞报告，构建一个由触发错误的代码片段组成的数据集。基于该数据集，FuzzGPT采用以下策略。**(1) 上下文学习** [38]：我们向大语言模型提供若干历史触发漏洞的程序示例（少样本学习 [8, 53]），或提供一段不完整的触发漏洞程序（零样本学习 [49, 60]），以生成新的代码片段或补全不完整的代码。**(2) 微调** [52]：我们使用提取出的历史触发漏洞程序进行训练，以调整模型权重，从而获得专门用于生成类似触发漏洞代码片段的微调后的大语言模型。从通过这两种学习策略，FuzzGPT 可以引导大语言模型生成触发漏洞的程序，方法是从局部上下文示例或微调数据集中捕获代码要素。

T摘要 ize, thi本文的主要kes the followi贡献包括tributions:

- **维度**本文为通过大语言模型生成异常输入程序以实现有效模糊测试开辟了一个新方向。本文首次表明，只需简单提示或微调，大语言模型就能模仿历史触发漏洞程序，甚至直接遵循人类指令，生成用于模糊测试真实世界系统的异常程序。与用于生成异常程序的传统模糊测试器相比，FuzzGPT 完全自动化、具有良好的泛化能力，并适用于具有挑战性的应用领域。
- **技术**虽然这一想法具有普适性，但本文将 FuzzGPT 实现为一种面向深度学习库的基于大语言模型的模糊测试器。我们基于上下文学习和微调实现了 FuzzGPT 的三种变体：1）少样本学习：提供少量以往触发缺陷的代码片段作为示例；2）零样本学习：提供一个未完成的触发漏洞程序；3）微 - 调：通过从历史程序中学习漏洞要素来训练专用大语言模型。虽然 FuzzGPT 可与不同的大语言模型配合使用，但我们基于最先进的专有和开源 GPT 风格代码大语言模型 Codex [11] 和 CodeGen [45]构建了相关策略。此外，我们还通过直接利用 ChatGPT [47]的指令遵循能力，在不使用任何历史信息的情况下构建了一种专门的零样本 FuzzGPT 变体。
- **广泛研究**我们在两个流行的深度学习库（PyTorch [51] 和 TensorFlow [64]）上研究了所有 FuzzGPT 变体。结果表明，FuzzGPT 的覆盖率比当前最先进的 TitanFuzz 在 PyTorch/TensorFlow 上分别高出 60.70%/36.03%。 - 总体而言，FuzzGPT 在 PyTorch 和 TensorFlow 的最新版本中发现了 76 个软件缺陷。 . 其中 49 个已确认为新软件缺陷，包括 11 个高优先级软件缺陷或安全漏洞。

## 2 背景与相关工作

### 2.1 大语言模型

大语言模型（LLM）通过在从互联网抓取的大规模文本语料上进行训练，在广泛的自然语言处理任务中展现了出色的性能 [21]。近年来，研究人员开始将大语言模型应用于代码相关任务，例如，使用开源代码库中的代码片段对大语言模型进一步微调 [18]。根据流行的 Transformer架构 [66] 的不同变体，大语言模型可分为：仅编码器、仅解码器和编码器-解码器模型。仅解码器大语言模型（例如 GPT [8, 48],、Codex [11] 和 CodeGen [45]）专注于自回归补全任务，通过学习根据先前生成的词元预测下一个词元的概率来实现。另一方面，仅编码器模型（例如 CodeBERT [18] 和 GraphCodeBERT [25]）以及编码器-解码器模型（例如 CodeT5 [81] 和 PLBART [4]）则专为填充任务而设计，其预训练目标是利用双向上下文恢复训练数据中被遮蔽的词元或词元片段。

为了将大语言模型用于下游应用，主要有两种范式：微调 [52] 和上下文学习 [8, 53]。微调是通过学习特定下游任务数据集中给定输入对应的期望输出，来更新模型权重的过程。由此得到的微调后的大语言模型可以视为专门设计用于执行特定任务（如代码摘要）的

模型。 [4]与微调不同，微调通常需要大量下游数据集来更新模型，而上下文学习只需要下游任务的少量示例。上下文学习无需修改权重，直接使用预训练大语言模型：构造一个输入，其中包含多个（即少样本）输入/输出示例，然后附上待解决的最终任务。通过查看上下文，大语言模型无需微调即可直接学习特定任务的目标和预期输出格式。人们提出了多种改进上下文学习的技术，尤其是通过示例选择 [36, 55],，以及使用 Scratchpad [46] 或思维链 [72]进行多步推理。总的来说，上下文学习和微调是让大语言模型适应下游任务的两种不同方法。

尽管 FuzzGPT 的方法适用于不同的大语言模型，但本文主要关注近期功能强大的 GPT 风格模型：Codex [11]和 CodeGen [45],，它们分别是当前最先进的专有代码模型和开源代码模型。此外，我们还直接利用 ChatGPT [47] 的指令遵循能力进行模糊测试。

## 2.2 利用历史漏洞进行模糊测试

利用历史漏洞进行模糊测试的研究已有十多年。其中一项开创性技术是 LangFuzz [29]：LangFuzz 首先解析已知会导致错误行为的历史测试，并提取单独的代码片段；随后，LangFuzz 会生成代码，并进一步用提取的代码片段替换部分代码。研究人员还从历史触发漏洞程序中挖掘模糊测试器配置（例如语法/语句概率），以实现更多样化的生成 [10, 61]。近来，JavaTailor [84] 利用从历史漏洞中提取的代码要素，为 Java虚拟机（JVM）模糊测试合成新的输入程序。除了这类基于生成或混合的方法外，基于变异的模糊测试器也广泛利用这类先前的漏洞报告或回归测试，将其作为高质量输入种子进行后续变异，目标包括C编译器[32],、SMT 求解器 [73],和数据库系统 [68, 86]。此外，近期一项研究 [85] 表明，直接从历史漏洞报告中复用几乎未经修改的代码片段，也能在其他 C/C++ 编译器中发现有趣的软件缺陷。

FuzzGPT 通过微调和大语言模型的上下文学习，直接从历史触发漏洞的代码中学习。不同于以往基于生成或变异的技术——这些技术需要投入大量人力构建专用生成器，并确保生成程序的语法和语义有效性—— [29, 84], FuzzGPT 完全自动化且具有良好的泛化能力。此外，由于语言本身复杂，且存在张量/算子约束，为深度学习程序构建此类传统生成器/变异器极具挑战性，而 FuzzGPT 可以利用现代大语言模型隐式地学习所有这些约束。

## 2.3 通过深度学习进行模糊测试

除了传统的模糊测试方法，研究人员还开发了基于深度学习（DL）技术的模糊测试工具。Se-qFuzzer 是一种基于长短期记忆（LSTM） [28],和循环神经网络（RNN） [12]构建的网络协议模糊测试器。基于 RNN 的模糊测试器也已用于测试其他系统，例如 OpenCL 编译器 [13],、C编译器 [39], 和 PDF 文件解析器 [23]。类似地，Montage [34] 被提出用于模糊测试 JavaScript 引擎，其方法是直接训练基于树的循环神经网络来变异现有的种子抽象语法树。此外，COMFORT [79]提出对预训练的 GPT-2 模型进行微调，以生成 JavaScript 程序，并进一步依赖额外的启发式方法为合成的程序生成输入。尽管上述工作利用了深度学习模型的发展来进行模糊测试，但它们尚未研究用于代码的现代大语言模型。最近，TitanFuzz [14] 首次证明，现代大语言模型无需微调即可直接用于以零样本方式对真实世界系统进行端到端模糊测试。TitanFuzz 首先利用 Codex [11]生成高质量的种子程序，然后利用 InCoder [20] 结合演化式模糊测试策略生成多样化的代码片段。TitanFuzz 在深度学习库模糊测试方面取得了最先进的成果，并能发现只有通过复杂的 API 调用序列才能揭露的软件缺陷。

在这项工作中，我们提出利用历史触发漏洞的代码信息，进一步引导大语言模型实现更有效的模糊测试。据我们所知，这是首项证明大语言模型能够轻松执行历史驱动模糊测试（这一方法已被广泛研究十余年）的工作，同时还能实现全自动化、具有良好的泛化能力，并适用于极具挑战性的深度学习库模糊测试领域。

# 3 方法

本节介绍我们提出的 FuzzGPT 方法，该方法利用大语言模型（LLM）通过历史软件缺陷自动对深度学习库进行模糊测试。FuzzGPT 的核心思想是利用大语言模型，直接从历史报告的软件缺陷中学习，生成类似的可触发软件缺陷的代码片段，以发现新的软件缺陷。现有的相关工作需要投入大量人工，且几乎无法推广到深度学习库模糊测试这一充满挑战的领域。相比之下，大语言模型的最新进展提供了一种自然、可泛化且完全自动化的解决方案——现代大语言模型只需通过提示或微调，就能轻松吸收此类历史程序，进而生成与历史程序相似的程序，并有效利用其中的代码要素。

图 2 展示了 FuzzGPT 的整体概览。首先，我们系统地挖掘目标深度学习库代码仓库中的漏洞报告，收集历史触发错误的代码片段（第 3.1 节）。由于 FuzzGPT 旨在针对特定的深度学习库 API，因此每个触发错误的代码片段都需要有对应的存在缺陷的 API 标签。然而，漏洞报告中往往没有明确指出具体存在缺陷的 API。因此，FuzzGPT 采用自训练方法，通过提示大语言模型并提供少量人工标注的示例，自动生成存在缺陷的 API 标签。接下来，利用提取出的触发错误的代码片段与存在缺陷的 API 标签对，FuzzGPT 可以启动模糊测试流程，生成边缘情况代码片段（第 3.2 节）。在这项工作中，我们研究了两种学习方法：**1）上下文学习：**我们直接向预训练大语言模型提供触发漏洞的代码与存在缺陷的 API 配对示例（少样本），或部分/完整的触发漏洞代码（零样本），让模型为目标 API 生成/编辑程序；**2）微调：**利用提取的数据集对大语言模型进行微调，使其从触发漏洞的代码示例和模式中学习，为目标 API 生成新的触发错误的代码片段。最后，运行生成的程序，并使用测试预言机（例如 CPU/GPU [71],或自动微分预言机 [77]）检测漏洞（第 3.3 节）。

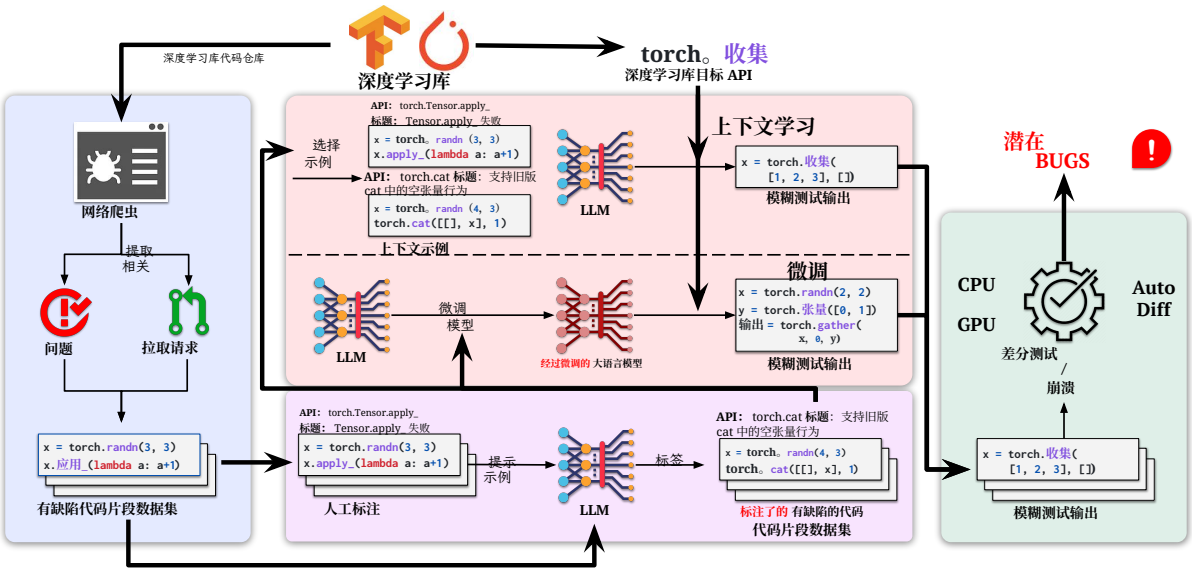



**图 2：FuzzGPT 概览。**

虽然 FuzzGPT 适用于任何生成式大语言模型，但本文聚焦于两个特定的大语言模型：Codex [11] 和 CodeGen [45]。Codex 是一种先进的生成模型，基于 GPT-3 [8] 模型权重初始化，并在开源代码仓库上进行微调。与 Codex 不同，CodeGen 是一种开源生成模型；Codex 的模型权重和训练数据并未公开。由于我们的目标是 Python 中提供的深度学习库 API，因此我们使用经过 Python GitHub 代码微调的 CodeGen Python 版本 [45]。在 FuzzGPT 中，我们直接使用 Codex 和 CodeGen 模型：规模更大的 Codex 模型有助于展示使用大语言模型进行模糊测试的全部潜力，而开源的 CodeGen 模型则可用于评估规模效应并测试微调策略。接下来，我们将更详细地介绍 FuzzGPT 的各个步骤。

## 3.1 数据集构建

为了促进从历史漏洞中学习，FuzzGPT 需要一个触发漏洞的代码数据集，其中包含暴露错误的代码片段及其对应的有漏洞的深度学习库 API。

*3.1.1 从 GitHub 挖掘漏洞历史。* 首先，我们实现一个 HTML 爬虫，从目标库在 GitHub 问题跟踪系统中的所有问题和拉取请求（PRs）中收集数据。然后，我们重点从两个来源识别触发漏洞的代码片段：（1）与已接受或待处理拉取请求相关联的问题。我们在这些漏洞报告中搜索所有代码块，其中包含用于复现漏洞的代码片段，并将它们拼接在一起。（2）在其提交信息中包含代码块的拉取请求。我们进一步考虑拉取请求，因为一些拉取请求可能会修复漏洞，但没有对应的问题。对于每个提取出的问题或拉取请求，我们还会提取其标题，并将其纳入少样本学习和微调的提示中。在挖掘出触发错误的代码片段后，我们接下来将执行自动化标注，为每个代码片段进一步标注对应的有漏洞 API。

*3.1.2 有漏洞 API 自动标注。* 我们的数据集中的每个代码片段通常涉及多个深度学习 API。因此，无法直接提取出确切的有漏洞 API。为了标注

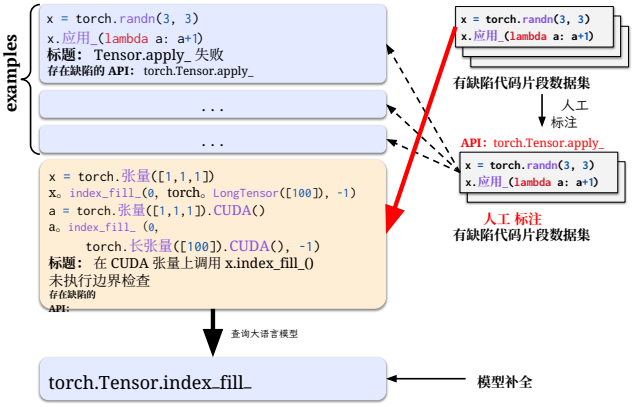



**图 3：错误 API 标注提示。**

每个触发错误的代码示例中的有漏洞 API，我们提出了一种自训练 [57, 87]方法：向大语言模型输入少样本提示（如图 3 所示），并将模型补全作为有漏洞 API。我们首先为若干随机选取的触发错误代码片段人工标注有漏洞 API 名称。这些人工标注的代码片段与 API 名称配对后，成为输入大语言模型的少样本示例。如图 3 所示，少样本示例由选定的触发错误代码片段、提取出的对应问题/拉取请求标题（例如 `Tensor.apply_ 失败`），以及人工标注的有漏洞 API 名称（例如 `torch.Tensor.apply_`）组成。接下来，我们将少样本示例与一个目标触发错误代码片段结合，构造提示并查询大语言模型，以获取该代码片段对应的预测有漏洞 API 名称。值得注意的是，由于现代大语言模型具备上下文学习能力，对于目标库，我们只需标注少量示例（本研究中为 6 个，如第4节所示），因此标注工作量极少。

我们提出的方法与自训练类似 [57, 87]：首先使用一个小型有标签数据集训练分类器，再用它为更大的无标签数据集添加标签。在我们的场景中，我们直接使用大语言模型，通过

少样本学习，利用少量人工标注，为数据集中大量提取出的触发错误的代码片段提供标注。请注意，标注不必完全精确 [43]：标注的主要目的是通过提供成对的 API 及其对应代码示例，引导大语言模型针对每个目标深度学习 API 生成大量程序。也就是说，即使某个代码示例不巧被标注为与真实有缺陷的 API 不同的“错误” API，它仍可引导模型学习从深度学习 API 到包含有趣的触发缺陷模式、并有望调用该 API 的程序之间的非严格映射。在评估中，我们还观察到，针对某个深度学习 API 时，FuzzGPT 确实能够生成揭示另一 API 中软件缺陷的代码片段。

## 3.2 上下文学习与微调

使用提取并标注的有缺陷代码片段数据集后，FuzzGPT 开始通过学习来生成边界情况代码。从宏观层面来看，基于大语言模型的学习方法可分为两种主要方式：**上下文学习**和**微调**。上下文学习是指直接使用预训练大语言模型，而不调整任何模型参数。具体而言，在提供当前输入之前，会先将指令和示例添加到大语言模型的输入前面，以展示任务要求。上下文学习使大语言模型能够通过周围的输入上下文进行预热，不仅学习期望的输出格式（例如，提供是或否的答案），还学习任务领域（例如，将英语翻译成法语）。另一方面，微调旨在通过使用特定数据集进行训练来修改大语言模型的参数，从而创建专用模型。

在 FuzzGPT 中，我们遵循已有工作 [7]，系统地探索不同的学习设置。具体而言，我们设计了三种不同的学习策略：少样本、零样本和微调；每种策略都以不同方式利用已标注的漏洞触发代码数据集来生成模糊测试输出。图4展示了这三种策略，下文将详细介绍。

*3.2.1* 少样本学习。我们遵循经典的少样本上下文学习方法 [7]，在目标查询前添加来自已标注的触发缺陷代码片段数据集中的示例。图4（ ）展示了少样本学习提示的构造方式。我们的示例格式包括 API 名称（例如 `torch.Tensor.apply_`）、从问题/拉取请求标题中提取的缺陷描述（例如 `Tensor.apply_` 失败），以及触发缺陷的代码片段。这些示例有两个目的。首先，它们旨在引导大语言模型生成所需的输出格式。其次，通过观察历史触发错误的代码片段，它们使模型无需修改模型参数，就能学会生成类似的边缘情况代码片段。在少样本学习中，每个提示都由 $K$ 个示例（记作 $K$-Shot）和实际查询（目标 API 与缺陷描述标题）组成。然后，大语言模型可以基于提示生成新的预测。

**思维链提示**。少样本学习的一个重要组成部分是所使用的提示。我们的提示设计受到思维链（CoT） [72] 提示的启发：提示不会直接生成最终输出，而是要求模型按步骤完成任务。遵循上下文示例的格式，我们先在查询中加入目标 API 名称（例如，API：torch.gather），然后要求模型生成一段描述可能存在的“缺陷”（缺陷描述：），再生成实际触发“缺陷”的代码片段，以调用目标 API。预测的缺陷描述会为大语言模型提供额外提示，指出生成的代码应尝试覆盖特定的潜在缺陷行为。

设 M 为输出生成序列概率的大语言模型。令 $E_K = \{< p_1, d_1, c_1 >, \ldots\ldots < p_K, d_K, c_K >\}$ 表示由 $K$ 个示例串接而成的序列，每个示例均由示例 API $p_i$、缺陷描述 $d_i$ 和代码片段 $c_i$ 组成。令 $p$target 为目标 API 查询。使用少样本学习生成输出代码 $c$fs 的概率，可形式化为以下条件概率：$\mathrm{M}(c\text{fs}|E_K, p\text{target}) = \mathrm{M}(c\text{fs}|E_K, p\text{target}\ d\text{fs}) \cdot \mathrm{M}(d\text{fs}|E_K, p\text{target})$

*3.2.2* 零样本学习。现在我们介绍 FuzzGPT 在零样本学习场景下的两种变体：

**零样本补全（默认）**。输入仅由部分代码片段构成。该部分代码片段取自历史错误触发代码片段数据集，我们会随机删除一部分后缀代码。图4（ ）展示了如何构造零样本补全输入。我们首先加入一条自然语言注释 `#` 以下代码揭示了 `{target_api}` 中的一个缺陷，这使我们能够将零样本学习应用于任意目标 API。随后，我们从数据集中随机选取一个暴露错误的代码片段。接着，随机删除所选触发缺陷的代码片段中的部分后缀，仅保留前缀代码行作为输入，供大语言模型补全。

再次设 M 为输出生成序列概率的大语言模型，$c_e$ 为选定的代码片段，其前 $j$ 行（$c_e[:j]$），$p$comp 为补全提示。生成零样本补全代码 $c$zs-comp 的概率可形式化为以下条件概率：$\mathrm{M}(c\text{zs-comp}|p\text{comp}\ c_e[:j])$，其中 $c_e[:j]$+$c$zs-comp 是完整的模糊测试代码片段，包括部分代码。

**零样本编辑**。在此变体中，输入取自我们历史触发漏洞数据集中的完整代码片段。图4（ ）也展示了零样本编辑的示例。首先，我们使用自然语言注释 `#` 编辑代码以使用 `{target_API}`，向大语言模型表明我们希望执行编辑，并且应直接复用原始代码的大部分内容。与零样本补全类似，我们随机选取一段触发漏洞的代码，并将其附加到输入末尾。不同于零样本补全中模型自动补全代码片段末尾的方式，编辑旨在让大语言模型复用历史触发漏洞代码片段的所有部分，以生成新的模糊测试输出。令 $p$edit 为编辑提示。零样本编辑输出 $c$zs-edit 可形式化为以下条件概率：$\mathrm{M}(c\text{zs-edit}|p\text{edit}\ c_e)$

与少样本学习将多个历史软件缺陷作为示例提供给大语言模型不同，零样本学习可以直接利用历史软件缺陷中的现有代码片段，只需生成部分程序（补全），或替换一小部分代码，以使用新的目标 API（编辑）。这些来自以往触发漏洞的代码中的具体代码行有助于测试新的 API，因为它们可能包含特殊值（例如，非数）、边界情况的张量形状/维度，以及其他有助于发现软件缺陷的代码要素（例如，不常见的 API 用法）。通过采用零样本学习，FuzzGPT 可以直接利用这些历史触发错误的代码片段来测试更多 API。

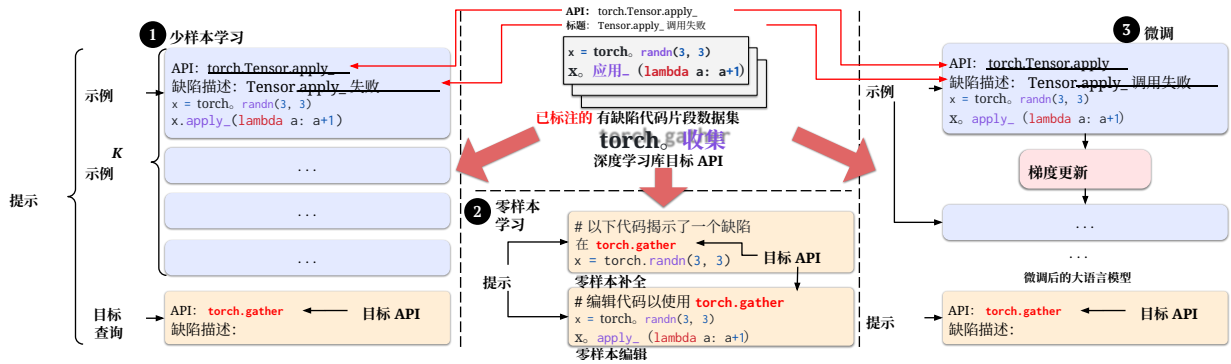



**图4：用于深度学习库模糊测试的微调、零样本学习和少样本学习。**

*3.2.3* 微调。 与上下文学习（少样本和零样本）不同，后者仅使用原始预训练大语言模型从历史软件缺陷中学习；微调则通过在历史缺陷代码片段数据集上训练，直接修改模型参数。图4

(●) 展示了微调过程的工作方式，以及推理时输入微调后模型以生成模糊测试输出的内容。每个训练样本都遵循与少样本示例相同的格式，由 API 名称、缺陷描述和触发缺陷的代码片段组成。我们从原始的预训练大语言模型开始，然后通过梯度下降训练模型自回归地预测每个训练样本，从而更新模型权重。

令 $T=\{t_1, t_2.....t_n\}$ 表示对训练样本进行令牌化后得到的训练令牌序列，令 $T_{<s}=\{t_1, t_2.....t_{s-1}\}$ 表示模型迄今生成的令牌序列，M 表示大语言模型，它根据先前生成的令牌输出下一个令牌的生成概率。微调损失函数定义为：$L_{微调}=-\frac{1}{n}\sum_{i=1}^{n} log\,(M(t_i \mid T_{<i}))$

对于每个深度学习库，我们使用从该库收集的触发缺陷的代码片段，分别微调一个模型。通过使用这些历史触发错误的代码片段进行微调，大语言模型能够学习目标库中不同类型的触发缺陷模式/要素。微调模型的输入遵循与少样本方法相同的模式，即根据特定目标 API 构造提示。设 Mft 为微调后的大语言模型，它输出生成包含目标 API 提示 $p_{target}$ 的序列的概率。微调模型的代码输出 $c_{ft}$ 可形式化为以下条件概率：$M_{ft}(c_{ft} \mid p_{target}) = M_{ft}(c_{ft} \mid p_{target}\ d_{ft}) \cdot M_{ft}(d_{ft} \mid p_{target})$

## 3.3 预言机

利用生成的模糊测试输出，我们通过通用预言机和深度学习专用预言机来测试深度学习库：

**崩溃**。我们检测执行模糊测试输出时发生的意外崩溃所导致的软件缺陷。这些崩溃包括中止、段错误以及 内部_断言_失败。此类崩溃暴露的软件缺陷甚至可能进一步引发安全漏洞。

**CPU/GPU 预言机**。 我们通过识别两个执行后端（CPU 和 GPU）输出值之间的不一致性，来检测计算错误缺陷。我们沿用已有工作 [14, 71]，使用显著性容差阈值进行比较，以应对特定库 API 在不同后端上执行时的非确定性。

**自动微分（AD）预言机**。 我们检测深度学习库中关键自动微分引擎的梯度计算缺陷，这些引擎支持深度学习模型的高效训练。我们应用已有工作 [77] 提出的自动微分预言机，并比较反向模式自动微分（深度学习库中最常用的模式）、前向模式自动微分和数值微分（ND）计算出的梯度。

## 4 实现

**数据集构建**。 我们使用*requests* 库抓取目标库的 GitHub 仓库 [3],，最终从它们的历史问题或拉取请求中，分别收集了 1750 个 PyTorch 触发缺陷的代码片段和 633 个 TensorFlow 触发缺陷的代码片段。TensorFlow 的数量较少，是因为其拉取请求数量少于 PyTorch；TensorFlow 开发者确认软件缺陷和拉取请求的积极性不如 PyTorch 开发者，而且很少在拉取请求中加入代码块。请注意，我们还对提取出的代码进行了额外清理，以过滤错误消息，并删除仅包含执行输入和输出的代码行。我们也不考虑语法检查未通过或长度超过 256 个 token 的代码片段。

对于错误 API 标注，我们手动标注随机抽取的 $K=6$ 个示例，并将其用作上下文示例。对于每个未标注的示例，我们按照 [69]，通过 *temperature* = 0 （确定性贪心解码）请求 Codex 补全错误 API，以获得置信度最高的预测。我们还在随机抽取的 100 个 PyTorch 问题/拉取请求样本上评估标注准确率，发现自动标注的精确率达到 76%。事实上，即使是标注错误的示例（其中被错误标注的 API 总会出现在代码中），仍能帮助 FuzzGPT-FS 大幅超越零样本基线；该基线无法使用任何示例（第 6.3.1 节）。**语言模型**。 我们在两种特定的生成式大语言模型上进行评估：Codex（code-davinci-002）和 CodeGen （ 350M/2B/6B-mono ）。与训练数据和模型权重未公开的 Codex 不同，CodeGen [45] 是一种广泛使用的开源生成式模型 [16,42, 74],，提供多种规模的训练模型，并支持微调。我们通过其 API 使用 Codex，并使用 Hugging Face [30] 上 CodeGen 模型的 PyTorch 实现。

**微调**。 由于 Codex 并非开源，我们对 CodeGen 模型进行微调。针对每个研究的深度学习库，我们分别微调一个模型。为更新每个目标库的模型参数，我们采用批量大小_、学习率=5 e-5，并使用 AdamW 优化器 [40] 训练模型 10 个训练轮次，同时使用预热比例为 10% 的线性学习率调度器。

# 5 评估

## 5.1 研究问题

我们在实验中探讨以下研究问题：

- **RQ1：** FuzzGPT 的不同学习范式之间如何比较?
- **RQ2：** FuzzGPT 与现有模糊测试器相比如何?
- **RQ3：** FuzzGPT 的关键组件如何影响其有效性?
- **RQ4：** FuzzGPT 能否检测到新的软件缺陷?

## 5.2 实验设置

**受测系统。** 我们评估 FuzzGPT 对 PyTorch 和 TensorFlow 进行模糊测试的效果。这两者是最受欢迎的开源深度学习库。对于 RQ1，我们分别以少样本、零样本和微调设置运行 FuzzGPT，以评估各设置在模糊测试中的效果（分别记为 FuzzGPT-FS、FuzzGPT-ZS 和 FuzzGPT-FT）。除非另有明确说明，我们均使用 FuzzGPT-FS 作为 FuzzGPT 的默认实现。对于 RQ2，我们在 PyTorch（v1.12）和 TensorFlow（v2.10）上将我们的方法与已有工作进行比较；这与最新工作 TitanFuzz 使用的版本相同[14],，我们也使用了与 TitanFuzz 相同的一组公开 Python API。对于 RQ3，我们在 PyTorch 和 TensorFlow 的 nightly 版本上运行所有生成的测试，以查找此前未知的软件缺陷（第 6.4 节）。对于 RQ4，鉴于成本高昂，我们针对 FuzzGPT 的三种设置开展消融研究实验，从一个示例深度学习库 PyTorch 中随机抽取 50 个 API，并按照已有工作的方法报告 5 次运行的平均结果 [14]。

**基线。** 我们将 FuzzGPT 与最先进的深度学习库模糊测试器进行比较，包括最先进的 API 级模糊测试器（FreeFuzz [71], Deep- REL [15], and ∇Fuzz [77]）和模型级模糊测试器（Muffin [24]），以及最新的 TitanFuzz [14]。除 Muffin 仅在 TensorFlow 上运行（因为它不支持 PyTorch）外，我们均使用各工具的默认配置，在两个库上运行它们。

**环境。** 我们使用一台配备 256 GB RAM 的 64 核工作站，运行 Ubuntu 20.04.5 LTS，并配有 4 块 NVIDIA RTX A6000 GPU。**模糊测试预算。** 默认设置下，我们为每个目标 API 生成 100 个程序。在少样本方法中，对于每个目标 API，我们独立构建 10 个提示，每个提示包含随机选取的 6 个示例，并将每个提示输入大语言模型以采样 10 次生成结果。同样，在零样本方法中，我们从数据集中随机选取 10 个不同的示例，并使用部分或完整代码构建 10 个提示，以执行补全或编辑。在微调方法中，我们使用固定的任务描述，并查询模型 10 次，为一个目标 API 生成全部 100 个程序。

**生成。** 按照已有工作 [14]，我们在使用所有大语言模型进行生成时，默认采用 top-p采样，参数为 $p = 0.95$、$temperature = 0.8$ 和 $max_token = 256$。

## 5.3 指标

**检测到的漏洞。** 按照深度学习库模糊测试领域已有工作[14, 50, 67, 70, 71, 75, 77]，我们报告检测到的唯一漏洞数量。

**唯一崩溃。** 除了统计所有软件缺陷外，我们还统计唯一崩溃的数量，将其作为衡量模糊测试效果的另一项指标。文献中广泛使用唯一崩溃来评估模糊测试技术 [19, 31, 41]。

**代码覆盖率。** 代码覆盖率已广泛应用于软件测试，并且近期也应用于深度学习库/编译器测试 [14, 24, 37, 71, 77]。我们遵循近期的深度学习库模糊测试工作 [14, 24, 71, 77]，并使用 *coverage.py* 工具 [1]衡量 Python行覆盖率。为确保公平比较，我们排除了预言机检查额外增加的覆盖率。**API 覆盖率。** 按照已有的深度学习库模糊测试工作 [14, 15, 71, 77]，我们将已覆盖的深度学习 API 数量作为衡量测试充分性的重要指标。

**唯一有效程序。** 如果生成的程序能够成功执行且不引发异常，并且至少调用一次目标 API，则视为有效程序。我们还会移除已生成的程序，只统计唯一程序。

# 6 结果分析

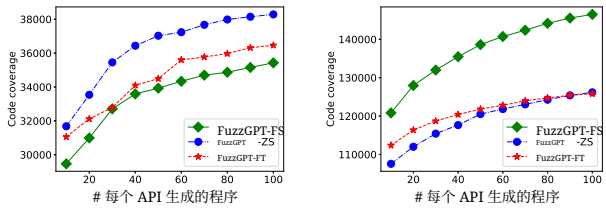
## 6.1 学习范式比较。

我们首先比较 FuzzGPT 的三种变体（少样本、零样本、微调），以了解它们各自的性能。表 1 总结了结果。列 **#APIs**、**# Prog.** 和 **Cov** 分别表示 API 数量、唯一程序数量和覆盖行数。具体而言，**Valid** 表示仅考虑没有运行时错误的唯一程序，**All** 表示考虑所有生成的唯一程序。最后，**Valid(%)** 表示有效程序占所有生成的唯一程序的比例。图 5 进一步展示了覆盖率随每个 API 生成的程序数量变化的趋势。

我们可以观察到，在 PyTorch 和 TensorFlow 上，FuzzGPT-FS 覆盖的 API 数量以及生成的唯一（有效）程序数量都最多。原因可能是它为大语言模型（即 Codex）提供了丰富的上下文（包括 $K = 6$ 个漏洞示例，这些示例很可能涉及不同的漏洞 API），使其能够学习并组合多种漏洞模式，并使用多样化的 API。

与其他变体相比，FuzzGPT-ZS 的有效率低得多。原因在于它需要补全已有的部分程序。因此，与其他变体相比，其搜索空间受到更多限制，且任务更具挑战性，因为新生成的代码需要兼容。与此同时，FuzzGPT-ZS 可能会触发已有部分程序中的 API 与新生成的 API 之间更有趣的交互。此外，从漏洞历史中复用的部分代码也可能非常有价值，并且可能已经覆盖了有趣的程序路径/行为。因此，FuzzGPT-ZS 甚至在 PyTorch 上取得了最高的覆盖率。FuzzGPT-ZS 在 TensorFlow 上的表现相对较差，这可能是因为可供复用的代码片段较少（仅 633 个，几乎无法覆盖 TensorFlow 的全部 3316 个 API）。

FuzzGPT-FT 在两个库上都实现了相当的代码覆盖率，并且即使使用规模更小的模型（CodeGen-6B），在 PyTorch 上也达到了最高的有效率。结果表明，微调是对特定库进行模糊测试的一种非常有效的方法，因为微调后的模型通过更新模型参数，从收集到的所有缺陷模式中学习，并能在生成过程中“选择”或“混合”所学到的缺陷要素，以针对特定 API。相比之下，少样本会消耗数量有限的上下文示例，而零样本在每个推理步骤中都依赖一个不完整的示例。尽管如此，微调需要收集一个（高质量的）微调数据集，并针对每项不同的任务训练一个不同的大语言模型（这会消耗大量计算资源和存储）。




(a) PyTorch (b) TensorFlow

**图 5：FuzzGPT-FS/-ZS/-FT 的覆盖率趋势。**

FuzzGPT 展现了运用上下文学习和微调技术生成模糊测试输入的能力。从图 5 中的覆盖率趋势可以观察到，在三种变体中，即使生成了全部 100 个代码片段后，覆盖率仍未饱和，这表明大语言模型能够从历史触发漏洞的数据集中学习，持续生成有价值的模糊测试程序，从而获得更高的覆盖率。接下来我们将考察 FuzzGPT 与最先进的深度学习库模糊测试器的比较。

**表 1：学习范式比较。**

| | 范式 | # 应用程序接口 | | # 程序 | | 有效率（%） | Cov |
|---|---|---|---|---|---|---|---|
| | | 有效 | All | 有效 | All | | |
| PyTorch | FuzzGPT-FS | **1377** | **1588** | **42496** | **154904** | 27.43% | 35426 |
| | FuzzGPT-ZS | 1237 | 1553 | 7809 | 132111 | 5.91% | **38284** |
| | FuzzGPT-FT | 1223 | 1546 | 31225 | 112765 | **27.69%** | 36463 |
| TensorFlow | FuzzGPT-FS | **2309** | **3314** | **54058** | **310483** | **17.41%** | **146487** |
| | FuzzGPT-ZS | 1460 | 3157 | 4650 | 233887 | 1.99% | 126193 |
| | FuzzGPT-FT | 1834 | 3292 | 31105 | 253216 | 12.28% | 125832 |

## 6.2 与以往工作的比较

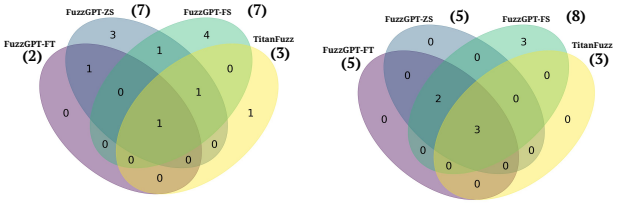
我们将 FuzzGPT-FS/-ZS/-FT 与最先进的模糊测试器 TitanFuzz 及其他近期的深度学习库模糊测试器进行比较。所有技术均采用默认配置。

**API 和代码覆盖率**。如表 2 所示，FuzzGPT 的三个变体在代码覆盖率方面均显著优于所有现有模糊测试器，包括最先进的 TitanFuzz。具体而言，表现最佳的变体 FuzzGPT-FS/FuzzGPT-ZS 在 TensorFlow/PyTorch 上分别达到 54.37%/33.72% 的行覆盖率，相较于 TitanFuzz 提升了 36.03%/60.70%。我们还发现一个有趣的现象：FuzzGPT 的 API 覆盖率与 TitanFuzz 相近，但代码覆盖率高得多。这表明 FuzzGPT 能够覆盖深度学习库中更多有趣的代码行为/路径。FuzzGPT 和 TitanFuzz 都依靠大语言模型全自动生成（或变异）程序，并且在 API 覆盖率方面显著优于以往技术（FreeFuzz、DeepREL、∇Fuzz、Muffin），这体现了大语言模型在模糊测试方面的优势。**崩溃检测**。我们以唯一崩溃数量为指标，在一个 PyTorch 示例库上比较 FuzzGPT 与 TitanFuzz 的漏洞发现能力。我们未纳入不一致性

在此比较中使用软件缺陷，是因为崩溃更容易衡量，也可作为漏洞发现能力的近似指标 [31]。我们使用两种工具的默认设置，并直接执行程序以检测崩溃。除了直接执行外，我们还使用 AD 预言机执行所有程序，以检测更多崩溃。图 6 展示了 FuzzGPT-FT/-ZS/-FT 与 TitanFuzz 的维恩图比较，括号中的数字表示每种技术发现的崩溃总数。首先，我们观察到，我们的三种技术总共都能发现更多崩溃（包括

**表 2：与已有工作的比较。**

| | PyTorch | | TensorFlow | |
|---|---|---|---|---|
| | 代码覆盖率 | API 覆盖率 | 代码覆盖率 | API 覆盖率 |
| 被测代码库 | 113538 (100.00%) | 1593 | 269448 (100.00%) | 3316 |
| FreeFuzz | 15688 (13.82%) | 468 | 78548 (29.15%) | 581 |
| DeepREL | 15794 (13.91%) | 1071 | 82592 (30.65%) | 1159 |
| ∇模糊测试 | 15860（13.97%） | 1071 | 89722（33.30%） | 1159 |
| Muffin | NA | NA | 79283（29.42%） | 79 |
| TitanFuzz-seed-only | 22584 (19.89%) | 1329 | 103054 (38.35%) | 2215 |
| TitanFuzz | 23823 (20.98%) | 1329 | 107685 (39.97%) | 2215 |
| FuzzGPT-FS-25 | 32305 (28.45%) | 1296 | 130312 (48.36%) | 1937 |
| FuzzGPT-FS | 35426 (31.20%) | **1377** | **146487 (54.37%)** | **2309** |
| FuzzGPT-ZS | **38284 (33.72%)** | 1237 | 126193 (46.83%) | 1460 |
| FuzzGPT-FT | 36463 (32.12%) | 1223 | 125832 (46.70%) | 1834 |




(a) 直接崩溃。 (b) AD 崩溃.

**图 6：唯一崩溃的维恩图**

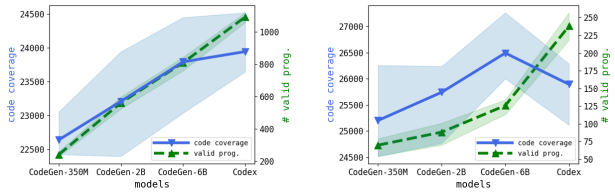
（同时包括 AD 和直接执行导致的崩溃）相比 TitanFuzz 基线，例如，我们默认的 FuzzGPT-FS 检测到的唯一崩溃数量是 TitanFuzz 的2.5 倍。此外，通过组合 FT/ZS/FT，FuzzGPT 总共检测到 19 个不同的崩溃，其中 14 个是 TitanFuzz 无法发现的唯一崩溃，而 TitanFuzz 仅独有地发现了 1 个崩溃，这凸显了借助历史触发错误的代码片段，FuzzGPT 在生成触发异常崩溃的程序方面的有效性。

**生成效率**。我们进一步讨论 FuzzGPT 与最强基线 TitanFuzz 的生成效率。由于 TitanFuzz 使用 Codex 先生成 25 个种子程序，然后使用 InCoder 进行变异，我们使用 FuzzGPT-FS 仅通过 Codex 生成 25 个程序（记为 FuzzGPT-FS-25）。根据表 2，即使 FuzzGPT-FS-25 的表现也能显著优于 TitanFuzz，且成本低得多（TitanFuzz 还会针对每个 API 使用 InCoder 进行额外变异），这表明 Codex 这样的 大语言模型能够有效利用历史触发漏洞程序，生成有价值的模糊测试程序。此外，与仅使用种子的版本（即 TitanFuzz-seed-only）相比，TitanFuzz 的变异阶段并未带来显著的覆盖率增益，而 FuzzGPT 的覆盖率即使在生成 100 次后仍未饱和（图 5）。请注意，由于使用了不同的大语言模型（以及不同的 CPU/GPU/云成本），总体而言，很难精确比较不同技术的效率；我们已尽最大努力确保此处的讨论公平。

## 6.3 消融研究

*6.3.1* 少样本学习。我们首先研究 FuzzGPT-FS 的各种设计选择，这些选择会在提示中向大语言模型提供若干能触发真实错误的代码示例。我们比较不同的模型以及各种提示策略。

**模型规模**。我们首先评估随着模型规模扩展，FuzzGPT-FS 的性能。图 7a 展示了随着模型规模增大，代码覆盖率以及生成的有效程序数量的变化。我们可以看到，较大的模型能够生成更多（唯一的）、语法和




（a）少样本　　（b）零样本

**图7：模型规模扩展的比较**

语义我们通过程序观察到，随着模型参数增加（从 CodeGen 350M 到 2B，最后到 6B），覆盖率明显提高；就覆盖率而言，CodeGen-6B 已经能够达到与 Codex 相当的性能。**思维链提示。**现在，我们来考察缺陷描述在少样本模板中的有效性。我们的默认提示策略**采用思维链**为有缺陷的代码提供自然语言解释（如图4所示），这可视为思维链提示，因为它指示大语言模型先生成可能的漏洞原因（以自然语言表述），再据此生成代码。基线策略**不采用思维链**会从提示中移除缺陷描述：……部分，只要求模型根据指定 API 生成程序。如表3所示，加入缺陷描述能显著提高代码覆盖率，这表明为大语言模型提供一些中间上下文信息，有助于其对缺陷模式进行“推理”。表3还显示，在每个示例中加入自然语言描述，会促使模型生成更多独特的 API，这可能意味着模型先生成了更多样的自然语言描述，进而影响后续的代码生成。有效率较低，可能是因为生成的程序更有可能覆盖某些边界情况并触发运行时异常。

**表 3：使用或不使用思维链提示的 FuzzGPT-FS。**

| 提示 | 有效 API | 所有 API | 有效程序 | 所有程序 | 有效率（%） | Cov |
|---|---|---|---|---|---|---|
| 使用思维链 | **190** | **428** | 1092 | **4885** | 22.36% | **23945** |
| 不使用思维链 | 181 | 377 | **1346** | 4798 | **28.05%** | 22922 |

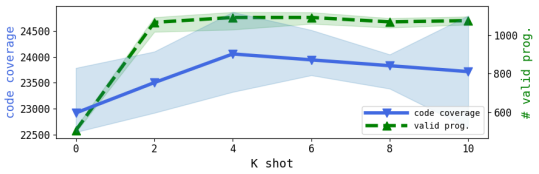



**图 8：随着示例数量 K 的增加，FuzzGPT-FS 的表现。**

**# 个示例。**我们研究 $K$ 的影响，即上下文中的示例数量。图 8 展示了随着示例数量增加，代码覆盖率和有效程序数量的变化趋势。我们首先注意到，在覆盖率和生成的有效程序数量方面，不提供示例（$K=0$）是最差的。随着我们逐渐增加 $K$，覆盖率显著提升，这表明使用少样本学习为大语言模型提供上下文示例供其学习是有益的。不过，我们发现，随着示例数量进一步增加，覆盖率实际上开始下降。这可能是因为历史触发错误示例过多，导致大语言模型限制其生成创造力并受到干扰。事实上，它已经有了已有工作 [76] 中也观察到，加入更多少样本示例的干扰性提示结构可能会降低大语言模型的性能。

*6.3.2* 零样本学习。**模型规模。**图 7b 展示了随着所用模型规模的变化，使用 FuzzGPT-ZS 生成的代码覆盖率和有效程序数量。我们首先观察到，随着模型规模增大，整体表现明显提升，但 Codex 的覆盖率有所下降。一个可能的原因是，Codex 生成的有效程序多得多，因此可能无法覆盖通过异常处理代码获得的覆盖率（这部分由无效程序覆盖）。此外，这也可能是因为零样本设置中使用了部分程序，即直接复用历史触发漏洞代码的一部分。因此，像 CodeGen 这样较小的模型无需生成少样本设置所要求的完整代码片段，也能取得较高的覆盖率。尽管如此，我们仍然观察到，像 Codex 这样较大的模型能够生成数量最多的有效程序，而这些程序对于测试深度学习库十分重要。

**提示策略。**我们评估了三种不同的零样本提示策略：**编辑**（编辑完整程序）、**补全**（补全部分程序，这是我们的默认零样本变体），以及基线策略 **completion-NL**，即仅向大语言模型提供自然语言描述、不提供代码来进行补全。如表 4 所示，与 completion-NL 相比，completion 的覆盖率显著更高，这凸显了在提示中加入历史触发漏洞的部分程序的有效性。值得注意的是，completion 的有效率低于 completion-NL，因为部分代码包含不常见的模式，因此更难生成语义上有效的补全。编辑的有效率极低，是因为要全自动地编辑现有程序以使用新的 API，对 Codex 来说更具挑战性。

**表 4：采用不同提示策略的 FuzzGPT-ZS。**

| 提示 | 有效 API | 所有 API | 有效程序 | 所有程序 | 有效率（%） | Cov |
|---|---|---|---|---|---|---|
| 编辑 | 22 | 304 | 20 | 1609 | 1.22% | 19440 |
| completion-NL | **192** | **400** | **506** | **4972** | **10.17%** | 22917 |
| 补全 | 112 | 362 | 238 | 3470 | 6.86% | **25893** |

*6.3.3* 微调。**模型规模。**我们研究了不同模型规模下的微调性能。表 5 显示，CodeGen-6B 是所研究的规模最大的 CodeGen 模型，确实实现了最高的代码覆盖率。然而，经过微调后，模型越大并不总是越好。例如，2B 模型拥有数量最多的独特有效程序。这或许可以用有效性与非寻常性的权衡来解释：当我们微调大语言模型，使其生成更偏向非寻常性时，可能会损失一些保持有效性的信息，而这两方面都有助于提高代码覆盖率和最终的漏洞发现能力。

**表 5：不同大语言模型规模下的 FuzzGPT-FT。**

| 模型 | 有效 API | 所有 API | 有效程序 | 所有程序 | 有效率（%） | Cov |
|---|---|---|---|---|---|---|
| 350M | 118 | **329** | 682 | **4236** | 16.10% | 23352.2 |
| 2B | 118 | 292 | **835** | 3393 | **24.60%** | 22688.8 |
| 6B | **131** | 311 | 768 | 3532 | 21.75% | **24420.8** |

**提示策略。**根据表6，与少样本结果类似，在微调过程中加入缺陷描述，也有助于微调后的大语言模型生成更多样化的输出（即更多独特且有效的 API/程序），并获得更高的覆盖率。

**表6：采用不同提示策略的 FuzzGPT-FT。**

| 提示 | 有效 API | 所有 API | 有效程序 | 所有程序 | 有效率（%） | Cov |
|---|---|---|---|---|---|---|
| 含思维链 | 131 | 311 | 768 | 3532 | 21.75% | 24420.8 |
| 不含思维链 | 83 | 250 | 434 | 2659 | 16.34% | 24242.8 |




**图9：FuzzGPT发现的软件缺陷示例。**

## 6.4 漏洞发现

鉴于漏洞发现和报告需要投入大量人工成本，本研究问题主要关注默认设置：采用第3.3节中的所有预言机的 FuzzGPT-FS。同时，我们预计 FuzzGPT-ZS/FuzzGPT- FT 在漏洞发现方面也会有效（鉴于其在代码覆盖率和崩溃检测方面的表现），并可能发现更多软件缺陷。

软件缺陷统计汇总见表7。FuzzGPT共检测到76个软件缺陷，其中61个已得到确认，包括49个此前未知的软件缺陷（其中6个已修复）。此外，**待确认** 列显示尚未确认的软件缺陷，**不予修复** 列显示被开发者驳回的软件缺陷（通常是由于精度问题或效率问题）。值得注意的是，**高优先级** 列显示FuzzGPT新发现的高优先级软件缺陷或安全漏洞数量。需要注意的是，TensorFlow中得到确认或修复的软件缺陷较少，因为TensorFlow的开发者活跃度较低（如第4节所述，TensorFlow中的拉取请求较少）。在49个已确认的新软件缺陷中（包括25个崩溃和24个不一致性，其中30个与自动微分相关），只有11个可以通过运行TitanFuzz（结合我们的预言机）发现，另有2个可以通过使用我们的预言机直接重新运行历史触发漏洞程序发现。接下来，我们将介绍FuzzGPT检测到的两个典型软件缺陷。

**表7：检测到的漏洞汇总。**

| | 总计 | 已确认（已修复） | | 待处理 | 不予修复 | 高优先级 |
|---|---|---|---|---|---|---|
| | | 未知 | 已知 | | | |
| PyTorch | 43 | 33 (6) | 5 (1) | 1 | 4 | 3 |
| TensorFlow | 33 | 16 (0) | 7 (0) | 5 | 5 | 8 |
| 总计 | 76 | 49 (6) | 12 (1) | 6 | 9 | 11 |

图 9a 展示了一个崩溃错误：当我们对形状为零维的特殊张量应用 `PixelShuffle`，然后使用 `jacrev`计算梯度时，就会触发该错误。通常，`PixelShuffle` 接受一个四维张量，其最后一个维度表示图像的宽度，通常大于 1。Fuz- zGPT 从一个上下文中的历史错误示例 [22]中了解到，零维张量可能会触发崩溃，并生成了形状为 `(1,1,1,1,0)` 的异常输入传给 `PixelShuffle`，导致梯度计算期间发生浮点异常。由于 `PixelShuffle` 是计算机视觉应用中常用的 API [59]，而崩溃可能导致安全风险，因此 PyTorch 开发者将此错误标记为**高优先级**并立即修复。

图 9b 展示了一个 TensorFlow 缺陷：生成的模糊测试代码片段先复制了 NumPy 数组`x`，然后修改其值。在这种情况下，张量`x_copy` 应保持不变。然而，我们发现，在 CPU 上，为`x[3]`赋予新值后，复制得到的张量`x_copy` 的值也会随之改变！有趣的是，FuzzGPT 生成的缺陷描述（位于第一行）似乎准确捕捉到了这一缺陷的原因！以往的工作（包括同样使用大语言模型进行生成的 TitanFuzz）无法触发这一缺陷，因为它需要先调用`copy` API，再修改原始数据的值——这一系列操作并不自然。FuzzGPT 成功发现了这一缺陷，是因为我们的思维链提示指示它先预测一个合理的漏洞原因“缓冲区共享问题”，并将其触发条件“复制数组时……”一并作为缺陷描述。遵循这一描述，FuzzGPT 能够生成这种在训练数据集中极为罕见、因而难以由 TitanFuzz 生成的非常规程序。该缺陷可通过悄无声息地更改原始数组的副本来实施数据操纵攻击，并已由 Google 安全团队进一步确认属于**安全漏洞**。

# 7 讨论

**表 8：不含历史信息的 FuzzGPT 与 ChatGPT。**

| 系统消息：你是一个 PyTorch 模糊测试器。 | # 应用程序接口 | | # 程序 | | | |
|---|---|---|---|---|---|---|
| 提示：请生成一个使用二维卷积的程序…… | 有效 | All | 有效 | All | 有效率（%） | Cov |
| 以展示示例用法**(基线)** | 102 | 183 | 2298 | 4209 | **55.00%** | 20454 |
| 以一种你在训练数据集中未曾见过的方式 | 190 | 297 | **2306** | 4756 | 48.00% | **21077** |
| 以一种非常有创意的方式 | 201 | 318 | 2104 | 4801 | 44.00% | 20971 |
| 以一种非常规的方式 | **212** | **330** | 2038 | **4880** | 42.00% | 20759 |
| 以开发者在实践中很少采用的方式 | 200 | 306 | 2056 | 4841 | 42.00% | 20759 |
| 以一种非常奇怪的方式 | 181 | 291 | 1735 | 4854 | 36.00% | 20377 |

**FuzzGPT 不使用历史信息。**迄今为止，我们利用历史触发漏洞程序来引导大语言模型生成非常规程序。与此同时，随着大语言模型遵循指令的能力不断提升，也可以直接指示大语言模型（不提供任何历史信息）生成用于模糊测试的非常规程序。为此，我们使用一系列具有代表性的提示，对消融研究中采用的 50 个 PyTorch API 测试了最先进的 ChatGPT [47]。具体而言，我们首先指示 ChatGPT 生成某个 API 的典型用法示例，作为基线；随后使用其他各种提示，指示 ChatGPT 生成非常规程序。表 8 展示了精简后的提示和主要结果。从表中可以看出，所有研究过的提示都能帮助 ChatGPT 覆盖远多于基线的 API，这表明 ChatGPT 及类似模型（如 GPT-4 [48]）能够理解指令，并生成更有趣、可能覆盖有趣库路径/行为的程序。另一个有趣的发现是，与基线相比，其他提示的有效率都较低，因为不常见的程序更容易运行失败。**示例选择的影响。**除了默认 FuzzGPT-FS 变体采用的随机示例选择外，我们还研究了用于少样本模糊测试的上下文示例选择策略。直观而言，与目标 API 在语义或语法上相似的 API 示例，可能提供更相关的触发缺陷的模式，从而更便于大语言模型利用。反之，多样化的示例也能提供互为补充的触发缺陷模式或要素，供大语言模型利用或组合。因此，受已有工作 [80], 启发，我们设计了一组经过平滑处理的最大边际相关性（MMR） [9]引导选择策略，包括偏向与目标 API 相似的 API、偏向多样化 API，以及介于两者之间的策略。有趣的是，我们发现默认的随机策略与所有研究过的变体相比也颇具竞争力。主要原因可能是，现代大语言模型即使从不相似的示例中也能充分学习

程序生成；这样一来，随机选择就能提供多样化的要素，促进有效生成。

**有效性威胁**。 内部效度的主要威胁在于我们实现和实验中可能存在的软件缺陷。为减轻这些威胁，我们对代码进行了严格审查。外部效度的主要威胁在于所使用的目标系统。为降低这些威胁，我们选择了两个最受欢迎的深度学习库 PyTorch 和 TensorFlow，它们也在近期研究中得到了广泛研究 [14, 15, 26, 71, 77]。最后，我们还采用了以往模糊测试研究中广泛使用的指标 [14, 71, 77], , 例如真实缺陷检测和代码覆盖率。

## 8 结论

我们介绍了 FuzzGPT，这是首个利用历史上触发漏洞的程序来启发大语言模型进行边界情况模糊测试的方法。与十多年来一直在研究如何利用此类历史信息的传统模糊测试技术相比，FuzzGPT 完全自动化、具有通用性，并且适用于深度学习库模糊测试等具有挑战性的领域。此外，FuzzGPT 还展示了 ChatGPT 在无需任何历史信息的情况下进行边界情况程序生成的潜力。实验结果表明，FuzzGPT 显著优于现有的深度学习库模糊测试器，并能够检测出 PyTorch 和 TensorFlow 中的各种软件缺陷。

**工件可用性**。我们在 [2] 提供工件。

## 参考文献

[1] https://github.com/nedbat/coveragepy. [2] 2023. FuzzGPT 工件。https://github.com/ise-uiuc/FuzzGPT. [3] 2023. requests. https://pypi.org/project/requests/. [4] Wasi Uddin Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-Wei Chang。2021。用于程序理解与生成的统一预训练。arXiv: 2103 06333 [ ]。cs.CL [5] Marcel Boehme、Cristian Cadar 和 Abhik ROYCHOUDHURY。2021。模糊测试：挑战与思考。*IEEE Software* 38，3（2021），79–86。[6] Dalvin Brown。2021。医院借助人工智能解决一个由来已久的问题：医生床边态度欠佳。*TheWashingtonPost*（2021）。https://www.washingtonpost.com/technology/2021/02/16/virtual-ai-hospital-patients/.[7] Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。2020。语言模型是少样本学习器。神经信息处理系统进展 33（2020），1877–1901。[8] Tom B. Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Pra- fulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。2020。语言模型是少样本学习器。arXiv:2005.14165。[9] Jaime Carbonell 和 Jade Goldstein。1998。使用 MMR 和基于多样性的重排序对文档重新排序并生成摘要。载于第*21*届 *ACM SIGIR* 信息检索研究与开发年度国际会议论文集。335–336。[10] Junjie Chen、Guancheng Wang、Dan Hao、Yingfei Xiong、Hongyu Zhang 和 Lu Zhang。2019。用于编译器测试程序生成的历史引导配置多样化。载于*2019*年第*34*届 *IEEE/ACM* 自动化软件工程国际会议（*ASE*）。IEEE，305–316。[11] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman 等。2021。评估在代码上训练的大语言模型。*arXiv* 预印本 *arXiv:2107.03374*（2021）。[12] Kyunghyun Cho、Bart van Merriënboer、Caglar Gulcehre、Dzmitry Bahdanau、Fethi Bougares、Holger Schwenk 和 Yoshua Bengio。2014。用于统计机器翻译的 RNN编码器-解码器短语表示学习。载于*2014*年自然语言处理实证方法会议（*EMNLP*）论文集。计算语言学协会，多哈，卡塔尔， 1724 1734 10 3115 1 14 1179 – 。https://doi.org/ 。/v /D - [13] Chris Cummins、Pavlos Petoumenos、Alastair Murray 和 Hugh Leather。2018。通过深度学习进行编译器模糊测试。载于第27届 ACM SIGSOFT 软件测试与分析国际研讨会。95–105。[14] Yinlin Deng、Chunqiu Steven Xia、Haoran Peng、Chenyuan Yang 和 Lingming Zhang。2023。大语言模型是零样本模糊测试器：通过大语言模型对深度学习库进行模糊测试。载于第 32 届 ACM SIGSOFT 软件测试与分析国际研讨会论文集（ISSTA 2023）。[15] Yinlin Deng、Chenyuan Yang、Anjiang Wei 和 Lingming Zhang。2022。通过自动化关系 API 推断对深度学习库进行模糊测试。载于第 30 届 ACM 欧洲软件工程联合会议与软件工程基础研讨会论文集（新加坡，新加坡）（ESEC/FSE 2022）。计算机协会，美国纽约州纽约市，44–56。https://doi.org/10.1145/3540250.3549085[16] Yangruibo Ding、Zijian Wang、Wasi Uddin Ahmad、Murali Krishna Ramanathan、Ramesh Nallapati、Parminder Bhatia、Dan Roth 和 Bing Xiang。2022。CoCoMIC：通过联合建模文件内和跨文件上下文实现代码补全。arXiv 预印本 arXiv:2212.10007（2022）。[17] Alastair F Donaldson、Hugues Evrard、Andrei Lascu 和 Paul Thomson。2017。图形着色器编译器的自动化测试。《ACM 编程语言》1，OOPSLA（2017），1–29。[18] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。2020。CodeBERT：面向编程语言和自然语言的预训练模型。arXiv:2002.08155。[19] Andrea Fioraldi、Dominik Maier、Heiko Eißfeldt 和 Marc Heuse。2020。AFL++：结合模糊测试研究的渐进步骤。载于第 14 届 USENIX 攻击性技术会议论文集（WOOT'20）。USENIX 协会，美国，文章 10，共 1 页。[20] Daniel Fried、Armen Aghajanyan、Jessy Lin、Sida Wang、Eric Wallace、Freda Shi、Ruiqi Zhong、Wen-tau Yih、Luke Zettlemoyer 和 Mike Lewis。2022。InCoder：用于代码填充与合成的生成模型。arXiv 预印本 arXiv:2204.05999（2022）。[21] Leo Gao、Stella Biderman、Sid Black、Laurence Golding、Travis Hoppe、Charles Foster、Jason Phang、Horace He、Anish Thite、Noa Nabeshima 等。2020。The Pile：用于语言建模的多样化文本数据集，规模达 800GB。arXiv 预印本 arXiv:2101.00027（2020）。[22] GitHub。2022。当最后三个维度的输入大小为 0 时，torch.nn.PixelShuffle 因浮点异常而崩溃。（2022）。https://github.com/pytorch/pytorch/issues/85155。[23] PatriceGodefroid、HilaPeleg 和RishabhSingh。2017。Learn&Fuzz：用于输入模糊测试的机器学习。载于 2017 年第 32 届 IEEE/ACM 自动化软件工程国际会议（ASE）。 50 59 10 1109 2017 8115618 – 。[24] 2022 https://doi.org/ 。/ASE。。J. Gu、X. Luo、Y. Zhou 和 X. Wang。。Muffin：通过神经网络架构模糊测试来测试深度学习库。载于 2022 年 IEEE/ACM 第 44 届国际软件工程会议（ICSE）。IEEE 计算机学会，洛斯阿拉米托斯，美国，1418–1430。https://doi.org/10.1145/3510003.3510092[25] Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、Shujie Liu、Long Zhou、Nan Duan、Alexey Svyatkovskiy、Shengyu Fu、Michele Tufano、Shao Kun Deng、Colin Clement、Dawn Drain、Neel Sundaresan、Jian Yin、Daxin Jiang 和 Ming Zhou。2021。GraphCodeBERT：利用数据流预训练代码表示。arXiv:2009.08366 [cs.SE][26] Qianyu Guo、Xiaofei Xie、Yi Li、Xiaoyu Zhang、Yang Liu、Xiaohong Li 和 Chao Shen。2020 2020 35。Audee：深度学习框架的自动化测试。载于第 届 IEEE/ACM 自动化软件工程国际会议（ASE）。IEEE，486–498。[27] Abram Hindle、Earl T. Barr、Zhendong Su、Mark Gabel 和 Premkumar Devanbu。2012。论软件的自然性。载于第 34 届国际软件工程会议论文集（ICSE '12）。IEEE 出版社，837–847。[28] Sepp Hochreiter 和 Jürgen Schmidhuber。1997。长短期记忆。《神经计算》9（12，1997），1735–80。[29] Christian Holler、Kim Herzig、Andreas Zeller 等。2012。使用代码片段进行模糊测试。载于 USENIX 安全研讨会。445–458。[30] HuggingFace 2022。Hugging Face。https://huggingface.co。[31] George Klees、Andrew Ruef、Benji Cooper、Shiyi Wei 和 Michael Hicks。2018。评估模糊测试。载于 2018 年 ACM SIGSAC 计算机与通信安全会议论文集。2123–2138。[32] Vu Le、Mehrdad Afshari 和 Zhendong Su。2014。通过输入模等价验证编译器。《ACM SIGPLAN 通讯》49，6（2014），216–226。[33] Vu Le、Chengnian Sun 和 Zhendong Su。2015。通过引导式随机程序变异发现深层编译器缺陷。《ACM SIGPLAN 通讯》50，10（2015），386–399。[34] Suyoung Lee、HyungSeok Han、Sang Kil Cha 和 Sooel Son。2020。Montage：由神经网络语言模型引导的 JavaScript 引擎模糊测试器。载于第 29 届 USENIX 安全研讨会会议论文集。2613–2630。[35] Jiawei Liu、Jinkun Lin、Fabian Ruffy、Cheng Tan、Jinyang Li、Aurojit Panda 和 Lingming Zhang。2023。NNSmith：为深度学习编译器生成多样且有效的测试用例。载于 ASPLOS。530–543。[36] Jiachang Liu、Dinghan Shen、Yizhe Zhang、Bill Dolan、Lawrence Carin 和 Weizhu Chen。2022。什么样的上下文示例适合 GPT-3？ 载于《深度学习由内而外》（DeeLIO 2022）：第三届深度学习架构知识抽取与集成研讨会论文集。计算语言学协会，爱尔兰都柏林及线上，100–114。

https://doi.org/ 10 18653 1 2022 1 10 。/v / 。deelio- 。Jiawei Liu、[37] Yuxiang Wei、Sen Yang、Yinlin Deng 和 Lingming Zhang。2022。通过联合 IR-Pass 变异进行覆盖率引导的张量编译器模糊测试。《ACM 编程语言》6，OOPSLA1，文章 73（2022 年 4 月），共 26 页。https://doi.org/10.1145/3527317[38] PengfeiLiu、WeizheYuan、JinlanFu、ZhengbaoJiang、HiroakiHayashi 和 Graham Neubig。2023。预训练、提示与预测：自然语言处理提示方法的系统综述。《计算机调查》55，9（2023），1–35。[39] Xiao Liu、Xiaoting Li、Rupesh Prajapati 和 Dinghao Wu。2019。Deepfuzz：自动生成语法有效的 C 程序以进行模糊测试。载于 AAAI 人工智能会议论文集，第 33 卷。1044–1051。[40] Ilya Loshchilov 和 Frank Hutter。2019。解耦权重衰减正则化。载于国际学习表征会议。https://openreview.net/forum?id = 6 7 Bkg RiCqY M. Zalewski。《American Fuzzy Lop 白皮书》。[41] https: //lcamtuf.coredump.cx/afl/technical_ 2016 details.txt。[42] Aman Madaan、Alexander Shypula、Uri Alon、Milad Hashemi、Parthasarathy Ran- ganthan、Yiming Yang、Graham Neubig 和 Amir Yazdanbakhsh。2023。学习能够提升性能的代码编辑。arXiv 预印本 arXiv:2302.07867（2023）。[43] Sewon Min、Xinxi Lyu、Ari Holtzman、Mikel Artetxe、Mike Lewis、Hannaneh Hajishirzi 和 Luke Zettlemoyer。2022。重新思考示例的作用：什么让上下文学习奏效？arXiv 预印本 arXiv:2202.12837（2022）。[44] Tanya Mohn。2022。人工智能能否彻底终结车祸？潜力确实存在。《纽约时报》（2022）。https://www.nytimes.com/2022/04/19/technology/ai-road-car-safety.html。[45] Erik Nijkamp、Bo Pang、Hiroaki Hayashi、Lifu Tu、Huan Wang、Yingbo Zhou、Silvio Savarese 和 Caiming Xiong。2022。Codegen：面向多轮程序合成的开放式大语言模型。arXiv 预印本 arXiv:2203.13474（2022）。[46] Maxwell Nye、Anders Johan Andreassen、Guy Gur-Ari、Henryk Michalewski、Jacob Austin、David Bieber、David Dohan、Aitor Lewkowycz、Maarten Bosma、David Luan 等。2021。展示你的工作：语言模型中用于中间计算的草稿区。arXiv 预印本 arXiv:2112.00114（2021）。[47] OpenAI。2023。ChatGPT。（2023）。https://openai.com/blog/chatgpt。[48] OpenAI。2023。GPT-4 技术报告。arXiv:2303.08774 [cs.CL][49] Fabio Petroni、Tim Rocktäschel、Sebastian Riedel、Patrick Lewis、Anton Bakhtin、Yuxiang Wu 和 Alexander Miller。2019。语言模型是知识库吗？载于《2019 年自然语言处理实证方法会议与第九届自然语言处理国际联合会议论文集》（EMNLP-IJCNLP）。2463–2473。[50] Hung Viet Pham、Thibaud Lutellier、Weizhen Qi 和 Lin Tan。2019。CRADLE：通过跨后端验证检测并定位深度学习库中的软件缺陷。载于 2019 年 IEEE/ACM 第 41 届国际软件工程会议（ICSE）。1027 1038 10 1109 2019 00107 – 。https://doi.org/ 。/ICSE。 。PyTorch。[51] 2023 PyTorch。http://pytorch.org。[52] Alec Radford、Karthik Narasimhan、Tim Salimans、Ilya Sutskever 等。2018。通过生成式预训练提升语言理解。（2018）。[53] Alec Radford、Jeffrey Wu、Rewon Child、David Luan、Dario Amodei、Ilya Sutskever 等。2019。语言模型是无监督的多任务学习者。《OpenAI 博客》1，8（2019），9。[54] Maithra Raghu 和 Eric Schmidt。2020。深度学习与科学发现综述。arXiv 预印本 arXiv:2003.11755（2020）。[55] Ohad Rubin、Jonathan Herzig 和 Jonathan Berant。2022。学习为上下文学习检索提示。载于《2022 年计算语言学协会北美分会人类语言技术会议论文集》。2655–2671。[56] Adam Satariano 和 Cade Metz。2023。利用人工智能检测医生漏诊的乳腺癌。《纽约时报》（2023）。https://www.nytimes.com/2023/03/05/technology/artificial-intelligence-breast-cancer-detection.html。[57] H. Scudder。1965。某些自适应模式识别机器的错误概率。《IEEE 信息论汇刊》11，3（1965），363–371。https://doi.org/ 10 1109 1965 1053799 。/TIT。 。[58] 2023 Danny Shapiro。。交通运输新纪元：了解 AI 和元宇宙如何在 GTC 重塑汽车行业。Nvidia 博客（2023）。https://blogs.nvidia.com/blog/2023/02/16/ai-metaverse-shaping-automotive-industry-gtc/。[59] Wenzhe Shi、Jose Caballero、Ferenc Huszár、Johannes Totz、Andrew P Aitken、Rob Bishop、Daniel Rueckert 和 Zehan Wang。2016。使用高效亚像素卷积神经网络进行实时单幅图像与视频超分辨率。载于 IEEE 计算机视觉与模式识别会议论文集。1874–1883。[60] Taylor Shin、Yasaman Razeghi、Robert L Logan IV、Eric Wallace 和 Sameer Singh。2020。AutoPrompt：利用自动生成的提示从语言模型中引出知识。载于 EMNLP 2020。4222–4235。[61] Ezekiel Soremekun、Esteban Pavese、Nikolas Havrikov、Lars Grunske 和 Andreas Zeller。2020。地狱般的输入。《IEEE 软件工程汇刊》48，4（2020），1138–1153。[62] Chengnian Sun、Vu Le 和 Zhendong Su。2016。通过实时代码变异发现编译器缺陷。载于 2016 年 ACM SIGPLAN 面向对象编程、系统、语言与应用国际会议论文集。849–863。[63] Michael Sutton、Adam Greene 和 Pedram Amini。2007。《模糊测试：蛮力漏洞发现》。Addison-Wesley Professional。[64] TensorFlow 2023。TensorFlow。https://www.tensorflow.org。[65] Alina Tugend。2021。一款更智能的应用正在关注你的钱包。《纽约时报》（2021）。https://www.nytimes.com/2021/03/09/business/apps-personal-finance-budget.html。[66] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。2017。注意力就是你所需要的一切。神经信息处理系统进展 30（2017）。[67] Jiannan Wang、Thibaud Lutellier、Shangshu Qian、Hung Viet Pham 和 Lin Tan。2022。EAGLE：创建等价图以测试深度学习库。（2022）。[68] Mingzhe Wang、Zhiyong Wu、Xinyi Xu、Jie Liang、Chijin Zhou、Huafeng Zhang 和 Yu Jiang。2021。覆盖率引导的企业级数据库管理系统模糊测试的行业实践。载于《2021 IEEE/ACM 第43届国际软件工程会议：软件工程实践》（ICSE-SEIP）。IEEE，328–337。[69] Xingyao Wang、Sha Li 和 Heng Ji。2022。Code4struct：从自然语言生成代码，以实现少样本结构化预测。arXiv 预印本 arXiv:2210.12810（2022）。[70] Zan Wang、Ming Yan、Junjie Chen、Shuang Liu 和 Dongdi Zhang。2020。通过有效的模型生成测试深度学习库。载于《第28届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。788–799。[71] Anjiang Wei、Yinlin Deng、Chenyuan Yang 和 Lingming Zhang。2022。测试的免费午餐：对开源深度学习库进行模糊测试。载于《2022 IEEE/ACM 第44届国际软件工程会议》（ICSE）。995–1007。https://doi.org/10.1145/3510003.3510041[72] Jason Wei、Xuezhi Wang、Dale Schuurmans、Maarten Bosma、Ed Chi、Quoc Le 和 Denny Zhou。2022。思维链提示可激发大语言模型的推理能力。arXiv 预印本 arXiv:2201.11903（2022）。[73] Dominik Winterer、Chengyu Zhang 和 Zhendong Su。2020。类型感知运算符变异在测试 SMT 求解器方面异乎寻常的有效性。ACM 编程语言 4，OOPSLA（2020），1–25。[74] Chunqiu Steven Xia 和 Lingming Zhang。2023。对话式自动程序修复。arXiv 预印本 arXiv:2301.13246（2023）。[75] Danning Xie、Yitong Li、Mijung Kim、Hung Viet Pham、Lin Tan、Xiangyu Zhang 和 Michael W Godfrey。2022。DocTer：文档引导的模糊测试，用于测试深度学习API函数。载于《第31届 ACM SIGSOFT 软件测试与分析国际研讨会论文集》。[76] Sang Michael Xie、Aditi Raghunathan、Percy Liang 和 Tengyu Ma。2021。将上下文学习解释为隐式贝叶斯推断。arXiv 预印本 arXiv:2111.02080（2021）。[77] Chenyuan Yang、Yinlin Deng、Jiayi Yao、Yuxing Tu、Hanchi Li 和 Lingming Zhang。2023。对深度学习库中的自动微分进行模糊测试。载于《国际软件工程会议》（ICSE）。即将发表。[78] Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。2011。发现并理解 C编译器中的软件缺陷。载于《第32届 ACM SIGPLAN 编程语言设计与实现会议论文集》。283–294。[79] Guixin Ye、Zhanyong Tang、Shin Hwei Tan、Songfang Huang、Dingyi Fang、Xiaoyang Sun、Lizhong Bian、Haibo Wang 和 Zheng Wang。2021。通过深度编译器模糊测试对 JavaScript 引擎进行自动一致性测试。载于《第42届 ACM SIGPLAN 编程语言设计与实现国际会议论文集》。435–450。[80] Xi Ye、Srinivasan Iyer、Asli Celikyilmaz、Ves Stoyanov、Greg Durrett 和 Ramakanth Pasunuru。2022。用于有效上下文学习的互补解释。arXiv 预印本 arXiv:2211.13892（2022）。[81] Shafiq Joty Yue Wang、Weishi Wang 和 Steven C.H. Hoi。2021。CodeT5：面向代码理解与生成的标识符感知统一预训练编码器-解码器模型。载于 EMNLP 2021。[82] Andreas Zeller、Rahul Gopinath、Marcel Böhme、Gordon Fraser 和 Christian Holler。2019。《模糊测试之书》。[83] Qirun Zhang、Chengnian Sun 和 Zhendong Su。2017。用于严格编译器测试的骨架程序枚举。载于《第38届 ACM SIGPLAN 编程语言设计与实现会议论文集》。347–361。[84] Yingquan Zhao、Zan Wang、Junjie Chen、Mengdi Liu、Mingyuan Wu、Yuqun Zhang 和 Lingming Zhang。2022。用于 JVM测试的历史驱动测试程序合成。载于 ICSE 2022。1133–1144。[85] Hao Zhong。2022。利用来自软件缺陷报告的真实程序丰富编译器测试。载于《第37届 IEEE/ACM 自动化软件工程国际会议》。1–12。[86] Rui Zhong、Yongheng Chen、Hong Hu、Hangfan Zhang、Wenke Lee 和 Dinghao Wu。2020。Squirrel：利用语言有效性和覆盖率反馈测试数据库管理系统。载于《2020 ACM SIGSAC 计算机与通信安全会议论文集》。955–970。[87] Barret Zoph、Golnaz Ghiasi、Tsung-Yi Lin、Yin Cui、Hanxiao Liu、Ekin Dogus Cubuk 和 Quoc Le。2020。重新思考预训练与自训练。神经信息处理系统进展 33（2020），3833–3845。