DiverseVul：一种用于基于深度学习的漏洞检测的新型易受攻击源代码数据集

Yizheng Chen 马里兰大学
yzchen@umd.edu

Zhoujie Ding 加州大学伯克利分校
zhoujie.ding@berkeley.edu

Lamya Alowain 阿卜杜勒阿齐兹国王科技城
lalowain@kacst.edu.sa

Xinyun Chen Google DeepMind
xinyunchen@google.com

David Wagner 加州大学伯克利分校
daw@cs.berkeley.edu

arXiv:2304.00409v2 [cs.CR] 9 Aug 2023

摘要

我们提出并发布了一个新的易受攻击源代码数据集。我们通过爬取安全问题网站、从相关项目中提取漏洞修复提交和源代码来整理该数据集。我们的新数据集包含18,945个易受攻击函数，涵盖150种常见弱点枚举，以及从7,514次提交中提取的330,492个非易受攻击函数。与所有以往数据集的总和相比，我们的数据集多覆盖了295个项目。

将我们的新数据集与以往数据集相结合后，我们分析了使用深度学习检测软件漏洞所面临的挑战和有前景的研究方向。我们研究了属于4个类别的11种模型架构。结果表明，由于误报率高、F1 分数低，以及难以检测较难的常见弱点枚举，深度学习尚未做好用于漏洞检测的准备。特别是，我们展示了基于深度学习的模型在部署时面临的一项重要泛化挑战。我们发现，增加训练数据的规模可能无法进一步提升深度学习模型在漏洞检测方面的性能，但可能有助于提高其对未见项目的泛化能力。

我们还指出了值得期待的未来研究方向。我们的实验表明，大语言模型（LLM）是基于机器学习的漏洞检测领域一个很有前景的研究方向，其表现优于利用代码结构特征的图神经网络（GNN）。此外，开发源代码特定预训练目标也是提升漏洞检测性能的一个有前景的研究方向。

关键词

数据集、漏洞检测、深度学习、大语言模型

ACM 引用格式：
Yizheng Chen、Zhoujie Ding、Lamya Alowain、Xinyun Chen 和 David Wagner。2023。DiverseVul：一种新的易受攻击源代码数据集，用于基于深度学习的漏洞检测。在 第 *26* 届攻击、入侵与防御研究研讨会（*RAID '23*），*2023*年*10*月*16*日至*18*日，香港，中国香港。ACM，纽约，纽约州，美国，15 页。https://doi.org/10.1145/3607199.3607242

允许免费将本作品的部分或全部内容制作成数字版或纸质版，仅供个人或课堂使用，但不得为营利或商业利益而复制或传播，并且副本须保留本声明及首页的完整引用信息。本作品中第三方组件的版权必须得到尊重。其他用途请联系所有者/作者。*RAID '23*，*2023*年*10*月*16*日至*18*日，香港，香港© 2023 版权所有归所有者/作者所有。ACM ISBN 979-8-4007-0765-0/23/10。
https://doi.org/10.1145/3607199.3607242

1 引言

检测软件漏洞对于预防网络犯罪和经济损失至关重要，但截至目前，这仍然是一个难题。传统的静态和动态漏洞检测技术存在不足。鉴于深度学习在图像和自然语言应用中的巨大成功，我们自然会思考，深度学习是否能够增强我们检测漏洞的能力 [4, 15, 18, 25, 33]。然而，正如本文所展示的，在深度学习能够在易受攻击源代码检测方面取得出色性能之前，我们仍需克服许多挑战。

要使深度学习取得成功，我们需要一个大规模的易受攻击源代码数据集。我们发布了一个新的开放漏洞数据集，用于 C/C++，名为 DiverseVul。为整理该数据集，我们爬取安全问题网站，收集漏洞报告，提取每个漏洞对应的漏洞修复提交，克隆相关项目，并从中提取易受攻击和不易受攻击的源代码。我们的数据集包含从 7,514 次提交中提取的 18,945 个易受攻击函数和 330,492 个不易受攻击函数，涵盖 150种CWE漏洞类型。其规模是之前规模最大且最多样化的数据集 CVEFixes 中 C/C++ 数据的两倍以上。我们的数据集更加多样化，涵盖的项目数量比此前发布的所有数据集的总和多近 50%。我们在 https://github.com/wagner-group/diversevul 上向社区公开发布了 DiverseVul数据集。

我们的新数据集使我们能够研究最先进的深度学习方法，并深入了解有前景的研究方向以及基于机器学习的漏洞检测所面临的挑战。具体而言，我们研究了几个问题。更多训练数据是否有帮助，还是模型已经趋于饱和？模型架构是否会造成很大差异？使用依赖代码结构特征的最先进模型更好，还是使用大语言模型更好？更大的大语言模型是否优于更小的？进一步提升深度学习漏洞检测能力，最有前景的方向是什么？

为研究模型架构的影响，我们使用来自4个代表性模型家族的11种不同深度学习架构进行实验：图神经网络（GNN） [13], RoBERTa [10, 11, 16],GPT-2 [17, 23, 30], 和 T5 [3, 24, 29]。许多关于深度学习的研究

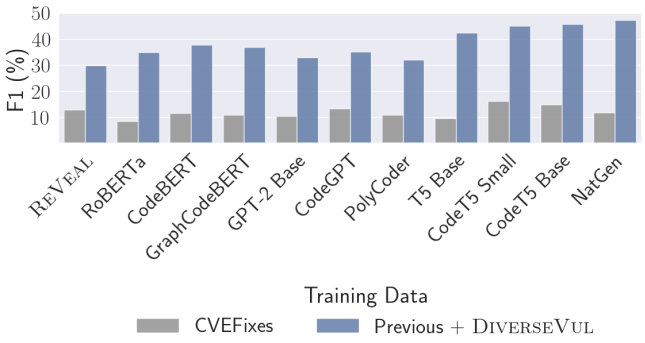


图 1：我们若干项研究结果概览。仅使用 CVEFixes 数据集训练时，ReVeal 的表现与大语言模型相当。如果数据量足够（Previous + DiverseVul），大语言模型（例如 NatGen）的表现优于上一代模型（例如 ReVeal——一种具有代码结构特征的图神经网络模型），但要看到这些优势，我们需要大型数据集。与上一代模型相比，大语言模型能更好地利用大型数据集（蓝色条形与灰色条形）。在此任务中表现最好的大语言模型 CodeT5 和 NatGen，经过了代码专属任务的预训练。

将图神经网络与代码结构特征结合用于漏洞检测 [4, 18, 33]。我们还探索将大语言模型（LLM）应用于漏洞检测，因为尽管大语言模型不使用代码结构特征，但它们在自然语言处理和代码理解方面已取得最先进的成果。我们研究了这些模型在三个数据集上的性能：（1）CVEFixes [2]，迄今已发布的最大的 C/C++ 漏洞数据集；（2）所有先前发布的数据集的组合（Devign [33],ReVeal [4], BigVul [9], CrossVul [19], CVEFixes [2]），并进行了去重；（3）这些先前数据集与我们提出的 DiverseVul 的组合（详见表3）。

我们的实验表明，在先前的数据集 CVEFixes [2], 上进行评估时，模型架构影响甚微，且大语言模型的表现与图神经网络大致相同。具体而言，在规模最大的既有数据集 CVEFixes 上，ReVeal 模型（一种图神经网络）的 F1 分数为 12.8，而大语言模型的 F1 分数为 8.5–16.3（见图 1）。人们或许会据此得出结论，认为具体的架构影响甚微。然而，在更大的数据集上进行评估时，我们会发现结论恰恰相反：大语言模型的表现显著优于图神经网络。具体而言，将所有先前发布的数据集与我们的 DiverseVul 合并后，表现最佳的大语言模型 F1 分数达到 47.2，而 ReVeal 为 29.8。

这些实验表明，我们需要大型数据集，才能可靠地评估用于漏洞检测的深度学习方法，因为随着可用训练数据量的增加，不同架构的相对性能会发生巨大变化：训练数据量增加 a 5×（从 CVEFixes 增至所有数据集），使我们最佳模型的性能从 10.5 提升至 48.9 F1 分数。这些实验还表明，大语言模型比图神经网络更能利用大型数据集：更大的数据集只能适度提升 ReVeal 的性能，却能显著提升大语言模型的性能。

然而，我们的实验表明，收集更多数据带来的性能提升可能已经停滞。将我们的数据集与先前的数据集组合后，11 个模型中有 7 个的测试性能得到提升。不过，对于表现最佳的 3 个模型，我们要么看不到提升，要么提升幅度很小（详情见第4.2节）。

遗憾的是，目前最先进的深度学习技术仍未能满足漏洞检测的实际需求。我们表现最好的模型 F1 分数为 47.2%，真正率为 43.3%，误报率为 3.5%。对于模型的实际应用而言，误报率仍然高得多。一个项目可能包含数万个函数，而这一误报率意味着会产生数百个假正例，大多数分析师恐怕都不愿逐一筛查这么多结果 [1]。

尽管面临这些挑战，图 1 表明，大语言模型（LLM）可能更适用于基于深度学习的漏洞检测。以往论文中的研究人员认为，结合代码结构特征的图神经网络在漏洞检测方面很有前景，因为它将领域知识与深度学习相结合 [4, 18, 33], 。相比之下，我们的结果表明，大语言模型（RoBERTa、GPT-2 和 T5 系列）的表现显著优于最先进的图神经网络，尤其是在使用更多数据进行训练时。具体而言，CodeT5 模型（CodeT5 Small、CodeT5 Base、NatGen）表现最佳。

与普遍认为模型规模是大语言模型取得良好表现的最重要因素这一观点相反，我们的结果表明，最重要的因素可能是大语言模型的训练方式。在代码理解任务上进行预训练似乎能带来显著提升。例如，CodeT5 Small 经过预训练，能够预测变量名和函数名，其 F1 分数平均比规模是其两倍、但未在代码上进行预训练的模型高出 8 个百分点。令人惊讶的是，我们发现，对自然语言有效的预训练任务并不能显著提升漏洞检测效果。相反，我们似乎需要代码特定预训练任务。我们认为，开发更优的代码特定预训练任务，是改进基于深度学习的漏洞检测的一条有前景的研究方向。

此外，我们发现了深度学习模型部署面临的一项重要泛化挑战。部署模型时，我们需要检测来自训练集未涵盖的新软件项目中的漏洞。我们发现，深度学习模型在这种情况下的表现非常差。具体而言，过去的研究会随机划分漏洞，将数据分为训练集和测试集，而不考虑每个漏洞所属的项目。然而，在实际应用中，我们往往希望在新项目上运行漏洞检测工具，因此训练集中不会包含来自该项目的任何漏洞。为了评估深度学习在这种情况下的性能，我们留出一组项目，称之为“未见项目”；我们使用其他项目（“已见项目”）中的漏洞进行训练，然后在未见项目中的漏洞上进行测试。所有模型在未见项目上的性能都显著下降，例如，F1 分数从已见项目上的 49% 降至未见项目上的仅 9.4%。造成这一现象的原因尚不明确；模型或许过拟合了训练集中那些特定项目所特有的模式或编码习惯。这种泛化失败很可能会成为深度学习漏洞

检测在实际应用中的一大障碍。我们希望未来的研究能够探索如何解决这一问题。我们建议采取一种简单的干预措施：在训练损失中使用类别权重，这朝着解决问题的方向迈出了一小步，但差距仍然很大，还需要开展更多工作。

最后，我们量化了我们数据集以及以往数据集中的标签噪声。标签噪声是基于机器学习的漏洞检测研究面临的一项重大挑战。为了从漏洞修复提交中提取易受攻击函数，我们遵循当前最先进的方法（Devign [33], ReVeal [4], BigVul [9], CrossVul [19],CVEFixes [2]均采用此方法），将这些提交修改过的函数标记为易受攻击。为了解这类标注方法的标签准确率，我们从自己的数据集中随机抽取50个易受攻击函数，并从三个收集国家漏洞数据库中提交的数据集（BigVul、CrossVul 和 CVEFixes）的并集中再抽取50个易受攻击函数。随后，我们人工分析漏洞及标记为易受攻击的函数。结果显示，DiverseVul 中易受攻击函数标签的准确率为60%，比 CVEFixes、BigVul 和 CrossVul 的并集高24%，但仍存在许多标签错误。主要挑战包括漏洞跨多个函数分布，以及漏洞修复提交中对非易受攻击函数所做的修改。我们希望本研究能迈出理解标签噪声问题的第一步，并凸显深入研究标签噪声影响的必要性。

本文的贡献如下：

- 我们发布了 DiverseVul，这是一个全新的 C/C++ 易受攻击源代码数据集。我们的数据集比此前最大的 C/C++数据集大 60%，且在所有此前的数据集中最多样化。
- 我们研究了来自 4 个不同模型家族的 11 种模型架构。结果表明，大型语言模型的表现优于基于深度学习的漏洞检测领域最先进的图神经网络；开发针对源代码的预训练目标也是一个有前景的研究方向。
- 我们指出了深度学习用于漏洞检测所面临的挑战。具体而言，我们强调了将模型泛化到训练集之外未见项目的困难。
- 我们评估了本数据集以及依赖漏洞修复提交的此前数据集中的标签噪声。

2 相关工作

在本节中，我们分析了以往公开的 C/C++ 易受攻击源代码数据集、其标注方法，以及相关的漏洞检测深度学习研究如何使用这些数据集。

合成数据集： SATE IV Juliet [22] 和 SARD [21] 是以往论文 [15, 18, 25]中常用的合成数据集。SARD 在 Juliet v1.0 测试套件的基础上进行了扩展，包含多种编程语言的测试用例。这些测试用例准确度很高，涵盖了多种常见弱点枚举。然而，这些用例是在孤立环境中根据已知的易受攻击模式构建的，旨在评估静态和动态分析工具。它们无法充分反映真实世界项目中漏洞的复杂性。VulDeePecker [15] 数据集只关注两种常见弱点枚举。研究人员根据国家漏洞数据库（NVD） [20], 中的通用漏洞披露信息，从 19 个项目中选取漏洞，并还结合了 SARD [21] 中属于这两种常见弱点枚举的测试用例。VulDeePecker 和 SARD 都是半合成数据集。

静态分析器标签： Draper [25] 数据集使用三种静态分析器（Clang、Cppcheck 和 Flawfinder）的警报生成标签。部分类别的警报被标记为易受攻击，另一些则被映射为非易受攻击。该标注数据集以函数为粒度。标签质量尚不明确，但静态分析器的标签准确率往往较低。D2A [32]对六个开源代码仓库上的静态分析器（Infer）输出进行了差分分析。对于给定 GitHub 仓库的数千组版本对，如果静态分析器在某次 Git 提交之前的版本中生成警报，但在提交之后的版本中没有生成警报，那么 D2A 就将该提交视为修复了一个漏洞。对于其余警报，D2A 将其标记为与漏洞无关。

人工标注： Devign [33] 数据集由三位安全研究人员进行标注。他们首先使用关键词，从四个代码仓库中找出可能修复漏洞的提交，以及与漏洞无关的提交。对于第一类提交，三位安全研究人员通过多数表决对这些提交进行人工审查，以确定哪些提交修复了安全漏洞。根据每个提交的标签，Devign 提取提交前发生变更的函数作为数据样本，并依据该提交的标签将其标记为易受攻击或非易受攻击。Devign 的作者发布了两个代码仓库（FFMPeg 和 Qemu）的数据。该数据集的标签质量很高，但人工标注成本高昂，耗费约600人时。

安全问题： 若干先前的数据集通过爬取安全问题来识别漏洞修复提交。ReVeal [4] 数据集使用 Chromium 安全问题和 Debian 安全跟踪器中已知安全问题的补丁进行标注。ReVeal 将安全补丁（提交）前发生变更的函数视为易受攻击，将补丁后的函数视为非易受攻击，并将所有未发生变更的函数视为非易受攻击。相比之下，我们的 DiverseVul 数据集包含 1.8万个易受攻击函数，规模是 ReVeal 的 11× 倍（表3）。

BigVul [9],CrossVul [19] 和 CVEfixes [2]从国家漏洞数据库（NVD） [20]中的通用漏洞披露（CVE）记录收集漏洞修复提交。具体而言，CVEFixes 涵盖截至 2022年8月27日发布的所有 CVE。CVEFixes 和 CrossVul 数据集涵盖多种编程语言，本文使用其中的 C/C++ 数据。这三个数据集涵盖了广泛的项目和常见弱点枚举。相比之下，我们的数据集包含更多项目、更多常见弱点枚举，漏洞修复提交的数量也翻了一倍。

其他一些 C/C++ 易受攻击源代码数据集未提供易受攻击函数，因此我们没有将其纳入实验。例如，AOSP [5] 从 Android 开源项目（AOSP）的安全公告中收集修复 CVE 的提交，其中包含针对 Android 操作系统、Linux 内核和片上系统制造商相关漏洞的补丁。PatchDB [28] 提供补丁信息，即代码差异，但没有提供足够的信息来识别其来源项目或 Git 存储库，因此无法据此还原发生更改的函数的完整代码。

安全问题有助于有效识别漏洞修复提交，因为它们基于开发人员的人工分析。

它们也能代表真实项目中的野外漏洞。因此，我们通过爬取安全问题，也收集了新的数据集 Diverse- Vul。与所有 Previous 数据集相比，DiverseVul 的多样性最高，涵盖的项目数量最多。具体而言，DiverseVul 包含来自 295个新项目的漏洞，这些项目此前未被任何真实世界数据集收录（表3）。

用于易受攻击源代码检测的深度学习：以往的研究使用长短期记忆网络 [15],卷积神经网络和循环神经网络 [25], 双向循环神经网络 [14],以及图神经网络 [4, 18, 33]来检测易受攻击的源代码。Thapa等人 [27] 最近的一篇论文表明，在涵盖两种常见弱点枚举的 VulDeePecker [15] 数据集上，大语言模型的表现优于双向长短期记忆网络和双向门控循环单元模型。然而，他们没有与图神经网络（GNN）进行比较。GNN 将程序表示为图，其中包含有助于漏洞检测的领域知识。ReVeal [4]使用从代码属性图 [31], 中提取的特征，而 VulChecker [18] 则提出了一种新的增强型程序依赖图。这些研究使用的数据集相对较小，例如 ReVeal 和 Juliet。如果我们使用更大的数据集训练模型，目前尚不清楚采用代码结构特征的 GNN 相较于大语言模型是否仍然有效。

3 数据收集

我们的目标是从多种真实世界项目中收集高质量的漏洞修复提交。我们重点收集安全问题相关数据，因为这些问题反映了开发人员和安全分析师社区提供的高质量标签。我们首先确定了 29 个安全问题网站，然后将范围缩小到 Git 系统提交数量最多的 2 个网站 [1]。我们从这些网站抓取问题标题、正文以及相关的 Git 提交URL。由于开发人员的讨论可能会提及漏洞修复提交和引入漏洞的提交，我们采用两种启发式方法排除引入漏洞的提交。首先，排除评论中包含“introduced”和“first included”关键词的所有提交URL；其次，人工检查所有修改了至少 10 个函数的提交，并排除引入漏洞的提交。我们将剩余的提交保留在数据集中。

接下来，我们解析 Git 提交 URL，以提取项目和提交 ID。我们克隆这些项目，并从中提取提交。我们识别提交中的 C/C++ 相关代码文件。然后，我们提取这些提交中所有发生更改的函数，以及文件中未发生更改的函数。与 ReVeal [4], 一样，我们将更改函数的提交前版本标记为易受攻击，将提交后版本标记为非易受攻击。我们将相关代码文件中的所有未更改函数标记为非易受攻击。与以往工作一样，我们根据函数的 MD5哈希值对函数进行去重，并且在去重前不对代码进行规范化处理。处理函数时，我们会跟踪唯一 MD5值的集合。我们先处理所有易受攻击函数，再处理非易受攻击函数。如果某个函数的 MD5值已存在于该集合中，我们就不会再次将该函数纳入数据集。总计而言，我们从 797个项目中收集了 7,514次提交，得到 18,945个易受攻击函数和 330,492个非易受攻击函数，涵盖 150种CWE漏洞类型。表1展示了 DiverseVul 中数量最多的前 10 个项目和前 10 种CWE漏洞类型，其中包含数量最多的

[1]snyk.io 和 bugzilla.redhat.com。

项目	# 提交数
Linux	1,458
ImageMagick	330
php-src	301
OpenSSL	261
TensorFlow	243
qemu	205
Linux 2.6	179
vim	134
FFMPeg	134
tcpdump	112

CWE	# 提交
CWE-787	2,896
CWE-125	1,869
CWE-119	1,633
CWE-20	1,315
CWE-703	1,228
CWE-416	1,005
CWE-476	975
CWE-190	783
CWE-200	747
CWE-399	509

(b) 表1:(a)DiverseVul 中排名前 10 的项目和常见弱点枚举，以及对应的漏洞修复提交数量。

漏洞修复提交。请注意，CWE-703“异常条件检查或处理不当”不在 MITRE 最常见的 25 种CWE漏洞类型之列。

对于标题中提及通用漏洞披露编号的问题，我们会查询国家漏洞数据库的应用程序接口，获取该问题及其对应提交的 CWE 信息。对于开发者标注了漏洞类别的问题，我们会手动将其映射到最常见的前 25 种常见弱点枚举。我们的数据中约有 85% 可以映射到 150 个 CWE 类别。我们不会专门处理分层的 CWE。根据国家漏洞数据库的查询结果，一个通用漏洞披露编号可能映射到多个 CWE。

4 实验

在本节中，我们研究新的数据集如何提升基于深度学习的漏洞检测性能。我们研究了来自 4 个模型家族的 11 种模型架构。此外，我们还讨论了从这些实验中获得的见解。

4.1 模型架构

我们研究了 4 个模型家族，其中 3 个是基于 Transformer 的大语言模型（LLM）。在每个 LLM 家族中，都有使用不同目标函数进行预训练的模型变体。表 2 总结了所有模型架构的参数数量。

4.1.1 图神经网络。在图神经网络（GNN）家族中，我们选择复现一项具有代表性的先前工作 ReVeal [4]。

给定一个函数，ReVeal 模型会构建一个图来表示该函数，计算该图的嵌入向量，并将该向量分类为易受攻击或不易受攻击。具体而言，该函数的图表示为代码属性图 [31] (CPG)。CPG 结合了抽象语法树 (AST)、控制流图 (CFG)、数据流图 (DFG) 和程序依赖图 (PDG)。每个节点都包含相应的源代码和类型，每条边也都有类型。图的嵌入是图中各节点嵌入的总和。为了学习节点嵌入，ReVeal 使用门控图神经网络 (GGNN) [13]

模型家族	模型架构	# 参数
GNN	ReVeal	1.28M
RoBERTa	RoBERTa	125M
	CodeBERT	125M
	GraphCodeBERT	125M
GPT-2	GPT-2 Base	117M
	CodeGPT	124M
	PolyCoder	160M
T5	T5 Base	220M
	CodeT5 Small	60M
	CodeT5 Base	220M
	NatGen	220M

表 2：不同模型的参数数量。

以递归方式更新节点的嵌入。节点的初始嵌入由代码的 Word2Vec 嵌入与类别类型向量拼接而成。随后，GGNN 训练过程使用消息传递机制，根据图中节点的邻居节点更新每个节点的嵌入。最后，在训练 GGNN 后，ReVeal 添加两个全连接层，并重新平衡训练集，以学习最终的分类器。ReVeal 模型的参数总数为 1.28M。

4.1.2 RoBERTa 系列。 我们从 RoBERTa 系列中选取了三种模型架构：RoBERTa [16], CodeBERT [10], 和 GraphCodeBERT [11]。它们都具有 12 层 Transformer 编码器、768 维隐藏状态、12 个注意力头，总共有 1.25 亿个模型参数。该系列模型采用的通用预训练目标是掩码语言建模（MLM）。MLM 预训练过程会随机掩盖输入词元中的一定比例，从而将其移除；训练目标是预测缺失的词元。

RoBERTa [16] 是对 BERT [8] 的扩展，对重要超参数进行了调整，包括移除预测下一句的预训练目标，以及在训练期间使用更大的小批量和更高的学习率。RoBERTa 在五个数据集的联合数据上进行了预训练：BookCorpus、英文维基百科、CC-News、OpenWebText 和 Stories。

CodeBERT [10] 使用包含来自六种编程语言（Go、Java、JavaScript、PHP、Python 和 Ruby）的 230 万个函数的 CodeSearchNet [12] 数据集对模型进行预训练。CodeBERT 执行掩码语言建模（MLM）预训练和替换词元检测预训练。在预训练期间，每个输入都是自然语言描述和源代码组成的配对文本，其中自然语言文本描述代码的含义。CodeBERT 的 MLM 预训练会同时遮蔽自然语言部分和源代码部分中的词元，而替换词元检测也会对输入的这两个部分进行扰动。CodeBERT 在自然语言代码搜索和代码文档生成这两项下游任务上的表现优于 RoBERTa。

GraphCodeBERT [11]也使用 CodeSearchNet [12]训练数据集。除了输入中的自然语言描述和源代码部分外，GraphCodeBERT 的预训练还会构建输入的第三部分，用于捕捉源代码中变量之间的数据流。除了掩码语言建模（MLM）预训练之外，GraphCodeBERT 还提出了两种新的预训练目标：边预测和节点对齐。如果两个节点之间存在边，边预测任务就会最大化它们嵌入的点积；如果变量代表代码词元，节点对齐任务就会最大化代码词元与变量词元嵌入的点积。在基准数据集上，GraphCodeBERT 在代码克隆检测、代码翻译和代码优化任务上的表现优于 CodeBERT 和 RoBERTa。

请注意，CodeBERT 和 GraphCodeBERT 的训练数据集中没有用 C/C++ 编写的程序。

4.1.3 GPT-2 系列。 我们从 GPT-2 系列中选择了三种模型架构：GPT-2 Base [23], 、CodeGPT [17], 和 PolyCoder [30]。它们均包含 12 层 Transformer 解码器、768 维隐藏嵌入和 12 个注意力头。模型规模见表 2，范围为 1.17 亿至 1.6 亿。该系列模型共同采用的预训练目标是因果语言建模，即下一个词元预测。模型在因果语言建模任务上的预训练效果通过困惑度衡量。困惑度越低，表示模型效果越好。

GPT-2 [23] 是在一个未公开的 WebText 数据集上进行预训练的，该数据集通过抓取 Reddit 上的网页链接收集而成。

CodeGPT [17] 使用了与 GPT-2 相同的训练目标和架构，但训练数据不同。作者从 CodeSearchNet [12] 中选取 Python 和 Java 代码作为训练集，并发布了多个预训练 CodeGPT 模型变体。本文使用的是一个基于 Java 代码进行预训练的 CodeGPT 改编版本。CodeGPT 模型以 GPT-2 权重初始化，随后使用来自 CodeSearchNet 的 Java 代码，通过下一个词元预测任务进行预训练。请注意，训练集中没有 C/C++ 程序。

PolyCoder [30] 使用与 GPT-2 相同的模型架构和预训练目标，但从头开始预训练模型。作者使用来自 GitHub 的数据对模型进行预训练，这些数据包含源代码以及代码文件中的自然语言注释。他们克隆了共 147,095 个项目，这些项目是 12 种热门编程语言中最受欢迎、且至少有 50 个星标的代码仓库。其训练数据包含超过 24K 个 C/C++ 代码仓库。作者整理了一个由未见过的代码仓库中的代码构成的评估数据集。在 C 编程语言方面，与 GPT-Neo、GPT-J 和 Codex 相比，PolyCoder 的困惑度最低。

4.1.4 T5 系列。 我们从 T5 系列中选取了四种模型架构：T5 Base [24], CodeT5 Base、CodeT5 Small [29], 和 Nat-Gen [3]。所有模型都采用编码器-解码器 Transformer 层。CodeT5 Small 有 6 个编码器层和 6 个解码器层，隐藏状态维度为 512，包含 8 个注意力头和 6,000 万个参数。其他模型有 12 个编码器层和 12 个解码器层，隐藏状态维度为 768，包含 12 个注意力头和 2.2 亿个参数。

T5 [24] 使用掩码语言建模目标来预训练模型。具体而言，T5 的预训练过程会随机遮盖若干词元跨度。预训练数据集为 C4（Colossal Clean Crawled Corpus）。作者通过处理

Common Crawl 数据集，获得数百 GB 的干净英文文本。

CodeT5 [29] 采用与 T5 相同的底层 Transformer 架构。我们在实验中考虑了两种模型规模：CodeT5 Base 和 CodeT5 Small。CodeT5 Small 是规模最小的大语言模型，其模型规模仅为其他基于 T5 的模型的三分之一，约为 RoBERTa 和 GPT-2 系列模型的一半。CodeT5 使用 CodeSearchNet 数据以及来自 GitHub 的其他 C/C# 项目进行预训练。除了掩码跨度预测目标外，CodeT5 还利用词元是否为标识符（变量名或函数名）的信息，并设计了两项新的预训练任务：掩码标识符预测（屏蔽所有标识符）和标识符标注（预测词元是否为标识符）。

NatGen [3] 提出了一种名为"自然化"预训练的新预训练目标。自然化预训练类似于代码编辑过程：将一些奇异的合成代码转换为开发者可读代码。作者通过保留语义的代码变换生成非自然代码，包括添加死代码、将 while 循环改为未初始化变量的 for 循环、重命名变量，以及插入令人困惑的代码元素等。随后，预训练目标要求模型将代码自然化为原始的、对开发者友好的形式。NatGen 模型以 CodeT5 Base 权重为起点进行预训练，然后使用新的预训练目标继续预训练。要在自然化预训练目标上取得良好表现，模型必须充分理解代码。与 CodeT5 相比，NatGen 在代码翻译、文本到代码生成和漏洞修复等多种下游任务中的性能都有提升。

4.2 更多数据下的模型性能

4.2.1 数据集设置。深度学习模型在使用大量数据训练时表现良好。因此，我们将来自真实项目、带有高质量漏洞标签的非合成数据集结合起来，包括 Devign、ReVeal、BigVul、CrossVul 和 CVEFixes。然后，我们将它们与 DiverseVul 数据集合并，并删除重复样本，以创建 Previous +DiverseVul 数据集，如表3所示。

表 3 展示了之前五个数据集、我们的数据集 DiverseVul 以及合并后数据集的统计信息。与所有先前的数据集相比，DiverseVul 包含更多项目、更多常见弱点枚举、更多易受攻击函数以及更多漏洞修复提交。具体而言，DiverseVul 包含 18,945个易受攻击函数，其中 16,109 个包含 CWE 信息，数量是任何先前数据集的两倍以上。拥有更多带有 CWE 信息的数据，将使我们能够更全面地了解模型预测结果。表 3 的最后两行展示了去重样本后，DiverseVul 为合并数据集带来的独有新数据。比较 Previous 和 Previous + DiverseVul 数据集可以发现，DiverseVul 包含 295个新项目，这些项目在任何先前的数据集中都不存在。此外，DiverseVul 还提供了 10,845 个独有的新易受攻击函数。

在我们的实验中，我们从 Previous + DiverseVul 数据集中随机抽取 80% 的样本作为训练集，10% 作为验证集，另外 10% 作为测试集。我们还构建了仅包含此前五个数据集的 Previous 训练集和验证集，以及仅包含 CVEFixes 数据的训练集和验证集。这样，我们就可以使用不同数量的数据训练模型，并评估增加数据量对提升模型性能、预测同一 Previous + DiverseVul 测试集有多大帮助。

4.2.2 结果。对于表 2 中的每种模型架构，我们分别使用 CVEFixes、Previous 和 Previous + Diverse- Vul 训练数据集训练三个模型。ReVeal 模型从头开始训练，而大语言模型（LLM）则从预训练模型权重出发，针对漏洞检测任务进行微调。总计训练了 33 个模型。实验中的详细训练设置见附录 A。

表 4 展示了模型在来自 Previous + DiverseVul 的同一测试集上的性能。以下总结了相关结果。

结果 1：在使用所有可用数据进行训练时，大语言模型的性能显著优于当前最先进的基于图神经网络的 ReVeal 模型。 在使用所有可用数据（Previous + DiverseVul）进行训练时，大语言模型的表现显著优于 ReVeal 模型：ReVeal 模型取得了 29.76 F1 分数，而大语言模型取得了 31.96 至 47.15 的 F1 分数。表现最佳的大语言模型在这个大规模训练集上的性能显著优于 ReVeal。比较 ReVeal 与大语言模型可能并不公平，因为 ReVeal 的参数量比大语言模型少 1 至 2 个数量级。我们不知道更大的图神经网络是否能够与大语言模型竞争。遗憾的是，即使是表现最佳的模型 NatGen，目前也不适合部署于漏洞检测，其误报率为 3.47%，F1 分数为 47.15%。这一误报率仍然过高，无法满足实际应用需求，而 F1 分数也仍然偏低。尽管如此，我们相信大语言模型在基于深度学习的漏洞检测方面具有发展潜力。

有趣的是，大语言模型需要大量训练数据才能超越 ReVeal。仅使用规模小得多的 CVEFixes 数据进行训练时，大语言模型并未明显优于基于图神经网络的 ReVeal 模型；在这种设置下，ReVeal 甚至优于 10 个大语言模型中的 6 个。

结果 2：在三个基础大语言模型中，T5 Base 在漏洞检测方面的表现优于 RoBERTa 和 GPT-2 Base。 RoBERTa 仅使用编码器，GPT-2 仅使用解码器，而 T5 使用编码器-解码器 Transformer 层。在使用 Previous + DiverseVul 进行训练时，T5 Base 的测试F1分数分别比 RoBERTa 和 GPT-2 Base 高 7.35% 和 9.3%。因此，编码器-解码器架构可能优于仅使用解码器或编码器的架构。

结果 3：如果我们只使用自然语言预训练任务，那么在代码上进行预训练并不会显著提升漏洞预测性能。 代码模型 CodeBERT、GraphCodeBERT、CodeGPT 和 PolyCoder 并不显著优于对应的文本模型 RoBERTa 和 GPT-2 Base。具体而言，在 Previous 数据集上训练时，CodeBERT 和 GraphCodeBERT 的表现与 RoBERTa 相近。在 Previous + DiverseVul 数据集上训练时，CodeBERT 和 GraphCodeBERT 的 F1 分数比 RoBERTa 提高了至多 2.8%。另一方面，在 Previous 数据集上训练时，CodeGPT 和

数据集	# 项目	# 常见弱点枚举	# 函数	# 漏洞函数	# 包含CWE信息的漏洞函数	# 提交
Devign	2▽	N/A	26,037	11,888	N/A	N/A
ReVeal	2◇	N/A	18,169	1,664	N/A	N/A
BigVul	348	91	264,919	11,823	8,783	3,754
CrossVul*	498	107	134,126	6,884	6,833	3,009
CVEFixes*	564	127	168,089	8,932	8,343	3,614
DiverseVul	**797**	**150**	**330,492**	**18,945**	**16,109**	**7,514**
Previous†	638	140	343,400	30,532	14,159	17,956
Previous +DiverseVul	**933**	**155**	**523,956**	**41,377**	**22,382**	**21,949**

†：我们通过合并并去重 Devign、ReVeal、BigVul、CrossVul 和 CVEFixes 中的样本，汇总了前五个数据集。*：CVEFixes 和 CrossVul 是多语言数据集。我们报告 C/C++ 代码的数据。▽：Devign 的作者发布了来自两个代码仓库的数据：FFMPeg+Qemu。◇：Chromium 和 Debian 软件包。**表3：前五个数据集、DiverseVul、合并后的 Previous 数据集以及 Previous + DiverseVul的统计信息。**

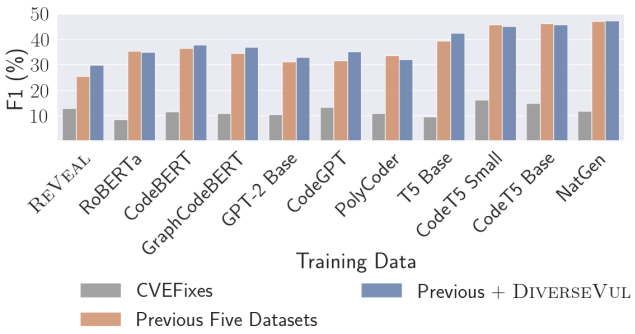


图 2：我们展示了在 CVEFixes、Previous 和 Previous + DiverseVul 上训练的模型的性能。将 DiverseVul 添加到合并后的 Previous 数据集中，有助于提升 11 个模型中 7 个模型的测试性能，但对 CodeT5 模型没有帮助。

PolyCoder 的 F1 分数比 GPT-2 高出至多 2.3%；但在 Previous + DiverseVul 数据集上训练时，PolyCoder 的表现不如 GPT-2。我们的研究结果表明，使用掩码语言建模或下一个词元预测技术在代码上预训练模型，并不能显著提升 C/C++ 漏洞检测效果。CodeBERT、GraphCodeBERT 和 CodeGPT 均未在 C/C++上进行预训练，而 PolyCoder 则在 C/C++ 代码上进行了下一个词元预测预训练，但这仍无助于检测 C/C++ 漏洞。

结果 4：面向 C/C 的代码专用预训练任务++ 对提升漏洞检测性能有显著作用。两种 CodeT5 模型和 NatGen 模型的 F1 分数最高。它们使用面向 C/C++ 的代码专用预训练任务进行预训练。CodeT5 模型采用标识符感知预训练任务：掩码标识符预测和标识符标注。NatGen 在 CodeT5 的基础上进一步进行代码自然化预训练，例如删除死代码和重命名变量。这些预训练任务旨在让模型学习代码理解的基础知识，这显著提升了微调模型在漏洞检测任务上的性能。需要注意的是，GraphCodeBERT 也进行了一些代码特定预训练，从具有数据流的变量对中学习嵌入，使其点积值较大。然而，由于它未在 C/C++ 数据上训练，因此无法确定此类预训练任务是否有助于漏洞预测。

结果 5：代码特定预训练任务比模型规模更重要。在表 4 中表现最佳的三个模型（CodeT5 Small、CodeT5 Base、NatGen）中，CodeT5 Small 模型仅有 60M 个参数，规模只有 RoBERTa 和 GPT-2 模型的一半，不到其他 T5 模型的三分之一。然而，CodeT5 Small 的表现与最大的 CodeT5 Base 和 NatGen 模型非常相近，并且优于所有其他模型。与“大模型往往能带来更好性能”这一观点相反，我们的结果表明，对于漏洞检测而言，代码特定预训练任务比模型规模更重要。

结果 6：通过收集更多数据集获得的性能提升可能已趋于饱和。图 2 展示了与 Previous 数据集相比，在 DiverseVul + Previous 数据集上训练能在多大程度上提升漏洞检测性能。与仅使用 Previous 数据集训练相比，将 DiverseVul 加入训练集后，7 个模型的 F1 分数平均提升了 2.4%。不过，这并未提升表现最佳的 CodeT5 模型，对 NatGen 的提升也有限。尽管与仅使用 CVEFixes 训练相比，使用合并后的 Previous 数据集训练显著提升了模型性能，但收集其他数据集可能无法进一步提升性能。

4.3 数据集规模

4.3.1 数据集设置。 我们希望测量数据量对漏洞检测模型性能的影响。我们将以下实验重复进行十次。每次实验中，我们随机将 Previous + DiverseVul 划分为训练集、验证集和测试集。然后，我们通过对训练集和验证集进行子采样，模拟不同数据量的影响。具体而言，我们从 Previous + DiverseVul 的完整训练数据和验证数据中，随机抽取 10% 至 90%。然后，我们训练

模型家族	模型 Arch	预训练在代码上	预训练在 C/C++ 上	代码专用预训练任务	训练集	在先前的 +DiverseVul (%) 上测试 Acc	Prec	召回率	F1	FPR
GNN	ReVeal				CVEFixes	82.12	11.56	14.37	12.81	11.06
					Previous	86.30	25.35	25.63	25.49	7.59
					上一页 + DiverseVul	82.81	23.75	39.83	29.76	12.87
RoBERTa	RoBERTa				CVEFixes	91.71	34.24	4.85	8.50	0.80
					Previous	90.98	40.97	31.11	35.37	3.86
					Prev + DiverseVul	91.68	46.02	28.22	34.98	2.85
	CodeBERT	✔			CVEFixes	91.62	35.64	6.98	11.67	1.09
					Previous	91.07	41.83	32.20	36.39	3.86
					Prev + DiverseVul	90.48	39.25	36.54	37.85	4.87
	GraphCodeBERT	✔		✔	CVEFixes	91.76	38.28	6.35	10.89	0.88
					Previous	91.65	45.71	27.61	34.43	2.83
					Previous + DiverseVul	90.32	38.18	35.51	36.79	4.96
GPT-2	GPT-2 Base				CVEFixes	91.45	31.02	6.37	10.57	1.22
					Previous	91.80	46.62	23.46	31.21	2.32
					Previous + DiverseVul	91.73	46.18	25.71	33.03	2.58
	CodeGPT	✔			CVEFixes	90.77	26.22	8.98	13.38	2.18
					Previous	91.59	44.51	24.48	31.58	2.63
					Prev + DiverseVul	91.36	43.48	29.62	35.23	3.32
	PolyCoder	✔	✔		CVEFixes	91.12	26.56	6.78	10.81	1.62
					Previous	91.28	42.44	27.66	33.49	3.23
					Prev + DiverseVul	91.97	48.76	23.78	31.96	2.15
T5	T5 Base				CVEFixes	91.57	32.23	5.65	9.61	1.02
					Previous	92.15	50.80	32.15	39.38	2.68
					Prev + DiverseVul	91.96	49.14	37.17	42.33	3.32
	CodeT5 Small	✔	✔	✔	CVEFixes	90.89	30.03	11.18	16.29	2.24
					Previous	91.98	49.34	42.53	45.68	3.76
					Prev + DiverseVul	91.85	48.41	42.22	45.10	3.88
	CodeT5 Base	✔	✔	✔	CVEFixes	91.41	34.76	9.39	14.79	1.52
					Previous	92.16	50.68	42.46	46.20	3.56
					Prev + DiverseVul	92.11	50.36	41.81	45.69	3.55
	NatGen	✔	✔	✔	CVEFixes	91.64	36.17	7.07	11.83	1.08
					Previous	92.30	51.81	42.92	46.94	3.44
					Prev + DiverseVul	**92.30**	**51.81**	**43.25**	**47.15**	**3.47**

表 4：我们在与 Previous + DiverseVul 相同的测试集上评估这些模型。如果仅使用 CVEFixes 数据集进行训练，不同架构的模型性能差异不大。然而，如果使用更大的数据集进行训练，大语言模型的表现显著优于基于 GNN 的 ReVeal 模型。其中，CodeT5 Small、CodeT5 Base 和 NatGen模型的 F1 分数最高。我们将 F1 分数最高的行加粗显示。使用 C/C++ 上的代码特定预训练任务对模型进行预训练非常有效。

这些模型，并在同一个未经子采样的原始测试集上对其进行评估。

4.3.2 结果。我们针对 10 次实验运行中的不同数据集设置，对 100 个 CodeT5 Small 模型进行微调。在每次运行中，我们都使用相同的最终测试集评估这些模型，该测试集来自 Previous +DiverseVul；此外，我们通过使用不同比例的训练数据和验证数据来训练 10 个模型。图3绘制了在相应数据集设置上微调模型时，测试集 F1 分数的平均值和 95% 置信区间。

结果7：增加来自相同分布的训练数据集规模有助于漏洞检测。我们的结果表明，使用更大的来自相同分布的数据集进行训练可以提升测试性能。图3显示，随着训练数据量增加，测试F1 分数呈上升趋势。如果我们能在模型部署之前了解测试数据分布

模型家族	模型 Arch	预训练在代码上	预训练在 C/C++ 上	代码专用预训练任务	训练集	在未见项目上测试（%） Acc	Prec	召回率	F1	FPR
GNN	ReVeal				Previous	82.88	5.06	20.92	8.15	14.78
					Prev + DiverseVul	85.88	5.67	18.46	8.67	11.58
RoBERTa	RoBERTa				Previous	94.69	6.20	3.23	4.25	1.85
					Prev + DiverseVul	95.59	10.46	2.78	4.40	0.90
	CodeBERT	✔			Previous	94.94	9.53	4.57	6.17	1.64
					Prev + DiverseVul	**94.19**	**13.34**	**10.80**	**11.94**	**2.65**
	GraphCodeBERT	✔		✔	Previous	95.32	4.64	1.45	2.21	1.12
					Prev + DiverseVul	94.74	12.48	7.35	9.25	1.95
GPT-2	GPT-2 Base				Previous	94.92	6.19	2.78	3.84	1.60
					上一页 + DiverseVul	95.06	9.82	4.34	6.02	1.51
	CodeGPT	✔			上一页	94.32	5.98	3.79	4.64	2.25
					上一页 + DiverseVul	94.47	9.86	6.35	7.72	2.19
	PolyCoder	✔	✔		上一页	95.41	8.54	2.67	4.07	1.08
					上一页 + DiverseVul	92.73	10.25	12.81	11.39	4.24
T5	T5 Base				上一页	95.67	20.21	6.35	9.66	0.95
					上一页 + DiverseVul	96.16	34.00	5.68	9.73	0.42
	CodeT5 Small	✔	✔	✔	上一页	95.02	12.21	5.90	7.96	1.60
					上一个 + DiverseVul	94.91	13.35	7.24	9.39	1.78
	CodeT5 Base	✔	✔	✔	Previous	96.21	32.32	3.56	6.42	0.28
					上一个 + DiverseVul	95.56	18.03	6.12	9.14	1.05
	NatGen	✔	✔	✔	Previous	95.48	17.86	6.68	9.72	1.16
					Prev + DiverseVul	95.49	17.38	6.35	9.30	1.14

表 5：我们随机选取 95 个项目作为未见项目进行测试。其余项目用于训练。我们在已见项目上训练每个模型，并在未见项目上测试这些模型。我们用粗体标出 F1 分数最高的行。总体而言，F1 分数表明，这些模型在未见项目上的泛化性能较差。将 DiverseVul 添加到 Previous 训练集中有助于提升所有模型的泛化性能，NatGen 除外。

时间，从该分布中收集更多训练数据，或许能进一步提升漏洞检测性能。

4.4 泛化

4.4.1 数据集设置。 在现实世界的部署场景中，漏洞检测模型需要预测其未曾训练过的新开发者项目中的易受攻击的源代码。因此，我们希望测试模型在未见项目上的性能。

我们从合并后的 Previous 数据集中随机选取 95 个不同的项目作为未见项目测试集，用于评估本实验中的所有模型。然后，其余项目在训练集和验证集中均视为已见项目。对于 Previous 和 Previous + DiverseVul 数据集，我们随机抽取 90% 的已见项目作为训练集，其余 10% 作为验证集。Previous +DiverseVul 的训练集和验证集分别是 Previous 中对应集合的超集。

4.4.2 结果。我们在 Previous 和 Previous + DiverseVul 的已见项目训练集上训练 ReVeal，并对每个大语言模型进行微调，共得到 22 个模型。我们确保这些模型都经过充分训练，因为它们的验证性能与训练性能相近。表 5 展示了这些模型在未见项目上的测试性能。

所有模型在未见项目上的 F1 分数都很低。表现最好的模型是 CodeBERT、PolyCoder、CodeT5 Small、在 Previous + DiverseVul 上训练的 CodeT5 Base 模型，以及在 Previous 已见项目上训练的 NatGen 模型。除 NatGen 外，将 DiverseVul 加入 Previous 训练集有助于提升所有模型的泛化性能。一项近期的并行研究 [26] 也发现，在未见项目上测试时，性能会显著下降。在我们的实验中，训练集包含的项目数量比 [26], 多出数百个，但我们仍然观察到泛化效果不佳。

结果 8：对于深度学习模型而言，在漏洞检测任务中泛化到未知测试项目是一项重大挑战。面向代码的人工智能的一个热门应用是 GitHub CoPilot：开发人员编写代码时，人工智能模型会向他们建议如何补全代码。如果基于深度学习的检测也要充当编程助手，就需要建议潜在的

模型架构	方案	在已见项目上训练 在未见项目上测试（%）					在随机样本上训练 在随机样本上测试（%）				
		Acc	Prec	召回率	F1	FPR	Acc	Prec	召回率	F1	FPR
CodeBERT	无权重	94.19	13.34	10.8	11.94	2.65	90.48	39.25	36.54	37.85	4.87
	项目均衡	95.09	11.6	5.23	7.21	1.51	90.7	34.43	18.97	24.46	3.11
	加权软 F1 损失	91.38	11.41	20.16	14.57	5.92	90.72	34.55	18.9	24.43	3.08
	类别权重	92.16	12.21	18.6	14.74	5.06	89.39	36.97	47.89	41.72	7.04
PolyCoder	无权重	92.73	10.25	12.81	11.39	4.24	91.97	48.76	23.78	31.96	2.15
	项目均衡	94.37	8.33	5.46	6.59	2.27	90.77	30.08	12.33	17.49	2.47
	加权软 F1 损失	93.17	11.24	12.69	11.92	3.79	89.88	36.07	35.72	35.9	5.46
	类别权重	89.76	9.84	22.16	13.63	7.68	86.48	29.19	49.36	36.68	10.32
CodeT5 Small	不加权	94.91	13.35	7.24	9.39	1.78	91.85	48.41	42.22	45.10	3.88
	项目均衡批次采样器	95.3	14.52	5.90	8.39	1.31	90.69	39.36	31.96	35.27	4.24
	加权软 F1 损失	96.34	48.18	5.90	10.52	0.24	91.31	44.69	39.78	42.09	4.24
	交叉熵损失的类别权重	**93.87**	**16.95**	**17.48**	**17.21**	**3.24**	**89.57**	**39.80**	**61.33**	**48.28**	**7.99**

表6：使用类别权重进行交叉熵损失计算，可提升模型的泛化性能：在已见项目上训练，并在未见项目上测试时，效果尤为明显。使用类别权重后，CodeBERT 的未见项目测试 F1 分数从 11.94% 提升至 14.74%，PolyCoder 从 11.39% 提升至 13.63%，CodeT5 Small 从 9.39% 提升至 17.21%。此外，如果训练样本和测试样本取自相同分布，使用类别权重也能提高测试 F1 分数。我们用粗体标出 F1 分数最高的行。

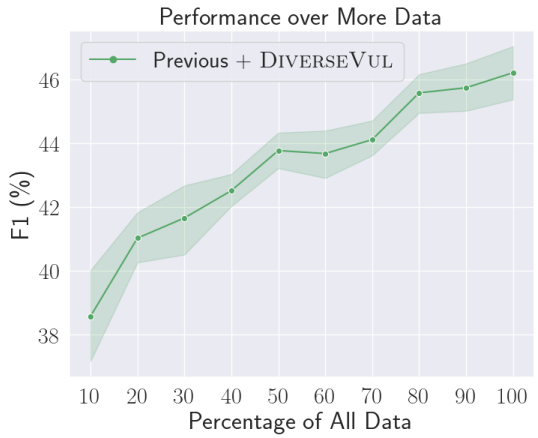


图3：对于易受攻击源代码检测，使用与测试数据来自相同分布的更多数据进行深度学习训练会带来益处。我们使用不同数量的易受攻击源代码数据对 CodeT5 Small 模型进行微调，并报告测试 F1 分数。每种数据集设置均运行 10 次。曲线表示平均值，阴影区域表示 95%置信区间。此图表明，扩大训练集可以提高来自相同分布的测试数据上的漏洞检测 F1 分数。

开发人员正在编写的易受攻击函数，即使这些函数属于模型未曾训练过的新项目。另一种做法是使用静态分析器检查不同项目中的漏洞。在类似的应用场景中，基于深度学习的检测模型需要分析一个此前从未见过的新项目（开发完成后）。这两种应用场景要求深度学习模型对新项目具有较强的泛化性能，而这仍是社区需要解决的一个开放研究问题。

4.5 加权

在本节中，我们探讨三种简单的加权方案是否有望提升模型对未见过的测试项目的泛化性能。这些加权方案如下。

4.5.1 项目均衡批次采样器。我们的想法是让模型在不同项目上的表现同样出色。因此，我们提出一种批次采样器，使训练集中每个项目被采样的概率相同。选中一个项目后，再从属于该项目的所有函数中随机采样。

4.5.2 加权软 *F1* 损失。由于我们关注的是 F1 分数这一最终性能指标，因此希望探究采用不同的损失函数是否有助于提升泛化性能。我们使用训练样本的归一化预测概率（介于 0 和 1 之间），以浮点数的形式计算真正例、真负例、假正例和假负例。然后，分别计算预测为正标签（易受攻击函数）和负标签（非易受攻击函数）的两个 F1 分数。正标签的损失为 1 - 正标签 F1 分数，负标签的损失为 1 - 负标签 F1 分数。最后，我们根据数据中非易受攻击函数与易受攻击函数的比例，为前一个损失值赋予更高的权重。然后，根据真实类别标签选择相应的损失值，作为最终的训练损失。

4.5.3 交叉熵损失的类别权重。在此方案中，我们仍使用交叉熵损失进行训练。我们按非易受攻击样本与易受攻击样本的比例，提高

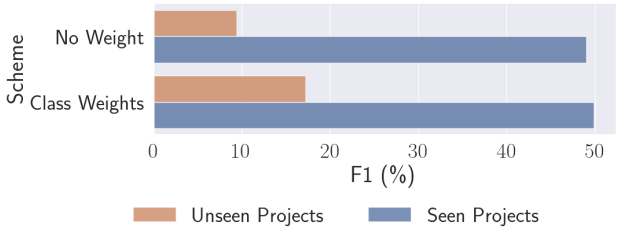


图4：在训练损失函数中使用类别权重，可提高 CodeT5 Small 在未见项目上的泛化性能，同时也略微提升其在已见项目上的性能。不过，在未见项目上的测试F1分数仍然相当低。

正类（易受攻击类）的损失值。负类的损失值则保持不变。

4.5.4 结果。我们采用与第4.4节所述相同的项目划分数据集设置。我们使用来自前述 + DiverseVul数据集的已见项目训练集，对 CodeBERT、PolyCoder 和 CodeT5 Small 模型进行微调，并在95个未见项目上测试。对于每种模型架构，我们采用四种方案分别微调四个模型：不加权、项目均衡批次采样器、加权软 F1 损失，以及交叉熵损失的类别权重。此外，我们还针对每种架构，使用这些方案在另一种数据划分上再微调四个模型；该划分为第4.2节所述的随机数据划分。

结果 9：使用交叉熵损失的类别权重可以提升模型对未见项目的泛化性能，但仍有很大的改进空间。如果训练样本和测试样本来自同一分布，类别权重也能提升模型性能。表6展示了采用不同方案进行微调的模型评估结果。对于已见项目/未见项目实验，使用类别权重可以提升三种模型架构的 F1 分数。项目均衡批次采样器无助于泛化。加权软 F1 损失有助于 CodeBERT 和 CodeT5 Small 的泛化，但会降低它们在已见项目上的性能。总体而言，类别权重是最佳方案，因为它能提升模型在已见项目和未见项目上的性能。使用类别权重训练的 CodeT5 Small 在未见项目上的测试 F1 分数最高，为 17.21%。

图4展示了两种 CodeT5 Small 模型在已见项目与未见项目上 F1 分数的差距：一种模型微调时未采用加权方案，另一种则使用类别权重进行交叉熵损失微调。从柱状图可以看出，使用类别权重缩小了模型在已见项目与未见项目上的 F1 分数差距：已见项目上的 F1 分数略有提高，未见项目上的 F1 分数则显著提高。这表明，使用类别权重能够提升模型在与训练数据分布相同的样本以及来自不同分布的新项目样本上的性能。不过，模型在已见项目上的 F1 分数仍为 49.9%，在未见项目上则为 17.21%，两者仍存在较大差距。作为泛化问题未来的研究方向，进一步提升模型在未知项目上的性能仍有很大潜力。

4.6 常见弱点枚举上的性能

为了解不同常见弱点枚举的学习难度，我们选取了 37 种常见弱点枚举，考察 CodeT5 Base 模型在 Previous +DiverseVul 上训练后的预测性能。这 37 种常见弱点枚举包括 MITRE [6], 列出的前 25 种，以及 DiverseVul 中排名不在前 25、但最常见的 12 种。我们从通过第4.2节随机划分得到的 Previous + DiverseVul 测试集中，选取属于这 37 种常见弱点枚举的易受攻击函数以及所有非易受攻击函数。

结果 10：无论训练数据量多少，有些常见弱点枚举都比其他弱点更容易学习。表7展示了 CodeT5 Base 模型在 37 种常见弱点枚举上的预测性能。我们用粗体标出了训练集中最常见的 10 种常见弱点枚举，以及真正率（TPR）最高的 10 个数值。请注意，所有常见弱点枚举的假阳性率（FPR）都相同，因为 FPR 仅与非易受攻击函数有关。我们观察到，在训练集中，某种常见弱点枚举的样本更多，并不一定意味着模型能比对训练样本更少的常见弱点枚举学习得更好。此外，有些训练样本极少的常见弱点枚举也能被模型准确检测。例如，CWE-502、CWE-79 和 CWE-89 在训练数据中的占比均低于 2%，但其真正率最高。这表明，有些常见弱点枚举更容易学习，无需大量训练数据；另一些则更难学习，即使有更多训练样本也是如此。例如，CWE-416 占训练样本的 5.46%，但其真正率仅为 17.86%。

对于某些常见弱点枚举，我们没有足够的测试样本，因此真正例率极低。“测试集编号”列显示了测试集中属于该常见弱点枚举的易受攻击函数数量。对于真正例率为 0% 的常见弱点枚举，大多数在测试集中的样本数少于 10。

5 标签错误分析

尽管我们的数据集旨在尽可能准确，但某些函数仍可能被错误标注。为了标注易受攻击函数，我们遵循 Devign [33], ReVeal [4], BigVul [9], CrossVul [19], 和 CVEFixes [2], 所采用的方法：根据安全问题跟踪系统，如果某个函数被一项被认定为修复漏洞的提交所修改，则将其视为易受攻击函数。虽然我们的标注技术处于业界领先水平且可有效扩展，但我们无法保证此类提交所修改的每个函数都易受攻击，因此部分标签可能不准确。

为了量化这种标注方法造成的标签噪声，我们人工评估了 DiverseVul、CVEFixes、BigVul 和 CrossVul 数据集的标签准确率。在以往的数据集中，我们选择了 CVEFixes、BigVul 和 CrossVul，因为它们提供了修改易受攻击函数的提交 ID，这使我们能够核实某个函数在项目的特定版本中是否易受攻击。

我们从 Diverse-Vul 中随机抽取 50 个易受攻击函数，并从前三个数据集（CVEFixes ∪ BigVul ∪ CrossVul）的并集中随机抽取 50 个易受攻击函数。然后，我们手动分析这些易受攻击函数的标签是正确还是错误。我们通过检查被标记为易受攻击的函数在提交前后的代码、据称修复该函数的提交、CVE 描述以及开发者

CWE	训练集（%）	测试集编号	真正率（%）	假正率（%）	
CWE-119	**15.16**	313	**39.30**	3.55	内存缓冲区边界内操作限制不当
CWE-120	2.29	49	**40.82**	3.55	未检查输入大小的缓冲区复制（“经典缓冲区溢出”）
CWE-125	**11.08**	239	27.20	3.55	越界读取
CWE-189	2.97	57	**31.58**	3.55	数值错误
CWE-190	**4.77**	100	21.00	3.55	整数溢出或回绕
CWE-200	**5.10**	131	**31.30**	3.55	向未授权主体暴露敏感信息
CWE-20	**10.76**	224	**32.59**	3.55	输入验证不当
CWE-22	1.13	20	25.00	3.55	未能正确限制路径名位于受限目录内（“路径遍历”）
CWE-264	3.55	73	28.77	3.55	权限、特权和访问控制
CWE-269	1.14	23	8.70	3.55	特权管理不当
CWE-276	0.19	3	0	3.55	错误的默认权限
CWE-284	3.35	77	25.97	3.55	不当访问控制
CWE-287	0.58	10	10.00	3.55	不当身份验证
CWE-306	0.00	0	N/A	3.55	关键功能缺少身份验证
CWE-310	1.95	44	25.00	3.55	密码学问题
CWE-352	0.10	1	0	3.55	跨站请求伪造（CSRF）
CWE-362	2.62	61	16.39	3.55	竞态条件
CWE-369	1.26	31	29.03	3.55	除零
CWE-399	**5.29**	110	**41.82**	3.55	资源管理错误
CWE-400	2.38	34	5.88	3.55	失控的资源消耗
CWE-401	1.83	33	24.24	3.55	有效生命周期结束后未释放内存
CWE-415	1.55	30	30.00	3.55	二次释放
CWE-416	**5.46**	112	17.86	3.55	释放后使用
CWE-434	0.07	1	0	3.55	不受限制地上传危险类型的文件
CWE-476	**5.00**	106	17.92	3.55	空指针解引用
CWE-502	0.05	3	**66.67**	3.55	不可信数据反序列化
CWE-611	0.09	3	0	3.55	XML 外部实体引用限制不当
CWE-703	**6.39**	133	10.53	3.55	异常条件检查或处理不当
CWE-77	0.18	6	16.67	3.55	命令注入
CWE-78	0.38	7	0	3.55	操作系统命令注入
CWE-787	**15.57**	311	**33.76**	3.55	越界写入
CWE-79	0.47	12	**50.00**	3.55	跨站脚本攻击
CWE-798	0.01	0	N/A	3.55	使用硬编码凭据
CWE-862	0.26	6	16.67	3.55	缺少授权
CWE-89	0.31	9	**33.33**	3.55	SQL 注入
CWE-918	0.02	4	0	3.55	服务器端请求伪造（SSRF）
CWE-94	0.69	15	0	3.55	代码生成控制不当（‘代码注入’）

表7：我们评估了 CodeT5 Base 模型在 DiverseVul 中排名前 25 的常见弱点枚举和最常见的 12 种常见弱点枚举上的预测性能。我们用粗体突出显示训练样本百分比最高的 10 项和真正例率最高的 10 项。针对特定常见弱点枚举增加训练样本，并不一定能提升模型的预测性能；有些常见弱点枚举也比其他的更难学习。真正例率为 0% 的常见弱点枚举中，大多数在测试集中的样本数不足 10 个。

数据集	正确标签	错误标签		
		漏洞传播跨多个函数	相关一致性	不相关
DiverseVul	60%	10%	12%	18%
CVEFixes ∪ BigVul ∪ CrossVul	36%	12%	12%	40%
CVEFixes	51.7%	10.3%	17.3%	20.7%
BigVul	25%	15.6%	9.4%	50%
CrossVul	47.8%	13%	21.8%	17.4%

表8：在随机抽取的易受攻击函数样本上评估的四个数据集的标签准确率。

安全问题跟踪器中的讨论。如果漏洞确实存在于某个函数中，且未跨多个函数扩散，我们就确认该函数被正确标记为易受攻击。我们观察到三类标签错误：1）漏洞跨多个函数扩散；2）该函数本身并不易受攻击，但修改该函数与修复漏洞有关（例如调整调用参数）；3）该函数本身并不易受攻击，且与该漏洞无关（例如，某个漏洞修复提交修改了某些非易受攻击函数中的空格，或对非易受攻击函数进行了无关的功能修改）。

表8展示了我们的分析结果。DiverseVul中易受攻击函数的标签准确率为60%，比之前的三个数据集（CVEFixes ∪ BigVul ∪ CrossVul）高出24个百分点。在这三个数据集中，CVEFixes的准确率最高，而BigVul的标签准确率很低，只有25%。我们发现，BigVul中许多来自Chromium和Android项目的提交与修复漏洞完全无关。我们还发现，在四个数据集中，不相关函数的比例高得惊人，介于17.4%至50%之间。这些函数与漏洞无关，但由于它们被漏洞修复提交修改，自动标注流程便将其标记为易受攻击。

同期工作也研究了标签噪声，并发现 BigVul 和 Devign 数据集中存在大量标签错误 [7]。与他们的分类标准相比，我们将函数标记为易受攻击函数的标准更为严格：我们将易受攻击函数的调用方视为非易受攻击；而他们将其视为易受攻击。此外，如果某个函数只是漏洞的一部分，且无法仅凭该函数的代码识别出漏洞，我们就将其视为错误标签；而他们认为该标签正确。考虑到分类标准的差异，我们对 BigVul（他们的工作与我们的工作中唯一共有的数据集）的研究结果总体上与他们的发现一致。

6 局限性

我们数据集及以往数据集中的标签噪声可能会导致对所有模型在测试集上性能的测量出现误差。我们希望发布数据集能够使社区探索未来减轻标签噪声影响的方法。

事后看来，我们数据集及以往数据集中的去重流程仍有改进空间。在标签噪声分析中，我们发现，DiverseVul 中有 4% 的标签以及（CVEFixes∪ BigVul ∪ CrossVul）中的 6% 标签存在错误，因为相关提交仅对某些函数进行了空白字符变更，而这些变更在标注时被视为安全修复。因此，在去重前规范化所有函数中的空白字符，可以略微提高标签准确率，也可能带来其他益处。

存在污染风险，即测试数据泄漏到预训练数据中，因为大语言模型是在文本和代码上进行预训练的，而这些内容理论上可能包含与我们测试集中安全漏洞相关的博客文章或代码补丁。我们的许多模型（CodeBERT、GraphCodeBERT、PolyCoder、CodeT5 Small、CodeT5 Base、NatGen）仅在代码上进行预训练，而没有在其他文本或代码变更上进行预训练，因此可能接触过测试集中的代码，但不太可能接触过关于哪些代码易受攻击的描述。这可能会以我们无法衡量的方式影响结果。其他模型（RoBERTa、GPT-2 Base、CodeGPT、T5 Base）是在文本上进行预训练的，因此可能接触过描述易受攻击源代码的博客文章。我们认为这种情况极为罕见，但无法衡量，因此不能排除测试集受到污染的可能性。无论如何，后面这些模型（RoBERTa、GPT-2 Base、CodeGPT、T5 Base）在我们的实验中表现相对较差。

此外，克隆代码也可能导致测试集污染：如果克隆代码随后经过轻微修改（从而避开我们的去重处理）。

7 结论

本文提出了一个用于利用深度学习检测软件漏洞的新数据集 DiverseVul。该数据集包含 18,945个易受攻击函数，涵盖 155 种常见弱点枚举，以及 330,492 个非易受攻击函数；这些函数从 7,514次提交中提取。其多样性更高，规模是此前规模最大、最多样化数据集 CVEFixes 的两倍。我们使用这一新数据集研究了各种深度学习架构在漏洞检测方面的有效性。我们测试了来自四个模型家族的 11 种不同深度学习架构：图神经网络（GNN）、RoBERTa、GPT-2 和 T5。结果表明，本文考察的训练数据在多样性和规模上的增加有助于漏洞检测，尤其有利于大语言模型；但更大规模的数据集是否会带来进一步提升，尚不明确。代码特定预训练任务似乎是深度学习漏洞检测中一个有前景的研究方向。我们的结果凸显了未来研究面临的一项重大挑战：改进深度学习模型，使其能够泛化到未知项目。我们已在 https://github.com/wagner-group/diversevul 向社区发布 DiverseVul数据集。

致谢

我们感谢 Bryce Casaje 在探索多种数据集构建方法方面所作的贡献，包括对提交进行人工标注、自动文本标注等。我们感谢 Kexin Pei 就大语言模型微调提供的建议。我们衷心感谢匿名评审提出的诸多宝贵意见，这些意见显著提升了论文质量。本研究得到美国国家科学基金会（NSF）项目 CNS-2154873、阿卜杜勒阿齐兹国王科技城（KACST）与加州大学伯克利分校联合设立的安全计算卓越中心、C3.AI 数字化转型研究所，以及人工智能安全中心计算集群的支持。本文所表达的任何观点、研究发现、结论或建议均属于作者本人，并不一定反映资助方的观点。

参考文献

[1] Al Bessey、Ken Block、Ben Chelf、Andy Chou、Seth Hallem、Bryan Fulton、Charles Henri-Gros、Asya Kamsky、Scott McPeak 和 Dawson Engler。2010年。数十亿行代码之后：使用静态分析在真实世界中查找错误。《美国计算机学会通讯》53，2（2010年2月）。[2] Guru Bhandari、Amara Naseer 和 Leon Moonen。2021年。CVEFixes：从开源软件中自动收集漏洞及其修复。在第17届软件工程预测模型与数据分析国际会议论文集。30–39。[3] Saikat Chakraborty、Toufique Ahmed、Yangruibo Ding、Premkumar T Devanbu 和 Baishakhi Ray。2022年。NatGen：通过“自然化”进行生成式预训练。

源代码。载于第30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。18–30。[4] Saikat Chakraborty、Rahul Krishna、Yangruibo Ding和Baishakhi Ray。2021年。基于深度学习的漏洞检测：我们已经做到这一步了吗？《IEEE软件工程汇刊》（2021年）。[5] Alexis Challande、Robin David和Guénaël Renault。2022年。构建真实世界漏洞的提交级数据集。载于第十二届ACM数据与应用安全和隐私会议论文集。101–106。[6] MITRE公司。最后访问于2023年3月28日。2022年CWE最危险软件弱点前25名。https://cwe.mitre.org/top25/archive/ 2022/2022_cwe_top25.html [7] Roland Croft、M Ali Babar和Mehdi Kholoosi。2023。软件漏洞数据集的数据质量。载于2023年IEEE/ACM第45届国际软件工程会议（ICSE）。[8] Jacob Devlin、Ming-Wei Chang、Kenton Lee和Kristina Toutanova。2018。BERT：面向语言理解的深度双向Transformer预训练。arXiv预印本 arXiv:1810.04805（2018）。[9] Jiahao Fan、Yi Li、Shaohua Wang和Tien N Nguyen。2020。包含代码变更和CVE摘要的AC/C++ 代码漏洞数据集。载于第17届国际软件仓库挖掘会议论文集。508–512。[10] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang等。2020。CodeBERT：面向编程语言和自然语言的预训练模型。arXiv预印本 arXiv:2002.08155（2020）。[11] Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、Shujie Liu、Long Zhou、Nan Duan、Alexey Svyatkovskiy、Shengyu Fu等。2020。GraphCodeBERT：通过数据流预训练代码表示。arXiv预印本 arXiv:2009.08366（2020）。[12] Hamel Husain、Ho-Hsiang Wu、Tiferet Gazit、Miltiadis Allamanis和Marc Brockschmidt。2019。CodeSearchNet挑战：评估语义代码搜索的现状。arXiv预印本 arXiv:1909.09436（2019）。[13] Yujia Li、Daniel Tarlow、Marc Brockschmidt和Richard Zemel。2015。门控图序列神经网络。arXiv预印本 arXiv:1511.05493（2015）。[14] Zhen Li、Deqing Zou、Shouhuai Xu、Hai Jin、Yawei Zhu和Zhaoxuan Chen。2021年。SySeVR：一种利用深度学习检测软件漏洞的框架。《IEEE可信与安全计算汇刊》19，4（2021年），2244–2258。[15] Zhen Li、Deqing Zou、Shouhuai Xu、Xinyu Ou、Hai Jin、Sujuan Wang、Zhijun Deng和Yuyi Zhong。2018。VulDeePecker：一种基于深度学习的漏洞检测系统。arXiv预印本 arXiv:1801.01681（2018）。[16] Yinhan Liu、Myle Ott、Naman Goyal、Jingfei Du、Mandar Joshi、Danqi Chen、Omer Levy、Mike Lewis、Luke Zettlemoyer和Veselin Stoyanov。2019。RoBERTa：一种稳健优化的BERT预训练方法。arXiv预印本 arXiv:1907.11692（2019）。[17] Shuai Lu、Daya Guo、Shuo Ren、Junjie Huang、Alexey Svyatkovskiy、Ambrosio Blanco、Colin Clement、Dawn Drain、Daxin Jiang、Duyu Tang等。2021。CodeXGLUE：用于代码理解与生成的机器学习基准数据集。arXiv预印本 arXiv:2102.04664（2021）。[18] Yisroel Mirsky、George Macon、Michael Brown、Carter Yagemann、Matthew Pruett、Evan Downing、Sukarno Mertoguno和Wenke Lee。2023。VulChecker：基于图的源代码漏洞定位。载于USENIX安全会议2023。[19] Georgios Nikitopoulos、Konstantina Dritsa、Panos Louridas和Dimitris Mitropoulos。2021。CrossVul：一个包含提交数据的跨语言漏洞数据集。载于第29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。1565–1569。[20] 美国国家标准与技术研究院。最后访问于2023年3月19日。国家漏洞数据库。https://nvd.nist.gov/ [21] 美国国家标准与技术研究院。最后访问于2023年3月19日。NIST软件保障参考数据集。https://samate.nist.gov/SARD [22] Vadim Okun、Aurelien Delaitre、Paul E Black等。2013。静态分析工具展示（SATE）IV报告。NIST特别出版物500（2013），297。[23] 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器。OpenAI博客1，8（2019），9。[24] Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li和Peter J Liu。2020。通过统一的文本到文本Transformer探索迁移学习的极限。《机器学习研究杂志》21，1（2020），5485–5551。[25] Rebecca Russell、Louis Kim、Lei Hamilton、Tomo Lazovich、Jacob Harer、Onur Ozdemir、Paul Ellingwood和Marc McConley。2018。使用深度表示学习自动检测源代码中的漏洞。载于2018年第17届IEEE国际机器学习与应用会议（ICMLA）。IEEE，757–762。[26] Benjamin Steenhoek、Md Mahbubur Rahman、Richard Jiles和Wei Le。2023。一项关于用于漏洞检测的深度学习模型的实证研究。载于2023年IEEE/ACM第45届国际软件工程会议（ICSE）。[27] Chandra Thapa、Seung Ick Jang、Muhammad Ejaz Ahmed、Seyit Camtepe、Josef Pieprzyk和Surya Nepal。2022。用于...的基于Transformer的语言模型

软件漏洞检测。载于第*38*届年度计算机安全应用大会论文集。481–496。[28] Xinda Wang、Shu Wang、Pengbin Feng、Kun Sun 和 Sushil Jajodia。2021年。Patchdb：大型安全补丁数据集。载于*2021*年第*51*届*IEEE/IFIP*年度可靠系统与网络国际会议（*DSN*）。IEEE，149–160。[29] Yue Wang、Weishi Wang、Shafiq Joty 和 Steven CH Hoi。2021年。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。*arXiv*预印本 *arXiv:2109.00859*（2021年）。[30] Frank F Xu、Uri Alon、Graham Neubig 和 Vincent J Hellendoorn。2022年。代码大语言模型的系统评估。*arXiv*预印本 *arXiv:2202.13169*（2022年）。[31] Fabian Yamaguchi、Nico Golde、Daniel Arp 和 Konrad Rieck。2014年。利用代码属性图对漏洞建模并发现漏洞。载于*2014*年*IEEE*安全与隐私研讨会。IEEE，590–604。[32] Yunhui Zheng、Saurabh Pujar、Burn Lewis、Luca Buratti、Edward Epstein、Bo Yang、Jim Laredo、Alessandro Morari 和 Zhong Su。2021年。D2A：通过差分分析构建的、面向基于人工智能的漏洞检测方法的数据集。载于*2021*年*IEEE/ACM*第*43*届软件工程国际会议：软件工程实践（*ICSE-SEIP*）。IEEE，111–120。[33] Yaqin Zhou、Shangqing Liu、Jingkai Siow、Xiaoning Du 和 Yang Liu。2019年。Devign：通过图神经网络学习全面的程序语义，实现有效的漏洞识别。神经信息处理系统进展32（2019年）。

A 模型训练设置

A.1 ReVeal 设置

我们使用 GitHub 上的 Joern [2] 来获取代码属性图。这个版本比 ReVeal 使用的版本更新，因为如果使用与 ReVeal 论文相同的旧版 Joern，所有数据集中的函数将近有一半无法提取为图。

对于门控图神经网络，我们将 Previous + DiverseVul 数据集的最大训练轮数设为 50，将 Previous 数据集的最大训练轮数设为 100，并选择验证集 F1 分数最高的模型，用于第4.2节中的实验。对于第 4.4 节中的实验，我们将最大训练轮数设为 60。我们遵循 ReVeal 源代码中的原始设置，使用 Adam优化器，学习率为 0.0001，权重衰减为 0.001。

为了训练 ReVeal 中的分类层，我们将最大训练轮数设为 100，并遵循作者的设置：如果验证集上的 F1 分数连续 5 个训练轮次没有提高，就停止训练过程。我们遵循 ReVeal 源代码中的原始设置，使用 Adam优化器，学习率为 0.001，且不使用权重衰减。

A.2 微调设置

按照标准方法，我们在 Transformer 模型上添加线性分类头，以微调大语言模型。对于 RoBERTa、CodeBERT 和 GraphCodeBERT，我们在线性层中使用表示第一个词元（[CLS]）的嵌入。对于 GPT-2- Base、CodeGPT 和 PolyCoder，我们在线性层中使用最后一个词元的嵌入。对于 T5 Base、CodeT5 Small、CodeT5 Base 和 NatGen，我们在线性层中使用最后一个解码器状态的嵌入。

我们使用 32 的训练批次大小、2e-5 的学习率和 Adam优化器，训练 10 个训练轮次。我们采用线性学习率衰减，并预热 1,000 步。每 1,000 步检查一次模型的验证性能，并保存验证性能最佳的模型用于测试。除一项例外，所有模型和训练数据设置均使用相同的学习率。在第4.2节所述的随机数据划分中，使用来自 Previous + DiverseVul 的数据训练 RoBERTa 时，我们使用 1e-5 的学习率，因为更大的学习

[2]在2022年10月11日提交 a6aa08ee9842eedb52e149695e3a34500b6ceab0之后。

比率会导致模型退化，使其始终将函数预测为不易受攻击。