TRACED: Execution-aware Pre-training for Source Code

Yangruibo Ding
Columbia University
New York, NY, USA

Ben Steenhoek
Iowa State University
Ames, IA, USA

Kexin Pei
Columbia University
New York, NY, USA

Gail Kaiser
Columbia University
New York, NY, USA

Wei Le
Iowa State University
Ames, IA, USA

Baishakhi Ray
Columbia University
New York, NY, USA

摘要

*警告：该PDF由GPT-Academic开源项目调用大语言模型+Latex翻译插件一键生成，版权归原文作者所有。翻译内容可靠性无保障，请仔细鉴别并以原文为准。项目Github地址 https://github.com/binary-husky/gpt_academic/。当前大语言模型: gpt-4o-mini，当前语言模型温度设定: 0。为了防止大语言模型的意外谬误产生扩散影响，禁止移除或修改此警告。

大多数现有的预训练语言模型针对源代码，侧重于学习静态代码文本，通常辅以静态代码结构（抽象语法树（abstract syntax tree）、依赖图（dependency graphs）、*etc.*）。然而，程序语义在实际执行之前并不会完全暴露。在没有理解程序执行的情况下，静态预训练模型无法全面捕捉动态代码特性，例如分支覆盖率和运行时变量值，因此在代码理解任务（如检索语义克隆和检测软件漏洞）中效果较差。

为了缩小语言模型的静态特性与程序的动态特性之间的差距，我们引入了TRACED，一种针对源代码的执行感知预训练策略。具体而言，我们通过源代码、可执行输入和相应的执行轨迹的组合来预训练代码语言模型。我们的目标是在预训练期间教会代码模型复杂的执行逻辑，使模型能够静态地估计动态代码特性，而无需在任务特定的微调过程中反复执行代码。

为了说明我们提出的方法的有效性，我们在三个下游任务上对TRACED进行了微调和评估：静态执行估计、克隆检索和漏洞检测。实证结果表明，TRACED在完整执行路径预测方面相对提高了静态预训练代码模型12.4%，在运行时变量值预测方面提高了25.2%。TRACED在克隆检索和漏洞检测方面也显著优于静态预训练模型，跨越四个公共基准测试。

1 INTRODUCTION

机器学习（ML）在源代码方面使许多软件工程任务成为可能，例如自动程序修复 [11, 21–23]、错误查找 [8, 54]和重构 [7]。最近，训练用于源代码理解的ML模型的常见做法是基于在源代码上预训练基于transformer的语言模型。这些方法将源代码程序视为静态文本 [1, 6, 16, 49]，有时还会增强程序特定的结构，例如抽象语法树和依赖图 [10, 17, 18, 35]，并调整自然语言的预训练策略以学习程序表征（representation）。

然而，许多源代码理解任务需要对程序行为有更全面的理解。例如，检测语义克隆[32]涉及确定两段代码在相似输入下是否表现相似，即使它们的结构显然不同。同样，检测漏洞通常要求开发人员分析潜在问题位置是否可以被执行，以及什么样的价值流可能暴露任何漏洞。虽然现有的代码模型主要训练以捕捉静态代码属性，但它们在推理程序行为方面并不有效。事实上，许多更深层次的程序语义只有在代码执行时才会显现。因此，在需要更深层次语义理解的任务中，它们往往表现不佳。

激励示例。. 图 1 展示了一个具有简单执行逻辑的示例，以说明静态预训练代码模型在分支覆盖预测上的失败。我们查询了三个预训练代码模型，CodeX [13] (code-davinci-002)，UnixCoder [17]，和 TRACED

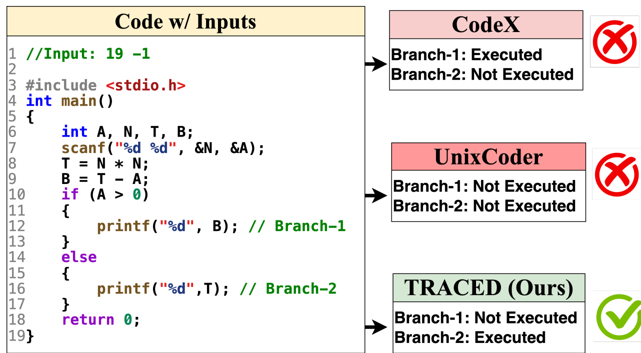

图 1: 来自CodeNet编码挑战第3597号的一个激励性例子[41]揭示了静态预训练的代码语言模型，无论其大小如何，都无法针对特定输入推理分支覆盖率，而TRACED则通过增强程序执行特性，正确识别执行路径。

(我们的模型)，根据给定的程序输入预测分支覆盖。对于 CodeX，我们使用精心设计的问题提示模型，类似于 [36]，以请求在零样本 (zero-shot) 设置下的分支覆盖预测。具体而言，我们通过在第 12 行和第 16 行的末尾添加注释来增强提示：`// 这一行会被执行吗？是或否？`。为了提供更多关于数据流的提示，我们在第 10 行的末尾进一步添加了一个注释：`// A 是 -1,因为它接受输入的第二个值`。不幸的是，即使提供了关于分支预测所需数据流的额外提示，CodeX 仍然未能预测正确的覆盖标签，这表明它无法解释这个简单的执行。

除了零样本提示外，我们还研究了微调预训练代码模型以预测执行是否能导致更好的分支预测。具体而言，我们微调了另一个流行的预训练代码模型 UnixCoder [17]，以预测分支执行，同时确保在训练期间未见到激励示例。从图 1 的推理结果中，我们注意到即使经过微调，UnixCoder 也无法预测被覆盖的分支。它预测没有任何分支会被覆盖，这表明它没有基本的理解，即对于这个特定示例，至少有一个分支在有效输入时总是会被执行。

我们的方法。. 为了解决静态预训练代码模型的局限性，我们提出了 TRACED，一种执行感知的预训练策略，以捕捉源代码的静态和动态视角。具体而言，我们使用多任务目标对基于 Transformer 的语言模型进行预训练，预测源代码、程序状态和执行覆盖，迫使模型同时推理程序的运行时行为和源代码的自然性 [43]。我们解决了几个技术挑战，例如表示程序执行状态、编码运行时变量值和表示代码覆盖，以实现预训练策略。

表示程序状态。. 在程序执行过程中，变量用于存储程序使用的数据。这些变量可以具有不同的类型，例如整数、浮点数、指针和数组。随着程序的执行，这些变量的值会发生变化，反映程序状态的变化。因此，软件开发人员通常通过调试工具监控变量值，以观察执行事实 [52] 并理解程序的动态行为。

在本工作中，我们将特定时间步的程序状态定义为当前作用域中每个定义变量的值集合。换句话说，程序状态等同于调试器的值映射表，开发人员在程序被特定断点暂停时监控该表。

值量化。. 虽然运行时变量值被追踪为具体值，但直接学习它们给机器学习模型带来了挑战。具体值跨越了广泛的可能值范围，尤其是在考虑不同数据类型（整数、浮点数、数组、指针等）时，导致高维、复杂但稀疏的数据分布。这种数据复杂性和稀疏性增加了模型学习变量值之间模式和关系的挑战，因为它必须处理许多独特的输入，这导致模型过拟合并记忆特定实例，而不是推广到更广泛的模式。此外，具体值的噪声、异常点 (outlier) 和不规则性也误导了模型的学习过程。我们将在§6.3中实证展示这些局限性。

为了降低数据复杂性并增加密度，我们定义了三十个值类别，覆盖广泛的变量类型，以将连续但稀疏的变量值映射到离散的区间。我们将此过程称为值量化，其设计类似于信号处理中的量化[1]。这种简化可能有助于模型对噪声和异常点更具韧性，使其能够专注于学习潜在的执行模式和变量之间的关系，而不是对特定实例或不规则性敏感。

表示执行覆盖率.. 虽然程序状态标签提供了关于程序当前状态的重要信息，但它们并未捕捉到程序如何到达该状态的信息。为了通过更全面的执行特征来增强训练，除了变量*value*（价值）之外，我们还记录

[1]https://en.wikipedia.org/wiki/Quantization_(signal_processing)

了执行期间的执行覆盖率，具体包括哪些行被执行，哪些行未被执行，并为模型（model）构建执行覆盖特征以供学习。

结果. 我们使用三个任务对TRACED的性能进行微调和评估：静态执行估计、克隆检索和漏洞检测。在静态预测程序执行方面，TRACED在执行路径预测上显著提高了静态预训练代码模型12.4%，在运行时变量*value*（价值）预测上提高了25.2%。TRACED在代码理解任务中也取得了最先进的结果：TRACED在CodeXGLUE-POJ104 [32]上报告了91.2%的MAP@R，在ReVeal [8]上报告了50.4%的F1，在CodeXGLUE-defect-detection [32]上报告了65.9%的准确率。

贡献. 我们做出了以下贡献：

- 我们提出了一种简化和紧凑的程序执行表征（representation），包括程序状态（state）和执行覆盖，以有效指导代码模型（model）学习程序语义并推理程序行为。
- 我们提出了一种新颖的多任务预训练策略，以共同学习静态和动态代码属性。因此，采用我们方法的预训练模型将具备良好的执行意识。
- 我们使用提出的追踪表征 (trace representation) 和执行感知策略 (execution-aware strategy) 对TRACED进行预训练，并在多个下游任务上评估其性能。实验结果表明，TRACED在这些任务中显著优于静态预训练的代码模型。
- 我们将公开发布我们的数据、代码和预训练模型（pre-trained models），以促进开放科学。

2 OVERVIEW

图 2 显示了 TRACED的概述，包含三个主要阶段：（1）追踪源代码并工程化特征，（2）使用程序追踪进行执行感知的预训练，以及（3）加载预训练权重并执行特定任务的微调。

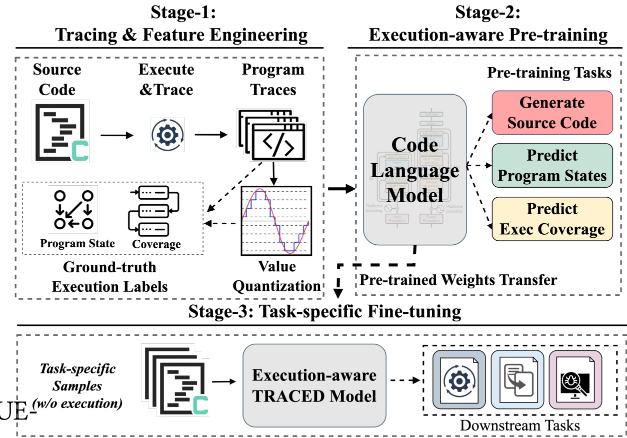

图 2: TRACED工作流程概述。

阶段一:追踪与特征工程。. 本阶段的目标是为预训练准备数据。该过程始于提供源程序及其可执行输入。第一步是使用每个输入执行程序，以生成相应的追踪。追踪记录了运行时变量的值，以及执行覆盖，记录了程序的完整执行历史，并揭示了程序状态在执行过程中的变化。为了减少数据的复杂性和稀疏性，并使模型更容易学习变量值之间的模式和关系，我们将追踪中记录的具体运行时值量化为预定义的价值（value）范围。量化过程将连续值映射到一组固定的离散值或区间。通过量化这些值，我们创建了一组有限的可能输出，可以在训练期间用作真实标签（ground-truth labels）。量化后，我们创建程序状态标签和执行覆盖标签，这将帮助模型捕捉程序执行。最终数据集包含一组样本和标签，其中每个样本包括源代码及其程序输入，标签表示该样本的执行追踪。

阶段二:基于追踪的执行感知预训练。. 我们利用从阶段一获得的预处理样本和标签进行监督预训练。具体而言，我们使用基于Transformer编码器的模型 [31] 来学习程序追踪，并提高模型对程序执行的理解。该模型可以从头开始训练，也可以加载现有代码语言模型的预训练权重。为了实现生成执行感知代码表征的目标，我们提出了三个预训练目标。第一个目标是学习生成源代码。我们认为理解代码文本的自然性 [20, 43] 对于模型捕捉更复杂的信号（如程序执行）是基础。该目标通过掩码语言建模（MLM）实现，该方法掩盖源代码中一定比例的标记（token），并训练模型

根据周围上下文重建被掩盖的标记。第二个目标是学习预测程序状态。通过预测在阶段一生成的程序状态标签，模型学习捕捉数据流和代码执行的副作用。第三个目标是预测执行覆盖。通过预测阶段一生成的执行覆盖标签，模型学习捕捉动态控制流，并帮助模型理解程序状态是如何达到和演变的。

阶段三:任务特定的微调。. 最后，我们将TRACED应用于几个下游任务。我们加载TRACED的预训练权重，对模型进行特定任务的微调，并持续更新模型权重。微调不需要执行程序；相反，TRACED将根据其在预训练期间学习到的执行信号静态推理执行，并学习相应地完成任务。在许多有用的应用中，我们可能没有可用的程序追踪。我们考虑TRACED的三个下游任务：静态执行估计，包括执行覆盖和运行时变量值预测、克隆检索和漏洞检测。

3 TRACING & FEATURE ENGINEERING

在本节中，我们介绍 TRACED如何从程序跟踪中构建可学习特征，以便模型（model）学习程序执行。

3.1 Representing Program States

为了模仿人类开发者监控变量价值（value）以理解程序行为的方式，我们提出使用运行时变量价值的日志来训练神经模型，以识别执行模式并推断动态程序行为，这种方式类似于人类直觉。通过在执行过程中记录变量价值的日志，我们可以以更紧凑和可解释的形式表示程序状态（state），使其更易于深度神经网络处理。

我们通过在执行过程中在程序点拍摄变量价值的快照来构建程序状态。当我们在特定时间步拍摄快照时，类似于程序在第 l 行之后设置的调试断点暂停的时刻，我们维护一个价值映射 M，将变量映射到其当前价值，类似于调试器的价值映射表。为了记录程序状态，我们在每行执行后拍摄价值快照并记录变量的当前价值。

定义: *程序状态（Program State）*。 正式地，我们将特定行 l 执行后的程序状态定义为 $s(l)$，表示为此时的变量价值集合：

$$s(l) = \{M(v, l) \mid v \in V,\ l \in L\}$$

V 表示所有追踪变量的集合，L 是源代码的行集合。图 3 显示了一个简单阶乘程序的示例，源代码后的注释指示了该行执行后的程序状态（state）。此外，对于没有可执行代码的行，例如图 3 的第 8 行，我们不记录程序状态（state）。

```
1  // INPUT: 4
2  int factorial() {
3    int x, y; // {'x': 32767, 'y': 32767}
4    x = atoi(argv[1]); // {'x': 4, 'y': 32767}
5    if (x < 0) { // {'x': 4, 'y': 32767}
6      y = -1;
7      return y;
8    }
9    y = 1; // {'x': 4, 'y': 1}
10   for (int i = 1; i <= x; i++) // {'x': 4, 'y': 24, 'i': 5}
11   {
12     y *= i; // {'x': 4, 'y': 24, 'i': 5}
13   }
14   return y; // {'x': 4, 'y': 24, 'i': 5}
15 }
```

图 3: 具有具体运行价值 (runtime values) 的程序状态 (program states).

注意，由于循环或递归，源代码行可能会被执行多次。虽然对程序执行的更详细表征（representation）可能提供额外的见解，但它也增加了模型（model）的复杂性和计算要求。作为复杂性与性能之间的权衡，我们使用每行的最后一次执行来确定程序状态（state），以便 $s(l)$ 在执行终止之前不断更新。

我们基于对真实执行的观察应用这种权衡。具体而言，最后出现的价值（value）通常足以捕捉循环和递归的结果。例如，在调用递归函数时，仅会取返回变量的最后出现的价值来满足调用者的后续执行。同样，当循环结束时的最终价值也会参与未来的执行。如图 3 的第12行所示，变量 y 在循环内被乘以以计算阶乘。它的价值在每次迭代中变化，但由于只有最终

价值被用作返回值（第14行），因此推理程序的整体行为时信息量较少。因此，我们将使用循环最后一次执行的价值来表征（representation） y。

3.2 Quantized Variable Values

在§1中我们介绍过，具体价值（concrete values）的分布是稀疏且复杂的，因此对于统计模型来说很难拟合。此外，具体价值并不总是必要的。一些常见的程序行为伴随着极大或极小的变量值——例如，在C语言中，未初始化的变量通常被设置为零或非常大的变量，但具体价值并没有实际意义，因为它们仅依赖于栈上剩余的数据，这些数据可能是随机的大或小。该模型可以通过估计变量的价值范围来表示这些行为，而不需要准确预测它们的具体价值，因为这些具体价值并不具有信息性或意义。图3展示了这些情况中的一些：在执行第3行之后，x和y未初始化，并随机初始化为32,767，这没有具体意义，只会使训练数据变得嘈杂和稀疏。

为了减少数据复杂性并增加密度，我们在表 1中定义了30个量化值的类别。为了全面表示变量的价值（value），所提出的量化类别考虑了两种类型，即静态定义的数据类型和价值类型，以及动态运行时的价值（value）。我们的量化类别涵盖了最常见的变量类型和价值（value）类型，我们发现这些类别足以捕捉重要的程序执行行为和关系。通过关注最频繁的价值（value）类型，我们可以捕捉程序执行的基本特征。这使得我们的方法在捕捉一般化的程序执行行为和模式方面非常有效。我们在§ 6.3中实证说明了我们量化策略的有效性。

3.3 Building Learnable Labels for Code Models

我们使用带有痕迹的监督预训练。我们为代码模型构建标签，以学习执行的两个主要视角：程序状态和执行覆盖。

程序状态标签。. 正如我们在前面的章节中讨论的，我们首先在执行过程中跟踪程序变量并记录它们的运行时**值**。然后，我们将这些**值**量化为预定义的类别。这个过程产生了一系列程序状态，每个状态由一组量化的变量**值**表示（如图 3所示），我们在这些程序状态的基础上构建可学习的特征用于代码模型。具体而言，我们为可以量化为表 1类别的变量构建标签，并训练模型在给定其源代码**表征**的情况下预测这些标签（§ 4.1.2）。每个变量的标签表示为一个元组：*(数据类型, 价值类型, 量化**值**)*。例如，在图 3中，出现在第3行的变量X的标签为*(基本, 整数, 正大)*, 因为X的当前**值**为32,767。我们为所有可以量化的有效变量的所有出现构建这样的标签，所有标签的集合被视为代码示例的程序状态标签。

表 1: TRACED的量化变量价值（variable values）设计。

Data Type	Value Types	Concrete Value	Quantized Value
Basic	Integer	$0 < v \leq 10,000$	Positive Regular
		$10,000 < v$	Positive Large
		0	Zero
		$-10,000 \leq v < 0$	Negative Regular
		$v < -10,000$	Negative Large
	Float/Double	$0.0 < v \leq 1.0$	Positive Small
		$1.0 < v \leq 10,000.0$	Positive Regular
		$10,000.0 < v$	Positive Large
		0.0	Zero
		$-1.0 < v < 0$	Negative Small
		$-10,000.0 \leq v < -1.0$	Negative Regular
		$v < -10,000.0$	Negative Large
	Character	'\0'	Null
		$v \in$ {a-zA-Z}	Alphabetic
		$v \neq$ '\0'; $v \notin$ {a-zA-Z}	Non-alphabetic
	Boolean	0	False
		1	True
	Void	-	Void
Array	Integer	$[v_1, v_2, \ldots, v_n]$; $quantize(v_i) \in$ Integer	Initialized
			Not Initialized
	Float/Double	$[v_1, v_2, \ldots, v_n]$; $quantize(v_i) \in$ Float/Double	Initialized
			Not Initialized
	Character	"⟨string⟩"	Initialized
			Not Initialized
Pointer	Integer	0x0	Null
		Not 0x0	Not Null
	Float/Double	0x0	Null
		Not 0x0	Not Null
	Character	0x0	Null
		Not 0x0	Not Null

执行覆盖标签。. 为了统一我们的设计并减少模型学习过程的复杂性，我们还为每个变量的出现构建执行覆盖标签，以与程序状态标签对齐。具体而言，

我们用二进制标签表示变量的覆盖，“是"或“否"。在执行行中的变量将被标记为“是"，而在未执行行中的变量将被标记为“否"，并进一步分配一个“未知"的量化值，同时保留它们的数据类型和**价值**类型标签。例如，在图3中，第6行未被执行，因此该行的y的覆盖标签为“否"，程序状态标签为(基本, 整数, 未知)，而第9行的y的覆盖标签为“是"，程序状态标签为(基本, 整数, 正常)。

4 MODEL

在本节中，我们解释了TRACED的组件和在预训练与微调期间的学习目标的细节。

模型架构. 图 4 显示了TRACED的预训练的高层架构。TRACED的主干是一个12层的Transformer编码器，类似于BERT [9]和RoBERTa [31]，它学习通用的代码表征（representation）。在主干Transformer层之上，TRACED堆叠了多个多层感知机（MLP）层作为不同任务的预测头。在预训练期间，如图4所示，TRACED使用了一个语言模型预测头，即LM层，以根据其上下文表征预测被掩码（mask）的标记（token），一个程序状态预测头以预测我们在§ 3.3中定义的程序状态标签，以及一个执行覆盖头以预测执行覆盖标签。对于特定任务的微调，主干Transformer层加载了预训练的权重，而预测头则被替换为一个为特定下游任务定制的新初始化头。

4.1 Execution-aware Pre-training

4.1.1 Model Input of Pre-training. 每个预训练样本包括一个可执行程序的源代码和一个有效的可执行输入。如图 4所示，可执行输入和源代码被展平并连接为一个序列。为了区分输入和源代码，因为它们是不同的模态，TRACED使用特殊的[SEP]标记（Token）来分隔它们并指示各自的位置。为了缓解编程语言的词汇外问题 [25]，TRACED采用了一个预训练的SentencePiece [28]子词标记（Token）器，词汇量为50,000。它使用这个标记（Token）器将连接的序列划分为一个新的子标记（Token）序列。

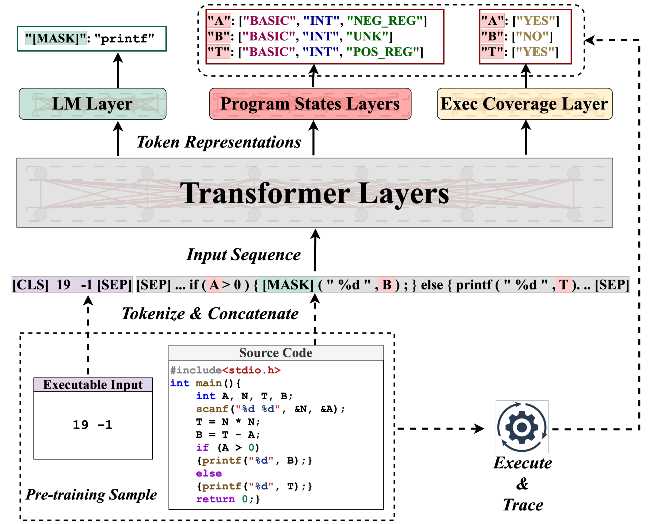

图 4: TRACED的高级模型架构。在程序状态层的标签中，NEG_REG 意味着 “负常规” (Negative Regular)，UNK 意味着 “未知” (Unknown)，而 POS_REG 意味着 “正常规” (Positive Regular)，这些我们在表 1 中进行了定义。

正式地，我们将可执行输入定义为$E = \{e_1, ..., e_i\}$，将展平的源代码定义为$C = \{c_1, ..., c_j\}$，那么最终的模型输入将是$\mathcal{I}$ $=\{\texttt{[CLS]}, e_1, ..., e_i, \texttt{[SEP]}, \texttt{[SEP]}, c_1, ..., c_j, \texttt{[SEP]}\}$。如果可执行输入和源代码过长，TRACED会分别截断它们。TRACED将可执行输入序列的最大长度设置为64个标记（Token），源代码的最大长度设置为960个标记（Token）。这些数字是根据我们预训练数据集的可执行输入长度的统计数据选择的（§5.2.1），并适应源代码的其余模型输入。

请注意，执行跟踪不是模型输入的一部分，而是作为模型在预训练期间进行预测的真实值（ground truth）标签使用。

4.1.2 Learning Execution-aware Code Representations with Traces. TRACED通过多重目标进行预训练，以共同捕捉源代码的静态和动态视角。

学习代码文本。. 学习代码文本是理解程序执行的基本第一步，因为代码文本是捕捉代码自然性 [20] 和其他静态属性的主要来源。我们通过调整掩码语言模型目标 [9, 16, 31] 来实现代码文本学习目标。具体而言，给定模型输入序列 $\mathcal{I}$，TRACED随机选择 15% 的标记（Token） [9, 31] 仅来自源代码序列 C 部分，

并用特殊的`[MASK]`标记替换（例如，图4中的`printf`被掩码）。它保持可执行输入序列 E 不变。模型被训练以将`[MASK]`的上下文编码到其代码表征 r_{masked} 中，并在该表征的条件下重构具体的掩码标记。我们将学习代码文本的损失表示为：

$$\mathcal{L}_{code-text} = \sum_{masked} -logP(c_{masked} \mid r_{masked}) \quad (1)$$

在图4中，LM（语言模型）层接收由最后一个transformer层生成的掩码标记（Token）表征。然后，LM层通过将标记（Token）表征映射到词汇表中每个标记的概率，使用MLP（多层感知器）层来预测具体的标记（Token）。这个过程可以被视为一个分类任务，其中类别的数量等于词汇表的大小。目标是学习从掩码标记（Token）表征到词汇表中最可能的标记（Token）之间的映射，给定其上下文。

学习程序状态。. 第二个预训练目标，程序状态预测（PSP），旨在使模型通过预测被追踪变量的程序状态标签来学习程序执行行为。这些程序状态标签，如§3.3中所定义，包含有关程序执行结束时变量的数据类型、价值类型和量化值的信息。具体而言，TRACED首先识别源代码序列中的变量标记（Token），记作 $\{c_{var} \mid c_{var} V\} \subseteq C$，其中 V 是所有被追踪变量的集合，C 是源代码序列。然后，它提取每个变量标记（Token）的表征 r_{var}，并将其输入到程序状态层。程序状态层预测变量作为真实数据类型 d_{var}、价值类型 t_{var} 和量化值 q_{var} 的联合可能性。请注意，如果一个变量被标记（Token）化为多个子标记（Token），所有属于的子标记（Token）共享相同的程序状态标签。最后，TRACED将PSP的损失计算为用于预测其程序状态的所有变量标记（Token）损失的总和。从数学上讲，损失表示如下：

$$\mathcal{L}_{program-state} = \sum_{var} -logP(\ d_{var}, t_{var}, q_{var} \mid r_{var}) \quad (2)$$

学习变量覆盖。. 第三个预训练目标，变量覆盖预测（VCP），旨在学习执行覆盖，这对于理解给定特定输入的代码控制流至关重要。与PSP目标类似，VCP的目标是对变量标记（Token）进行预测。变量覆盖的标签是二元的，其中1表示该变量所属的行被覆盖，否则为0。此外，属于同一变量的子标记（Token）将被分配相同的覆盖标签。VCP的损失如下：

$$\mathcal{L}_{var-cov} = \sum_{var} -logP(cov_{var} \mid r_{var}) \quad (3)$$

最后，TRACED将所有三个目标的损失结合起来，并计算它们的总和作为预训练样本的最终损失。它通过预测层和主干 transformer（transformer）层反向传播梯度，以更新它们的权重。我们将 TRACED的全套可学习参数表示为 θ，并将损失表示如下：

$$\mathcal{L}(\theta) = \mathcal{L}_{code-text}(\theta) + \mathcal{L}_{program-state}(\theta) + \mathcal{L}_{var-cov}(\theta) \quad (4)$$

4.2 Task-specific Fine-tuning

TRACED加载了预训练的 Transformers 层的模型权重，这些权重用于生成执行感知的代码表征，并进一步对模型进行微调以适应下游任务。我们将三个下游任务视为TRACED的主要应用：（1）程序执行的静态估计，包括执行覆盖预测和运行时变量值预测；（2）语义克隆检索；（3）漏洞检测。

静态执行估计。. 我们的预训练目标是将执行模式编码到代码表征中，以便模型能够静态估计程序执行。作为直接应用，TRACED微调模型以使用源代码和程序输入预测（1）执行覆盖和（2）运行时变量值。TRACED评估微调后的模型，以相同的方式估计未见程序的执行。

具体而言，对于执行覆盖预测，TRACED确定所有分支语句以定位源代码中的分支，$B = \{b_1, b_2, ..., b_m\}$。它训练模型为每个 $b_i \in B$ 预测一个二元标签，0 表示该分支未被当前执行覆盖，1 表示已覆盖。为了方便模型进行预测，在每个分支的开头插入特殊标记`[MASK]`。例如，以下`if-else`语句有两个分支，已为分支预测进行预处理：`if (condition) {[MASK] ...} else {[MASK] ... }`。在微调过程中，Transformer 层学

习将分支信息编码到相应的 [MASK] 标记表征中，利用内置的双向注意力和位置编码。然后，分类头使用 [MASK] 表征来预测某个分支是否被当前执行覆盖。对于变量值预测，TRACED确定变量 $V = \{v_1, v_2, ..., v_n\}$ 并训练模型在执行过程中预测它们的量化值（§3.2）。

语义克隆检索。. 检测语义克隆对软件维护具有重要意义 [26, 30]，但在实践中非常具有挑战性，因为语义克隆之间的标记和语法结构重叠可能非常有限。此任务要求模型在不执行程序的情况下估计程序行为，并捕捉它们之间的相似性。它评估模型的语义推理能力，以识别代码相似性并检索克隆：给定一个程序作为查询，以及一组任意程序作为候选，模型需要从可能成千上万的候选中识别查询的语义克隆。

漏洞检测。. 漏洞检测是软件安全中的一项关键任务，旨在识别软件代码中可能被攻击者利用的潜在安全漏洞。这些漏洞可能由于多种原因而存在，包括编程错误、设计缺陷或配置问题。在软件开发生命周期的早期检测这些漏洞可以防止潜在攻击、降低风险并节省资源。我们在包含漏洞和非漏洞代码样本的数据集上微调 TRACED的预训练模型，以便模型通过估计其执行行为来学习将代码功能分类为漏洞或非漏洞。

5 EXPERIMENTAL SETUP

5.1 Trace Collection

在本节中，我们解释了如何在给定源代码和程序输入的情况下，追踪程序中的动态信息以生成具体的追踪。

首先，我们使用 gcc 编译程序，选项为 -g -O0。选项 -g 保留调试信息，这对于使用调试器读取变量和源代码位置是必要的，而选项 -O0 禁用编译器优化，这可能会优化掉某些变量，从而在运行时无法读取它们。我们使用此选项是因为我们希望以变量 值（value）的语义来建模 源代码（source code），而不是优化后的机器代码。

其次，我们将程序加载到给定的标准输入重定向到 stdin，并附加 gdb[2] 调试器，使用 Python API 实现追踪命令。从入口点（main）开始，我们使用 step 命令逐行执行程序。在每一行，我们打印出所有作用域内变量的具体 值（value）。我们还在每个用户定义函数的入口处设置断点，在那里记录每个参数的 值（value）。对于数值类型，我们简单地记录它们的字符串 表征（representation）。对于 char 和 char *（字符串）类型，我们记录字符/字符串的人类可读值。我们使用 gdb 的漂亮打印器来打印 struct 类型和静态分配的数组类型，例如 int[<size>]. 对于指针类型，我们将指针的内存地址以十六进制代码的形式打印。我们仅追踪在源代码中定义的函数，并跳过所有标准库函数。

5.2 Dataset

5.2.1 Pre-training Dataset. IBM的CodeNet数据集 [41] 包含来自AIZU在线评测和AtCoder平台的4,053个编程挑战，涉及多种编程语言，每个问题都有来自不同程序员提交的多达数千个实现。在本研究中，我们将C语言作为预训练和下游任务的主要资源，因此我们构建了包含C语言解决方案的编程挑战的预训练数据集。除了样本数量庞大和编程挑战的复杂性外，我们选择CodeNet来构建我们的数据集，因为它为每个挑战至少维护一个、最多二十个可执行输入，这样我们就可以执行和追踪挑战的实现，从而为模型学习构建我们的执行标签。

在1,900个具有C语言解决方案的编程挑战中，我们选择了1,805个来构建预训练数据集，其余95个问题作为保留问题，用于评估模型在下游静态执行估计任务中的能力。通过挑战严格划分样本有效避免了训练集到保留集的数据泄漏问题。我们随机为每个挑战抽样最多200个执行轨迹，最终得到了121,319个训练轨迹。

5.2.2 Downstream tasks. 在本节中，我们介绍用于每个下游任务的数据集，并解释相应的评估指标。这些数据集的统计信息见表 2。

[2] https://www.sourceware.org/gdb

静态执行估计. 我们使用 CodeNet 构建此任务的数据集。我们从经过预训练选择的 1,805 个挑战中构建训练样本，并从保留的 95 个挑战中构建评估样本，以避免模型记忆和数据泄漏。

指标. 对于执行覆盖预测，我们考虑两种粒度的评估指标：完整执行路径和分支覆盖。具体而言，对于一个具有 m 个分支的样本，我们将其标签的完整集合表示为 $LB = \{lb_1, lb_2, ..., lb_m\}$，模型预测集表示为 $\hat{LB} = \{\hat{lb}_1, \hat{lb}_2, ..., \hat{lb}_m\}$。如果 $LB == \hat{LB}$，我们认为预测与完整执行路径匹配。对于分支覆盖，我们计算 $lb_i == \hat{lb}_i$ 的发生次数，其中 $1 \leq i \leq m$，并报告准确率、精确率、召回率和 F1。类似地，对于程序中的 n 个量化变量值，$QV = \{qv_1, qv_2, ..., qv_m\}$，我们的模型做出预测为 $\hat{QV} = \{\hat{qv}_1, \hat{qv}_2, ..., \hat{qv}_m\}$。如果 $QV == \hat{QV}$，我们说模型准确预测了完整执行。对于单个值匹配，我们计算 $qv_i == \hat{qv}_i$ 的发生次数并报告准确率。

语义克隆检索. 我们使用 CodeXGLUE-POJ104 [32, 33] 作为此任务的数据集。CodeXGLUE-POJ104 包含 104 个编程挑战，每个挑战有 500 个由不同程序员提交的 C/C++ 解决方案。CodeXGLUE [32] 通过将数据集拆分为训练集（64 个挑战）、开发集（16 个挑战）和测试集（24 个挑战）来重建它作为一个公共基准，且任意两个集合之间没有重叠的挑战。

指标. MAP@R（平均精度 @ R）[3] 是此任务的主要指标，我们遵循 CodeXGLUE 基准的设计。R 的平均精度是评估信息检索质量的常用指标；它测量一组针对查询程序呈现的前 R 个克隆候选者的平均精度分数。对于 CodeXGLUE，"R" 为 499，因为每个挑战有 500 个解决方案。

漏洞检测. 我们利用了三个公开可用的数据集：REVEAL (RV) [8]、D2A [53] 和 CodeXGLUE-Devign (CXG) [32, 54]。REVEAL 数据集由 Chakraborty *et al.*精心策划，以模拟一个真实世界场景，其中错误相对较少，导致有缺陷样本与良性样本之间的比例约为 1:10。D2A 数据集是一个平衡的数据集，专注于修复错误的提交。它将修改函数的先前版本标记为有缺陷的，将修复后的版本标记为良性的。最后，Zhou *et al.*引入的 CodeXGLUE-Devign 数据集也是一个平衡的数据集，已被 CodeXGLUE 重建为公共基准，确保所有模型可以使用相同的训练/验证/测试划分进行评估。

指标. REVEAL 是一个不平衡的数据集，因此我们使用 F1 作为评估指标。D2A 和 Devign 是平衡的数据集，因此我们遵循原始基准报告分类准确率。

[3]https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Mean_average_precision

表 2: 下游任务数据集的详细信息。

Task	Dataset	Train	Valid	Test
Execution Estimation	CodeNet	121,319	13,116	13,116
Clone Detection	CXG-POJ104	32,000	8,000	12,000
Vulnerability Detection	REVEAL	15,867	2,268	4,535
	D2A	4,644	597	619
	CXG-Devign	21,854	2,732	2,732

5.3 Model Configuration

TRACED的骨干是一个标准的RoBERTa$_{\text{BASE}}$架构 [31]，具有12层Transformer-encoder，每层有12个注意力头，隐藏维度为768。TRACED使用来自UnixCoder的预训练权重进行初始化 [17][4]，我们使用其BPE标记器将稀有标记拆分为BPE子标记。最大序列长度为1024个BPE标记，较长的序列将被截断。当代码样本与可执行输入配对时，可执行输入的最大长度为64，源代码为960。我们的实验在2 × 24GB NVIDIA GeForce RTX-3090 GPU上进行。我们进一步对模型进行10轮次的预训练，以学习程序执行，使用两个学习率，5e-5和2e-5，并报告下游任务的最佳表现模型。对于所有微调任务，我们使用8e-6的学习率。学习率通常在后期阶段下降 [10, 16, 18]，因此TRACED遵循相同的设计。我们使用Adam优化器 [27]，并采用线性学习率衰减。我们的模型主要使用Pytorch [12]和Huggingface [14]实现。

6 EVALUATION

在本节中，我们提出以下四个研究问题（RQs）：

[4]具体来说,我们加载unixcoder-base-nine,因为它的预训练考虑了C语言代码样本:https://huggingface.co/microsoft/unixcoder-base-nine。请注意，此检查点仅使用MLM目标进行预训练，而原始论文 [17]报告了其他未公开发布的表现更好的变体。

- **RQ1:** TRACED在静态估计程序执行方面的有效性如何？
- **RQ2:** 我们提出的训练策略如何有助于学习程序执行（program execution）？
- **RQ3:** 我们提出的量化价值（quantized values）对程序有效吗，以指导模型（model）学习程序执行？
- **RQ4:** TRACED在代码理解任务中相对于静态预训练基线的表现如何？

6.1 RQ1. Effectiveness of TRACED in Static Estimation of Execution

在本节中，我们展示了TRACED在静态估计程序执行方面的有效性。评估比TRACED的预训练更具挑战性和现实性，因为它要求模型不仅预测单个变量，还要预测分支和完整的执行路径。

基线。 在这个RQ中，我们主要将执行感知的TRACED与UnixCoder [17]进行比较。现在我们解释选择这个基线的原因。首先，TRACED是用预训练的UnixCoder权重初始化的，因此将TRACED与UnixCoder的性能进行比较是对我们提出的预训练影响的直接评估。其次，UnixCoder在许多任务中报告了最先进的性能，包括克隆检测、代码搜索和摘要、代码生成和补全，显著优于其他预训练代码模型，如CodeBERT [16]和GraphCodeBERT [18]。第三，它最多消耗1,024个标记（Token），而大多数预训练代码模型 [1, 6, 16, 18, 49]最多只消耗512个标记（Token）。通过消耗更长的序列，UnixCoder能够处理更长的程序，并在许多情况下进行完整的预测而不截断代码。由于TRACED也被设计为消耗1,024个标记（Token），因此在这个任务中将其与最大长度为512的基线进行比较是不公平的，因为基线必然会考虑更少的分支进行预测。

结果. 比较结果如表 3所示， 行-1 *vs.*行-2. TRACED在执行覆盖的静态估计和变量的动态价值（dynamic values）方面显著优于UnixCoder，尤其是在评估粒度较粗时，即完整执行路径（表 3中的Full Path列）和完整执行的运行时价值（表 3中的Full Exec列）。TRACED正确预测了71.6%的保留样本的完整执行路径，并准确预测了49.2%执行的所有变量值，显示出执行感知的预训练相较于UnixCoder的性能分别提高了12.4%和25.2%。

案例研究与定性示例. 我们在图 5和6中展示了两个定性示例，以具体比较TRACED与UnixCoder在执行覆盖和运行时价值预测方面的表现。两个样本从人类的角度来看都有简单的执行逻辑，但静态预训练的UnixCoder仍然无法正确估计它们。 图 5说明了UnixCoder对触发不同执行覆盖的不同输入不敏感，而TRACED能够确定不同值之间的数值关系。 图 6展示了TRACED在揭示异常程序行为方面的能力。

表 3: 静态执行估计的性能 (static execution estimation).

Model	Coverage					Runtime Value	
	Full Path	Branch				Full Exec	Var
	Acc	Acc	Prec	Rec	F1	Acc	Acc
UnixCoder	63.7	79.7	81.7	85.4	83.5	39.3	87.8
TRACED	**71.6**	**83.1**	84.6	**88.1**	**86.3**	**49.2**	**89.2**
-w/o MLM	70.4	82.6	**85.3**	86.0	85.6	49.0	**89.2**
-w/o PSP	69.0	81.4	83.0	86.9	84.9	44.0	87.4
-w/o VCP	66.1	80.3	82.4	85.6	84.0	46.7	89.0
-MLM-only	65.6	81.0	83.1	86.0	84.6	43.0	87.5

```
//Input: 19 100
#include <stdio.h>
int main(){
    int A, N, T, B;
    scanf("%d %d", &N, &A);
    T = N * N;
    B = T - A;
    if (A > 0) {printf("%d", B);} // Branch-1
    else {printf("%d", T);} // Branch-2
    return 0;
}
```

UnixCoder Predictions (Wrong)

Branch-1: Not executed

Branch-2: Not executed

TRACED Predictions (Correct)

Branch-1: Executed

Branch-2: Not Executed

图 5: 执行覆盖预测的定性示例。源代码与图 1相同，但输入触发了不同的执行路径。 **TRACED**正确地翻转了预测，而 **UnixCoder** 保持相同的预测。

```
//Input: 4 4320 4320 4320
#include <stdio.h>
int main (void) {
    int n, a, max = 0, sum = 0, i;
    for (i = 0; i < n; i++){ // Quantized value of n?
        scanf("\%d", &a);
        if (a > max) max = a;
        sum += a;
    }
    printf("\%d\\n" , sum - max / 2);
    return 0;
}
```

UnixCoder Prediction (Wrong)

n: Zero

TRACED Prediction (Correct)

n: Negative Large

图 6: 运行时价值（value）预测的定性示例。样本包含一种类型为CWE-457的漏洞“使用未初始化变量（Use of Uninitialized Variable）”。未初始化的n被随机赋值为-32767，并在for-loop中使用。TRACED成功地通过将n识别为“负大（Negative Large）”值静态地暴露了这种异常行为，而UnixCoder未能做到。为了更好地说明，其他变量的预测被隐藏。

Result-1: With a similar number of learnable parameters, TRACED outperforms the state-of-the-art pre-trained code model in the static estimation of program execution task. Our proposed pre-training successfully encodes the execution awareness into TRACED's code representations.

6.2 RQ2. Effectiveness of TRACED's Pre-training Objectives

本论文的主要贡献之一是提出多任务预训练以有效学习执行感知代码表征（code representations）。在这个研究问题（RQ）中，我们研究了TRACED的每个目标的有效性和贡献，并因此说明了多个任务的重要性。

为了进行这些实验，我们一次移除一个预训练目标，并在与主模型完全相同的设置下对变体进行预训练。然后，我们在静态执行估计任务上对变体进行微调，并将其性能与主模型进行比较。我们还考虑了一个在我们的数据集上预训练但仅具有MLM目标的变体。结果显示在表3的第3-6行。移除任何目标都会损害TRACED的性能，这表明全面学习静态和动态代码属性比单独学习一个视角更有效。

Result-2: TRACED's multi-task pre-training helps the model comprehensively learn both static and dynamic aspects of source code. Removing any one of TRACED's three pre-training objectives noticeably hurts the model's performance in statically estimating program executions.

6.3 RQ3. Effectiveness of TRACED's Quantized Variable Values

另一个贡献是，程序执行的简化和紧凑的表征（representation）有助于代码模型（model）捕捉动态代码属性。在这个研究问题（RQ）中，我们通过实证研究揭示了量化变量价值（value）的设计特别有助于代码模型的有效学习，因为它减少了变量价值的数据稀疏性，但仍然定义了足够详细的价值类别以区分不同的价值。

为了隔离对TRACED的量化价值的评估，我们通过仅重新创建量化价值标签（即方程 2中的q_{var}），使用不同的价值抽象策略（abstraction）预训练几个变体。例如，当我们预训练一个研究具体价值影响的变体时，我们用具体的追踪值替换TRACED定义的q_{var}。由于不同的策略在不同的粒度上抽象价值，因此在价值预测任务中比较它们并不可行，因为粗粒度策略将受益。因此，我们仅对研究的变体进行微调，以进行执行覆盖预测。

基线（Baseline）。 首先，我们考虑与具体价值进行比较，因为这是表示变量价值的最直观策略。然后，

我们考虑来自LExecutor [45]的两种数据抽象：粗粒度和细粒度。它们与我们的高层直觉相似，将具体价值映射到预定义的区间以减少数据复杂性，从而帮助模型的学习。请注意，LExecutor的数据抽象服务于与TRACED同的目标，并专注于Python，而TRACED则专注于C，因此我们无法直接重用它们的预定义区间。由于它们对数据抽象的定义清晰明了，我们为C语言重新实现了它们的数据抽象，并将其集成到我们的框架中进行比较。我们在相关工作部分（§7）中更详细地讨论和比较LExecutor与TRACED。

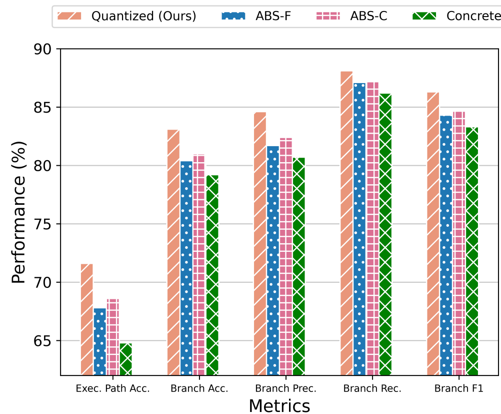

图 7: 比较 TRACED的量化变量价值 (quantized variable values) 设计与其他价值 (value) 抽象策略。

结果. 价值（value）抽象的比较如图 7所示。不出所料，具体价值（value）相比其他数据抽象表现较差，实证揭示了代码模型在适应稀疏和复杂数据分布方面的困难。有趣的是，我们注意到LExecutor的两种抽象表现略逊于TRACED。我们推测LExecutor对条件语句中的数值关系不如TRACED敏感，因为它们并未区分小的、常规的和大的价值（value）。请注意，执行覆盖（execution coverage）并不是LExecutor的主要关注点，因此不需要更细粒度的类别来服务于其目标，而这些类别在TRACED的范围内被实证证明是必要的。

Result-3: TRACED's quantized variable values directly contribute to the effectiveness of its execution-aware pre-training. It reduces the data sparsity of concrete values but defines sufficiently detailed value categories to distinguish dissimilar values for reasoning about execution paths.

6.4 RQ4. TRACED's Performance in Code Understanding Tasks

在这个研究问题（RQ）中，我们研究TRACED在两个代码理解任务上的表现：语义克隆检索和函数级漏洞检测。请注意，这些任务的样本并未与可执行输入配对，因此模型（model）需要推理一般代码语义以进行预测。

基线。 我们考虑五个与TRACED具有相似参数规模的预训练代码模型。CodeBERT [16]通过掩码语言模型（MLM）和替换标记检测（RTD）任务预训练了一个RoBERTa模型。GraphCodeBERT [18]以CodeBERT为基础进行初始化，并继续使用增强的数据流图进行预训练，以学习静态数据依赖关系。PLBART [1]和CodeT5 [49]都应用了seq2seq神经架构，其中PLBART调整了BART [29]模型以学习代码翻译和摘要，而CodeT5调整了 [42]以预测缺失的代码标记（token）并定位标识符。我们还再次考虑UnixCoder作为基线。

结果. 我们在表 4中展示了结果。尽管这些基准中的样本没有可执行的输入，TRACED仍然明显优于静态预训练模型。我们推测原因在于TRACED能够在没有特定输入的情况下估计一般的执行行为，而这两个代码理解任务的程序语义可以通过这种一般性的理解得到更好的捕捉。具体而言，克隆检索要求模型识别代码的行为相似性，因为语义克隆在代码文本和语法上大多存在差异。此外，具有潜在异常的脆弱代码在某些情况下可以被TRACED直接识别，如图 6所示。

表 4: 克隆检索（Clone Retrieval）与缺陷检测（bug detection）比较。

Task	Clone Retrieval	Vulnerability Detection		
Dataset	**POJ-104**	**RV**	**D2A**	**CXG**
Metric	**MAP@R**	**F1**	**Acc**	**Acc**
CodeBERT	82.7	47.3	59.2	63.4
GraphCodeBERT	86.7	46.6	61.0	62.9
PLBART-base	75.9	46.9	61.7	63.3
CodeT5-base*	65.9	46.5	**62.1**	64.4
UnixCoder	89.5	47.4	61.2	65.3
TRACED	**91.2**	**50.4**	**62.1**	**65.9**

*CodeT5-base has 223M parameters, roughly twice as large as other baselines and TRACED. We report its performance as CodeT5-small has only 60M parameters and performs poorly, and CodeT5 does not provide a ~110M model.

Result-4: TRACED outperforms statically pre-trained models in clone retrieval and vulnerability detection tasks, suggesting TRACED's general estimation of execution helps it capture the code semantics more effectively.

7 RELATED WORK

用于源代码的预训练模型. 研究界对开发用于源代码的预训练 transformer 模型表现出越来越大的兴趣。这些模型可以大致分为三种主要架构：仅编码器 [5, 6, 10, 16, 18, 24, 48]、仅解码器 [2, 13, 50]和编码器-解码器 [1, 7, 15, 17, 35]。仅编码器模型主要采用 MLM 目标和序列理解任务（例如预测下一个语句 [24]和对比语义 [10]）。这种架构在理解静态代码特征方面表现出色。另一方面，仅解码器模型通常通过从左到右预测代码标记（Token）进行训练。这种架构专注于基于学习到的模式生成代码文本。编码器-解码器模型结合了仅编码器和仅解码器模型的优点，并通过各种任务进行预训练，包括去噪自编码以重构错误排列的标记（Token） [1]、预测代码中缺失的标识符 [49]和从源代码中恢复方法名称 [35]。

这些模型主要关注学习源代码的静态方面，但往往无法捕捉代码执行的动态特性。这一局限性限制了这些模型准确推断运行时行为、调试问题和理解复杂程序状态的能力。

程序执行建模. Pei 等人 [38–40] 提出了系列开创性工作，以学习基于 transformer 的模型执行 二进制程序。他们使用了来自寄存器的具体值，这在他们的范围内是可行的，因为与源代码相比，二进制程序具有更小的可能值和效果空间。另一方面，我们的工作专注于通过模仿开发者的代码实践在源代码级别编码执行。源代码中的变量具有比机器寄存器更复杂的数据和价值类型。我们引入量化值以降低数据复杂性和稀疏性。

一些工作 [3, 4, 36, 44, 51] 尝试将程序执行学习作为直接目标。Souza 和 Pradel [45] 也提出了 LExecutor 来预测执行过程中的缺失值。虽然它与将具体值映射到离散类别的直觉相似，但 LExecutor 在几个方面与 TRACED有所不同。首先，LExecutor 仅专注于预测值，而 TRACED提出了一个通用的预训练策略，以将全面的执行意识编码到代码表征中，不仅包括值，还包括执行覆盖。此外，为了生成更高质量的代码表征，TRACED联合学习代码文本和动态执行，而不是仅限于单一视角。由于目标和设计的不同，我们在 RQ3 (§6.3) 中实证说明 LExecutor 的价值抽象与我们的范围并不完全一致。

Nie 等人 [34] 在不执行代码的情况下注释程序，提供有关程序可能执行的信息，但仅提供静态可用的信息。相反，一些工作 [19, 37, 46, 47] 需要动态跟踪作为输入。我们展示了 TRACED的预训练能够将执行意识编码到代码表征中，并仅用静态信息估计动态语义。

8 THREATS TO VALIDITY

内部有效性. 首先，由于数据结构、价值 (value) 范围和/或内存分配的复杂性，目前量化价值 (value) 的设计并未覆盖程序中的所有变量。其次，目前我们仅通过提供有效且可执行的输入来跟踪程序，这些输入不会终止程序或抛出错误。这可能使模型 (model) 在捕捉程序终止和抛出错误行为方面的能力降低。

外部有效性. 目前，TRACED仅支持 C 编程语言。这个限制是由于依赖于用于记录执行历史的跟踪器（tracer）的能力，而这些能力可能在其他编程语言中并不容易获得或同样有效。为了扩展 TRACED的适用性，有必要确保所使用的跟踪器能够在不同语言中准确且一致地捕获所需的信息。将 TRACED适配到多种语言将需要开发或调整能够有效处理每种语言复杂性的跟踪器，并产生可比较的结果，从而实现对更广泛编程语言中代码行为的一致分析。

9 CONCLUSION

在本文中，我们提出了TRACED，一个执行感知的预训练模型，它联合学习静态和动态代码属性，以解决现有静态预训练代码模型的局限性。评估实证表明，TRACED在静态估计代码执行方面比静态预训练模型更有效。TRACED还成功地将执行感知转移到代码理解任务中。

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[3] David Bieber, Rishab Goel, Dan Zheng, Hugo Larochelle, and Daniel Tarlow. 2022. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions. https://openreview.net/forum?id=SIcz2sObJ-5

[4] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 8626–8637. https://papers.nips.cc/paper/2020/hash/62326dc7c4f7b849d6f013ba46489d6c-Abstract.html

[5] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *SIGIR '21* (Virtual Event, Canada). 511–521. https://doi.org/10.1145/3404835.3462840

[6] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. arXiv:2006.12641 [cs.CL]

[7] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by" Naturalizing" source code. In *2022 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep Learning based Vulnerability Detection: Are We There Yet. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3087402

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 6300–6312. https://doi.org/10.18653/v1/2022.acl-long.436

[11] Yangruibo Ding, Baishakhi Ray, Devanbu Premkumar, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. https://doi.org/10.1145/3324884.3416587

[12] Adam Paszke *et al.*. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*.

[13] Mark Chen *et al.*. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[14] Thomas Wolf *et al.*. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[15] Yujia Li *et al.*. 2022. Competition-Level Code Generation with AlphaCode. *ArXiv* abs/2203.07814 (2022).

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. https://doi.org/10.48550/ARXIV.2203.03850

[18] 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, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning*

Representations. https://openreview.net/forum?id=jLoC4ez43PZ

[19] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3236024.3236085

[20] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[21] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs.SE]

[22] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. arXiv:2302.01857 [cs.SE]

[23] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[24] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML 2020*.

[25] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1073–1085.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[27] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[28] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. 201–213. https://doi.org/10.1145/2991079.2991102

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[33] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[34] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. arXiv. https://doi.org/10.48550/arXiv.2302.10166 arXiv:2302.10166 [cs].

[35] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. *CoRR* abs/2201.01549 (2022). arXiv:2201.01549 https://arxiv.org/abs/2201.01549

[36] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. https://doi.org/10.48550/arXiv.2112.00114

[37] Jibesh Patra and Michael Pradel. 2022. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1469–1481. https://doi.org/10.1145/3510003.3510144

[38] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 690–702. https://doi.org/10.1145/3468264.3468607

[39] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 747–759. https://doi.org/10.1145/3540250.3549147

[40] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *CoRR* abs/2012.08680 (2020). arXiv:2012.08680 https://arxiv.org/abs/2012.08680

[41] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[43] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[44] Scott Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. https://doi.org/10.48550/arXiv.1511.06279 arXiv:1511.06279 [cs].

[45] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. https://doi.org/10.48550/arXiv.2302.02343 arXiv:2302.02343 [cs].

[46] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. https://openreview.net/forum?id=BJuWrGW0Z

[47] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 121–134. https://doi.org/10.1145/3385412.3385999

[48] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. https://doi.org/10.48550/ARXIV.2108.04556

[49] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[50] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv preprint arXiv:2202.13169* (2022).

[51] Wojciech Zaremba and Ilya Sutskever. 2015. Learning to Execute. https://doi.org/10.48550/arXiv.1410.4615

[52] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[53] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[54] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*. 10197–10207.