理解代码风格：高效的基于 CNN 的编译器优化识别系统

Shouguo Yang*†、Zhiqiang Shi*†、Guodong Zhang*†、
Mingxuan Li*†、Yuan Ma*†、Limin Sun*†
*中国科学院信息工程研究所，中国北京
†中国科学院大学网络空间安全学院，北京，中国{yangshouguo, shizhiqiang,
zhangguodong,limingxuan,mayuan2, sunlimin}@iie.ac.cn

arXiv:2302.04666v1 [cs.PL] 18 Jan 2023

摘要—编译器优化级别识别可用于漏洞发现和二进制分析。由于存在许多不同的编译优化选项，二进制文件内容的差异十分复杂。编译器优化算法有数千种，且处理器架构各不相同，因此，手动分析二进制文件并通过规则识别其编译器优化级别十分困难。本文首先提出了一种基于 CNN 的编译器优化级别识别模型：BinEye。该系统提取语义和结构差异，并自动识别编译器优化级别。该模型专为二进制文件处理而设计，且易于理解。我们构建了一个包含 80028 个二进制文件的数据集，用于模型训练和测试。所提出的模型准确率超过 97%。与此同时，BinEye 是一个完全基于 CNN 的系统，前向计算速度更快，至少比普通的基于 RNN 的模型快 8 倍。通过分析模型输出，我们成功发现了不同编译器优化级别所导致的汇编代码差异。这意味着我们提出的模型具有可解释性。基于该模型，我们提出了一种分析不同编译器优化级别所导致代码差异的方法，对于分析闭源编译器和进行二进制安全分析具有重要指导意义。

索引词—**编译优化、软件安全、卷积神经网络、二进制分析、位置嵌入、模型可解释性**

I. 引言

随着物联网设备的规模不断扩大，在物联网设备上运行的软件系统安全性变得越来越重要。物联网软件安全日益受到不同编译器优化级别的影响。采用不同的编译优化级别时，相同的源代码可以生成不同的二进制代码。二进制代码在函数控制流、数据流、函数内联方式等方面可能存在差异。这些差异很容易导致安全风险。根据以往的研究，编译器优化级别识别可用于漏洞发现 [1] 和二进制相似性检测 [2]等领域。不同的编译器优化级别会对软件安全产生重大影响 [3]。编译器优化级别是软件安全分析的重要参考。例如，编译器优化算法“死存储消除”会导致一个非常著名的安全问题 CWE-14 [4]。“未定义行为”是指编译器可以自由决定如何处理 C/C++ 中的代码优化，在某些情况下也会造成安全漏洞，例如 CVE-2016-9843 和 CVE-2016-9840。

如今常见的编译器（如 GCC）内置了数千种编译器优化算法，每种算法都对应一个编译时选项。为便于使用，GCC 等编译器将众多编译器优化级别整合为五个主要级别：-O0、-O1、-O2、-O3 和 -Os。编译器生成的最小优化相关单元是目标文件。采用不同编译优化级别编译的目标文件可能会链接为同一个可执行二进制文件，因此我们无法直接识别该可执行二进制文件所采用的编译器优化；所以，我们的系统将编译器生成的目标文件作为目标文件。然而，各种编译器优化选项会使目标文件中的二进制代码产生细微差异，这些差异由人工观察十分耗时，且需要具备丰富的汇编语言经验。因此，我们提出了一种端到端深度学习模型，用于识别目标文件所采用的不同编译器优化级别。为了让模型更易于理解，我们成功构建了一个无需编译器知识、也无需对二进制文件进行分析的模型，这意味着它能够轻松处理由编译器优化引发的软件安全问题。

鉴于 Facebook 在语言处理领域的机器翻译应用中成功使用卷积神经网络，以及卷积神经网络在恶意软件检测中的应用，卷积神经网络在语言处理和二进制安全分析中的表现同样出色。此外，与循环神经网络相比，卷积神经网络可以充分利用图形处理器进行并行计算，从而更高效地完成参数梯度计算和训练。因此，我们的模型完全基于卷积神经网络技术，可以直接从可执行二进制文件的原始数据中识别编译器优化级别。根据先前的研究，由编译优化级别 -O2 或 -O3 编译得到的二进制文件差异并不大。由于编译器在编译大多数源代码时，-O2 和 -O3 这两个编译优化级别几乎相同，我们将任务定义为四分类：从二进制文件中识别 -O0、-O1、-O2/-O3 和 -Os 这四种不同的编译器优化级别。我们针对四分类任务对模型进行了大量实验。实验结果表明，我们的模型在四分类任务中的准确率可达 97%。与

Chen 提出的面向 X64 平台二进制文件的模型相比，我们的模型面向 ARM 平台下的二进制文件。由于模型直接将原始二进制字节作为输入，因此不需要任何先验知识或复杂的特征工程。模型的训练和收敛速度更快，规模更小，前向计算速度也更快。通过模型的输出，我们成功发现了不同编译器优化级别导致的若干汇编代码差异，并总结出一种利用该模型进行代码分析的方法。得益于精简指令集计算机的特性，该模型可以轻松扩展到其他精简指令集计算机平台上的编译器和指令集。

本文的贡献总结如下：

- 我们提出了一种用于识别编译器优化级别的模型：**BinEye** ，该模型不需要任何先验知识或特征工程。目标文件可直接作为输入传入我们的模型。基于卷积神经网络的模型能够更快速地进行并行训练和测试，并执行非常快速的前向计算。
- 我们构建了一个由**80028** 个目标文件组成的数据集，总大小为 691MB。我们使用该模型进行了实验，模型取得了较高的识别准确率。
- 借助模型的输出，我们提出了一种能够快速、清晰地分析不同编译器优化级别所导致差异的方法。

II. 相关工作

此前，已有大量研究从编译器优化级别的角度探讨软件安全问题。D'Silva 等人 [3] 提出了一项广泛的研究计划，旨在识别、理解并缓解不同编译器优化引入的安全错误所造成的影响。Wang 等人 [1]提出了一种新颖的模型，从利用未定义行为的各种编译器优化角度对不稳定代码进行分析。引入不稳定代码的不同编译器优化可能导致安全问题。他们利用该模型发现了 160 个新的漏洞，这些漏洞已得到开发人员的确认和修复。由此可知，不同的编译器优化级别可能导致安全漏洞。

由于现实世界中存在多种编程语言和 CPU 架构，使用基于规则的方法处理安全问题会变得更加复杂，同时也需要更强的专业知识和能力，并且需要花费很长时间进行分析和制定规则。根据 Chen 等人的研究， [5]，静态二进制可执行文件分析过程涉及大量手动且重复性的工作，例如调整加载偏移量和处理反汇编错误。由于反汇编存在困难，因此难以获取语义信息。所以，我们尝试直接从二进制代码中捕获语义和结构模式。随着深度学习在图像处理、语言处理等领域的成功应用，深度学习也被应用于安全领域。神经网络可以自动学习和记忆编程语言中的复杂模式。比如，Edward 等人提出的基于卷积神经网络的恶意软件识别模型 [8] 能够更好地识别软件的恶意性。该模型可以自动学习恶意软件中二进制代码的特征，从而区分恶意软件和非恶意软件。Shihab[10]提出了一种高效且可扩展的计算机网络安全技术，该技术使用多层神经网络实现解密方案和公钥生成。Xu 等人 [13] 使用图编码神经网络执行漏洞函数匹配任务，也取得了良好效果。Shin [14] 提出将人工神经网络应用于解决二进制分析中的重要但困难的问题，即函数识别；这是二进制分析技术中至关重要的第一步。结果表明，与其他基于机器学习的方法相比，深度学习技术能够更好地识别二进制文件中的函数。Chen 等人提出的基于循环神经网络的深度学习模型 [11] 也被应用于编译器优化级别识别，但该方法需要较强的专业知识来进行特征工程和复杂的数据预处理，并且其工作仅关注 X86 架构。他们从汇编代码中进行的特征提取需要经过反汇编过程；如前所述，反汇编过程中存在许多反汇编错误。因此，Chen 提出的方法不够准确。

受上述工作的启发，我们将一种新型基于CNN的神经网络应用于编译器优化级别识别任务，并取得了良好效果。与 Chen [11], 此前的工作相比，我们提高了4类分类任务的准确率，并简化了编译器优化级别识别任务：无需提取特征或进行复杂的预处理，直接识别目标文件的编译器优化级别。输入模型的数据无需经过可能引入缺陷的反汇编过程。在模型大小、前向计算速度等方面，我们模型的性能远优于使用循环神经网络的模型。

III. 详细设计

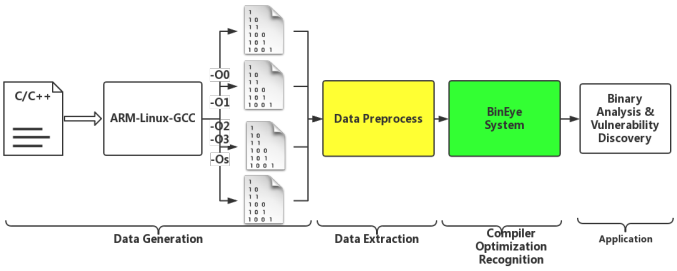

图1. 系统架构。

整体系统架构如图 1 所示。第一部分是数据生成部分。在这一部分中，我们使用交叉编译工具链 arm-linux-gnueabi-gcc 编译源代码，分别采用五种不同的编译器优化级别。编译后，我们得到四种不同类型的目标文件：-O0、-O1、-O2/-O3、-Os。随后，我们从目标

文件中提取代码段数据，作为 **BinEye** 的输入。代码段二进制数据被输入模型，模型执行编译器优化级别识别任务。借助模型的中间输出，我们可以分析汇编代码，相关内容已在第五节讨论。获取编译器优化级别信息后，我们还可以执行其他任务，例如二进制分析和漏洞发现；本文不讨论这些内容。

表 I 数据集详情

编译器优化级别	目标文件	总大小（MB）
-O0	19076	209
-O1	17136	131
-O2	13768	112
-O3	13992	128
-Os	16056	112
总计	**80028**	**691**

A. 数据生成与提取

我们使用 Buildroot [16] 交叉编译工具链，以五种不同的编译器优化级别（-O0、-O1、-O2、-O3、-Os）编译 463 个开源组件，共生成 80028 个目标文件（详情见表 I）。在 Buildroot 中进行配置的命令是**"make menuconfig"**，下载组件源代码的命令是"make source"。为了生成采用不同编译器优化级别的目标文件，我们在组件源代码包中找到与编译优化级别配置（"-O0、-O1、-O2、-O3、-Os"）相关联的字符串，用 shell 命令将其替换为目标优化级别。然后返回 Buildroot 根目录，执行"make"命令，按照指定的优化级别编译源代码。由于二进制文件由目标文件链接而成，且目标文件是最小的编译单元，因此本研究在构建数据集时仅考虑目标文件。需要注意的是，即使在前一步指定了编译优化级别，也无法保证所有源代码都能成功配置（因为部分源代码的 makefile 中未指定编译优化级别）。因此，我们按照宁缺毋滥的原则，从已编译的目标文件数据集中移除所有 MD5 签名一致的文件，以确保所选二进制文件具有正确的编译优化级别。

根据 ELF 文件的结构，许多内容区段不受编译器优化级别的影响。因此，我们只关注代码段的内容。使用"readelf"工具可以轻松获取代码段的内容。由于 ARM 架构属于精简指令集计算机，使用"readelf"工具获取按 4 字节对齐的二进制数据后，无需任何处理，便可将其直接输入**BinEye**模型。

B. 模型结构

该模型主要由三个部分组成：数据表示部分、卷积与池化部分以及输出部分。详细设计如图 2 所示。由于卷积与池化部分是卷积神经网络中较常用的模块，因此我们仅展示数据嵌入表示部分 III-B1 和 III-B2，以及卷积神经网络中的特殊结构 III-B3。以下各节将介绍这些部分。

1) 词嵌入：我们从二进制目标文件的代码段中提取 1024×4 字节，作为神经网络的原始输入。由于 ARM 架构属于精简指令集计算机，每条指令固定占用 4 字节，因此提取 1024×4 字节相当于提取 1024 条指令作为神经网络输入。这种数据提取方法也适用于所有精简指令集计算机架构，例如 MIPS、PowerPC 等。神经网络的第一层是词嵌入层，用于对输入的二进制指令字节进行编码。我们使用嵌入层将每个字节表示为一个四维向量，以便发现并表示特定指令之间的语义相似性。

为了更好地捕捉精简指令集计算机的特征，我们将输入 $x = (x_1, \ldots, x_i, \ldots, x_m)$ 嵌入到分布式空间中，表示为 $w = (w_1, \ldots, w_j \ldots, w_m)$，其中 $w_j \in R^f$ 是嵌入矩阵 $D \in R^{f \times V}$ 中的一列，V 是输入数据可能取值的数量。由于每个字节最多有 2^8 个取值，因此 $V = 2^8$、f 表示输入中的一个字节，并将其表示为一个 f 维向量。

从以往的研究 [15], 中可以得知，与独热数据表示相比，词嵌入具有降低数据维度的优势。另一方面，词嵌入能够表达指令之间的语义关系。

2）位置嵌入：为了让模型感知不同指令之间的顺序关系，我们将 m 条指令的二进制数据的绝对位置（1，...，m）嵌入为 $p = (p_1, \ldots, p_m)$，其中 $p_j \in R^g$ 是位置嵌入矩阵 $F \in R^{g \times S}$ 中的一列（S 表示输入指令的数量），这意味着将指令对应的固定位置嵌入为一个 g 维向量，从而使模型能够结合位置嵌入和词嵌入。由于目标文件尚未经过编译器链接，代码段数据的偏移地址从 0 开始，因此使用绝对地址和相对地址是等效的。值得注意的是，位置嵌入矩阵不同于上一节提到的嵌入矩阵，它是一种硬嵌入表示。该矩阵是固定的，即特定位置对应的嵌入表示是固定的，且不可训练。上一节提到的词嵌入矩阵则是可训练的。对于给定的位置 n，若嵌入表示的维度指定为 g，则位置 n 被嵌入为向量 p_n，如公式 1 所示。

$$p_n[k] = (1 - \frac{n}{S+1}) - (\frac{k}{g+1}) \times (1 - 2 \times \frac{n}{(S+1)}) \quad (1)$$

其中 $0 <= k < g$ 表示位置编码的 k_{th} 维。S 表示本文中的指令总数 $S = 1024$ 。

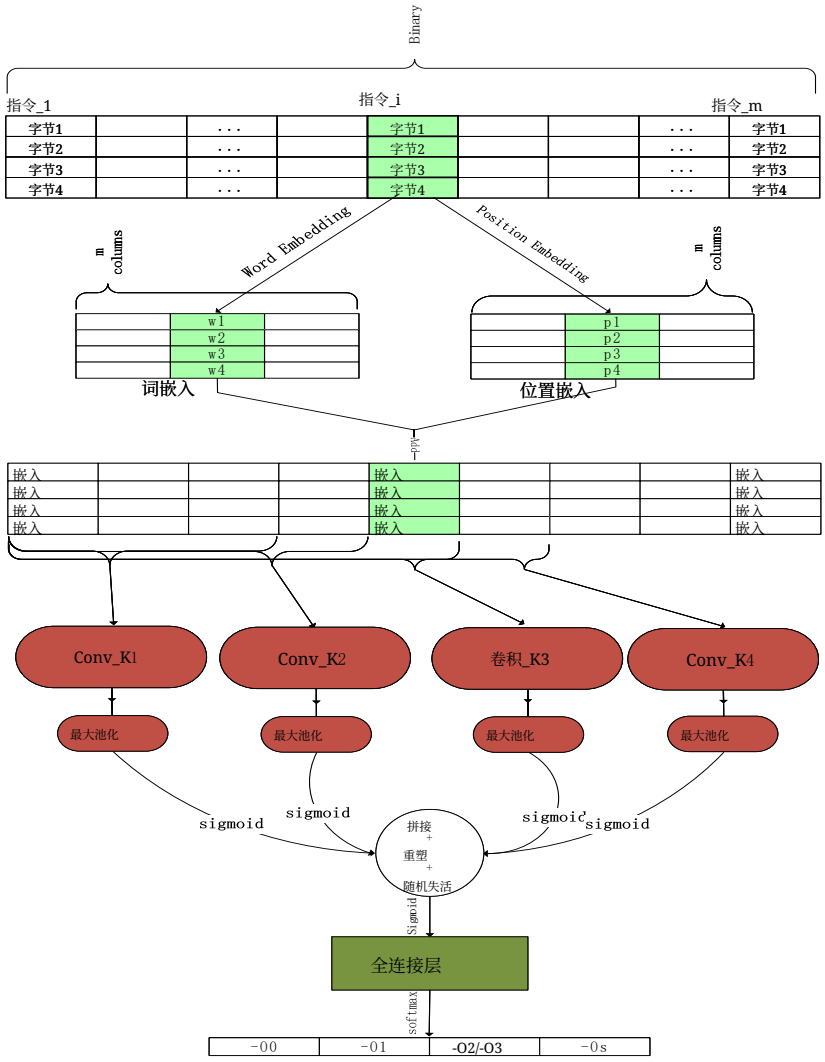

图 2. BinEye模型结构。

词嵌入和位置嵌入结合起来，将输入数据 $e = (w_1 + p_1, \ldots\ldots, w_m + p_m)$ 表示为下一层的输入数据。位置嵌入表示在模型中非常有用，它使模型能够捕获指令的位置信息，从而获得更高的准确率。

3) 卷积与池化：我们将所有卷积核的宽度设置为与前一个嵌入层的输出宽度相同，并将长度设置为$Conv\ K_1$、$Conv\ K_2$、$Conv\ K_3$、{v12>（这意味着使用四种不同类型的卷积核），且卷积后不进行填充。经过卷积操作后，我们可以得到一个 $m - k + 1$ 维向量，而不是矩阵（k 表示卷积核长度，m 表示序列长度）。由于对 $m * n$ 数据进行嵌入后，我们得到 $m * n * h$ （h 表示每个字节的分布式表示，即 h 维向量）（n 表示输入矩阵的宽度）。因此，卷积后我们得到一个大小为（$m - k + 1$）的矩阵 $\times\ 1 \times 1 \times num\ filters$（$num\ filters$ 表示每个类别的卷积核数量），然后使用最大池化提取每个通道的最大值，得到一个大小为$1 \times 1 \times num\ filters$的矩阵。接着，将所有卷积操作的输出拼接并重塑，得到一个$num\ filters \times filter\ sizes$（$filter\ sizes$ 表示卷积核的类型数量）向量，作为下一层全连接层的输入。

4) 输出：模型的最后一部分是全连接层，用于增强模型的表达能力。全连接层的输出使用 softmax函数获取每个类别的分类概率，如下方公式 2 所定义

$$S_i = \frac{e^{V_i}}{\sum_i^C e^{V_i}} \tag{2}$$

其中，V_i 是全连接层输出向量中的第 i 个值，C 是类别总数。由于我们总共有 -O0、-O1、-O2/-O3、-Os 四个类别，因此 C 的值为 4。模型的输出是一个四维向量，

表 II 实验结果

模型结构	含位置嵌入的精确率					模型参数	计算速度[1]
	-O0	*-O1*	*-O2/-O3*	*-Os*	准确率		
BinEye	0.9800	***0.9690***	***0.9807***	***0.9606***	***0.9724***	306439	***65.29***
LSTM	0.9805	0.9577	0.9525	0.9367	0.9408	101700	8.79
GRU	***0.9839***	0.9464	0.9391	0.9587	0.9507	***95492***	7.37
卷积神经网络+长短期记忆网络	0.9637	0.7699	0.7457	0.4757	0.7784	4349060	5.53
卷积神经网络+门控循环单元	0.9794	0.9464	0.8588	0.8966	0.9391	3298436	2.43

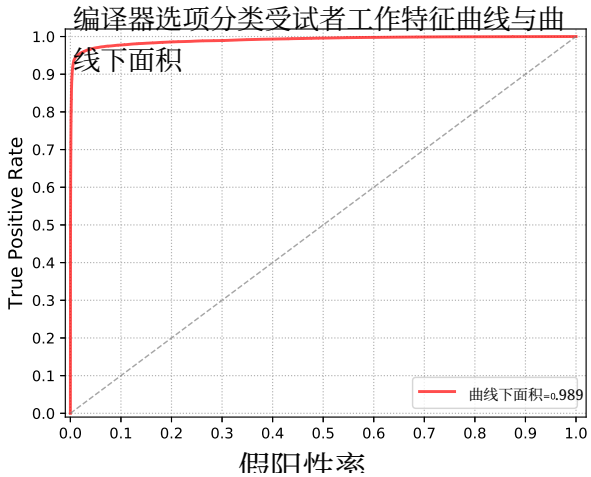

图 3. 编译器优化级别分类的受试者工作特征曲线。

其中第 i 个维度表示第 i 个类别的概率。我们使用 arg max函数，取概率最大的维度对应的下标作为预测类别。

IV. 实验

本部分主要介绍模型的训练环境、训练与测试结果，以及对实验结果的分析。

A. 模型性能

我们的模型 **BinEye** 完全基于卷积神经网络，并配备了位置嵌入。由于卷积神经网络具有权重共享的特点，我们的神经网络模型参数总数为 30,639，对 457 个样本进行前向计算仅需 7 秒，远快于循环神经网络模型。我们的模型准确率达到 97.24%，优于其他循环神经网络模型。

有时，程序语句的顺序对控制流至关重要，这要求我们的模型能够捕获序列的顺序。本文通过引入位置嵌入，为每个语句位置编号，并为每个编号分配一个向量。通过结合位置向量和指令嵌入向量，为每条指令引入特定的位置信息。我们针对能够识别不同编译器优化的模型进行了实验。为了让模型更好地捕获不同长度指令组合的语义差异，我们尝试了不同长度的卷积核及其不同组合，最终得到了性能较好的卷积核组合。实验模型使用了四种不同类型的卷积核。每个卷积核的尺寸仅长度不同，分别为 2、3、4 和 5。我们的模型在训练数据集上训练了数小时，最终在测试数据集上取得了一定的准确率。根据表 II 可知，对于每个编译器优化级别 -O0、-O1、-Os、-O2/-O3，使用尺寸为 2、3、4、5 的卷积核的模型 **BinEye** 识别准确率均超过 95%，四种编译器优化级别的准确率达到 97.24%。

B. 与循环神经网络模型对比

为了进行比较，我们构建了多个循环神经网络。我们也在循环神经网络中使用词嵌入表示技术。在词嵌入层之后，我们构建了循环神经网络层，采用以下一种结构：**长短期记忆网络、门控循环单元、卷积神经网络+长短期记忆网络、卷积神经网络+门控循环单元**。随后，我们构建了一个全连接层作为输出层。我们使用相同的数据集为所构建的循环神经网络模型提供输入。该模型在与**BinEye** 相同的机器上进行训练和测试，耗时较长。随后，我们获得了四种循环神经网络的测试结果，以便与基于卷积神经网络的模型进行比较；同时记录模型大小（参数数量）和前向计算速度（每秒计算次数）。比较结果如表 II 所示。我们可以看出，我们的模型在准确率和计算速度方面都远胜于其他模型。

C. 在目标文件上的应用

由于在模型训练和测试期间，我们只关注目标文件代码段中的一个 4K 大小的代码节，因此最终准确率 97.24 % 可能无法充分体现 BinEye 识别目标文件优化级别的能力。因此，我们重新加载训练好的模型，并测试其在我们构建的数据集上识别目标文件优化级别的能力，最终在整个数据集上取得了 97.49% 的准确率。对于单个目标文件，我们将其拆分为若干个 4K 大小的二进制代码块。我们使用 BinEye 识别每个二进制代码块的优化级别，然后将多个代码块的识别结果取众数，作为

[1]计算速度以每秒计算次数表示

目标文件的最终结果。我们统计了不同阈值下的真正例率和假正例率，并得到了图3所示的受试者工作特征曲线。

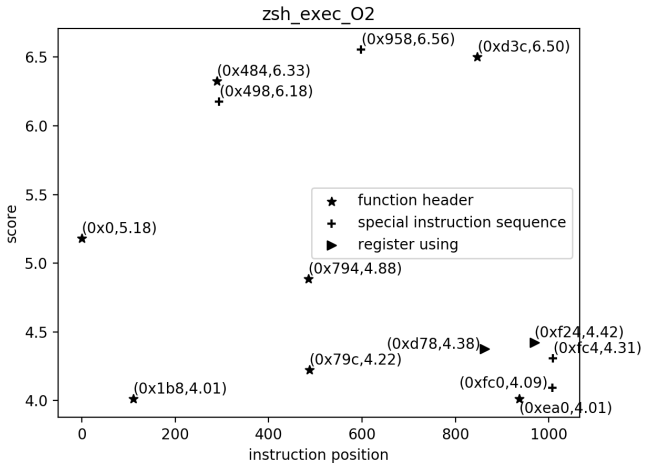

图 4. 卷积与池化输出

V. 编译优化分析方法

我们分析了卷积与池化部分的输出，发现只有少数输出为非零值，这意味着并非所有指令都对编译优化级别识别任务有贡献。我们开始关注与非零输出相对应的指令序列。在分析了数十个目标文件样本后，我们选取了两个典型文件 exec O0.o 和 exec O2.o 进行分析，它们由相同的源代码通过不同的优化级别编译而成。这两个文件可以从 GitHub [21] 获取。

我们从 exec O2.o 的卷积与池化运算输出中选取了几个较大的输出值，因为较大的值对最终分类的影响更大。图 4 展示了我们得到的输出。每个点的注释表示某个地址处相应位置代码块的得分。x 轴表示模型输入的 1024 条指令中的指令位置。通过分析得分较高的指令，并将其与 exec O0.o 文件中相应位置的指令序列进行比较，我们提出了一种分析同源二进制代码在不同编译优化级别下差异的方法。具体步骤如下。

1) 提取代码段的二进制数据，并将二进制数据划分为 4K 大小的块。2) 将代码段数据输入 BinEye 模型，进行前向计算，获得卷积与池化后的结果。3) 获取输出值大于 0 的对应指令地址，然后向后提取连续的 4 条指令。4) 总结连续指令对应的模式，并与同一源代码在不同编译优化级别下生成的其他代码进行比较。

5) 将模式差异与其他编译器优化级别下的代码进行比较，并总结不同级别编译优化之间的差异。

通过我们的方法，我们发现这两种不同的编译优化级别之间存在三类差异，这些差异在图 4 中分别以不同形状标记。对于这三类不同的差异：函数头、特殊指令序列和寄存器使用，我们将在
以下各节。

-O2	-O0
.text:00000484 dword DCD 0x201	.text:000112D4 STMFD SP!, {R11,LR}
.text:00000488 LDR R3, =选项	.text:000112D8 ADD R11, SP, #4
.text:0000048C STMFD SP!, {R4,R5,LR}	.text:000112DC SUB SP, SP, #0x80
.text:00000490 LDRB R1, [R3,#0x1E]	.text:000112E0 STR R0, [R11,#变量_80]

图5。用于函数头分析的 -O2 中 0x484 处的指令及其对应的 -O0 指令

A. 函数头差异

根据图4，我们首先分析了与**(0x484, 6.33)**对应的指令序列。此外，我们还分析并比较了 -O0 目标文件 exec O0.o 中对应函数头的代码。我们发现，-O0 与 -O2 编译优化级别之间存在明显差异。我们发现，使用 -O0 级别编译的代码总是在函数头处将 R11 寄存器保存到栈中，并在函数退出时恢复它。而在使用 -O2 级别编译的代码中，函数头处的指令 STMFD 会使用一个或多个 R4-R10寄存器。差异示例如图5所示（-O2 使用指令 *STMFDSP!,{R4,R5,LR}*，而 -O0 使用 *STMFDSP!,{R11,LR}*）。通过大量分析这两个优化级别下目标文件的函数头差异（例如 (0xd3c,6.50)、(0x0,5.18)），我们认为这可以作为区分不同编译器优化级别的良好依据。BinEye 能够识别函数头中寄存器使用的细微差别，令人印象深刻。

-O2	-O0
.text:00000958 CMP R8,#0	.text:000028A4 加载寄存器 R3, [R11, #var _28]
.text:0000095C STR R2, [R3]	.text:000028A8 AND R3, R3, #0x20
.text:00000960 BNE loc ACC_	.text:000028AC CMP R3, #0
.text:00000964 CMP R5, #0	.text:000028B0 BEQ loc_28DC

图6. 用于特殊指令序列分析的 -O2 中 0x958 处的指令及其对应的 -O0 指令

B. 特殊指令序列

我们根据图 4 中所示的 **(0x958, 6.56)** 长度为 4 的指令序列，以及 -O0 目标文件中的对应指令序列进行分析。随后，我们比较文件中其他位置的相似指令组合（例如 (0x498,6.18)）。我们发现了一种非常有趣的指令组合。在 -O0 优化级别下，指令组合“CMP BEQ”或“CMP CMPEQ BEQ”经常同时出现，但在 -O2 优化级别下，指令组合“CMP BNE”或

”CMP CMPEQ BNE”经常一起出现。代码示例如图6所示。

-O2	-O0
.text:00000F24 LDR R3, =文件描述符表 .text:00000F28 将 R2, #2 .text:00000F2C 加载 R3, [R3] .text:00000F34 STRB R2, [R3,R4]	.text:00006B54 LDR R3, =文件描述符表 .text:00006B58 LDR R2, [R3] .text:00006B5C LDR R3, [R11,#var_3C] .text:00006B60 ADD R3, R2, R3 .text:00006B64 MOV R2, #2 .text:00006B68 STRB R2, [R3]

图7. 用于寄存器使用分析的 -O2 在 0xF24 处的指令及其对应的 -O0 指令

C. 寄存器使用

寄存器的使用是 ARM 架构过程调用标准 [20] 中的一项重要规则。我们根据图 4 中所示的 **(0xf24, 4.42)** 长度为 4 的指令序列，以及 -O0 目标文件中的对应指令序列进行分析。我们发现，优化级别为 -O2 的代码会尽可能使用寄存器来存储局部变量。但在 -O0 优化级别下，代码总是将局部变量压入函数栈，并在需要时加载。如图 7 所示，优化级别为 -O0 的代码使用“LDR R3, [R11,#var 3C]”加载一个局部变量，并将立即数 2 保存到与“parameter+fdtable”对应的内存地址中。而在优化级别为 -O2 的代码中，则使用寄存器 R4 来保存该局部变量。

通过分析 BinEye 卷积与池化部分的输出，并比较优化级别 -O0 和优化级别 -O2 对应部分的指令序列，我们确实发现了不同编译优化级别之间的一些独特模式。也就是说，BinEye 通过持续学习输入的二进制代码，记住了与编译优化相关的一些特殊指令模式，从而能够识别不同目标文件的编译优化级别。

VI. 结论

本文主要提出了一种无需任何先验知识的高效深度学习模型：**BinEye**，用于识别目标文件的编译优化级别。该模型完全基于卷积神经网络，具有参数较少、前向计算速度快、训练速度快等特点。我们首先进行了大量准备工作，指定五种不同的编译优化级别，对数百个开源组件进行编译，并生成大型数据集用于模型训练和测试。根据模型的分类准确率，我们不断调整参数并修改网络结构。最终，模型在分类任务上的准确率约为 97.24%，达到了理想效果。通过分析模型输出，我们发现模型能够记忆不同编译优化级别之间的特殊指令模式。借助 BinEye，我们成功分析并验证了不同编译优化级别下汇编代码的若干差异。

VII. 致谢

本研究获国家自然科学基金（编号：U1636120）、中国科学院战略性先导科技专项（编号：X DC02020100）和国家自然科学基金重点项目（编号：U1766215）资助。

VIII. 参考文献

参考文献

[1] Wang、Xi 等人：《迈向优化安全的系统：分析未定义行为的影响》。第二十四届 ACM 操作系统原理研讨会论文集。ACM，2013 年。[2] David、Yaniv、Nimrod Partush 和 Eran Yahav：《通过重新优化实现二进制文件相似性》。ACM SIGPLAN 通知，第 52 卷，第 6 期。ACM，2017 年。[3] D’ Silva、Vijay、Mathias Payer 和 Dawn Song：《编译器优化中的正确性与安全性鸿沟》。2015 年 IEEE 安全与隐私研讨会（SPW）。IEEE，2015 年。[4] Cwe.mitre.org。（2018 年）。CWE - CWE-14：编译器移除用于清除缓冲区的代码（3.1）。[在线] 可访问：http://cwe.mitre.org/data/definitions/14.html [访问日期：10 月 10 日 2018]。[5] Chen、Jiongyi 等人：《IOTFUZZER：通过基于应用程序的模糊测试发现物联网中的内存损坏》。[6] 《使用 GNU 编译器套件（GCC）：优化选项》 https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html，2018 年。[7] Egele、Manuel 等人：《全面执行：针对程序二进制文件和组件的动态相似性测试》。USENIX，2014 年。[8] Raff、Edward 等人：《通过吃掉整个 exe 来检测恶意软件》。arXiv 预印本 arXiv:1710.09435（2017 年）。[9] Gehring、Jonas 等人：《卷积序列到序列学习》。arXiv 预印本 arXiv:1705.03122（2017 年）。[10] Shihab、Khalil：《用于计算机网络安全的反向传播神经网络》。计算机科学杂志 2.9（2006 年）：710–715。[11] YuChen：《HIMALIA：通过深度学习从二进制文件中恢复编译器优化级别》，2018 年智能系统会议，9 月 6–7 日。[12] Offutt、A. Jefferson 和 W. Michael Craft：《使用编译器优化技术检测等价变异体》。软件测试、验证与可靠性 4.3（1994 年）：131–154。[13] Xu、Xiaojun 等人：《基于神经网络的图嵌入，用于跨平台二进制代码相似性检测》。2017 年 ACM SIGSAC 计算机与通信安全会议论文集。ACM，2017 年。[14] Shin E C R、Song D、Moazzezi R：《使用神经网络识别二进制文件中的函数》[C]//USENIX 安全研讨会。2015 年：611–626。MLA[15] Bengio、Yoshua 等人：《神经概率语言模型》。机器学习研究杂志 3.Feb（2003 年）：1137–1155。[16] Buildroot.org。（2018 年）。Buildroot - 让嵌入式 Linux 更简单。 [在线] 可访问：https://buildroot.org/ [访问日期：10 月 27 日 2018]。[17] Hex-rays.com。（2018 年）。IDA：简介。[在线] 可访问：https://www.hex-rays.com/products/ida/index.shtml [访问日期：9 月 29 日 2018]。[18] Zynamics.com。（2018 年）。zynamics.com - BinDiff。 [在线] 可访问：https://www.zynamics.com/bindiff.html [访问日期：10 月 31 日 2018]。[19] Zsh.org。（2018 年）。Zsh。 [在线] 可访问：http://www.zsh.org/ [访问日期：10 月 31 日 2018]。[20] Infocenter.arm.com。（2018 年）。 [在线] 可访问：http://infocenter.arm.com/help/topic/com.arm.doc.dui0472k/DUI0472K armcc user guide.pdf [访问日期：11 月 11 日 2018]。[21] https://github.co m/yangshouguo/CompilerOptimizationLevelRecognition