面向代码重用检测的程序语义分析模型

郭曦[1]，王盼[2]
（1. 华中农业大学信息学院，湖北 武汉 430070；
2. 湖北工业大学太阳能高效利用及储能运行控制湖北省重点实验室，湖北 武汉 430068）

摘　要：程序相似性分析在代码缺陷检测、产权保护等领域有广泛的用途，但普遍存在计算开销过大等问题，为此提出了一种基于模糊匹配和统计推理的代码相似度分析方法。针对二进制程序，首先对其进行反汇编分析，然后进行函数边界识别操作，从而提取函数的执行边界信息。在此基础上，在基本块的粒度上使用动态规划的分析方法获得基本块之间的相似度结果，并在控制流图的基础上进行邻域搜索，从而将相似性分析从基本块级别扩展至函数级别。最后通过相似度函数的统计分析得出二进制文件的语义相似度。在该过程中对预训练模型进行优化分析，并对参数进行调优，从而可以对跨平台代码进行相似度分析。实验结果表明，相对于目前主流的分析工具，所提方法在分析精度方面较传统的分析工具有较大提高，其分析精度平均提高7.1%。

关键词：程序分析；模糊匹配；统计推理；机器学习
中图分类号：TP311
文献标志码：A
DOI：10.11959/j.issn.1000–436x.2024269

Program semantic analysis model for code reuse detection

GUO Xi[1], WANG Pan[2]
1. College of Informatics, Huazhong Agricultural University, Wuhan 430070, China
2. Hubei Collaborative Innovation Center for High-Efficiency Utilization of Solar Energy, Hubei University of Technology, Wuhan 430068, China

Abstract: Program similarity analysis had a wide range of applications in areas such as code plagiarism and property protection, but it generally suffered from problems such as excessive computational overhead, a code similarity analysis method based on fuzzy matching and statistical inference was proposed. For binary programs, first disassembly analysis was performed and then function boundary recognition operations was performed to extract the execution boundary information of the function. On this basis, dynamic programming analysis methods were used to obtain similarity results between basic blocks at the granularity of the basic blocks, and neighborhood search was performed on the basis of the control flow graph to extend similarity analysis from the basic block level to the function level. Finally, the semantic similarity of binary files was obtained through statistical analysis of similarity functions. During this process, the pre trained model was optimized and analyzed, and the parameters were tuned to enable similarity analysis of cross platform code. The experimental results show that the proposed method has a significant improvement in analysis accuracy compared to traditional analysis tools, with an average increase of 7.1% in analysis accuracy compared to current mainstream analysis tools.

Keywords: program analysis, fuzzy matching, statistical inference, machine learning

收稿日期：2024–05–27；**修回日期**：2024–11–29
通信作者：王盼，wp20210018@hbut.edu.cn
基金项目：国家自然科学基金资助项目（No.61502194）；国家重点研发计划基金资助项目（No.2023YFF1000100）；湖北省教育厅科学技术研究基金资助项目（No.Q20211405）；湖北工业大学博士科研启动基金资助项目（No.XJ2021003601）
Foundation Items：The National Natural Science Foundation of China (No.61502194), The National Key Research and Development Program of China (No.2023YFF1000100), Science and Technology Research Project of Hubei Provincial Department of Education (No.Q20211405), Doctoral Research Initiation Fund Project of Hubei University of Technology (No.XJ2021003601)

0 引言

在通信领域，存在大量的二进制文件以实现数据编码、网络通信等功能，在此过程中，网络通信的安全性极大程度依赖于底层二进制代码的安全性。为了降低系统开发周期和成本，各厂商通常会大量使用现有的代码模块和库文件，故某个代码模块或库文件中潜在的漏洞在该二次开发过程中会逐渐扩散，从而造成更大范围的安全事件。

代码相似性检测在程序安全分析、漏洞挖掘[1]、代码检测[2]等方面有广泛的应用。通过对现有代码的结构和语义特征进行分析，能有效提高对软件新功能识别和缺陷分析的效率，从而加速对可疑目标代码的新功能及组件的识别。例如，已泄露源代码的银行木马程序Citadel与Zeus具有相同的核心模块，而代码相似性检测能识别由恶意代码共享导致的相似功能，是通信和计算机领域的重点研究方向。

为了提升代码的安全性并降低被恶意利用的风险，软件开发人员经常选择将软件以二进制格式发布。这种方法虽然能有效保护代码不被轻易理解或篡改，但同时也因其缺乏类型信息和较低的抽象层次而难以被分析。为了进一步增加代码分析的难度，某些开发者会选择在编译时移除调试信息，使逆向工程变得更加复杂。此外，不同项目可能采用不同的编译器和优化策略，导致即便是语义相同的代码，在不同平台的汇编代码和控制流结构上亦会表现出差异。这些因素显著增加了代码相似性分析的难度和复杂性。

1 研究现状及分析

在二进制程序分析过程中，常见的问题有代码段中函数的识别、代码相似性的比较等。准确识别二进制代码中的函数是代码相似性分析的基础，而现有的研究方法往往跳过该过程直接进行后续的代码相似性分析，导致研究工作缺乏必要的前提条件。目前代表性的研究包括从内联函数中静态地识别函数的起始位置[3]，以及采用启发式的方法来识别函数的边界[4]，BAP和Dyninst等工具通过采用字节签名的方式识别函数的起始位置，但该类方法需要针对每类编译器人工地生成新的签名，缺乏灵活性和扩展性。同时基于深度学习的分析方法，如DeepDi[5]使用R-GCN模型在指令流图上学习指令的嵌入，但是此类方法的性能严重依赖于训练集，且对图形处理单元（GPU, graphics processing unit）等计算资源开销有较大的需求。

相对于这些方法，在函数边界识别方面，本文采用基于带权前缀树的学习函数的分析方法以加快函数边界的识别过程，通过二进制程序片段和函数签名的匹配，识别函数的起始位置。其中权重通过对数据集的传递进行计算，在识别函数起始位置的基础上，通过增量控制流恢复算法提取二进制程序中的函数体，从而实现函数边界的识别过程。

在二进制代码相似性分析方面，目前的研究方法依据代码特征的不同，主要从词法、语法、树和图的粒度展开研究。代表性的工作如下。1) 为了增强二进制分析的准确性，αDiff[6]利用二维卷积神经网络对二进制数据进行嵌入处理。该技术通过分析函数的调用图来构建特征向量，并利用这些特征向量来评估代码之间的相似性，然而混淆可以显著改变代码结构而不改变其功能，从而影响特征向量的准确性和相似性度量的有效性，故这种基于文本比较的方法在处理经过混淆技术的代码时面临较大挑战。2) 采用树或图等基于逻辑的分析方法采用基本块粒度的控制流图来进行相似度比较，如著名的检测工具Genius[7]采用优化的图匹配方法，但是难以对跨架构的代码相似性进行分析，同时此类采用树或图的方法通常只包含控制流信息，难以体现完整的程序语义。

在当前的学术研究中，众多研究者纷纷采用基于Transformer架构的预训练模型，如采用BERT[8]来深化其研究。这些模型包括但不限于OrderMatters[9]、PalmTree[10]和jTrans[11]，它们通过深入分析汇编语言代码中的语义特征，显著提高了代码相似性检测的效率。在这些模型中，jTrans因其卓越的性能脱颖而出，超越了其他常见的检测模型。Asm2vec[12]是另一种创新方法，它将指令和操作码视作基本的语言单元，构建了神经网络来处理这些单元。此外，还有一些研究采用了seq2seq[13]模型和Asteria[14]，后者通过将不同指令集架构的函数映射到一个统一的嵌入空间，实现了跨平台的代码相似性分析。然而，这些方法通常忽略了控制流的细节或者指令之间的相互关系[15]，可能会影响检测的准确性。

在编译过程中，由于编译器设置的差异，生成的二进制文件在语法层面可能会表现出一些变化。

本文采取的方法将相似性分析的焦点放在代码的基本结构块上，评估这些基本结构块之间的相似性得分，并将这种得分关系扩展到整个函数级别的相似性评估中。

针对上述方法存在的不足，本文提出了一种基于模糊匹配和统计推理的二进制程序相似性比较方法，本文的主要贡献如下。

1) 提出了基于模糊匹配的函数相似度识别方法。具体包含指令规范化、指令比较和路径搜索等过程。与现有的方法不同，该方法通过采用带权前缀树的分析方法识别函数的起始位置，并通过控制流增量恢复的方法识别函数的边界。通过该方法可以实现在函数粒度上，将来自不同平台或经过不同编译器优化的函数进行相似度评估，从而获得局部相似性分析的结果，确保在进行后续函数相似性分析之前，能够准确地识别和界定函数的结构和边界。

2) 提出了基于统计推理的二进制语义相似度分析方法。该方法包含函数分解、SubSlide置信度分析和函数语义相似度等过程。该方法主要通过统计推理的方法，计算函数之间的全局相似语义度。该方法不仅考虑了函数的局部特征，还考虑了函数在更广泛上下文中的语义相似度。

3) 开发了对应的实验原型系统，在函数边界识别方面，本文方法能够达到92.87%的精度，同时在函数相似度得分上平均提高了7.1%，实验结果证实了本文方法的有效性。

其整体框架如图1所示，对于由不同的编译器或在不同的编译条件下生成的程序，本文方法能够有效地比较二进制程序之间的相似性，有助于提高代码相似性分析的准确性和效率。

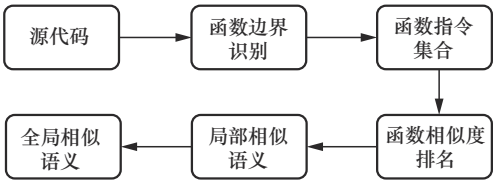


图1　本文方法整体框架

2　函数相似度匹配

函数边界识别是二进制重写和控制流完整性等操作的重要前提。本节通过带权前缀树的方法获取函数起始的签名信息，并将此签名信息与二进制代码进行匹配，从而识别函数的起始位置。同时在带权前缀树中，通过设定指令序列的权重，使用控制流增量恢复的方法识别函数边界。

在识别汇编代码中函数的基础上，本节重点介绍函数相似度的计算方法，包括匹配模型和比较算法。为了比较在不同编译环境及优化选项下所生成的指令，将汇编文件中的指令进行规范化操作以匹配相似指令。在相似度比较过程中，比较的粒度从指令级别通过统计推理的方法上升到函数级别。

2.1　模型描述

对于待分析的二进制文件，首先依据函数边界识别操作获得函数对应的指令集合。然后对汇编代码进行规范化操作，使用优化策略对规范化后的汇编代码进行约简，从而输出待匹配函数的候选集。

该候选函数与目标函数进行匹配操作，生成基本块的映射关系及相似性得分。其中规范化策略的作用对象为基本块及其相似度匹配过程，在匹配过程中通过最长公共子序列（LCS, longest common subsequence）获得目标函数与候选函数的执行路径，得到基本块之间的初始映射关系。并使用带有模糊匹配策略的邻域搜索方法，对基本块间的映射关系进行拓展，获得函数间的相似性得分。

2.2　函数边界识别

经过编译器优化后的函数可能不处于一片连续的地址空间，不同函数之间可能存在共享地址的情况，其原因是编译器在优化过程中，为了对函数进行填充或者对齐而引入额外的指令。如图2中灰色区域所示的指令并不属于任何函数，传统的IDA和BAP等分析工具都使用了类似这样的启发式分析方法从而导致了误报[16]。

```
0x00000c53: push   %rbp
0x00000c54: mov    %rsp, %rbp
0x00000c57: lea    0x3b(%rip), %rdi
0x00000c5e: pop    %rbp
0x00000c5f: jmpq   0x00000c7d <puts$stub>
0x00000c64: nopl   0x0(%rax)
0x00000c6b: nopl   0x0(%rax,%rax,1)
```

图2　不可达函数的汇编代码

本文在函数边界识别方面，将该问题通过机器学习的方法转换为分类问题，即通过标记指令的方法分析函数的起始信息。函数边界识别过程包含训练阶段、分类阶段等。

在训练阶段，对一定数量的候选过程产生二进制代码并确定起始地址，产生函数起始模式从而可

以匹配不同处理器及优化过程。依据起始地址对应的指令生成带权前缀树，通过计算候选过程中每个函数起始指令的识别比率$\frac{|\mathrm{TP}|}{|\mathrm{TP}|+|\mathrm{FP}|}$，获得顶点的权重值。

该过程的输入为候选函数对应的二进制文件T，以及用来设置带权前缀树最大树高的指令序列长度变量l。由于在训练阶段的二进制文件中保留了调试信息，故可以通过每个函数对应的起始位置和结束位置信息识别函数的边界。

在分类阶段，依据节点权重判断候选函数中的指令是否为函数起始位置。在确定了函数起始位置后，通过静态的控制流图恢复算法生成函数体对应的指令。即从指定的地址开始，通过递归的方式将与其直接连接的节点放入函数起始队列中。对于从起始地址可达的指令集合，将其添加到控制流图中，若新增的节点不存在于该队列中，则将该节点放在队尾。

2.3　反汇编的规范化

在二进制文件分析的反汇编阶段，生成基于基本块的控制流图是理解程序结构的关键。由于编译器在不同的编译环境下可能在助记符、寄存器编号等方面存在显著差异，这就需要规范化的方法来屏蔽这些差异，以便能够比较不同编译环境中的汇编指令。

在不同的编译环境下，即使是执行相同功能的指令，其表现形式也可能不同。例如，不同的编译器可能为相同的操作选择不同的助记符或寄存器编号。这种差异使得直接比较不同编译环境下生成的汇编代码变得更加复杂。为了有效地比较和分析这些代码，需要对基本块中的指令进行规范化处理。指令规范化处理过程如下。

1) 保持助记符不变。在规范化过程中，助记符直接反映了操作的类型，保持助记符不变从而保留了指令中的关键信息。

2) 规范化立即数和寄存器变量。立即数在不同编译环境中可能有不同的表示方式。规范化过程需要将其统一表示，如将所有立即数转换为统一的数值格式或范围。对于寄存器变量规范化，寄存器编号在不同的编译器或编译选项中可能不同，规范化操作将所有寄存器编号映射到一个统一的寄存器命名空间，使得相同的操作数在不同编译环境中可以被正确识别和比较。

通过这种规范化处理，即使是在不同的编译环境中生成的汇编指令，只要它们执行相同的操作，就可以被识别为相似或等效的指令，通过规范化操作可以匹配不同编译环境中相似的指令，如表1所示。

表1　　**规范化操作**

操作数	规则	描述	符号系统
立即数	Call	libc库调用	libc[name]
		递归调用	self
		函数调用	func
	Jump	跳转的目标	jmpdst
		字符串	dispstr
	Reference	变量	dispbss
		数据（非字符串型）	dispdata
	Default	其他立即数	immval
寄存器	Size	[*e*|*r*]*[*a*|*b*|*c*|*d*|si|di][*x*|l|*h*]*	reg[1|2|4|8]
	stack/base	[*e*|*r*]*[*b*|*s*|*i*]*p*[l]*	[*s*|*b*|*i*]*p*[1|2|4|8]
	special purpose	cr[0~15], dr[0~15]	reg[cr|dr|st]

2.4　指令比较

指令是计算机执行的基本单元，主要由操作码、操作数和数值常量构成，它们在进行指令相似度分析时的权重不同。算法1的作用为计算指令匹配得分[17]，为了优化匹配速度，必须对指令进行上述规范化操作。

操作码描述了具体的操作行为，当其操作数在标准化处理后保持一致时，相同的操作码应获得更高的得分。对于数值常量操作数，如果它们的值相等，这在指令比较中是一个显著的识别标志，因此会相应提高相似度得分。

对于操作码所描述的操作行为，当其操作数在标准化处理后保持一致时，应给予更高的得分。数值常量操作数的值若一致，则在指令比较中是一个明显的判断依据，因此会增加相似度得分。

算法1　计算指令匹配得分

输入　两条规范化指令ins1，ins2

输出　指令的匹配得分score

1)function CompIns(ins1, ins2)

2)　初始设置score的值为0

```
3)  if 伪指令 ins1 与伪指令 ins2 相等 then
4)    将 ins1 的操作数的数量保存在变量 n 中
5)    将当前伪指令的得分累加到 score 中
6)    for each i do
7)      if ins1 的第 i 个操作数与 ins2 的第 i 个操作数相等 then
8)        if ins1 的第 i 个操作数的类型为常量
9)          将常量的得分累加到 score 中
10)       else
11)         将操作数的得分累加到 score 中
12)       end if
13)     end if
14)   end for
15) else
16)   score 的得分为 0
17) end if
18)  return score
19)end function
```

2.5　路径搜索

在控制流图中，基本块由一系列指令构成，而每个节点可能连接多个后续节点。与此相对，最长公共子序列（LCS）的每个节点仅指向一个后续节点。为了在候选集中搜索出与 LCS 匹配度最高的执行路径，采用基于基本块间相似度的宽度优先搜索（BFS, breadth first search）方法。在候选集的控制流图上执行路径搜索操作，它以目标函数的 LCS 和候选函数的控制流图作为输入参数[17]。该操作使用 LCS 的计算策略，并在候选函数的路径搜索过程中，采用动态规划技术进行深入分析。考虑到路径搜索本质上是一个 NP 问题，该操作在启动时将存储表的长度初始化为 1，在每次迭代过程中，算法选取候选函数路径的第一个节点，并在存储表中为其创建新的一行，随后利用 updatePath() 函数更新存储表。

此外，该过程维护一个序列 s，用于记录每个节点在当前状态下的最高相似度得分。每当存储表更新后，序列 s 也会相应更新，确保序列 s 中始终反映的是每个节点的最高相似度得分。同时将当前节点的后续节点添加到待分析的候选路径集合中。这种方法不仅提高了路径搜索效率，还确保了在候选集中能够找到与 LCS 匹配度最高的执行路径。

2.6　相似性拓展

在实际分析过程中，由于程序的执行语义往往由多个基本块组合而成，仅对某一个基本块的相似性进行比较，难以发掘深层次的二进制相似代码。例如，在常见的内存破坏漏洞中，其内存边界检查对应的基本块代码可能在函数边界识别或代码优化等阶段被删除。故在上述基本块匹配算法的基础上，需要将某个基本块的相似性扩展到该函数对应的控制流图上，实现邻域搜索的功能，从而在函数粒度上有更广泛的适用性。

具体地，在上述函数边界识别操作的基础上，以函数控制流图为基础进行局部贪心算法的操作。在目标程序和候选程序中，以前文算法发掘的某一对匹配的基本块为起点进行邻域分析。注意到在该算法所生成的基本块集合中，两两匹配的基本块数量可能有多个，故对该步骤依次进行分析，直到匹配基本块集合为空。然后，在每一次基本块相似性邻域搜索过程中，以控制流图为基础，沿着其方向进行后续基本块的匹配分析。该邻域分析的匹配过程是单向的，即从起点基本块开始，依次朝着控制流图方向进行比较，而不会进行反方向基本块的比较。在该过程中所使用的匹配方法为匈牙利算法，从而实现在初始匹配基本块的基础上，发掘新的邻域基本块匹配结构。

对于在上述过程中生成的新的匹配基本块，为了衡量匹配程度，通过匈牙利算法中的距离变量的数值来构建优先队列。即依据该数值对新的匹配基本块进行实时排序，从而构建新的待匹配基本块结构。对于每一次的匹配分析，从优先队列中选择距离最短的基本块，扩展现有的基本块结构，直到优先级队列为空。

为了减少路径搜索的时间消耗并提高基本块映射方法的效率，引入了模糊匹配的方法，它通过上述邻域搜索策略，优化了基本块之间的映射过程，避免了对已分析基本块的重复匹配。

具体地，首先根据相似度得分，将路径搜索得到的匹配基本块对加入队列 Q 中。该队列按照相似度得分从高到低排序，每次从队首取出相似度得分最高的基本块进行处理。对于每一对基本块，算法通过分析其邻近的基本块，从而发掘匹配基本块。

由于候选函数由多个基本块构成，故函数的映射相似性计算可以使用基本块的映射得分进行累

加。例如，对于函数f和g，它们的相似度可以通过式(1)计算[17]。

$$\text{Similarity}(f,g)=\frac{2\sum_{\forall(u,v)\in\gamma}\text{CompBB}(u,v)}{\text{Score}(f)+\text{Score}(g)} \tag{1}$$

其中，$u(u\in f)$和$v(v\in g)$为基本块匹配集合γ中的基本块，函数Score()用于计算作为参数的函数f的自反映射得分。通过这种方法可以有效地扩展基本块映射结果，提高映射的准确性和效率。同时，邻域搜索策略也有助于发掘函数之间基本块的潜在映射关系，进一步提升整个映射过程的性能。

该过程中使用的匈牙利算法可以实现n个基本块的比较，同时将映射距离最小化，其运行时间为$O(n^3)$。故在分析过程中，只将某个基本块的邻域与其匹配的基本块的邻域进行分析，而不会同时比较多余2个相似的基本块。由于上述优先队列中的基本块实际上是程序中特定位置的特征向量，故在函数级别的分析中，该邻域搜索策略是上下文相关的。例如，在缓冲区溢出代码的分析场景中，如果使用函数级别的控制流图，则可能在边界检查过程中，某个基本块可能会在内存写入操作时丢失，从而导致误报。故该特征向量可以表示特定种类的缺陷，有助于研究人员对该类问题的理解。

此外，在邻域搜索过程中，基本块的粒度通常起到重要的作用，因为在上述优先级队列的操作过程中，需要对距离最短的基本块的数量进行选择。若基本块数量较少，则后期进行基本块比对的频率就越低，从而有更低的时间开销。在实际分析过程中，将该基本块数量设置为25，使得在相似度比较的性能和时间开销方面有较好的平衡。

3 语义相似匹配模型

二进制文件相似度分析的核心目标在于评估文件之间的语义相似性。然而，由于二进制文件在语法结构上的差异，这一分析过程可能会受到较大的干扰。首先将二进制代码划分为多个代码片段SubSlide集合，能够分别计算出局部和全局的相似性得分，然后利用程序验证器对这些代码片段进行语义层面的相似性分析，以确保分析结果的准确性和可靠性。

3.1 示例及分析

语义相似的SubSlide如图3所示，它展示了一个具有中间验证语言（IVL, intermediate verification language）形式的SubSlide示例[15]。该IVL形式使用64位的寄存器表示形式，在保留语义的基础上将具体的指令助记符进行抽象，为中间值的计算引入一个临时变量以保证寄存器之间的数据传递通过该临时变量。此外，在语义比较阶段，验证器通过对IVL形式的中间结构进行相似度比较，对具有相同输入参数及相同输出结果的过程都会被验证器认定为相似。为了更好地区分查询SubSlide和目标SubSlide，图3中使用了下标q和t。

$t1_q$=$r3_q$
$t2_q$=7h+$t1_q$
$t3_q$=int_to_ptr($t2_q$)
$r7_q$=$t3_q$
$v4_q$=9h
rsi_q=$t4_q$
$t5_q$=$t4_q$+$t3_q$
rax_q=$t5_q$

$t1_t$=23h
$x9_t$=$t1_t$
$t2_t$=rbx_t
$t3_t$=$t2_t$+$t1_t$
$t4_t$=int_to_ptr($v3_t$)
$x13_t$=$t4_t$
$t5_t$=$t1_t$+3
rsi_t=$t5_t$
$t6_t$=$t5_t$+$t4_t$
rax_t=$t6_t$

图3 语义相似的SubSlide

为了对SubSlide之间的相似度进行计算，首先计算两个SubSlidess_q与s_t之间具有等价指令的数量，再计算其在s_t的比率$\text{SIM}(s_q,s_t)$。特别地，若s_q与s_t的输入输出结构相似，则可以依据该比率$\text{SIM}(s_q,s_t)$计算条件概率$\Pr(s_q|s_t)$（详见3.3节）。如图3所示，左侧为查询SubSlidess_q，右侧为目标SubSlidess_t，由于s_q中的变量都能在s_t中找到有与之等价的，即条件概率$\Pr(s_q|s_t)=1$，反之$\Pr(s_t|s_q)=\frac{4}{5}$。

在编译器优化过程中，可能会产生一些共有的SubSlide，这些SubSlide可能会对相似度得分产生干扰。为了准确评估SubSlide的相对重要性，引入局部相似度得分的计算方法，即$S_{\text{local}}(s_q|t)=\log\frac{\max_{s_t\in t}\Pr(s_q|s_t)}{\Pr(s_q|H_0)}$，这种方法通过衡量与$s_q$等价的SubSlide的重要性，有助于区分等价的SubSlide[15]。为了使结果更加直观，采用对数变换将数据转换为平滑的线性变化的百分比值，这有助于将数据的分布范围缩小，突出数据间的差异并降低极端值对最终分析结果的潜在影响。

对于2个函数q和t，s_q在t中的重要程度可以

通过$S_{\text{local}}\left(s_q|t\right)$进行计算，在此过程中对$q$中所有$s_q$的$S_{\text{local}}\left(s_q|t\right)$进行累加，并获得$q$和$t$之间的全局相似度得分$S_{\text{global}}$。通过统计推理的方法以SubSlide之间的局部相似度为基础，计算全局函数之间的相似度得分$S_{\text{global}}\left(q|t\right)$，该过程如式(2)所示[15]。

$$S_{\text{global}}\left(q|t\right)=\sum_{s_q\in q} S_{\text{local}}\left(s_q|t\right)=\sum_{s_q\in q}\log\frac{\max_{s_t\in t}\Pr\left(s_q|s_t\right)}{\Pr\left(s_q|H_0\right)} \tag{2}$$

其中，$\max_{s_t\in t}\Pr\left(s_q|s_t\right)$表示$s_t$与$s_q$在语义上等价的最高概率，$\Pr\left(s_q|H_0\right)$表示与随机的SubSlide能够进行匹配的概率。

3.2　函数分解

根据3.1节所述，程序中的函数可以通过一系列基本块来表示，这些基本块保留了程序的控制流结构。算法2采用逆序遍历基本块的方法来构建SubSlide[15]，该过程从将基本块中的指令序列存储到集合unusedInsts开始，直到所有指令被整合到一个SubSlide中，则算法终止。对于每个尚未使用的指令，算法会计算出在SubSlide中定义的变量集varsDefined和引用的变量集varsReferenced。在for循环中，将定义变量的指令添加到SubSlide中，并相应更新varsReferenced和varsDefined。循环完成后，SubSlide将包含基本块中全部用于变量计算的指令。

算法2　从基本块中提取SubSlide

输入　一个基本块对应的指令序列b

输出　指令序列b对应的SubSlide

1)将unusedInsts的初始值设置为{1,2,⋯,|b|}，设置SubSlide为空数组
2)while 存在未使用的指令
3)　将unusedInsts的最大值保存在变量maxUsed中
4)　从unusedInsts中删除最大未使用的变量
5)　将b中最大未使用的值保存在newSubSlide中
6)　将b中最大未使用的值的引用保存在varsRefed中
7)　将b中最大未使用的值的定义保存在varsDefed中
8)　for each i in maxUsed
9)　将Def(b[i])∩varsRefed计算的结果保存在needed中
10)　if needed不为零
11)　将b[i]的值累加到newSubSlide中
12)　将Refed∪Ref(b[i])的值更新到Ref(b[i])中
13)　将varsDefed∪needed的值更新到varsDefed中
14)　从unusedInsts中删除i
15)　end if
16)　end for
17)　从varsRefed中删除varsDefed，并保存在inputs中
18)　将newSubSlide和inputs更新到SubSlide中
19)end while

3.3　SubSlide置信度分析

设s_q和s_t分别为函数q和t对应的SubSlide，则可以通过式(3)计算s_q在t中存在与之等价的SubSlide的概率[15]，记为$\Pr\left(s_q|t\right)$。

$$\Pr\left(s_q|t\right)=\max_{s_t\in H_t}\Pr\left(s_q|s_t\right) \tag{3}$$

其中，$\Pr\left(s_q|s_t\right)$为条件概率，使用sigmoid函数对s_q和s_t的匹配程度$\text{SIM}\left(s_q,s_t\right)$进行计算[15]。

$$\Pr\left(s_q|s_t\right)=g\left(\text{SIM}\left(s_q,s_t\right)\right)=\frac{1}{1+\text{e}^{-k\left(\text{SIM}\left(s_q,s_t\right)-0.5\right)}} \tag{4}$$

为了将相似度得分规范化到一个明确的区间内，使用sigmoid函数将相似度得分有效地压缩至0到1，从而使得输出结果更加平滑且易于解释。此外，为了量化代码片段之间的随机匹配概率[15]，通过引入相似比（SR, similarity ratio）的概念，SR用于衡量2个代码片段在随机情况下匹配的可能性。

$$\text{SR}\left(s_q|t\right)=\frac{\Pr\left(s_q|t\right)}{\Pr\left(s_q|H_0\right)} \tag{5}$$

为了获得随机语义的匹配值，可以通过使用$\Pr\left(s_q|H_0\right)=\frac{\sum_{s_t\in T}\Pr\left(s_q|s_t\right)}{|T|}$来计算所有$\Pr\left(s_q|s_t\right)$的平均值，其中$T$为目标SubSlide的集合，从而$S_{\text{local}}$的相似概率可以通过式(6)进行计算[15]。

$$S_{\text{local}}\left(s_q|t\right)=\log \text{SR}\left(s_q|t\right)=\log \Pr\left(s_q|t\right)-\log \Pr\left(s_q|H_0\right) \tag{6}$$

局部相似度得分的意义在于它可以通过计算某一个函数中存在与某个s_q语义等价的置信度，同时全局相似置信度$S_{\text{global}}(q|t)$表示q可以由t中相似的SubSlide组成，其值为所有$S_{\text{local}}(s_q|t)$的总和。

3.4 语义相似度计算

对于程序P的一个状态序列$\sigma_i\left(l_i,\text{var}_i,\text{val}_i\right)$，其中$\sigma_i\in\sum_P$，$\sum_P$为$P$所有状态集合，变量$\text{var}_i$的位置信息为$l_i$，其具体值为$\text{val}_i$。同时记$\sigma_0$到$\sigma_n$组成的序列为程序的一次具体的执行$\pi\in\sum_P^*$，first和last分别为该执行的首尾状态。

记2个状态σ_1和σ_2之间的关联集合为γ，它表示σ_1中var_1到σ_2中var_2的函数，即$\text{var}_1\nrightarrow\text{var}_2$。若$\forall\left(v_1,v_2\right)\in\gamma{:}\sigma_1\left(v_1\right)=\sigma_2\left(v_2\right)$，则称$\sigma_1$和$\sigma_2$在$\gamma$上等价，记为$\sigma_1\equiv_\gamma\sigma_2$。若2个执行$\pi_1$和$\pi_2$在$\gamma$上等价，则可以表示为$\pi_1\equiv_\gamma\pi_2$。

对于2个SubSlide s_1和s_2，若s_1和s_2之间存在等价关系，即$s_1\equiv_\gamma s_2$，需满足以下条件。

1) s_1和s_2之间存在相互对应某个的输入。

2) 对于一次执行$\left(\pi_1,\pi_2\right)\in\left(s_1,s_2\right)$，若其在输入处等价，即$\pi_1\equiv_\gamma\pi_2$，则可用式(7)表示。

$$\forall\left(i_1,i_2\right)\in\left(\gamma\cap\left(\text{inputs}\left(s_1\right)\text{inputs}\left(s_2\right)\right)\right):$$
$$\text{first}\left(\pi_1\right)\left(i_1\right)=\text{first}\left(\pi_2\right)\left(i_2\right) \tag{7}$$

2个SubSlide之间的相似度SIM可以表示为在关联集合γ上有最大关联匹配变量的比率[15]，即

$$\text{SIM}\left(s_q,s_t\right)=\frac{\max\left\{\left|\gamma\right|\left|\forall\left(\pi_q,\pi_t\right)\in\left(s_q,s_t\right):\pi_q\equiv_\gamma\pi_t\right\}}{\left|\text{Var}\left(s_q\right)\right|} \tag{8}$$

对于状态σ_q和σ_t，在σ_q中存在与其匹配的概率可以表示为$\text{SIM}\left(\sigma_q,\sigma_t\right)=\frac{\gamma_{\max}}{\sigma_q}$，其中$\left|\gamma_{\max}\right|$为$\sigma_q$和$\sigma_t$相互等价时，其关联的最大数值，即$\sigma_q\equiv_{\gamma_{\max}}\sigma_t$。SIM的优势在于它直接计算SubSlide中的关键执行语义的相似度，从而缓解仅通过判断SubSlide的输出值之间的相似性所带来的误差。

对于图3中的示例，通过对变量增加下标q和t进行别名操作从而区分每次的执行过程，算法3为SubSlide的SIM值计算过程[15]。对于查询函数p^q和目标函数p^t，通过为某一对程序输入增加等价假设，计算p^q和p^t的执行语义，同时在last状态下将断言信息写入与γ相匹配的变量中。

算法3 计算SubSlide的SIM值

输入 具有Boogie IVL形式的查询SubSlide p^q，目标SubSlide p^t

输出 SIM(p^q, p^t)

1)设置maxSIM的值为0
2)for each $\gamma\in\Gamma(p^q, p^t)$
3) 将NewProcedure(Inputs(p^q)∪(Inputs(p^t))的计算结果保存在p中
4) for each 对于(r∩(Inputs(p^q)∪Inputs(p^t))中的每一个(i^q, i^t)
5) 将assume $i^q==i^t$的计算结果追加到p中
6) end for
7) 将(p^q.body, p^t.body)的计算结果追加到p中
8) for each (Vars(p^q)Vars(p^q)∩γ)中的每个(v^q, v^t)
9) 将assume $i^q==i^t$的计算结果追加到p中
10) end for
11) 通过求解函数Solve(p)进行求解
12) if p^q与p^t等价
13) 将(|r|/|Vars(p^q)|, maxSIM)的最大值保存在maxSIM中
14) end if
15) end for

3.5 进一步讨论

在进行相似性分析时，通常关注于单一平台上的二进制程序。然而，在实际操作过程中，不同平台和不同编译器生成的二进制程序之间可能存在差异，这可能会影响分析的准确性。为了减少这些差异对分析结果的影响，第2.3节提出了指令规范化的方法。为了增强本文方法的泛化能力，本节首先阐述了函数微执行的概念及其重要性。函数微执行是指在不执行完整程序的情况下，对函数中的指令进行局部执行和分析的过程。这种方法有助于在不运行整个程序的情况下，理解和分析程序的行为。

然后引入基于迁移学习的模型用于分析不同体系结构的指令。迁移学习是一种机器学习方法，它允许模型将在一个任务上学习到的知识迁移到另一个相关任务上。这表示可以将在一个平台或编译器上训练得到的模型应用到另一个不同的平台或编译器上，从而提高分析的准确性和效率。通过结合函

数微执行和迁移学习，本文方法能够更好地适应不同平台和编译器的差异，提高二进制程序相似性分析的泛化能力和准确性。

3.5.1　函数微执行

在比较不同体系结构、不同编译器以及不同优化选项生成的执行文件的相似度时，提出了一种基于迁移学习的模型。该模型可以分析由指令和状态值组成的微执行序列，并计算它们之间的相似度，其优势在于它不需要人工标记数据，而是采用无监督学习的方式，通过预训练模型自动从微执行序列中提取和学习指令的执行语义。

与第3.4节中直接使用SubSlide进行相似度比较的方法不同，这种迁移学习模型能够从指令的上下文中学习到更丰富的语义信息。通过无监督预训练，迁移学习模型可以识别出在特定上下文中指令的执行模式和行为，收集到欠约束的指令集合。通过这种方式，迁移学习模型能够自动学习和理解当前指令在不同上下文中的执行语义，进而对不同体系结构和编译器生成的执行文件进行有效的相似度比较。

具体地，通过中间表示（IR, Intermediate Representation）的方法屏蔽不同架构上代码的语法差异，从而表示不同架构的汇编语法[18]。对于基于寄存器内存架构的x86平台和基于加载存储架构的ARM及MIPS平台可以用store(e_v,e_a)和load(e)分别表示读写操作，其中e_v为值表达式，e_a为地址表达式，jmp为有条件或无条件下的直接或者间接跳转指令，函数调用和返回分别使用call(e)和ret指令。

Function	f ::=[i_1,i_2,i_3,⋯]
Instrucion	i ::=nop|ret|call(e)|jmp(e_c,e_a)| store(e_v,e_a)|r:=load(e)|r:=e
Expression	e ::=c|r|r_1 op r_2
Operator	op ::={=,−,*,/,<,>,⋯}
Register	r ::={pc, sp, eax,⋯}
Const	c ::={true, false, 0x0,⋯}

基于IR，利用算法4来构建函数f的微执行过程[18]。这一过程涉及将函数f内的所有指令序列化并存储在一个队列结构中。同时，对栈指针和首个指令的地址进行必要的初始化设置。随后，按照指令的顺序逐一执行队列中的指令。当队列中的指令全部执行完成，算法随之结束其运行周期。

算法4　计算函数的微执行过程

输入　二进制函数f，寄存器r

输出　微执行t

1)将二进制函数f中的指令放入队列L中
2)设置空向量t
3)for each 对于r\{sp, pc}中的每一个寄存器r_i
4) 初始化寄存器值，并保存在r_i中
5)end for
6)while L和Φ不相等
7) 通过dequeue(L)操作，将结果保存在i中
8) if i的类型为load或者store
9) 通过memMap()函数计算i.accessAddr和i.accessSize的值
10) if i的类型为load
11) 通过参数i.accessAddr计算函数writeRandom()的值
12) end if
13) 通过t∪execute(i)更新t的值
14) else if i的类型为jmp或者call then
15) if i.targetAddr not in [cm.minAddr, cm.maxAddr] then
16) continue
17) end if
18) 通过计算t∪execute(i)更新t的值
19) else if i的类型为nop then
20) continue
21) else if i的类型为ret then
22) break
23) else
24) 通过计算t∪execute(i)更新t的值
25) end if
26)end while

3.5.2　迁移学习模型

迁移学习使用预训练模型，该模型的输入模块x由5个大小均为n的序列x_f,x_t,x_c,x_o,x_a组成[18]。其中指令位置序列记为x_c，指令中操作码或操作数的位置序列记为x_o，指令集的架构序列记为x_a。

在初始训练阶段，模型的损失函数由两部分组成，一部分是针对代码序列预测的交叉熵，另一部分是针对值预测的交叉熵，后者涉及8位微操作的值。由于这一过程不需要附加的标签数据，因此这种预训练策略是无监督的。其中包含一个双向长短期记忆（Bi-LSTM, bidirectional long short-term memory）网络，用来处理微操作的值，并将该值作为嵌入向量进行分析。同时模型包含自注意力机制和输出层，

这些部分用来对指令和值进行推断和预测。

模型的输入序列$x_f=\{\text{mov,eax,}\cdots\}$，它包括微跟踪算法中标记的汇编代码，$x_t$用于记录指令执行前每个符号用动态值序列。例如，对于mov eax, 0x1B；mov eax, 0x5C，在执行mov eax, 0x5C之前，eax中的值为0x1B。输入序列x_t为8位固定长度的序列$\{0x00,\cdots,0xff\}$，其中第i个值记为x_{t_i}。随后双向LSTM（Bi-LSTM）接收该x_{t_i}的输入，并使用$t_i=\text{Bi-LSTM}\left(x_{t_i}\right)$的值表示最后一个位置的嵌入，该嵌入的值作用在$x_{t_i}$上的计算结果保存在$t_i$中。

x_i为被屏蔽的嵌入，记为$m\left(E_i\right)$，同时被屏蔽的位置信息保存在集合MP中，嵌入序列$(E_1,\cdots,m(E_i),\cdots,E_n),i\in \text{MP}$作为预训练模型$g_p$的输入。预训练的目标是搜索$g_p$的参数$\theta$，从而减少预测值和预测令牌间的交叉熵损失。

$$\underset{\theta}{\operatorname{argmin}}\sum_{i=1}^{|\text{MP}|}\left(-x_{f_i}\log\left(\hat{x}_{f_i}\right)+\alpha\sum_{j=1}^{8}-x_{t_{ij}}\log\left(\hat{x}_{t_{ij}}\right)\right) \tag{9}$$

其中，第i个令牌x_{t_i}的第j个字节用$\hat{x}_{t_{ij}}$表示，同时使用参数α衡量预测值和预测令牌间的交叉熵损失[18]。

为预训练模型g_p输入2个函数，并生成由g_p中自注意层生成的嵌入序列$E_k^{(1)}=\left(E_{k,1}^{(1)},\cdots,E_{k,n}^{(1)}\right)$和$E_k^{(2)}=\left(E_{k,1}^{(2)},\cdots,E_{k,n}^{(2)}\right)$，从而对函数的相似性进行微调[18]。同时函数嵌入的计算过程如式(10)所示，使用2个多层感知器g_t，而输入为每个函数嵌入的平均值。

$$g_t\left(E_k\right)=W_2\tanh\left(\frac{W_1\sum_{i=1}^{n}E_{k,i}}{n}\right) \tag{10}$$

其中，$W_1\in R^{d_{\text{emb}}\times d_{\text{emb}}}$，$W_2\in R^{d_{\text{emb}}\times d_{\text{func}}}$可将$d_{\text{emb}}$维度的自注意层嵌入的平均值转换为$d_{\text{func}}$维的函数嵌入。在实际分析过程中，为了使该方法有较好的适应性以应对规模较大的函数，d_{func}的数值通常比d_{emb}小。该过程中微调的目标是计算两个函数嵌入之间的余弦距离的最小值，同时使得真值之间的余弦嵌入损失l_{ce}有最小值[18]。

$$\underset{\theta}{\operatorname{argmin}}\,l_{\text{ce}}\left(g_t\left(E_k^{(1)}\right),g_t\left(E_k^{(2)}\right),y\right) \tag{11}$$

其中，$l_{\text{ce}}\left(x_1,x_2,y\right)=\begin{cases}1-\cos\left(x_1,x_2\right), & y=1\\ \max\left(0,\cos\left(x_1,x_2\right)-\xi\right), & y=-1\end{cases}$

其中，ξ的取值范围为[0,0.5][18]，在实验过程中为了在处理不相似函数过程中的精度和时间开销上取得平衡，依据实验结果设置ξ的取值为0.1。

4 实验及分析

本节的实验比较涉及IDA 8.0、BAP 0.98、Genius[5]、Trex[18]以及jTrans[9]几款软件。IDA 8.0作为反汇编及控制流图构建的常用工具，而BAP 0.98和Genius为被广泛使用的二进制分析领域的标准工具。此外，Trex基于Bert技术进行相似性检测，是近年来较新的分析工具。jTrans则为最新开发，采用跳转感知编码技术对二进制指令进行分析，展现出了优异的分析能力。在选择测试数据集时，本节根据各个实验的测试目标，采用与之匹配的数据集以达到最佳的对比效果。在后续的实验模块中，将详细展示各个实验的具体设置和测试目标。

本文的实验环境为安装了Ubuntu 20.04.3 LTS操作系统的Linux服务器，配备了Intel Xeon E5-4600系列处理器，该处理器主频为2.70 GHz，由于采用了超线程技术，具有70个逻辑处理器核心。服务器配备了375 GB的随机存取存储器（RAM, random access memory），图形处理单元（GPU）为Nvidia GeForce GTX 1070。

4.1 函数边界识别实验及分析

本实验的目标为评估函数匹配模型的性能和函数起始位置的定位。实验数据集为来自coreutils、binutils和findutils的1 732个具有多种不同的格式的二进制文件。所使用的编译器包括GNU gcc 4.7.2和Intel icc 14.0.1，在准确性和执行时间等方面进行实验。

该实验过程将函数识别过程作为有监督的分类问题，表2为gcc与icc在模型签名匹配方面的精度和时间开销对比。

依据表2中的数据，本文方法在精度和召回率方面相对于对比工具有明显优势，其中IDA和BAP等工具进行反汇编并构建控制流图过程中，会因为间接跳转到未定义或错误的函数位置，被控制流完整性分析阻止，从而不能正确识别存在的函数。另外，如第2节所述，除了编译器在优化过程中可能会引入额外的指令，跳转表之类的结构以及函数之间共享代码也可能导致函数在地址空间上不连续，IDA和BAP等传统的分析工具难以处理此类问题，

故对函数边界识别的精度较低。需要指出的是，一方面采用高度为5的前缀树模型，虽然实现了较短的特征向量，这有助于减少大约2.8%的执行时间，但同时却导致了大约7.21%的精度下降。另一方面，对于未进行规范化处理的模型，虽然观察到大约1%的精度提升，但其识别率下降了约6.83%，且执行时间是规范化模型的12倍左右。

表2　匹配模型的精度和时间开销对比

方法	gcc			icc		
	精度	召回率	时间/s	精度	召回率	时间/s
IDA 8.0	0.572	0.493	1 893.5	0.627	0.534	2 692.7
BAP 0.98	0.623	0.567	1 723.1	0.652	0.593	2 547.6
Genius	0.486	0.531	2 284.3	0.565	0.642	2 292.5
Trex	0.912	0.824	1 901.5	0.871	0.787	2 117.2
jTrans	0.935	0.862	1 784.8	0.931	0.858	1 961.1
本文方法	0.961	0.921	1 631.9	0.948	0.919	1 878.3

在函数定位的实验中，测试集包含1 486个采用ELF文件格式和246个采用PE文件格式的文件。如表3所示的数据，其中单元格中横线上部依次为精度和召回率，横线下部为时间开销。本文方法在函数起始位置的定位精度和召回率上均优于其他对比工具。在执行时间方面，由于对比工具采用了较少的匹配模式，并且未进行指令规范化的预处理，因此IDA和BAP工具的执行时间相对较短，但其分析精度和召回率较低。

表3　函数识别的精度、召回率及时间开销对比

方法	PE x86	PE x86-64	ELF x86	ELF x86-64
IDA 8.0	0.723/0.534	0.771/0.573	0.634/0.762	0.671/0.759
	313.7	326.9	332.4	351.7
BAP 0.98	0.638/0.591	0.625/0.581	0.647/0.715	0.629/0.753
	389.5	369.4	406.1	411.5
Genius	0.632/0.718	0.674/0.734	0.721/0.754	0.735/0.768
	641.5	485.2	757.2	782.3
Trex	0.781/0.779	0.779/0.751	0.761/0.785	0.862/0.827
	432.8	428.6	585.2	618.4
jTrans	0.829/0.871	0.857/0.841	0.832/0.859	0.882/0.846
	341.9	387.1	462.1	463.2
本文方法	0.942/0.925	0.932/0.924	0.951/0.941	0.908/0.887
	289.5	365.9	423.8	443.2

4.2　函数相似度匹配实验及分析

本实验的数据集选取了在软件及操作系统领域内广泛部署的库文件，包括zlib和libpng。zlib是一个专门用于数据压缩的库，而libpng是一个专门处理PNG格式图像的库。

在分析函数复用及匹配的过程中，本实验目标为对相同二进制文件的不同版本进行函数复用分析。在实验过程中，采用了3种不同的相似性评估阈值，分别为0.6、0.65和0.7，以识别函数间的相似性。对于其Jaccard相似度低于这些设定阈值的函数，将应用前文提到的优化方法进行处理。此外，实验还通过计算基本块的数量来评估函数的规模，其规模的评估范围为4到35。

表4展示了函数识别实验的结果，该结果表明利用优化策略从函数库生成了候选集，随着相似度阈值的提高，候选集的规模逐步缩减，同时识别精度也呈现下降趋势。特别地，对于版本1.2.7的情况，识别精度出现了异常上升，原因在于当阈值设定为0.6时，某些其他函数的相似度得分意外超过了真正匹配的函数。然而，当阈值上调至0.7时，这些干扰函数被优化规则排除，使得真正的匹配函数获得了最高相似度得分。

表4　不同版本的zlib及与libpng的函数识别实验

版本	阈值	精度	候选数量	平均耗时/s
1.2.6	0.6	93.78%	11 824	2.62
	0.65	92.32%	2 867	1.33
	0.7	90.57%	1 863	0.93
1.2.7	0.6	97.41%	15 943	2.82
	0.65	96.83%	2 761	1.75
	0.7	93.61%	1 874	1.13
1.2.8	0.6	95.09%	15 871	2.75
	0.65	96.26%	2 785	1.72
	0.7	97.12%	1 957	1.27
1.2.9	0.6	94.63%	3 431	2.19
	0.65	93.16%	706	1.51
	0.7	90.12%	234	1.39

在程序分析部分，对两种常见的恶意软件Citadel和Zeus进行了深入的实验分析。Zeus恶意软件中采用了RC4加密算法的函数，而恶意软件Citadel则是在Zeus基础上发展而来，并对原有的

加密算法进行了调整。在实验过程中将从Zeus中提取RC4加密算法函数作为基准，与Citadel中提取相应函数进行相似度分析。

4.3 语义相似实验及分析

本文方法能够在程序发生变更时进行有效分析，这包括处理具有跨平台兼容性、编译器独立性以及不同编译优化级别的代码。在对目标函数集合进行分析之前，首先将其分解为单独的函数单元，随后执行相似性分析。此外，通过调用程序验证器，计算原始函数之间的相似度，以确保分析的准确性。

在测试集中使用8个真实的代码，包括具体的CVE（common vulnerabilities and exposures）信息，以及来自coreutils中的开源包，如表5所示。

本节实验研究主要围绕3个核心方面展开。首先，通过使用不同版本的gcc(4.6,4.8,4.9)、Clang(3.4,3.5)以及icc(14.0.4,15.0.1)编译器进行兼容性测试，以分析不同编译器版本间的函数兼容性。其次，实验通过交叉编译的方式，每次从不同的编译器中选取一个目标函数进行匹配测试，以探究编译器间的差异对函数匹配的影响。最后，实验还对经过语义修改的函数进行了深入的语义分析，即当源程序中的某个函数经历了语义变更时，这种变更同样会对其他相关函数的语义产生影响。

通过第3节的分析，对于表5中的数据集，其语义相似分析过程如下。

1) SubSlide之间相似性的计算过程（P_{SubSlide}）。该过程将SubSlide之间的SIM计算结果推广至函数间的SIM计算过程，其与目标SubSlide匹配的最大数量的具体计算方法为 $\sum_{s_t \in T} \max_{s_q \in Q} \left(\text{SIM}\left(s_t, s_q\right)\right)$。

2) 相似性推理全局相似性的过程（$P_{\text{local-global}}$）。在S_{local}和S_{global}的计算过程中，使用等式$\Pr\left(s_q, s_t\right) = \text{SIM}\left(s_q, s_t\right)$进行相应的代换，从而在没有sigmoid函数的情况下计算全局函数的相似度。

3) 在上述过程的基础上增加sigmoid函数的过程（P_{sigmoid}）。

通过以上3个步骤对测试集进行了相似度分析实验，旨在量化SubSlide之间的相似度[15]。表5展示了不同SubSlide之间的相似度得分。值得注意的是，较短的SubSlide往往对应较低的相似度得分，这可能会降低结果的置信度。通过调整相似度阈值来确保查询SubSlide至少与一半的目标SubSlide（即最小的阈值设定为0.5）相匹配。此外，还设定了最大匹配倍数为2，以减少因SubSlide数量过多而导致的真阳性数量减少的概率。这种阈值调整策略有助于提高匹配的准确性和可靠性。

在表5中展示的数据集中，选择来自openssl、bash和coreutils软件包的目标函数作为查询函数。图4中的柱状图直观地展示了每个目标函数的实验结果，其中每个柱状条代表一个特定的目标函数。柱状图的高度反映了目标函数的相似度得分，从而衡量目标函数与查询函数之间相似性。

在图4中，相同名称的编码器表示目标函数与查询函数具有相同代码基础，但可能在编译过程或代码版本上存在差异，分类比较有助于区分不同编译器和版本对函数相似度得分的影响。实验结果表明，即使目标函数来自不同的编译器或不同版本，本文方法仍然能够有效地识别出它们与查询函数之间的高度相似性，从而给出较高的相似度得分。该

表5 FP和ROC的查询搜索对比

文件名	基本块数量	SubSlide数量	P_{SubSlide}		$P_{\text{local-global}}$		P_{sigmoid}	
			FP	ROC	FP	ROC	FP	ROC
Heartbleed	19	97	108	0.964	0	1.030	0	0.962
Shellshock	152	463	241	0.875	4	0.962	4	0.975
Venom	17	67	1 353	0.963	0	1.020	0	1.060
Clobberin Time	46	261	340	0.831	63	0.956	23	0.952
ShellShock	94	27	169	0.848	45	0.962	0	0.918
ws-snmp	7	110	46	0.973	7	0.963	2	0.943
wget	89	183	321	0.846	16	0.975	0	0.938
ffmpeg	16	85	251	0.927	95	0.942	0	0.967

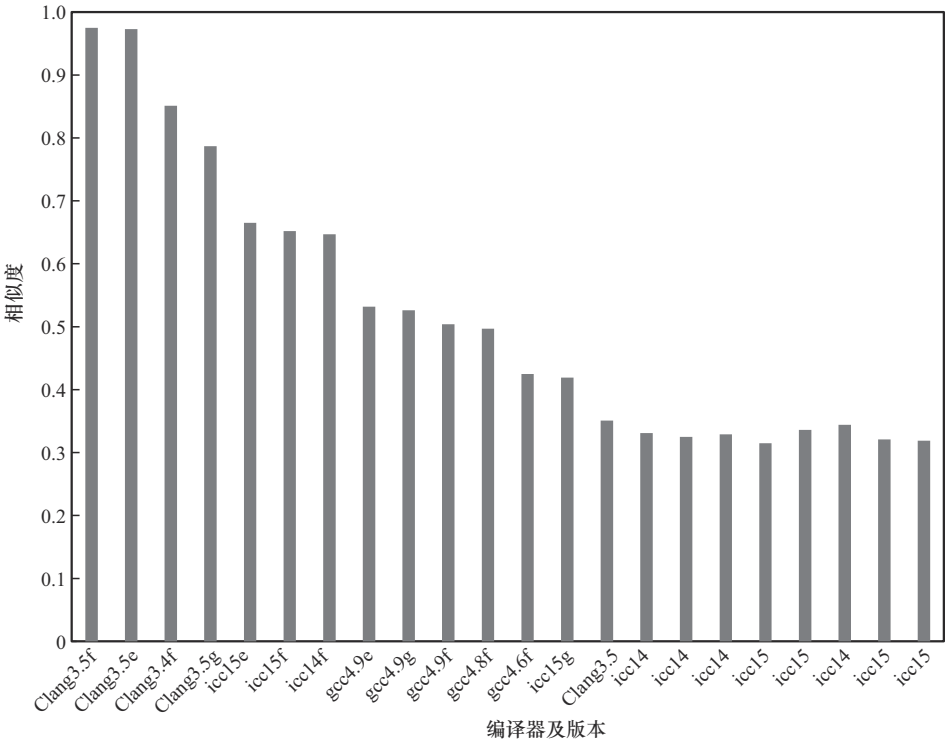


图4　相似度得分对比

实验结果验证了本文方法在处理跨编译器和跨版本函数相似度分析方面的有效性和可靠性。

对于变异代码的语义相似性分析，以程序的补丁修复为背景进行分析。对于程序中出现的漏洞，软件供应商通常采用发布补丁的方式进行修复，但是对于修复细节则往往不会予以公开，本文通过分析未修补版本和已修补版本的二进制文件，可以将修复过程定位在一个较小的区域。例如，在MS15-034中，对于特制的HTTP.sys请求，HTTP协议堆栈不能正确地进行解析，从而恶意用户可以执行任意代码，微软通过发布对应的补丁修复该缺陷。本文方法通过识别经过修改的函数来定位修改位置，经过分析对于相似度不为1的11个函数，其中5个函数的语义功能不同。

表6所示的补丁函数来自文献[17]，如对于UlpParseRange函数，在修复前后均有63个基本块，其中只有一个基本块的语义功能发生了改变。如图5所示，修复后的基本块的出度发生了改变（修复前为1，修复后为2），在指令中调用了_RtlULongLongAdd函数以修复该缺陷[17]。

依据表6，本文方法相较于其他3种对比工具，有更高的函数相似度得分，表明本文方法可以较好地应对不同版本函数或者变异函数的相似度分析需求。

表6　　补丁函数的相似度

函数名	本文方法	IDA	BAP	jTrans
Func1	0.972	0.631	0.672	0.867
Func2	0.935	0.713	0.691	0.875
Func3	0.859	0.574	0.517	0.794
Func4	0.841	0.515	0.488	0.725
Func5	0.794	0.483	0.416	0.642

```
sub   eax, edi
sbb   ecx, edx
add   eax, 3
adc   ecx, 5
mov   [esij], eax
mov   [esi+7], ecx
```

```
push  esi
push  2
sub   eax, edi
push  3
sbb   ecx, edx
push  ecx
push  eax
call  _RtlULongLongAdd
test  eax，eax
jl    loc_2A60C7
```

图5　基本块的补丁对比实验

4.4　跨平台指令相似性实验及分析

在本文的训练阶段，首先将数据集划分为训练集和测试集，以确保模型能够在其他数据上进行有效的泛化。为了全面评估模型的性能，采用了10折交叉验证的方法。这种方法涉及将数据集分成10个部分，每次使用其中9个部分进行训练，

剩余一个部分用于测试，循环进行 10 次，最终通过计算每次测试的速度与精度的平均值来评估模型的整体性能。

数据集由 2 200 个二进制文件组成，这些文件涵盖了广泛的数据来源。其中，2 064 个文件来自 13 个主流 Linux 开源项目，而其余文件来自 Windows 系统，包括 putty、7zip、vim 和 libsodium。这样的数据集设计旨在模拟真实世界中的多样化编译环境，包括 Linux 和 Windows 操作系统，以及 x86 和 x86-64 指令集。此外，还考虑了不同的编译器，如 GNU gcc 4.72、Intel icc 14.0.1 和 Microsoft Visual Studio，以及从无优化（O0）到全优化（O3）的 4 个优化级别，函数具体的信息如表 7 所示。

本实验定义函数比率作为测试集中出现相似函数的百分比，以量化测试集中相似函数的普遍性。此外，引入了 top-p ratio，它表示在所有测试函数中，相似度排名前 p 的函数所占的比率。相应地，bottom-p ratio 定义为相似度排名后 p 的函数所占的比率。这 2 个比率的差值被用作衡量函数相似性的置信度，即 top-p ratio 与 bottom-p ratio 的差值越大，表示函数相似性越高。图 6 展示了在 p=10% 的条件下，top-p ratio 与 bottom-p ratio 的差值在真阳性（TP）、假阳性（FP）、假阴性（FN）和真阴性（TN）上的典型值。这些值表明本文方法在正确性方面相较于其他对比工具有显著优势。

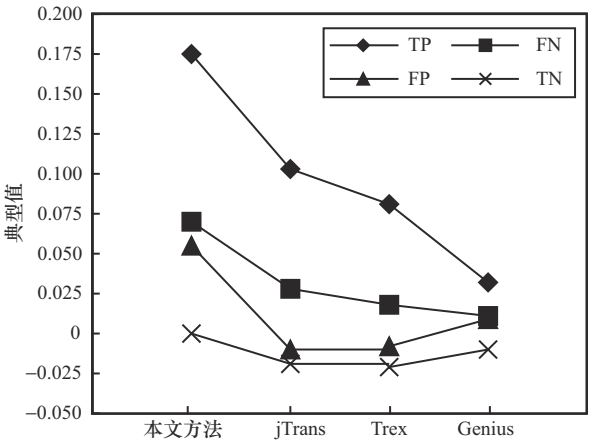


图 6 不同工具的典型值对比

表 7 带有 4 级优化的不同架构的函数数量

架构	优化级别	Binutils	Coreutils	Curl	Diffutils	Findutils	GMP	ImageMagick	Libmicrohttpd	LibTomCrypt	OpenSSL	Putty	SQLite	Zlib	合计
x86	O0	37 783	24 383	1 335	1 189	1 884	809	3 876	326	818	12 252	7 548	2 923	204	95 630
	O1	32 263	20 079	1 013	967	1 516	741	3 482	280	782	11 578	6 171	2 248	196	81 316
	O2	31 523	18 612	1 054	1 006	1 524	728	3 560	265	784	11 721	6 171	2 113	183	82 988
	O3	31 047	17 439	1 020	1 052	1 445	707	3 597	284	760	11 771	5 892	1 930	197	85 192
x64	O0	26 757	17 238	1 034	845	1 386	751	2 970	200	782	12 047	7 061	2 190	151	73 412
	O1	21 447	12 532	739	600	1 000	691	2 358	176	745	11 120	5 728	1 523	137	58 796
	O2	20 992	12 206	734	596	976	689	2 374	171	742	11 136	5 703	1 380	132	57 831
	O3	19 491	11 488	662	536	857	667	2 308	160	725	10 768	5 390	1 183	119	54 354
ARM	O0	25 492	19 992	1 067	944	1 529	766	2 938	200	779	11 918	7 087	2 286	147	75 152
	O1	20 043	14 918	771	694	1 128	704	2 341	176	745	10 991	5 765	1 614	143	60 033
	O2	19 493	14 778	765	693	1 108	701	2 358	171	745	11 001	5 756	1 473	138	59 180
	O3	17 814	13 931	697	627	983	680	2 294	160	726	10 633	5 458	1 278	125	55 406
MIPS	O0	28 460	18 843	1 042	906	1 463	734	2 840	200	779	11 866	7 003	2 199	153	76 488
	O1	22 530	13 771	746	653	1 059	670	2 243	176	745	10 940	5 685	1 530	139	60 887
	O2	22 004	13 647	741	653	1 039	667	2 260	171	743	10 952	5 677	1 392	135	60 081
	O3	20 289	12 720	673	584	917	646	2 198	161	724	10 581	5 376	1 197	121	56 187

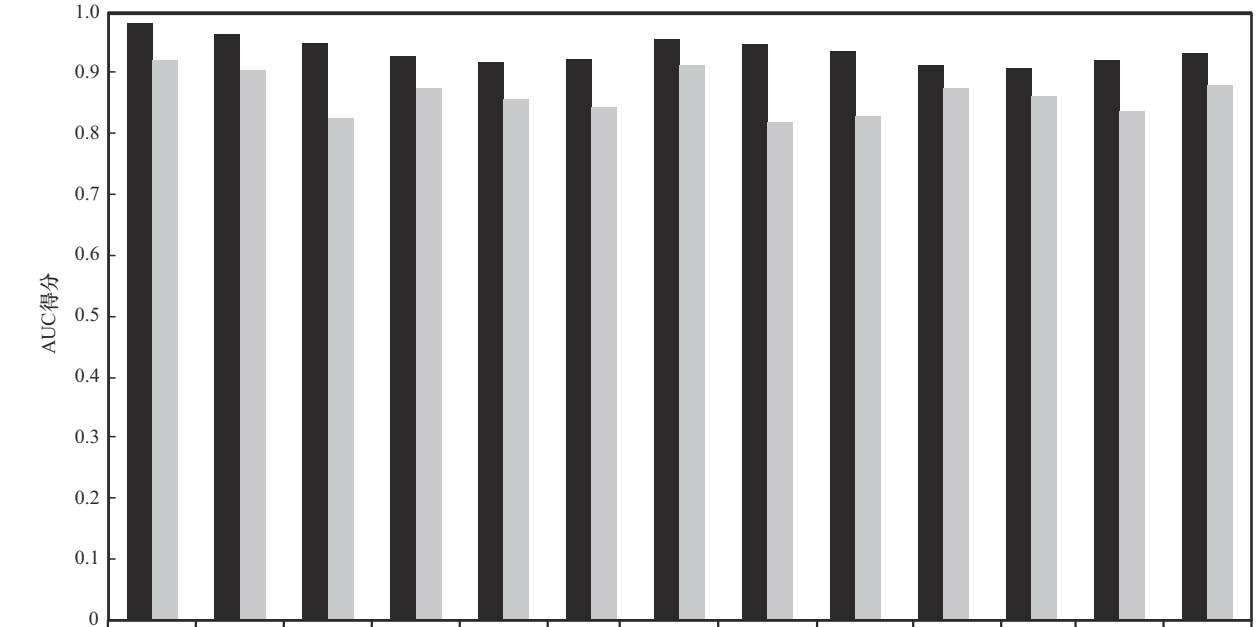
图 7 展示了表 7 中各测试集对应的 AUC（area under curve）得分对比，其中 AUC 得分是衡量模型性能的一个重要指标，它表示 ROC 曲线下的面积。AUC 得分越高，表明模型在区分不同类别（如正例和反例）方面的能力越强。在本实验中，为了直观展示本文方法与其他工具的性能差异，选择性能最优的对比工具 jTrans 进行比较。实验结果表明，本文方法在所有测试集上的平均 AUC 得分较 jTrans 高出 6.7%，表明本文方法在识别和匹配语义相似函数方面具有更高的准确性和可靠性。

在跨优化分析实验中，通过使用从静态汇编代码中提取的嵌入信息进行分析的方法对不同优化级别下的代码语义和结构进行分析。表 8 展示了在不同优化级别下，本文方法与对比工具 jTrans 的性能对比。结果表明，本文方法在精度上平均比 jTrans 高出 7.1%。同时在从 O0 到 O3 的优化级别变化中，代码的语法结构会发生显著变化，这通常会导致 AUC 得分下降。然而本文方法的 AUC 得分下降幅度较小，这表明本文方法在处理不同优化级别的代码时，能够更好地保持分析的稳定性和准确性。

表 8　本文方法和 jTrans 函数相似性得分对比

文件名	O2 和 O3		O0 和 O3	
	本文方法	jTrans	本文方法	jTrans
Coreutils	0.94	0.91	0.92	0.79
Curl	0.97	0.94	0.87	0.86
GMP	0.96	0.96	0.88	0.78
ImageMagick	0.96	0.95	0.92	0.84
LibTomCrypt	0.98	0.97	0.94	0.93
OpenSSL	0.97	0.95	0.93	0.80
Putty	0.98	0.87	0.92	0.82
SQLite	0.95	0.95	0.89	0.79
Zlib	0.87	0.89	0.87	0.86
平均值	0.94	0.92	0.89	0.83

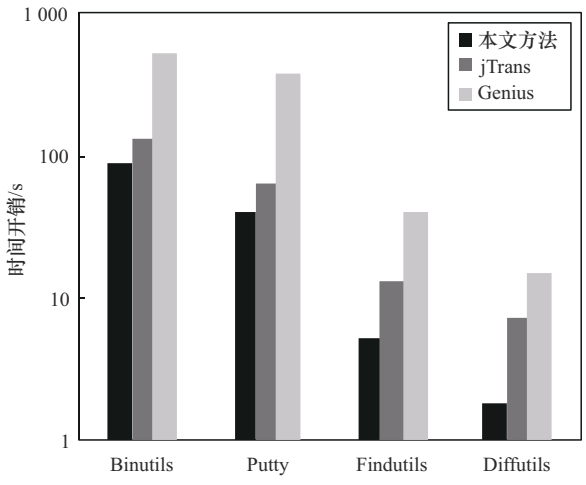
在运行时间方面，从表 7 中选取了具有不同规模的 4 个测试程序，分别为 Binutils、Findutils、Diffutils 和 Putty，这些程序在 x64 平台上编译，优化级别设置为 O3。在匹配过程中详细统计了预训练模型的构建时间和计算函数嵌入所需的时间，以全面评估各方法的效率。



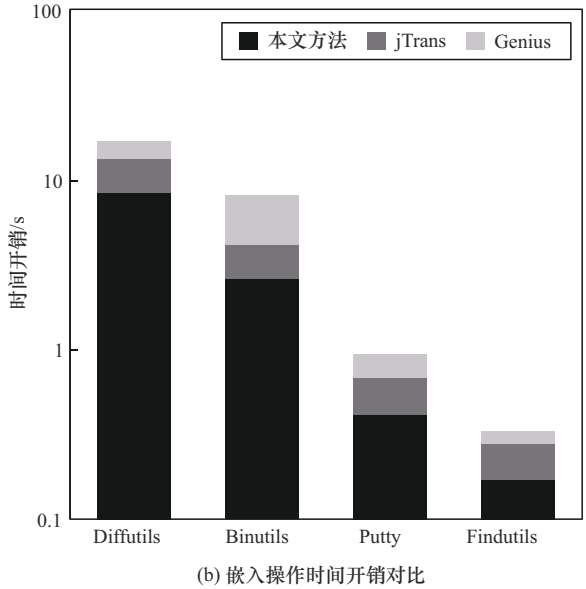
	Binutils	Coreutils	Curl	Diffutils	Findutils	GMP	ImageMag-ick	Libmicroh-ttpd	LibTomCr-ypt	OpenSSL	Putty	SQLite	Zlib
■本文方法	0.981	0.963	0.948	0.927	0.918	0.922	0.956	0.947	0.935	0.913	0.907	0.921	0.933
■jTrans	0.921	0.904	0.826	0.874	0.857	0.843	0.912	0.819	0.828	0.875	0.861	0.837	0.879

图 7　本文方法和 jTrans 在函数识别 AUC 得分方面的对比

图 8 展示了各方法的时间开销对比，其中纵坐标以对数方式缩放，以更清晰地展示不同方法的时间开销。如图 8(a)所示，Genius 由于需要人工构建控制流图并提取基本块特征，导致其时间开销相对较大。相比之下，如图 8(b)所示，本文方法引入了指令规范化操作，与 Genius 使用的循环神经网络和 jTrans 使用的 Bert 模型相比，实现了更少的时间开销。实验结果表明，通过引入指令规范化操作，本文方法在保持分析精度的同时，显著降低了运行时间。为了分析本文方法中各个模块对性能的具体影响，设计了一系列消融实验。首先评估预训练模型在学习和匹配二进制文件执行语义方面的作用，然后分析微执行在预训练过程中的作用。



(a) 函数扫描时间开销对比



(b) 嵌入操作时间开销对比

图 8　本文方法、jTrans 和 Genius 的时间开销对比（以对数为底）

在预训练模型的评估方面，通过比较本文方法在不同预训练函数比例下的 AUC 得分来进行。通过比较在 100% 预训练、66% 预训练、33% 预训练以及无预训练的情况下的 AUC 得分，从而量化预训练函数比例对模型性能的具体影响。

图 9 表明，在无预训练情况下，本文方法的 AUC 得分平均下降了 15.7%，表明预训练函数比例对于提高模型性能至关重要。值得注意的是，当预训练函数比例分别为 100%，66% 和 33% 时，AUC 得分之间的差距非常小，均在 1% 以内。这表明在实验过程中，为了减少时间开销，可以适当减少预训练集的规模，而不会显著影响模型的执行效果。

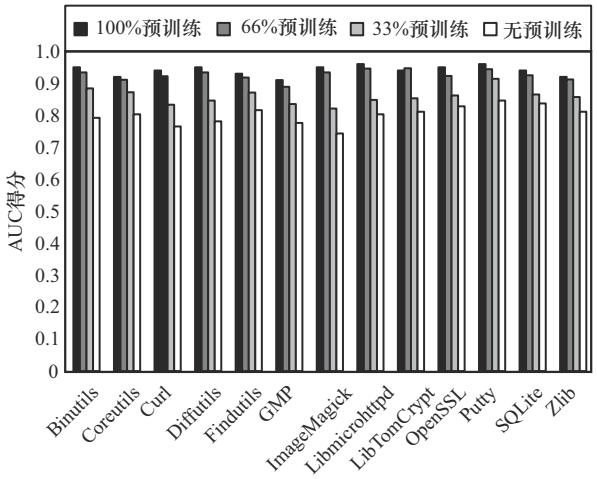


图 9　不同预训练函数比例下的 AUC 得分对比

在微执行方面，由于微执行能够更准确地捕捉执行语义，对具有屏蔽值的数据集进行了预训练，将输入值替换为屏蔽值，并在式(7)中删除了与预训练对应的值，该方法可以用于评估微执行在捕捉执行语义方面的有效性。

图 10 展示了在无微执行的情况下，本文方法的预训练 AUC 得分平均下降了 7.2%。这一下降与 jTrans 的 AUC 得分相比低了 3.5%，表明在无微执行的情况下，本文方法的性能略有下降。然而，值得注意的是，即使在无微执行的情况下，预训练仍然为系统带来了显著的性能提升。具体地，无微执行的预训练导致的性能损失（7.2%）相比于完全不进行预训练的系统的性能损失（15.7%）要小，这种差异明显表明预训练对于提高函数匹配精度具有正向影响。该实验表明即使在微执行

不可用的情况下，预训练也有助于模型学习到更深层次的语义特征，从而能够显著提高模型的性能。

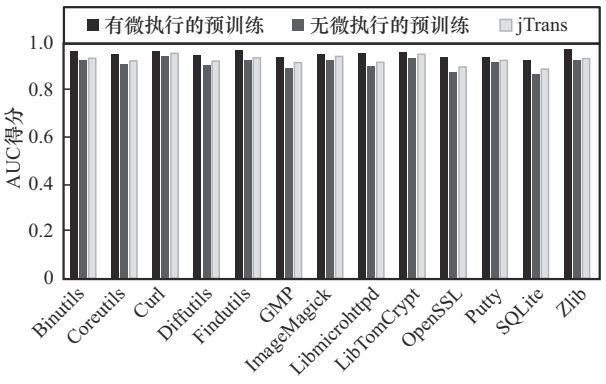


图10　不同测试集的AUC得分对比

5　结束语

本文提出了一个借助统计推理方法来比较二进制文件相似性的新型框架。该框架的分析焦点涵盖了从二进制文件中函数的识别、函数间的相似性评估，以及通过统计推理计算函数的语义相似性等方面。分析工作从二进制文件的函数识别入手，提出了一种基于加权前缀树的分析策略，旨在识别函数的起始点和函数体。这一策略使得反汇编后的文件能够被有效分割，进而在分割的基础上，对函数及整个二进制文件进行层次化的相似度评估。此外，通过邻域搜索策略，将映射关系细化至函数层面，并针对函数基本块的执行语义进行相似性统计分析，以此来估算二进制文件的相似度。

本文存在继续优化的空间，例如，在函数识别阶段，当前缀树的末端节点权重超过0.5时，可以确定函数的起始位置。这一阈值是基于现有对比工具设定的，而本文使用的固定长度指令序列也可能限制了方法的应用范围。未来的研究将考虑根据不同的训练数据集调整阈值和指令长度，以提高函数识别的准确性和效率。

参考文献：

[1] 陈锦富, 王震鑫, 蔡赛华, 等. 基于蜕变测试的区块链智能合约漏洞检测方法[J]. 通信学报, 2023, 44(10): 164-176.
CHEN J F, WANG Z X, CAI S H, et al. Vulnerability detection method for blockchain smart contracts based on metamorphic testing[J]. Journal on Communications, 2023, 44(10): 164-176.

[2] 王金伟, 陈正嘉, 谢雪, 等. 基于Ngram-TFIDF的深度恶意代码可视化分类方法[J]. 通信学报, 2024, 45(6): 160-175.
WANG J W, CHEN Z J, XIE X, et al. Deep visualization classification method for malicious code based on Ngram-TFIDF[J]. Journal on Communications, 2024, 45(6): 160-175.

[3] LIN W, GUO Q L, YIN J W, et al. FSmell: recognizing inline function in binary code[C]//Proceedings of the European Symposium on Research in Computer Security. Berlin: Springer, 2024: 487-506.

[4] KIM S, KIM H, CHA S K. FunProbe: probing functions from binary code through probabilistic analysis[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM Press, 2023: 1419-1430.

[5] YU S, QU Y, HU X C, et al. DeepDi: learning a relational graph convolutional network model on instructions for fast and accurate disassembly[C]//Proceedings of the USENIX Security Symposium. Berkeley: USENIX Association, 2022: 2709-2725.

[6] LIU B C, HUO W, ZHANG C, et al. αDiff: cross-version binary code similarity detection with DNN[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. New York: ACM Press, 2018: 667-678.

[7] FENG Q, ZHOU R D, XU C C, et al. Scalable graph-based bug search for firmware images[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM Press, 2016: 480-491.

[8] DEVLIN J, CHANG M W, LEE K, et al. BERT: pre-training of deep bidirectional transformers for language understanding[C]//Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics: Minnesota. 2019: 4171-4186.

[9] YU Z P, CAO R, TANG Q Y, et al. Order matters: semantic-aware neural networks for binary code similarity detection[J]. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, 34(1): 1145-1152.

[10] WANG H, QU W J, KATZ G, et al. jTrans: jump-aware transformer for binary code similarity detection[C]//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM Press, 2022: 1-13.

[11] DING S H H, FUNG B C M, CHARLAND P. Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization[C]//Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). Piscataway: IEEE Press, 2019: 472-489.

[12] ZHANG X C, SUN W J, PANG J M, et al. Similarity metric method for binary basic blocks of cross-instruction set architecture[C]//Proceedings 2020 Workshop on Binary Analysis Research. Reston: Internet Society, 2020.

[13] YANG S G, CHENG L, ZENG Y C, et al. Asteria: deep learning-based AST-encoding for cross-platform binary code similarity detection[C]//Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Piscataway: IEEE

Press, 2021: 224-236.

[14] YANG J, FU C, LIU X Y, et al. Codee: a tensor embedding scheme for binary code search[J]. IEEE Transactions on Software Engineering, 2022, 48(7): 2224-2244.

[15] DAVID Y, PARTUSH N, YAHAV E. Statistical similarity of binaries[J]. ACM SIGPLAN Notices, 2016, 51(6): 266-280.

[16] BAO T, BURKET J, WOO M, et al. BYTEWEIGHT: learning to recognize functions in binary code[C]//Proceedings of USENIX Security Symposium, 2014: 845-860.

[17] HUANG H, YOUSSEF A M, DEBBABI M. BinSequence: fast, accurate and scalable binary code reuse detection[C]//Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. New York: ACM Press, 2017: 155-166.

[18] PEI K X, XUAN Z, YANG J F, et al. Trex: learning execution semantics from micro-traces for binary similarity[J]. arXiv Preprint, arXiv: 2012.08680, 2020.

[作者简介]

郭曦（1983– ），男，湖北鄂州人，博士，华中农业大学副教授，主要研究方向为软件分析与测试、信息安全、大模型开发等。

王盼（1987– ），女，河南济源人，博士，湖北工业大学副教授，主要研究方向为功率变换器、新能源发电技术、电能质量控制、新型配电网技术等。