基于统计推理的二进制程序语义比较模型

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: 在程序缺陷分析、恶意代码发掘等过程中,通常需要对二进制程序的行为相似性进行分析. 目前基于语法的相似性分析方法忽略了程序的执行语义,存在分析精度不高的问题. 基于语义的相似性分析方法在符号逻辑公式生成过程中,频繁地调用约束求解器进行语义相似性比较,会产生巨大的计算开销. 提出一种基于统计推理的代码相似性模糊匹配分析方法,从指令级别相似度的计算开始,逐级对基本块及函数间的语义相似性进行推理. 首先将二进制代码按照一定的规则划分为具有规范形式的片段集合,在基本块粒度上使用动态规划的方法构建有相同执行语义的存储表,从而获得基本块间的指令初始语义映射. 然后通过邻域搜索的方法将该映射拓展到目标分析函数,并在该过程中提取函数的执行语义. 最后通过对相似函数的结果进行统计分析,进而计算二进制文件的相似度. 同时采用无监督的预训练分析方法,通过调优预训练模型的参数从而提高代码相似分析的精度. 从跨平台及优化选项的角度对 13 个主流的开源项目进行了实验,实验结果表明相较于对比工具,本文方法的分析精度平均提高 7.26%,同时消融实验表明,本文的预训练模型可以有效提高二进制程序语义匹配的性能.

关键词: 程序分析;语义比较;逆向工程;统计推理;迁移学习

基金项目: 国家自然科学基金(No.61502194);国家重点研发计划(No.2023YFF1000100);湖北省教育厅科学技术研究项目(No.Q20211405);湖北工业大学博士科研启动基金项目(No.XJ2021003601)

中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2025)01-0163-19

电子学报 URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20240408

Semantic Comparison Model for Binary Programs Based on Statistical Reasoning

GUO Xi[1], WANG Pan[2*]

(1. *College of Informatics*, *Huazhong Agricultural University*, *Wuhan*, *Hubei* 430070, *China*;
2. *Hubei Key Laboratory for High-Efficiency Utilization of Solar Energy and Operation Control of Energy Storage System*, *Hubei University of Technology*, *Wuhan*, *Hubei* 430068, *China*)

Abstract: In the process of program defects and malicious code discovery, it is necessary to analyze the behavioral similarity of binary programs. Currently, syntax-based similarity analysis methods often ignore the execution semantics of the program, resulting in low analysis accuracy; In the process of generating symbolic logic formulas, semantic based analysis methods frequently call constraint solvers for semantic similarity comparison, resulting in significant time overhead. This article proposes a code similarity fuzzy matching analysis method based on statistical inference for binary programs. Starting from the calculation of instruction level similarity, the semantic similarity between basic blocks and functions is inferred step by step. Firstly, the binary code is divided into a set of fragments with a standardized form according to certain rules, and dynamic programming is used at the basic block granularity to construct a storage table with the same execution semantics for the longest common subsequence, thereby obtaining the initial semantic mapping of instructions between basic blocks; Then, the mapping is extended to the target analysis code through neighborhood search, and the execution semantics of the fragments are learned during this process; Finally, statistical analysis is performed on the results of similar fragments to calculate the similarity of binary codes. During the experiment, an unsupervised pre training analysis method was used to improve the accuracy of code similarity analysis by tuning the pre training model parameters. Experiments were

收稿日期:2024-05-07;修回日期:2024-09-23;责任编辑:王天慧
*通讯作者:王盼

conducted on 13 mainstream open-source projects from the perspective of cross platform and optimization options. The experimental results showed that compared to the comparison tools, the analysis accuracy of our method improved by an average of 7.26%, Meanwhile, ablation experiments have shown that the pre trained model proposed in this paper can effectively improve the semantic matching performance of binary programs.

Key words: program analysis; semantic comparison; reverse engineering; statistical reason; transfer learning

Foundation Item(s): National Natural Science Foundation of China (No.61502194); National Key Research and Development Program of China (No.2023YFF1000100); Technology Research Plan of Hubei Provincial Department of Education (No.Q20211405); Doctoral Research Startup Foundation of Hubei University of Technology (No.XJ2021003601)

1 引言

二进制代码相似检测以二进制文件为输入,在语法、语义等层面上使用相似度比对的方法确认二进制文件之间的相似程度,该技术在程序安全分析、恶意软件检测、软件维护与重用、版权保护等方面有广泛的用途. 例如在恶意软件检测方面,恶意软件在开发过程中通常不会进行全新的开发,而是重用现有的源代码,典型例子包括:木马程序Citadel保留了已泄露源代码的银行木马程序Zeus的核心模块,恶意软件Flame使用了轻量级的数据库引擎SQLite中的功能模块. 故代码相似性检测可以通过分析恶意代码的语义特征,从而高效识别恶意代码之间存在的相似功能. 又如在逆向分析过程中,二进制相似性检测技术可以将未知函数的功能与已知恶意函数库中存在的恶意行为进行比较,从而快速识别可疑的目标代码,并及时检测目前已有或者变种的恶意行为. 故准确而高效的二进制代码相似性检测方法一直以来都是相关领域研究人员的重点研究方向.

然而,二进制代码相似性检测技术存在多方面的挑战,主要体现在:

(1)现有的相似性分析方法难以准确提取程序的语义信息. 目前的分析方法主要分为基于代码和基于图这两种形式,然而基于代码的方法主要针对指令上下文关系开展研究,忽略了数据流等动态信息;而基于图的方法则着重考虑控制流等信息,而缺乏指令之间的语义信息. 故二进制程序在执行过程中可能会根据输入和环境动态改变其行为,仅仅通过静态或动态分析可能无法完全捕捉到程序的所有行为.

(2)二进制程序抽象级别低,缺乏类型等关键信息. 为了提高代码的安全性,减少被恶意攻击及利用的概率,软件通常以二进制的形式进行发布. 有的二进制程序为了增加反汇编后理解的难度,通过编译器的设置删除了调试信息等关键内容. 此外不同项目使用的编译器及优化策略各不相同,从而相同语义的二进制代码在不同的指令集架构平台上有不同的语法表现形式,即有不同的汇编代码形式和控制流结构.

(3)二进制相似性分析过程中的性能和评价问题. 目前不同的检测方法往往针对特定的检测目标,或者使用特定的数据集,并使用诸如ROC(Receiver Operating Characteristic)等相似性评价指标. 而现实分析过程中往往需要处理多样化的数据集,同时需要平衡精确度(避免误报)和召回率(避免漏报)等指标,这些因素对相似性分析的效果会带来较大的影响.

基于以上挑战,本文的研究内容包括:

(1)如何从二进制代码中识别出函数起始位置;

(2)如何在函数级别上对基本块进行相似比较;

(3)如何在文件级别上比较文件之间的相似性.

针对研究目标,本文提出基于统计推理的代码相似性模糊匹配分析方法. 从二进制程序对应的指令开始,将语义相似性的分析和计算逐渐扩展到过程级别,从而依据二进制文件的相似度推理出文件执行语义的相似度.

本文在现实实验环境中进行相似性计算和分析,以主流的开源项目为测试集,从跨平台及不同优化选项的角度进行实验. 实验结果表明,相较于对比工具,本文方法的分析精度平均提高7.26%,同时引入消融实验,以评价预训练模型对相似性分析效果的贡献.

2 研究现状及分析

代码相似性分析由Whale等人[1]于20世纪80年代末提出,主要包含预处理、代码中间表示、匹配单元抽取和相似度度量这几个阶段. 其中预处理阶段主要对代码格式进行规范化处理,删除注释等无效信息;中间表示阶段主要通过编译器等工具提取控制流和语法结构等信息;匹配单元抽取阶段依据具体的目标和需求从不同粒度上对代码进行划分;相似度度量阶段依据不同的粒度分析单元,采用文本、图等方法进行相似度计算.

源代码通常具有类型等丰富的高层语义信息,同时还具有跨平台的特点,而二进制代码与源代码在形式上有较大的不同,体现在其包含了特定的指令结构和优化信息,但缺少高层语义信息. 虽然两者在代码形式上存在较大不同,但是基于二进制的代码相似性检测也遵循上述Whale提出的检测框架. 两者主要区别在于中间表示的方式存在较大差异,基于此,目前二进

制代码相似性检测技术主要分为基于文本、基于特征、基于逻辑和基于语义等几类,以下分别进行总结和分析.

基于文本的分析方法通常对二进制代码中的寄存器名称等信息进行处理,并生成标识符序列从而计算二进制代码的相似性. 例如静态反编译工具IDA Pro通过字符串等标识符定位库文件中的函数,然后在二进制文件中查找该函数从而获得相似的函数列表信息. 类似的还有Asteria-Pro[2],该工具将基本块中的指令序列对应的哈希值作为指纹信息,同时对代码中的符号信息进行匹配操作,但是该方法不能对二进制代码中的操作数进行分析. 基于文本分析的另一种思路是直接对二进制字节流进行处理,例如采用二维卷积神经网络的αDiff[3]对二进制字节进行嵌入操作,通过构建函数出入度来构建特征向量,并计算向量间相似度,此类基于文本的方法难以对抗经过混淆的代码克隆等操作. 由于基于文本的分析方法忽略了语法和语义等信息,故该类方法难以应对代码混淆等场景,通常作为对抗检测的辅助工具.

基于特征的分析方法主要针对二进制代码中属性的特征值,从代码段的角度构建多维特征向量(如函数调用、转移指令、局部变量等),并计算特征向量之间的距离从而衡量相似度. 包括通过人工的方式构建并筛选特征向量,具备跨架构的特征搜索和统计功能[4],该类分析方法通常利用KNN方法计算二进制代码中函数的距离[5],从而减少子图同构匹配过程的复杂度. 类似的还有BinSequence[6],该工具基于指纹和图信息,通过从控制流图中获取基本块节点和边的数量信息,并依此获取函数的调用信息,从而可以快速确定二进制代码的差异. 由于基于特征的分析方法通常依赖人工的方式设置和筛选特征值,且忽略了程序的逻辑结构等信息,从而难以平衡各属性所具有的维度信息,故在实际分析过程中,存在因特征属性种类过多而导致过拟合现象,从而影响相似性检测的精度.

基于逻辑的分析方法从数据流动和函数调用的角度,使用树或者图等结构记录数据的流向和控制转移,在节点或基本块等粒度展开相似性分析,包括使用函数调用序列、控制流图和逻辑树分别表示函数间、函数内和基本块粒度的控制逻辑. 例如MultiMH[7]使用基本块的输入输出信息以及语义签名的方式进行代码漏洞分析;Genius[8]采用基于属性优化控制流图的图匹配算法进行二进制相似性检测,同时诸如Faser[9]、VulHawk[10]等工具在控制流图的基础上加入符号执行的分析方法,从而更加有针对性地进行基本块或函数粒度的相似性检测. 此外有研究引入神经网络相关的技术,如Genius使用机器学习中的谱聚类方法对控制流图进行分类,从而将函数识别问题转换为特征编码的识别问题. 这些基于控制流图的分析方法使其具备了较好的跨语言和跨架构的相似性检测能力,但是此类方法存在诸如控制流提取开销较大、图匹配算法的时间复杂度函数难以在多项式时间内求解等问题.

基于语义的分析方法是目前研究较多的一类相似性检测技术. 它使用类似自然语言处理中的相关技术,并引入神经网络中的嵌入的概念,通过对比和查询嵌入向量的方式进行相似性比对操作,其中间表示包括规范化的中间语言或控制流图. 例如在Gemini[11]中采用基于神经网络的控制流嵌入方法,使用嵌入高维向量并计算向量间余弦相似度的方法,从而获得函数间的相似度. 此外Asm2Vec[12]是首个将表征学习引入到特征向量构建过程中的方法,其使用随机游走的策略,将控制流图以指令序列的形式进行表示,从而在不需要先验知识的前提下构建指令嵌入向量和函数语义嵌入,但其缺点是不能对跨平台的二进制代码进行分析. SAFE[13]方法首先提取汇编代码的指令嵌入,并使用循环神经网络的方法生成指令间的上下文,从而获得函数的嵌入,故该方法能够直接将二进制代码的语义嵌入到高维的特征向量中,从而减少生成控制流图的开销,但是该方法同样难以应对跨平台二进制代码的分析,导致其训练样本库的规模迅速扩大,从而限制了该方法的应用范围. 目前大量的研究采用基于Transformer的预训练模型Bert[14]来展开,例如OrderMatters[7]、PalmTree[15]和jTrans[16]等通过分析汇编代码中的语义信息,从而可以更高效地进行相似性检测,其中jTrans的性能优于目前常见的相似性检测模型. Asm2vec将指令和操作码作为单词并构建神经网络模型,类似的还有基于seq2seq模型的方法[17],Asteria[18]将不同指令架构的函数映射到同一嵌入空间,从而实现跨架构的代码相似性检测,但是该类方法存在缺少控制流信息或者汇编指令间的关系等问题[7,19]. 故基于神经网络的分析方法对于不同的任务,可以采用训练的方法进行反复学习,从而不断调整学习率等参数并优化特征向量的选择. 但是该方法同样存在诸如数据集的选择,以及多维向量所表达的含义难以用数学的方式进行证明等问题.

下面为一个具体的二进制代码相似性分析的示例[20],由于编译器在不同编译条件下所生成的二进制文件可能存在一定的语法变化,但该文件在整体上可以保持源代码和基本块之间的相对稳定. 故本文的相似性比较以基本块为中心开展,通过计算相互映射的基本块之间的相似性得分,并将该相似性得分的结果扩展至函数间的相似性比较. 例如对于二进制OpenSSL的代码片段进行反汇编操作,分别采用gcc4.9

和Clang3.5进行编译，得到图1所示的汇编代码[4].

```
① shr   ebx, 21
② lea   r1, [r5+8h]
③ mov   r7, rax
③ lea   rax, [r7+2]
  mov   [r7+1], al
  mov   [r7+2], r9
  mov   rdi, rax
```

(a) 经gcc编译的目标代码

```
② mov   r2, 25h
③ mov   r11, rax
  add   rbp, 5
  mov   rsi, rbp
③ lea   rdi, [r11+5]
  mov   [r11+3], bl
② lea   r2, [rcx+4]
① shr   ebx, 21
```

(b) 经Clang编译的查询代码

图1 不同编译器生成的汇编代码

如图1所示，不同的编译器产生的汇编代码存在较大差异. 为了提高相似分析的效率，本文首先将待分析的目标代码以基本块为粒度进行划分，再将基本块划分为由ShortString组成的集合(5.1节)，每条ShortString由基本块内对某个符号的值进行计算的若干指令组成. 通过在ShortString级别上进行相似性分析，再依据该分析结果进行目标代码的相似性分析. 图1(*a*)和图1(*b*)中存在3条相似ShortString，分别标记为①②③. 由于语法不同的ShortString的语义可能等价，故当两个ShortString指令计算等效时，则该两个ShortString等效，例如图1(*a*)和图1(*b*)中②对应的指令等效. 同时由于ShortString基于数据流依赖而非语法属性，故允许ShortString在语法上不连续，例如③对应的指令(mov r7, rax; lea rax, [r7+2])与指令(mov r11, rax; lea rdi, [r11+5])相匹配.

基于上述示例和分析，本文提出了一种基于统计推理的二进制程序相似性比较方法，其框架如图2所示，主要包含函数识别、函数相似度计算和语义相似度分析这三个部分，分别对应本文第3、第4和第5节. 其中函数识别模块通过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方法识别函数的边界. 函数相似度计算模块将识别出的函数指令经规范化操作，在指令级别计算其相似度，并扩展至基本块粒度进行相似度的计算. 其中通过启发式贪心策略对函数中的数据流信息进行搜索，从而获得局部函数的相似度信息. 在此基础上，语义相似度分析模块将待分析的函数进行分解，并通过统计推理的方法计算全局相似语义度.

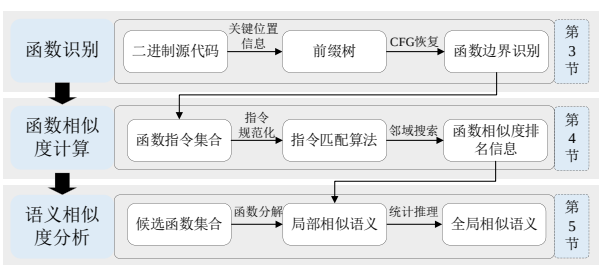


图2 本文方法框架

本文的主要贡献如下：

(1)针对无调试信息的二进制文件中函数识别精度较低的问题，提出基于前缀树及增量控制流分析的函数识别方法(对应研究目标1)；

(2)针对不同平台及优化选项生成的二进制代码难以交叉理解的问题，在函数粒度上提出基于模糊匹配的函数相似度比较方法(对应研究目标2)；

(3)在第2点基础上依据函数的相似度，通过二进制语义相似分析模型，推理并计算二进制文件的相似度(对应研究目标3)；

(4)开发了对应的实验原型系统，实验结果表明本文方法在函数识别方面的精度为94.2%，函数相似度得分平均提高7.26%.

3 基于前缀树及增量控制流分析的函数识别方法

函数识别是二进制重写和控制流完整性等操作的重要前提. 而现有的研究方法往往直接跳过该过程而直接进行后续的相似性分析，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[21]，以及采用启发式的方法来识别函数的边界[22,23]，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[24]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，对GPU等计算资源有较大的开销需求.

本节通过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.

3.1 问题分析

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，同时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等操作而引入额外的指令. 此外，由于跳转表等结构以及函数之间可能存在共享代码的情况，也导致函数在地址空间上可能不连续.

3.2 函数识别

函数起始识别问题可以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. 函数识别的整体框架如图3所示，其中包含训练阶段、分类阶段、函数确认阶段和函数边界确认阶段.

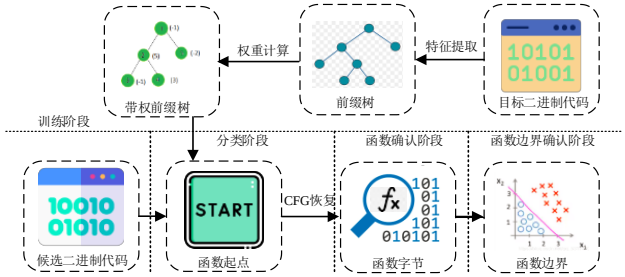


图3　函数识别框架

其中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$|\mathrm{TP}|/(|\mathrm{TP}|+|\mathrm{FP}|)$,从而获得顶点的权重值. 由于前缀树是一种高效的存储字符串集合的树型数据结构,它利用字符串之间的公共前缀,将重复的前缀合并在一起,这种特点使其常用来在一组字符串集合中快速查找某个字符串. 依据定义,前缀树上除根节点外每个节点都应有一个标识来表明其是否是一个词的结尾,同时叶子节点为一个字符串的结尾. 这些特点使得前缀树能够较好地满足函数识别问题,即识别函数的起始位置和函数的执行边界. 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.

在确定了函数起始位置后,通过静态的CFG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CFG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. 其中,对于调用指令,可达的指令包括该指令的下一条指令及被调用的指令. 对于跳转指令,则将跳转到的指令作为可达指令. 对于直接跳转或者函数调用等直接转移,使用递归式的反汇编进行分析. 而对于间接调用等间接转移,通过使用增加边的方式对控制流进行增量确认,从而计算函数的起始位置.

对于图3的训练阶段,其输入为候选函数对应的二进制文件T,以及用来设置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及结束位置信息(s,e)获得函数的边界. 该过程主要分为以下步骤:

(1)指令提取阶段. 对于从调试信息中获取的函数起始位置和对应的结束位置集合,从起始位置开始提取长度为l的指令集合,若当前(s,e)中指令长度小于l,则取(s,e)中所有的指令进行训练. 即对每个函数的(s,e)提取其前l个指令$B[s:s+l]$,若(s,e)的指令长度小于l,则保留整个(s,e)的信息.

(2)前缀树生成阶段. 依据前文所描述的前缀树的特点,通过第1阶段所产生的指令集合构建前缀树,从根节点开始一直延伸到叶子节点的路径就代表了一组指令序列. 此时l为指令序列长度的最长值,在实际分析过程中指令长度可能不为l,若长度小于l,则直接使用实际的长度进行训练,若长度大于l,则使用l进行训练. 在实验过程中,指令序列经过规范化操作,由于规范化操作使匹配操作可以处理相似但不等价的指令,该特点使得本文方法可以有效提高分析精度和召回率. 对长度为l的指令序列生成如图4所示的前缀树,树中每个非根节点n代表指令序列中的一条指令,此时函数对应的指令的执行过程可以表示为从根节点到n的指令序列.

(3)权值计算阶段. 在具体实验过程中将函数集划分为训练集和测试集,故通过为函数训练集中每个前缀树节点分配一个权值,从而判断第2阶段所生成的前缀树中语句序列(最长l个指令)是否为函数的起始位置. 权重计算过程如下,对于整个训练集T,若前缀树中的节点不为T中的函数起始位置,则将其存入集合T_-中,否则将其存入集合T_+中,此时记录T_+中真实函数起始的数量,注意到该过程中可能存在错误的匹配,即前缀树中某条路径对应的指令序列不为真实的函数起始,此时需要降低该权值,即设置该节点的权重$w=T_+/(T_++T_-)$. 如图4所示[23],节点push %ebp的权重为0.172,表明在训练过程中,对于带有push %ebp前缀的指令序列,其为函数起始的概率为17.2%.

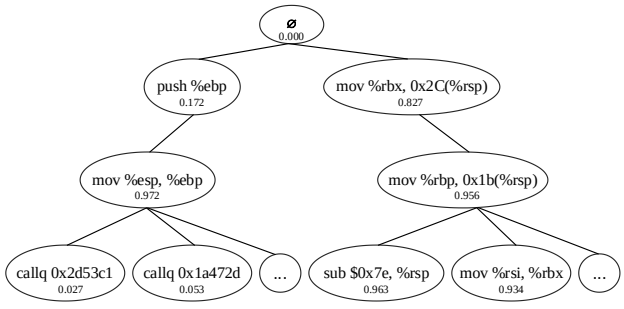


图4　未规范化的前缀树

对于规模较大的前缀树,采用优化策略对其进行简化从而增加匹配的速度. 具体地,若某节点的T_-为0,则删除该节点的所有子节点. 若一个子节点开始的指令序列与一个函数起始不匹配,则从其父节点开始的指令序列也必然与函数起始不匹配. 同理,若父节点无任何错误匹配,则其子节点也无错误匹配,故子节点的T_-值不为0,同时也不会超过其父节点的T_-值. 依据节点权重计算公式,若$T_-=0$同时$T_+>0$,此时权重为1,

故对于有相同权值的子节点也可以进行删除，且不影响分类精度.

在实际分析过程中，有的代码由于自身的逻辑问题导致存在不可达的函数，而在优化的时候没有使用相关的规则移除这些死代码，从而导致在二进制代码中存在不可达的函数. 可能的原因包括编译器在优化过程中可能对函数进行预处理操作，并作为内联函数进行后续分析，故在二进制代码中没有该函数的显式调用. 编译器对于规模较小的函数可能会通过函数内联的方法将其删除从而减少函数调用的开销.

4 基于模糊匹配的函数相似度比较方法

在识别汇编代码中的函数的基础上，本节重点介绍函数相似度的计算方法，包括匹配模型及比较算法. 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分析的方法上升到函数级别.

4.1 模型描述

图5为函数相似度计算的模型框架[6]. 主要流程为首先对于待分析的二进制文件，依据第3节的函数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，在匹配操作阶段，通过对候选函数使用最长公共子序列的动态规划方法进行指令比较，然后使用路径搜索和邻域搜索的方法，输出待匹配函数的候选集.

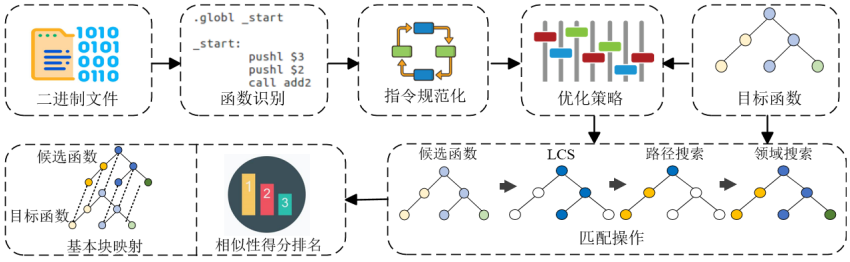


图5 相似度计算模型

为了更好描述该模型，给出如下相关概念：

动态规划DP(Dynamic Programming). DP是一种求解多阶段决策过程最优化问题的方法. 其核心思想为将原问题划分为若干子问题，并使用自下而上的递推方法或者自上而下的记忆化方法对子问题进行求解，并将已解决的子问题的结果保存在一张表格中，后续子问题的求解过程则可以直接参考表中的信息. DP中Programming指的就是一种表格处理过程，即将每次计算的结果保存在一张表格中，供后续计算过程查找使用. DP与分治法的区别在于，DP分解后的子问题之间往往存在一定的内在关联，即子问题之间可能会出现重叠的现象，而分治法则不存在内在关联.

最长公共子序列LCS(Longest Common Subsequence). LCS指的是在序列集合中，寻找所有序列中的最长子序列的过程. LCS与最长公共子串的区别在于，LCS不需要子序列中的元素在原序列中连续，LCS的求解过程可以通过上述DP的方法进行解决，其存储表中每个单元包含了两个较短序列的LCS长度. 通过不断填充该表，从而逐步计算出最终的最长公共子序列，LCS递归形式为

$$C[i,j]=\begin{cases}0, & \text{if } i=0 \text{ or } j=0\\ C[i-1,j-1]+1, & \text{if } i,j>0 \text{ and } x_i=y_i\\ \max(C[i,j-1],C[i-1,j]), & \text{if } i,j>0 \text{ and } x_i\neq y_i\end{cases}$$

其中符号$C[i,j]$用来记录两个序列$X_i=\{x_1,x_2,\cdots,x_i\}$，$Y_j=\{y_1,y_2,\cdots,y_j\}$的LCS的长度.

宽度优先搜索BFS(Breadth First Search). BFS主要针对树、图和矩阵等结构，从顶点开始逐层地向外搜索，直到达到目标节点或者搜索时间上限. 在BFS过程中通过建立一棵宽度优先树，搜索开始时只包含根节点s，在扫描已发现顶点u的邻接表的过程中每发现一个顶点v，该顶点v及边(u,v)就被添加到树中. 搜索对象的节点在BFS策略下会维持一个队列，具有先进先出(first in first out)的特点，使得BSF方法适用于求与源节点有最近距离的目标节点，故上述LCS的计算过程中采用BFS的搜索策略.

通过候选集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，在匹配过程中使用最长公共子序列LCS的动态规划方法获得目标函数与候选函数的执行路径，从而得到基本块之间的初始映射关系. 在此基础上，为了提高函数匹配的效率，使用带有模糊匹配策略的邻域搜索方法，对已获得的基本块间的映射进行拓展，从而获得函数间的相似性得分.

4.2 反汇编的规范化

在反汇编阶段为每个二进制文件生成基于基本块的控制流图. 在不同编译环境中，编译器在助记符、寄

存器编号的选择方面有较大的不同,故为了比较不同编译环境中的汇编指令,需要将基本块中的指令进行规范化处理.

在同一体系结构下指令规范化过程保持助记符不变,通过对立即数和寄存器变量进行规范化操作,可以匹配不同环境中相似的指令,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	调用	libc库调用	libc[name]
		递归调用	self
		函数调用	func
	跳转	跳转的目标	jmpdst
	引用	字符串	dispstr
		变量	dispbss
		数据(非字符串型)	dispdata
	默认值	其他立即数	immval
寄存器	数量	[e|r]*[a|b|c|d|si|di][x|l|h]*, r[8~15][b|w|d]*	reg[1|2|4|8]
	栈/instruction	[e|r]*[b|s|i]p[1]*	[s|b|i]p[1|2|4|8]
	特殊功能	cr[0-15], dr[0-15], st([0-7]),[c|d|e|f|g|s]s	reg[cr|dr|st] reg[c|d|e|f|s]s

经过规范化后的前缀树可以匹配相似但不是完全等价的指令,故assign、sub、sum等指令均可以被匹配为函数起始位置[23]. 分类阶段的输入为训练阶段生成的前缀树、候选二进制序列B以及权重. 对B的反汇编指令序列进行规范化处理,并对前缀树进行匹配操作. 执行过程的权重由最后一个匹配的节点决定.

4.3　指令比较

指令作为最小的执行单元,通常由助记符、操作数和常量这3个部分组成,它们在指令相似性比对过程中携带不同的信息量. 算法1为指令匹配算法[6],为了提高匹配效率,需要对指令进行上述规范化操作.

由于助记符包含具体的执行动作,若其对应的操作数在规范化操作后相同,则对相同的助记符赋予更高的权重得分. 对于常量操作数,若其数值相同,则在指令比较过程中具有明显的识别特征,则会额外增加相似度得分. 由于助记符包含具体的执行动作,若其对应的操作数在规范化操作后相同,则对相同的助记符赋予更高的权重得分. 对于常量操作数,若其数值相同,则在指令比较过程中具有明显的判断特征,故会额外增加其相似度得分.

4.4　路径搜索

一个基本块由若干指令组成,控制流图CFG中一个节点可能有若干个后续节点,而在最长公共子序列

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
Function CompIns(ins1, ins2)
  score = 0
  if ins1.mnemonic == ins2.mnemonic then
    n = numof(operands(ins1))   //操作数的数量
    score += MNEMONIC
    foreach i do
      if operand(ins1)[i] == operand(ins2)[i] then
        if type(operand(ins)[i]) == CONSTANTS then
          score += CONSTANT
        else
          score += OPERAND
        endif
      endif
    endfor
  else
    score = 0
  endif
  return score
end
```

中每个节点只有一个后续节点. 为了获得候选集中与最长公共子序列有最高匹配程度的执行路径,在基本块间相似度的基础上,采用上述宽度优先遍历的分析方法进行路径搜索.

对于候选集的控制流图CFG,算法2进行路径搜索操作[6],将具有相同执行语义的节点保存在存储表t中. 具体地,算法2以目标函数的最长公共子序列P以及候选函数的CFG为输入,采用类似最长公共子序列的计算方法,在候选函数的路径搜索阶段采用动态规划的方法进行分析[6]. 由于路径搜索是NP问题,算法在初始化时设定存储表长度为1,每次选择候选函数路径中的第一个节点,在存储表中新增一行并使用updatePath()函数更新存储表. 序列s保存每个节点在当前状态下的最高相似得分,当存储表更新后,同步更新序列s中的得分,以保证s中始终是每个节点的最高得分,并将该节点的后续节点插入到待分析的候选路径中.

4.5　邻域搜索

由于路径搜索有较大的时间开销,同时对于已经分析过的基本块,在后续不同的路径分析过程中会重复匹配,从而影响目前基本块映射方法的效率. 故对前文中获得的基本块映射结果,使用模糊匹配的分析方法进行拓展,即通过邻域搜索的方法提高发掘函数之间基本块映射的效率.

首先依据相似性得分将路径搜索产生的一对匹配

算法2 路径搜索

输入：目标函数的最长路径P,候选函数的CFG图G
输出：存储表t
Function PathExploration(P, G)
  t = InitTable(1, |P|+1)
  //s用于保存G中每个节点最大得分的序列
  s = InitArray(|G|)
  Q = InitQueue()
  foreach Q is not empty **do**
    currNode = Q.front()
    Q.popFront()
    t.addNewRow()
    updatePath (currNode, P)
    if s(currNode) < t(currNode, |P|) **then**
      s(currNode) = t(currNode, |P|)
      foreach successor s of currNode **do**
        Q.pushBack(s)
      endfor
    endif
  endfor
    return t
end

基本块(b_1,b_2)放入队列Q中,每次从队首取出相似性得分最高的一对基本块,对其近邻的基本块进行分析. 由于候选函数由若干基本块组成,故可以将函数内所有基本块的自反映射得分进行累加,从而获得当前函数的自反映射得分. 例如对于函数f和g,其相似度可以用式(1)进行计算[6]:

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

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,Score(·)函数计算当前函数的自反映射得分. 式(1)反映出在给定两个候选函数的条件下,通过计算所有(u,v)相似度得分的总和,以及f和g各自对应的自反映射得分,从而计算出函数f和g之间的相似度得分.

对于具有较大规模的候选函数,若对所有函数中的路径进行相似性分析及邻域搜索将会导致极大的开销,故必须采用一定的优化规则提高匹配效率. 本文使用启发式的分析方法从基本块数量和过程指纹相似度的角度进行分析:对于基本块数量的优化规则,由于采用了模糊匹配的方法,对于不完全同构的CFG也可以进行基本块级别的匹配,故通过设置阈值γ调整匹配过程. γ的值应在合理的范围,若太小则会漏掉匹配的基本块,若太大则会极大增加匹配过程的开销,即对于函数f,其基本块数量介于$(|f|-\gamma,|f|+\gamma)$之间的候选过程会被认定为具有较高的相似性.

为了减少匹配的次数,采用Minhashing和banding的方法[6],其中Minhashing采用k个不同的哈希函数生成minhash签名,banding方法将minhash签名划分为b个band,其中每个band有r行数据. 对于目标过程f,计算其指令指纹和minhash签名,则候选过程的minhash签名至少与f中minhash的某个band中的签名一致,在实验过程中采用文献[25]中的方法,将banding的Jaccard阈值设置为$\left(\frac{1}{b}\right)^{\frac{1}{r}}$.

5 基于统计推理的二进制语义相似分析模型

二进制文件相似性比较的主要任务是语义相似性分析,但实际的二进制文件存在不同的语法形式,从而会对相似性分析的效果带来较大干扰. 本节通过将二进制代码分解为代码片段(ShortString)的集合,并计算局部相似性和全局相似性得分,最后使用程序验证器对语义进行相似性分析.

5.1 示例及分析

如第1节所述,为了提高相似度计算效率,本文将代码分解为若干规模较小的代码片段ShortString,每个片段由一系列对操作数进行计算的指令组成,首先计算ShortString的相似度得分,依据该得分使用统计推理的方法计算代码之间的相似度.

图6为ShortString的示例,并将其转化为中间验证语言IVL(Intermediate Verification Language)形式[4],需要注意的是,图6中使用了下标q和t进行别名操作,以区别查询ShortString和目标ShortString以及每一次的执行过程. 在各ShortString的输入中构建等价假设(assume),在输出处添加判断变量相等的断言(assert),通过程序验证器对断言进行检查.

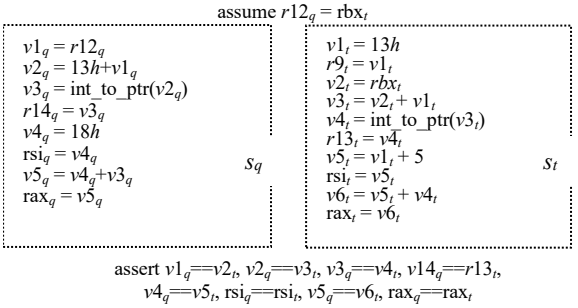


图6 语义相似的ShortString

为了量化ShortString之间的匹配程度,记$\mathrm{SIM}(s_q,s_t)$为ShortString s_q中的指令在ShortString s_t中有与其等价的指令的百分比,若s_q与s_t具有相似的"输入-输出",则可以基于$\mathrm{SIM}(s_q,s_t)$计算条件概率$\Pr(s_q|s_t)$(5.3节).

对于图6中左侧的查询ShortString s_q及右侧的目标ShortString s_t,因s_q中每个变量在s_t中都有等效的变量,即$\Pr(s_q|s_t)=1$,而$\Pr(s_t|s_q)=\frac{8}{10}$. 通过计算$\Pr(s_q|s_t)$在目标函数$t$中所有ShortString s_t的最大概率来表示s_q在t中的匹配程度.

由于编译器在优化操作过程中可能会引入公共的ShortString,这些ShortString会影响相似度计算的得分,故通过计算ShortString内的局部相似得分$S_{\text{local}}(s_q|t)=\log\frac{\max_{s_t\in t}\Pr(s_q|s_t)}{\Pr(s_q|H_0)}$,从而衡量与$s_q$等价的ShortString的重要程度. 为了使计算的结果更加直观地反映数据间的差异程度,使用对数操作将该结果转换为平滑的线性变化百分比数值,将数据分布压缩到一个较小的范围之内,从而减少极端值对分析结果的影响[4].

对于两个函数q和t,由于$S_{\text{local}}(s_q|t)$可以用来计算s_q在t中的重要程度,其计算过程通过5.2节中的式(6)进行表示. 通过对q中所有s_q的$S_{\text{local}}(s_q|t)$进行求和,从而计算q和t之间的全局相似得分S_{global}. 基于此可以将ShortString之间的局部相似度,通过统计推理的方法计算全局函数之间的相似度$S_{\text{global}}(q|t)$,其计算过程如式(2)所示[4].

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

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

5.2　函数分解

依据5.1节相关概念,程序中某个待分析的函数可以通过由基本块组成的序列进行表示,所以函数中保留了程序的控制结构关系.

算法3通过逆向迭代基本块的方法生成ShortString[4],该算法首先将基本块中的指令序列放入集合unusedInsts中,算法在所有指令都被放入一个ShortString的时候停止执行. 针对最后一个未被使用的指令,计算ShortString中定义的变量集合varsDefed和引用的变量集合varsRefed. 在for循环体中加入ShortString里定义变量的指令,并更新varsRefed和varsDefed. 当循环结束时,将基本块中所有用来计算变量的指令都保存在ShortString中. 特别的,若一个函数中某个ShortString可以由另外一个函数中的ShortString组成,则这两个函数是相似的.

算法3　从基本块中提取 ShortString

```
输入: 一个基本块对应的指令序列 b
输出: b 对应的 ShortString
unusedInsts = {1,2,⋯,|b|}
ShortString = [·]
while unusedInsts ≠ 0 do
  maxUsed = Max(unusedInsts)
  unusedInsts \= maxUsed
  newShortString = b[maxUsed]
  varsRefed = Ref(b[maxUsed])
  varsDefed = Def(b[maxUsed])
  foreach i in maxUsed do
    needed = Def(b[i]) ∩ varsRefed
    if needed ≠ 0 then
      newShortString += b[i]
      varsRefed ∪= Ref(b[i])
      varsDefed ∪= needed
      unusedInsts \= i
    endif
  endfor
  inputs = varsRefed \ varsDefed
  ShortString += (newShortString, inputs)
endwhile
```

5.3　ShortString置信度分析

对于函数q和函数t,各自对应的ShortString分别表示为s_q和s_t,则s_q在t中能够找到与其等价的ShortString的概率$\Pr(s_q|t)$可以表示为

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

其中条件概率$\Pr(s_q|s_t)$可以通过使用式(4)中sigmoid函数对s_q和s_t的匹配程度$\text{SIM}(s_q,s_t)$进行计算来获得[4]:

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

类似于分类问题中的逻辑回归算法,使用sigmoid函数可以将相似度的结果压缩到[0,1]的范围内,并将sigmoid函数的中值点x_0设置为0.5,该逻辑函数可以对概率进行相似性计算,使输出的结果更加平滑,即当$\text{SIM}(s_q,s_t)=1(0)$时,$\Pr(s_q|t)$的值趋近于1(0). 在实际分析过程中,一个s_q除了要与目标s_t进行匹配操作,还需要计算在整个二进制文件范围内随机匹配的概率,以及由此带来的影响. 为了计算代码片段的随机匹配概率,依据式(3)给出相似比SR(Similarity Ratio)的定义[4]:

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

其中 $\Pr(s_q|H_0)=\dfrac{\sum_{s_t\in T}\Pr(s_q|s_t)}{|T|}$ 表示对所有 $\Pr(s_q|s_t)$ 计算平均值，T 为所有目标 ShortString 的集合，该式反映出 s_q 在函数 t 中找到等价语义的概率与 s_q 在随机匹配中找到等价语义概率的比值. 从而获得随机语义的匹配值. 依据式(5)，S_{local} 的相似概率为

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

该局部相似得分反映出在函数 t 中，存在与 s_q 在语义上等价的 ShortString 的置信度[4]. 而全局相似 $S_{\text{global}}(q|t)$ 是所有 $S_{\text{local}}(s_q|t)$ 的总和，该置信度反映出 q 可以由 t 中的某些相似的 ShortString 组成.

5.4 语义相似度计算

设程序 P 在某时刻的状态 $\sigma_i\in\Sigma_P$（Σ_P 为全部状态集合）为序列 $(l_i,\text{var}_i,\text{val}_i)$，其中 l_i 为变量 var_i 在该时刻的位置信息，val_i 为 var_i 在该时刻的具体值. 程序的一次执行 $\pi\in\Sigma_P^*$ 是由 $\sigma_0,\sigma_1,\cdots,\sigma_n$ 组成的序列，同时设程序所有可能的执行为 P，π 的第一个和最后一个状态分别为 first 和 last.

记两个状态 σ_1 和 σ_2 之间的关联为 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 $\forall(v_1,v_2)\in\gamma$: $\sigma_1(v_1)=\sigma_2(v_2)$，则称 σ_1 和 σ_2 在 γ 上等价，记为 $\sigma_1\equiv_\gamma\sigma_2$.

对于两个执行 π_1 和 π_2，若 π_1 和 π_2 在 γ 上等价，则记为 $\pi_1\equiv_\gamma\pi_2$. 一对程序 (P_1,P_2) 的所有变量关联记为 $\Gamma(P_1,P_2)$. 此外给出以下定义.

定义 1 ShortString 之间的等价性[4]. 对于两个 ShortString s_1 和 s_2，若 s_1 和 s_2 相互等价，即 $s_1\equiv_\gamma s_2$，当且仅当：

(1) s_1 中的每个输入都能在 s_2 中找到与之对应的某个输入；

(2) 一对执行 $(\pi_1,\pi_2)\in(s_1,s_2)$ 在输入处等价，即 $\pi_1\equiv_\gamma\pi_2$，则存在如下关系：

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

定义 2 ShortString 的相似度 SIM. 对于两个 ShortString，它们之间的相似度 SIM 为在 γ 上有最大关联匹配变量的比率[4]，即

$$\text{SIM}(s_q,s_t)=\frac{\max\left\{|\gamma|\,|\forall(\pi_q,\pi_t)\in(s_q,s_t):\pi_q\equiv_\gamma\pi_t\right\}}{\left|\text{var}(s_q)\right|} \tag{7}$$

式(7)所表示的相似度 SIM 的意义在于它直接对 ShortString 中的核心语义进行相似度计算，减少仅通过 ShortString 输出的值来判断 ShortString 之间的相似性所带来的误差，从而可以为潜在无关联的代码产生匹配得分. 对于状态 σ_q 和 σ_t，记 σ_q 中有匹配值的比例为 $\text{SIM}(\sigma_q,\sigma_t)=\dfrac{\gamma_{\max}}{\sigma_q}$，其中 $|\gamma_{\max}|$ 为 σ_q 和 σ_t 相互等价的关联的最大值，即 $\sigma_q\equiv_{\gamma_{\max}}\sigma_t$.

依据式(7)，ShortString 的 SIM 计算过程如算法 4 所示[4]. 该算法中，对于 γ 中每一对程序输入，算法增加等价假设以及查询函数 p^q 和目标函数 p^t，获得 p^q 和 p^t 的执行语义，最后在 last 状态将断言添加到与 γ 相匹配的变量中.

算法 4 计算 ShortString 的 SIM 值

输入：具有 Boogie IVL 形式的查询 ShortString p^q，目标 ShortString p^t
输出：SIM(p^q, p^t)

```
maxSIM = 0
foreach γ∈Γ(p^q, p^t)
  p = NewProcedure(Inputs(p^q)∪(Inputs(p^t))
  foreach (i^q, i^t)∈(r∩(Inputs(p^q)∪Inputs(p^t))) do
    p.body.Append(assume i^q == i^t)
  endfor
  p.body.Append(p^q.body, p^t.body)
  foreach (v^q, v^t) ∈(((vars(p^q)×vars(p^q))∩γ) do
    p.body.Append(assert v^q == v^t)
  endfor
  Solve(p)
  if p^q ≡_γ p^t then
    maxSIM = Max(|r|/|vars(p^q)|, maxSIM)
  endif
end
```

5.5 进一步讨论

以上相似性分析的对象为单一平台对应的二进制程序，在实际分析过程中，不同的平台以及编译器使用的二进制程序之间存在一定程度的差异. 为了屏蔽这种差异带来的影响，4.2 节中使用了指令规范化的操作方法，为了使本文方法具有更好的泛化功能，本节首先介绍函数微执行的概念和作用，然后使用基于迁移学习的模型对不同体系结构的指令进行分析.

5.5.1 函数微执行

为了对不同体系结构上使用不同编译器和优化选项所生成的执行文件进行相似度比较，提出一种迁移学习的模型，对由指令及状态值组成的微执行序列进行相似度计算. 不同于 5.4 节中直接使用 ShortString 进行相似度比较，该框架无需人工标记，而是通过无监督的预训练模型，从微执行序列中收集欠约束的指令集合，同时学习指令在上下文中的执行语义.

为了屏蔽不同架构上汇编语法的差异，通过使用以下中间表示 IR 的方法[26]来表示不同架构的汇编

语法.

Function f ::= $[i_1, i_2, i_3, \cdots]$

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 ::= $\{=, -, *, /, <, >, \cdots\}$

Register r ::= {pc, sp, eax, $\cdots$}

Const c ::= {true, false, 0x0, $\cdots$}

在 IR 的基础上,通过算法 5 生成函数 f 的微执行[26],将函数 f 中所有指令存入一个队列中,并对栈指针及首指令地址等进行初始化操作. 依据顺序执行的方式执行 f 中的指令,当队列中所有指令执行完毕则算法终止运行.

算法 5　计算函数的微执行

```
输入: 二进制函数f,寄存器r
输出: 微执行t
L = getInstructions(f) //将f中指令放入队列中
t = emptyVector
foreach register r_i in r\{sp,pc} do
  r_i = randomInit()//初始化寄存器值
endfor
while L ≠ Φ
  i = dequeue(L)
  if i.type == load or i.type == store then
  memMap(i.accessAddr, i.accessSize)
    if i.type == load then
      writeRandom(i.accessAddr)
    endif
    t = t∪execute(i)
  elseif i.type == jmp or i.type == call then
    if i.targetAddr not in [cm.minAddr, cm.maxAddr] then
      continue
    endif
    t = t∪execute(i)
  elseif i.type == nop then
    continue
  elseif t.type == ret then
    break
  else
  t = t∪execute(i)
  endif
endwhile
```

5.5.2　迁移学习模型

图 7 为对应的预训练模型,主要包括处于图中间位置的输入模块,其输入 x 由 5 个序列 x_f, x_t, x_c, x_o, x_a 组成,每个序列的大小均为 n. 其中 x_c 表示指令位置的整数序列,x_o 表示当前指令中操作码或操作数的位置编码序列,x_a 表示指令集的架构序列[26].

图 7 中输入的灰色部分表示被屏蔽的指令或者操作数,在预训练阶段损失函数由代码预测和值预测的交叉熵损失组成,其中值预测由 8 位微执行的值组成. 由于对微执行进行操作而不需要额外的标签信息,故该预训练模型为无监督模型. 该图左侧为 bi-LSTM 模块,其中将微执行的值作为嵌入值进行分析. 该图右侧为自注意层和输出模块,对指令和值进行预测操作. 对于输入模块,第一个序列 x_f 由微跟踪算法中标记的汇编代码组成,即 x_f={mov, eax, $\cdots$}.

输入序列 x_t 为微执行的动态值序列,即在指令执行前每个符号用动态值,例如对于 mov eax, 0x1c; mov eax, 0x5e,在执行 mov eax, 0x5e 之前,eax 中的值为 0x1c. x_{ti} 表示 x_t 中第 i 个值,x_{ti} 为 8 位固定长度的序列{0x00, $\cdots$, 0xff},然后将 x_{ti} 反馈到双向 LSTM(Bi-LSTM),并将最后一个位置的值作为 embedding 以表示值 t_i=Bi-LSTM(x_{ti}),其中 t_i 为将 embedding 作用到 x_{ti} 所产生的输出.

设 $m(E_i)$ 为被屏蔽的 x_i 的嵌入,集合 MP 记录被屏蔽的位置信息. 预训练模型 g_p 将嵌入序列 $(E_1, \cdots, m(E_i), \cdots, E_n), i \in \mathrm{MP}$ 作为输入,其令牌随机进行屏蔽. 对于 g_p 的参数 θ,预训练的目标是搜索 θ,减少

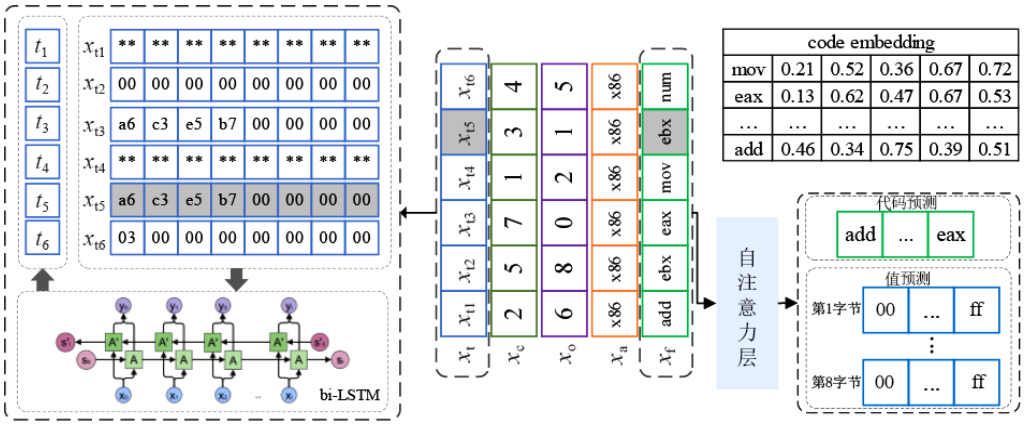


图 7　基于微执行的迁移学习模型

预测代码令牌与预测值之间的交叉熵损失：

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

其中$\hat{x}_{t_{ij}}$表示x_t中第i个令牌x_{ti}中的第j个字节，参数α用来衡量预测代码令牌和预测值之间的交叉熵损失[26]. 同时式(8)表示每个嵌入有9个感知器，其中1个感知器用于代码预测，而其余8个感知器用于字节预测.

为了对函数相似性进行微调，将两个函数作为预训练模型g_p的输入，生成由g_p中自注意层产生的嵌入序列$E_k^{(1)}=\left(E_{k,1}^{(1)},\cdots,E_{k,n}^{(1)}\right)$和$E_k^{(2)}=\left(E_{k,1}^{(2)},\cdots,E_{k,n}^{(2)}\right)$. 使用两个多层感知器$g_t$，将每个函数嵌入的平均值作为输入，产生函数嵌入$g_t\left(E_k\right)=W_2\tanh\left(\frac{W_1\left(\sum_{i=1}^{n}E_{k,i}\right)}{n}\right)$，其中$W_1\in\mathbf{R}^{d_{\mathrm{emb}}\times d_{\mathrm{emb}}}$，$W_2\in\mathbf{R}^{d_{\mathrm{emb}}\times d_{\mathrm{func}}}$将具有维度$d_{\mathrm{emb}}$的最后自注意层嵌入的平均值转换为$d_{\mathrm{func}}$维的函数嵌入. 为了应对大规模的函数，$d_{\mathrm{func}}$通常比$d_{\mathrm{emb}}$小，同理对于具有参数$\theta$的$g_t$，通过微调操作可以最小化两个函数嵌入之间的余弦距离[26]，同时最小化真值之间的余弦嵌入损失l_{ce}：

$$\underset{\theta}{\operatorname{argmin}}\, l_{\mathrm{ce}}\left(g_t\left(E_k^{(1)}\right),g_t\left(E_k^{(2)}\right),y\right) \tag{9}$$

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

其中ξ的取值范围为$[0,0.5]$[26]，在实验过程中依据实验结果设置ξ的取值为0.1，从而保证模型能够更好处理不相似的函数. 由于g_p和g_t均为神经网络，故在实验过程中，可以采用后向传播梯度下降的方法，对式(8)和式(9)进行优化操作.

6 实验及分析

实验对比对象为Genius[8]、Trex[26]和jTrans[17]，其中Genius为经典的二进制分析工具，Trex为近年基于Bert的相似性检测工具，jTrans为最新的二进制代码相似分析工具，其采用了跳跃感知的分析方法对指令进行编码，有较好的分析性能. 在测试数据集方面需要说明的是，为了更好对比各实验模块的效果，依据不同的实验目标会有针对性地选择不同的测试集并进行结果分析. 在下文各实验模块中，分别给出各实验所需的具体配置及测试对象.

本文实验环境为基于CUDA 11.5和cuDNN 8.3.2的PyTorch 1.7.0，操作系统为运行Ubuntu 20.04.3 LTS的Linux服务器，所使用的处理器为Intel Xeon E5-4600(2.70 GHz)，包含70个虚拟内核并使用超线程技术，内存大小为375 GB RAM，GPU为Nvidia GeForce GTX 1070.

本文的实验设置为：基于以上实验环境和后续各实验所使用的具体数据集，在指标设置方面，由于常见的余弦相似性取值为[0,1]之间的任意数值，故为了能够更加准确地表达相似的结果，在实验中使用ROC曲线以衡量模型在不同阈值下的TP(True Positive)和FP(False Positive)值. 具体地，使用AUC(Area Under Curve)的数值来量化本文方法的准确性，即AUC得分越高，则模型对应的准确性越高.

注意到有些实验工具使用其他形式进行精度对比，例如Genius使用Precision@k的方法，具体过程为对于需要分析的二进制程序，Precision@k用来计算目标程序中能够成功进行匹配的函数百分比，此处匹配指的是在目标程序中与真实程序具有前top-k的最高得分. 此外，对于预训练过程中预测掩码的效果，使用标准的perplexity形式进行评估，即perplexity得分越低，预训练模型的性能越高.

在微调方面，通过随机选择10 000个函数对，并随机选择其中的10%进行训练，其余用作测试集. 为了减少过拟合的现象，将微调集合中相似函数与不相似函数的比率设置为1∶5，之所以选择这样的比率，是因为现实测试集中，不相似的函数数量往往多于相似的函数数量，同时也可以体现对比学习过程中的具体实践效果，故在式(9)中，将参数ξ的值设置为0.1，从而使得本文的模型更加趋向于将测试集中的函数集合默认设置为不相似.

在超参数设置方面，在预训练与微调过程中设置epoch为10，并在式(8)中设置参数α的值为0.125，从而使得代码预测和值预测的交叉熵损失具有相同的权重. 在训练过程中将二进制文件经编码操作后的长度设置为256，若其他编码后的函数长度超过该值，则将其拆分为子串并进行预训练操作，在该过程中取子序列嵌入的平均值. 将批处理大小(batch size)设置为256，其主要原因是受制于GPU的显存大小(8 GB)，若设置为更大的值(如512)则不能保证存入显存. 故在实验过程中，每操作32个batch就进行梯度的聚合操作，并调整权重等参数，故在实验过程中设置batch size为8(32×8=256). 在预训练模型的学习率方面，在第一个epoch阶段设置学习率为一个较小的值(如10^{-7})，在后续epoch阶段逐渐增加该取值为2×10^{-4}.

6.1 函数识别实验及分析

该实验的任务主要包括对函数匹配模型的比较和函数起始位置的识别，从准确性和时间开销等方面进行测试和分析. 在Linux平台中，测试数据集包含coreutils、binutils、findutils中的1 732个不同格式的二进

制文件,编译器为GNU gcc 4.7.0及Intel icc 14.0.1. 在Windows平台,使用VS2022编译器,对包括7zip、putty、vim等59个开源的项目进行测试.

在匹配模型实验过程中,将函数识别作为有监督的分类问题,其对比工具jTrans使用的阈值为0.5,为了对比实验效果,本文的方法设置相同的阈值,并在函数匹配方面不进行递归函数的分析. 表2为gcc与icc在模型签名匹配方面的精度和时间开销. 为了体现对比效果,表中"本文方法(5)"和"本文方法(无规范化)",分别表示将本文函数识别模型的前缀树的高度设置为5,以及不对本文模型进行规范化处理的场景.

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
Genius	0.427	0.529	2 135.7	0.571	0.634	2 267.3
Trex	0.913	0.852	1 823.4	0.867	0.793	2 106.7
jTrans	0.935	0.867	1 769.1	0.925	0.861	1 975.2
本文方法(5)	0.921	0.867	**1 563.8**	0.915	0.839	**1 737.6**
本文方法(无规范化)	0.915	0.871	15 967.4	**0.956**	0.842	21 871.4
本文方法	**0.952**	**0.917**	1 619.2	0.943	**0.894**	1 869.5

表2表明,Genius在GCC和ICC编译器下,精度均不超过0.6且召回率较低(不超过0.64),而本文方法在精度和召回率方面优于Trex和jTrans,其主要原因是jTrans等工具在指令识别阶段没有采用指令规范化等操作,使其在匹配跨平台指令和各级优化作用下,对语义相似的指令集合有较低的分析精度. 在时间开销方面需要特别说明的是,前缀树高为5的模型具有较小签名长度,从而减少大约3.2%的时间开销,但是却损失了大约7.3%的准确性. 对于没有规范化的模型有大约1.2%的精度提升,并降低大约6.67%的识别率,但时间开销是规范化模型的大约11.3倍.

在函数识别方面,数据集依据中间文件形式划分为ELF x86、ELF x86-64、PE x86和PE x86-64的格式,其中包含1 486个ELF格式的文件,以及246个PE格式的文件. 对于ELF格式的二进制文件使用objdump对符号表进行解析,而对于PE格式的文件则使用dia2dump对PDB文件进行解析. 由于VS使用mov指令而不是push %rbp来开始一个函数,故该匹配方法不能识别PE格式的文件.

实验结果如表3所示,表3数据表明本文方法在函数起始位置的识别精度和召回率方面都优于对比工具. 在时间开销方面,本文在函数识别方面包含函数指令规范化、函数起始位置的识别和函数边界识别等操作,而相应的对比工具的匹配模式较少,且不进行指令规范化的预处理操作,故其时间开销较少,在将来工作中,可以进一步使用更加优化的规范化和匹配策略以进一步减少时间开销.

表3　函数识别的精度、召回率及时间开销(秒)对比(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)

方法	PE x86	PE x86-64	ELF x86	ELF x86-64
Genius	0.629/0.721 635.4	0.692/0.715 493.7	0.706/0.763 765.3	0.729/0.751 771.5
Trex	0.793/0.767 427.4	0.762/0.763 421.7	0.761/0.794 590.5	0.859/0.813 613.1
jTrans	0.815/0.857 **339.5**	0.839/0.834 **394.7**	0.828/0.861 **459.7**	0.879/0.835 **452.1**
本文方法	**0.936/0.912** 386.9	**0.927/0.924** 426.2	**0.942/0.932** 523.7	**0.911/0.874** 497.4

6.2　函数相似度匹配实验及分析

该实验使用的测试集为在软件及操作系统中广泛使用的库文件zlib和libpng,其中zlib为压缩库文件,libpng为处理.png格式的库文件.

在函数重用和函数匹配方面,实验对不同版本的相同二进制文件进行函数重用检测. 由于zlib(1.2.8-1.3.1)有良好的维护记录,这4个版本的程序具有相同的函数名符号以及2 055 584个函数. 实验过程中使用前一版本的zlib对后一版本进行匹配,同时引入了1 701个动态库文件作为噪声函数. 实验中依次使用3个不同的阈值0.6,0.65和0.7进行重复性检测[6],对于Jaccard相似性低于这些阈值的函数,将使用前文的优化策略进行删除,同时使用基本块的数量来衡量函数的规模,为了对比实验依据jTrans,其阈值范围为[4,35].

表4为函数识别实验的结果. 通过优化策略从函数库中生成候选集,随着指纹阈值的增加,候选集的数量会逐渐减少,同时分析精度也会逐渐降低. 需要说明的是,对于版本1.3.0,其分析精度反而上升,其原因是在阈值为0.6时,有其他函数的相似得分超过了具有真实匹配的函数相似得分. 而当阈值为0.7时,这两个函数被优化规则删除,使得真实匹配函数具有最高得分. 这也证实了3.2节中优化策略对于提高函数相似分析

精度的有效性.

对于最新的版本1.3.1,由于没有后续版本进行对比,实验采用不同的二进制文件libpng对其进行重用检测(libpng是一个依赖于zlib的.png格式图像处理文件,会大量重用zlib中的函数). 通过编译zlib 1.3.1和libpng 1.6.30,并进行函数提取操作,从中选择基本块数量大于4的40个函数作为目标过程并进行重用分析,其结果如表4灰色区域所示.

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
1.2.8	0.60	94.70%	11 903	2.51
	0.65	92.28%	2 851	1.29
	0.70	90.43%	1 841	0.88
1.2.9	0.60	97.81%	15 971	2.75
	0.65	96.83%	2 785	1.63
	0.70	94.01%	1 893	1.04
1.3.0	0.60	**95.47%**	15 895	2.69
	0.65	**96.26%**	2 794	1.64
	0.70	**97.03%**	1 960	1.13
1.3.1	0.60	94.47%	3 409	2.01
	0.65	93.61%	717	1.34
	0.70	90.92%	229	1.12

在软件分析方面,本文对Citadel和Zeus这两个常见的程序进行实验. Zeus为使用RC4算法的流密码函数, Citadel从Zeus发展而来并对该流密码函数做了修改. 故本实验目的为发掘Citadel中的流密码函数,使用IDA Pro对Zeus和Citadel进行反汇编,将从Zeus中抽出的RC4函数作为目标函数,与从Citadel中抽出的函数进行相似度对比. 图8显示了Citadel和Zeus中的RC4的CFG图[6],虽然它们不完全同构,但是本文方法通过模糊匹配的方法可以正确识别.

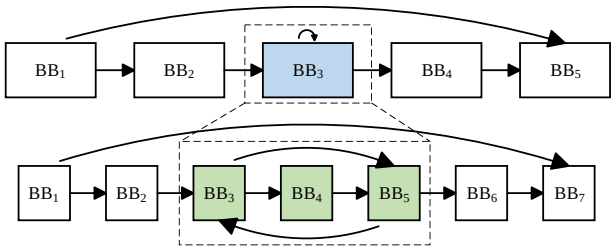


图8 RC4的整体结构(其上部为在Zeus中的结构,下部为在Citadel中的结构)

6.3 语义相似实验及分析

本文方法可以在程序有修改的情况下对其进行分析,包括跨平台和跨编译器以及不同优化级别的程序. 对于待分析的函数集合,首先将该集合划分为单个函数集合,然后进行相似性分析,调用程序验证器以计算原函数的相似度.

如表5所示,在语义相似实验过程中使用8个真实的代码,表中包含了具体的CVE(Common Vulnerabilities and Exposures)信息. 其中Heartbleed为允许攻击者通过发送带有错误负载长度的恶意心跳请求的漏洞, Shellshock通过特别设计的参数使得解析器错误地执行参数中的命令,Venom使得攻击者能够从受感染虚拟机中摆脱访客身份限制.

表5 FP和ROC的查询搜索对比

方法	基本块数量	ShortString数量	$P_{shortstring}$		$P_{local-global}$		$P_{sigmoid}$	
			FP	ROC	FP	ROC	FP	ROC
Heartbleed	17	95	102	0.95	0	**0.96**	0	0.97
Shellshock	143	457	238	0.85	4	0.93	4	**0.98**
Venom	15	64	1 340	0.95	0	0.95	0	1.00
Clobberin Time	45	257	339	0.81	61	0.93	21	0.97
ShellShock	92	279	167	0.83	44	0.94	0	0.93
ws-snmp	7	103	41	**0.97**	7	0.92	2	0.96
wget	87	179	317	0.85	14	0.94	0	0.94
ffmpeg	15	83	243	0.93	94	0.95	0	**0.98**

对于测试代码采用默认的编译设置生成x86架构的可执行代码,并删除调试信息,此时包含约1 500个不同的过程. 在实际操作过程中,由于代码库规模的增加,相似性的比较会出现较高的误报率. 例如在实验过程中,不同编译器依据不同的编译风格,产生了大量有明显特征的代码,以至于可以通过该代码来识别对应的编译器种类. 此时不同编译器或优化参数生成的代码,由于较低的比较精度导致较低的相似度得分[4]. 原因是原始的语义在一定程度上被编译器编译后的语义所取代,从而导致了相似度得分的偏移.

该部分实验主要包括三个方面:首先使用不同版本的同一编译器进行匹配试验,具体为gcc,Clang及icc编译器;再基于交叉编译每次从不同编译器中选择某一查询函数进行匹配实验;最后对有修改的函数的语义进行识别,即对于有语义修改的源程序,若某个函数的语义被修改,则对其他函数的修改也会影响到该

语义.

相似性比较的方法通常设置一个阈值以判断相似性,即高于该阈值即认定为匹配,反之亦然,但是在真实实验环境中,难以找到固定的阈值区分误报和漏报.由于两个函数嵌入间的余弦相似性取值范围为[-1,1],为了避免特定的阈值带来的偏差,本文使用基于分类器的阈值验证工具ROC,它通过连续地对可能的阈值进行测试从而给分类器进行评分,若评分高于阈值则分类值为1,反之亦然.

在表5中,根据第5节的分析,语义相似分析可以分为三组[4]:

(1)计算ShortString之间相似性的过程($P_{\text{ShortString}}$).该过程将ShortString之间的SIM推广至函数间的SIM,使用下式对能够与目标ShortString匹配的最大数量进行计算:$\sum_{s_t \in T} \max_{s_q \in Q} \left(\text{SIM}\left(s_t, s_q\right)\right)$.

(2)通过ShortString相似性推理全局相似性的过程($P_{\text{local-global}}$).在该过程中,通过使用$\Pr\left(s_q, s_t\right) = \text{SIM}\left(s_q, s_t\right)$对$S_{\text{local}}$和$S_{\text{global}}$的计算过程进行相应的替换,从而获得在没有sigmoid函数作用下的全局函数相似度.

(3)在过程2的基础上增加sigmoid函数的过程(P_{sigmoid}).

为了量化相似度的分析过程及结果,依据以上三个过程分别对测试集进行实验,结果如表5所示.由于较小的ShortString对应较低的S_{local}相似度得分,从而难以为S_{global}提供较高的置信度.对于相似度较低的ShortString,通过调整阈值确保查询ShortString与至少有一半(SIM的最小值为0.5)或者最多2倍(减少由于ShortString过多而导致True Positive较少的概率)变量的目标ShortString相匹配.

6.4　跨平台指令相似性实验及分析

本文方法在训练阶段需要将数据集划分为训练集和测试集,并进行10折交叉验证,通过计算速度与精度的平均值来评价最后的性能.

数据集由2 200个二进制文件组成,其中包含2 064个来自13个主流Linux的开源项目,其他为putty、7zip、vim、libsodium等来自Windows系统里的文件.编译环境包括Linux和Windows操作系统,指令集包括x86和x86-64,编译器包括GNU gcc 4.72,Intel icc 14.0.1以及Microsoft VS,优化级别包含4级,从无优化到全优化:-O0,-O1,-O2,-O3,函数信息如表6所示.

表6　带有4级优化的不同架构的函数数量

架构	优化	测试函数												
		Binutil	Coreutils	Curl	Diffutils	Findutils	GMP	ImageMagick	Libmicrohttpd	LibTomCrypt	OpenSSL	Putty	SQLite	Zlib
x86	O0	37 779	24 388	1 361	1 181	1 871	821	3 870	322	821	12 201	7 569	323	215
	O1	32 281	20 081	1 030	972	1 520	715	3 473	285	786	11 563	6 185	2 148	193
	O2	31 527	18 615	1 059	1 010	1 511	723	3 549	272	795	11 748	6 131	2 013	174
	O3	31 053	17 447	1 032	1 041	1 453	715	3 575	287	771	11 741	5 842	1 930	185
x64	O0	26 741	17 239	1 017	840	1 372	746	2 962	207	765	12 040	7 053	2 290	167
	O1	21 432	12 543	732	587	1 006	682	2 337	163	753	11 163	5 747	1 623	153
	O2	20 974	12 225	736	593	983	675	2 359	163	747	11 112	5 721	1 380	147
	O3	19 481	11 471	675	551	870	672	2 321	157	743	10 768	5 368	1 183	130
ARM	O0	25 475	19 969	1 089	930	1 541	751	2 915	207	797	11 936	7 047	2 186	137
	O1	20 059	14 905	762	676	1 136	687	2 318	185	730	10 974	5 751	18	151
	O2	19 478	14 762	757	684	1 127	717	2 351	187	725	11 015	5 743	15	143
	O3	17 823	13 953	697	635	989	663	2 280	172	717	10 610	5 437	14	117
MIPS	O0	28 482	18 827	1 030	918	1 475	737	2 863	205	739	11 839	7 081	23	148
	O1	22 536	13 782	751	657	1 031	685	2 217	187	714	10 927	5 673	17	146
	O2	22 017	13 653	735	647	1 047	643	2 239	183	746	10 948	5 695	1 367	128
	O3	20 273	12 743	662	571	929	630	2 171	157	739	10 552	5 340	1 144	134

对于一组测试集,记函数比率为测试集中出现了相似的函数的百分比,同时记top-p ratio为函数相似度为前p的函数比率.对于bottom-p ratio有类似定义,则top-p ratio与bottom-p ratio的差为函数相似的置信度.图9为在$p=10\%$的条件下,top-p ratio与bottom-p ratio的差在TP、FP、FN、TN上的典型值,表明本文方法在正确性方面相较于对比工具有较大优势.

图9表明相对于TN,本文方法可以有更高的置信度以进行TP分析,即对函数识别和语义匹配等操作有更高的典型值得分.需要说明的是,在语义匹配阶段本

文方法除了函数识别和函数调用的分析之外，还需要对数据之间的依赖关系进行分析．即某些函数彼此之间可能有直接的数据依赖关系，但没有任何顺序关系，导致在进行数据依赖分析的时候可能会对函数匹配的效果带来影响，将来的工作中会继续对语义与数据依赖之间的协同关系展开分析．

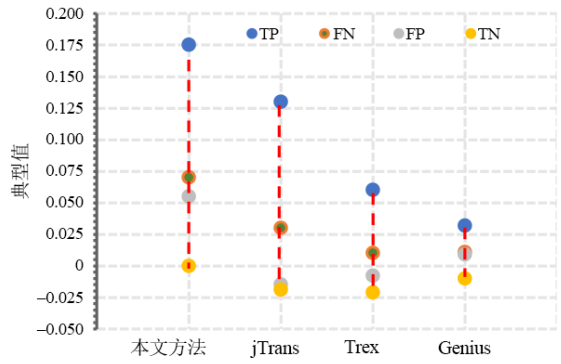


图9 不同工具的典型值对比

为了验证公共子序列对实验性能的影响，图10描述了实验中的平均误报数，对于一个查询序列，在候选序列中搜索与其语义相似的序列．实验中逐渐增加查询序列的数量并记录假阳性序列的数量，随着公共子序列的数量增加，假阳性序列数量快速减少，在公共子序列为73时到达固定点．

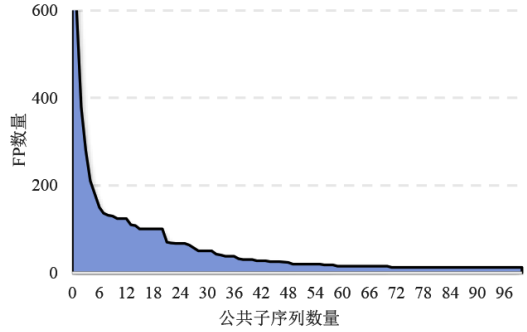


图10 LCS与FP的对应关系

为了进一步验证该结果，图11绘制了ROC用来表示TP比例与FP比例之间的关系）得分与公共子序列数量之间的热图，每个像素代表一个具体的过程，其颜色的强度代表ROC得分高低，其中F-score为精度与召回率的调和平均数，在公共子序列数量大约为70时，F-score的得分最高，表明本文方法可以通过预训练的方法学习执行语义，从而使用较小的训练集就可以高效地匹配语义相似的函数．

实验过程中ROC的最高值为0.972，由于式(7)中对于SIM的计算是一种非对称的计算，根据前文对式(7)的分析，即不同的s_q和s_t在相似度匹配过程中有不同的

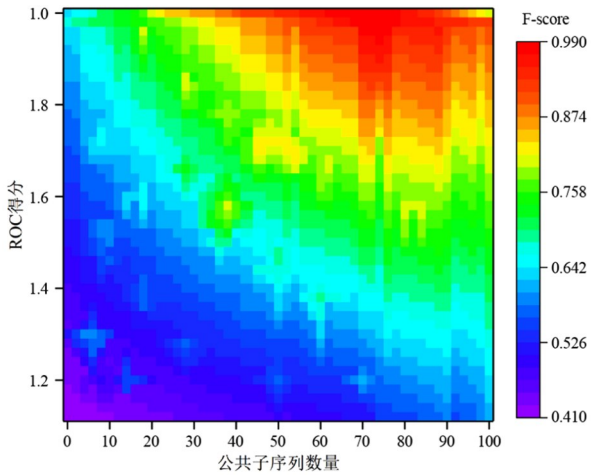


图11 LCS和ROC得分的热图

潜在无关匹配指令，从而导致有不同的匹配结果．同时图10表明对于语义相似的指令集合，本文方法可以产生较高的ROC得分，而对于随机匹配则会产生相对较低的ROC得分．

注意到图11的颜色强度并不是均匀过渡，而是在某些位置会有差异，其原因是某些函数的规模较小，依据前文分析，此类函数指令的匹配值较低，导致在整个函数集相似度计算过程中，该较短的指令序列会对整体的相似度结果产生一定的影响．

图12为表6中各测试集对应的AUC(ROC曲线下的面积)得分对比，受篇幅所限，该处与对比工具中性能最优的jTrans进行对比，本文方法的AUC平均得分较jTrans高6.7%．本文方法的AUC得分均超过90%，最高可以达到98%，其中包含了跨平台和多优化选项的环境．在此场景中同一函数经过不同优化选项生成的指令集，其操作码和操作数基本不变，而寄存器变量可能会发生变化，而这些因素都会影响指令位置的偏移．实验结果表明测试对象版本的变化给AUC得分带来的影响较小，而数据集之间的语义差异对AUC得分有较大影响，这也反映了本文方法具有较好的泛化能力．

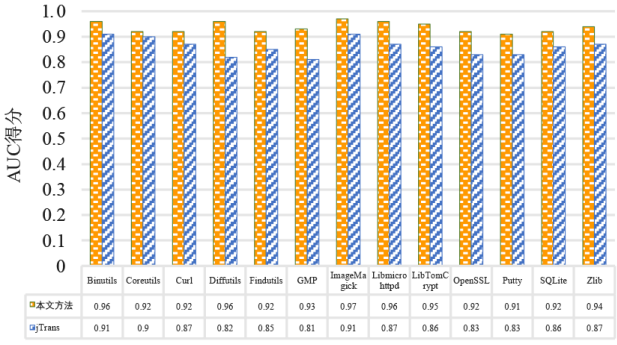


图12 本文方法和jTrans在函数识别得分方面的对比

在跨优化分析方面,本文依据从静态的汇编代码中获取的嵌入进行跨优化分析. 本文方法在不同优化级别下,使用precision@1作为精度评价指标,表7表明对于不同的优化级别,本文方法较jTrans在精度上平均高出6.5%. 在代码变化幅度方面,从-O0到-O3在代码的语法结构上的改变比从-O2到-O3的要大,故AUC得分均有不同程度的下降,但本文方法下降的幅度较小. 这表明编译器的优化选项对于生成的指令集合有较大影响,编译优化跨度越大,所生成的指令集合之间的差异也越大,这也会对AUC的得分带来直接影响.

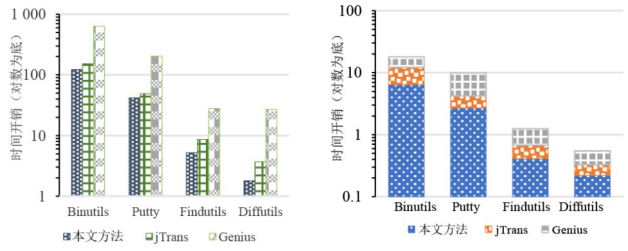
表7　本文方法和jTrans函数相似性得分对比

指令集	不同编译器优化级别下的相似度值			
	-O2 和 -O3		-O0 和 -O3	
	本文方法	jTrans	本文方法	jTrans
Coreutils	0.945	0.913	0.921	0.796
Curl	0.972	0.949	0.874	0.863
GMP	0.964	0.964	0.885	0.785
ImageMagick	0.963	0.951	0.923	0.844
LibTomCrypt	0.987	0.978	0.945	0.937
OpenSSL	0.971	0.953	0.933	0.803
Putty	0.983	0.875	0.926	0.822
SQLite	0.954	0.953	0.898	0.796
Zlib	0.873	0.891	0.877	0.868
平均值	**0.941**	**0.922**	**0.895**	**0.835**

在运行时间方面,从表6中抽取具有不同规模的4个测试程序binutils、findutils、diffutils和putty,编译环境为x64平台,优化等级为-O3. 在该过程中,主要包含二进制文件的函数解析过程,即将二进制文件转换为各平台所需格式的过程;以及生成嵌入的过程,即计算函数嵌入的过程. 在匹配过程中统计预训练模型构建时间以及计算函数嵌入所需要的时间. 图13为时间开销对比图,其y轴以对数的方式对时间开销进行缩放,由于Genius需要人工构建CFG图并提取基本块特征,故其时间开销较大(图13(a)). 相对于Genius使用的循环神经网络和jTrans使用的Bert模型,本文方法在指令相似性对比之前加入了指令规范化操作,并在图7的模型中加入了自注意力层,从而在GPU的加持下有更少的时间开销(图13(b)).

为了量化本文方法各模块对于性能的影响,需要进行消融实验. 首先对预训练模型在学习执行语义及匹配二进制文件中所起的作用进行实验,然后对微执行在预训练中所起的作用进行实验. 在预训练方面,比较本文方法在100%、66%、33%以及无预训练函数的AUC得分.

图14表示在无预训练的情况下,AUC得分平均下降了15.7%. 值得注意的是,对于具有100%、66%和33%的预训练函数,其AUC得分差距在1%以内,这表明在实验过程中为了减少时间开销,可以减少预训练集的规模以获得近似的执行效果. 在微执行方面,由于微执行可以更好地反映执行语义,在该项实验中对具有屏蔽值的数据集进行预训练,将输入值替换为屏蔽值,删除式(8)中预训练对应的值.



(a) 函数扫描时间开销对比　　(b) embedding生成时间开销对比

图13　本文方法, jTrans, Genius的运行时间开销(以对数为底)

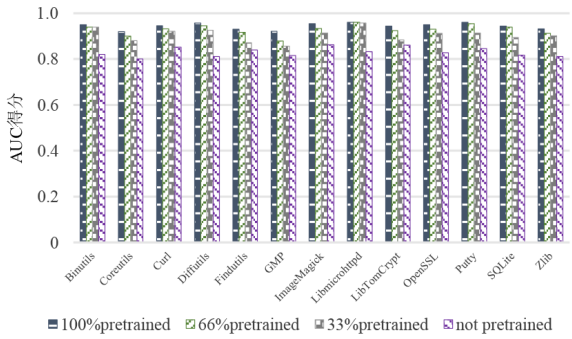


图14　不同比率预训练下的AUC得分对比

图15表明在没有微执行的情况下,本文方法预训练的AUC得分平均下降了5.3%,同时比jTrans的AUC得分低4.1%. 同时该图表明无微执行的预训练给系统带来的性能损失(3.6%)相对于不进行预训练的系统的性能损失(8.7%)要小,这表明预训练对于提高函数匹配精度是正相关的. 其原因是功能相似的函数在语法上是静态相似的,这反映在语义上的结果是AUC具有相似的得分,故在图7的模型中使用静态的汇编代码也可以进行预训练,并使得在模型的训练阶段具有较好的学习能力.

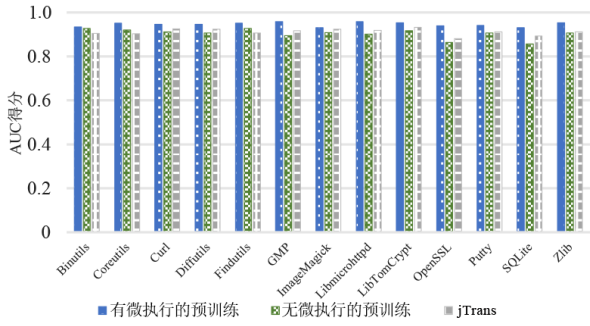


图15　不同测试集的AUC得分对比

7 总结

本文提出一种基于统计推理的二进制文件相似性比较框架从二进制文件中的函数识别开始,提出前缀树的分析方法识别函数的起始位置及函数体,从而对反汇编后的文件进行划分,在此基础上逐级地对函数和二进制文件进行相似度分析. 通过邻域搜索的方法将该映射关系拓展到函数粒度,同时对函数基本块的执行语义进行相似性统计分析,从而计算二进制文件的相似度.

本文存在如下改进空间:在函数识别过程中若前缀树的终端节点权重大于0.5,则表示可以识别函数的起始位置,该权值来自对比工具中的取值,同时本文使用固定长度的指令序列也在一定程度上限制了本文的适用范围. 在将来工作中将针对不同的训练集调整该权值的大小以及指令的长度,以取得更优的函数识别精度和效率. 另外,在相似性比较过程中,若匹配一个规模非常小的目标函数,此时数量及规模均较小的ShortString会对相似性统计的结果造成较大影响;此外高级语言中存在的泛型机制,会创建具有相似结构但类型不同的模板函数,由于它们具有相同的结构,从而导致在相似性比较过程中会对真阳性和误报带来一定程度的影响,在将来工作中需要对小规模函数和模板函数的特点做进一步的完善.

参考文献

[1] WHALE G. Plague: Plagiarism Detection Using Program Structure[R]. Sydney: University of New South Wales, 1988: 1-13.

[2] YANG S G, DONG C P, XIAO Y, et al. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge[J]. ACM Transactions on Software Engineering and Methodology, 2023, 33 (1): 1-40.

[3] 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, 2018: 667-678.

[4] DAVID Y, PARTUSH N, YAHAV E, et al. Statistical similarity of binaries[C]//Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2016: 266-280.

[5] MUJA M, LOWE D G. Fast approximate nearest neighbors with automatic algorithm configuration[C]//Proceedings of the Fourth International Conference on Computer Vision Theory and Applications. SciTePress - Science and and Technology Publications, 2009:331-340.

[6] HUANG H, YOUSSEF A M, DEBBABI M, et al. 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, 2017: 155-166.

[7] 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.

[8] 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, 2016: 480-491.

[9] COLLYER J, WATSON T, PHILLIPS I. FASER: Binary code similarity search through the use of intermediate representations[EB/OL]. (2023-11-29) [2024-05-07]. https://arxiv.org/abs/2310.03605v3.

[10] LUO Z H, WANG P F, WANG B S, et al. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search[C]//Proceedings 2023 Network and Distributed System Security Symposium. Internet Society, 2023: 1-13.

[11] XU X J, LIU C, FENG Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2017: 363-376.

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

[13] MASSARELLI L, DI LUNA G A, PETRONI F, et al. SAFE: Self-attentive function embeddings for binary similarity[M]//Detection of Intrusions and Malware, and Vulnerability Assessment. Cham: Springer International Publishing, 2019: 309-329.

[14] 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, Volume 1. Cham: Springer International Publishing, 2019: 4171-4186.

[15] LI X, QU Y, YIN H, et al. PalmTree: Learning an assembly language model for instruction embedding[C]//In Pro-

ceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2021: 3236-3251.

[16] 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, 2022: 1-13.

[17] 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: 1-13.

[18] YANG S G, CHENG L, ZENG Y C, et al. Asteria: Deep learning based AST-encoding for cross-platform binary code similarity detection[C]//2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Piscataway: IEEE, 2021: 224-236.

[19] 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.

[20] 于璞, 舒辉, 熊小兵, 等. 基于分片融合的代码隐式混淆技术[J]. 软件学报, 2023, 34(4): 1650-1665.
YU P, SHU H, XIONG X B, et al. Implicit code obfuscation technique based on code slice fusion[J]. Journal of Software, 2023, 34(4): 1650-1665. (in Chinese)

[21] LIN W, GUO Q L, YIN J W, et al. FSmell: Recognizing Inline function in binary code[M]//Computer Security-ESORICS 2023. Cham: Springer Nature Switzerland, 2024: 487-506.

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

[23] BAO T, BURKET J, WOO M, et al. BYTEWEIGHT: Learning to recognize functions in binary code[C]//Proceedings of the 23rd USENIX Conference on Security Symposium. New York: ACM, 2014: 845-860.

[24] 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]//USENIX Security Symposium. Piscataway: IEEE, 2022: 2709-2725.

[25] LESKOVEC J, RAJARAMAN A, ULLMAN J D. Mining of Massive Datasets[M]. 2nd Ed. Cambridge, UK: Cambridge University Press, 2014.

[26] PEI K X, XUAN Z, YANG J F, et al. Trex: Learning execution semantics from micro-traces for binary similarity [EB/OL]. (2021-04-26) [2024-05-07]. https://arxiv. org/abs/2012.08680v3.

作者简介

郭　曦　男,1983年10月出生于湖北省鄂州市.现为华中农业大学信息学院副教授、硕士生导师,主持和参与国家及省级科研课题10余项.主要研究方向为软件分析与测试、信息安全、大模型开发等.
E-mail: xguo@mail.hzau.edu.cn

王　盼　女,1987年4月出生于河南省济源市.现为湖北工业大学电气与电子工程学院副教授、硕士生导师,主持和参与国家及省级科研课题10余项.主要研究方向为功率变换器、新能源发电技术、电能质量控制以及新型配电网技术等相关研究.
E-mail: wp20210018@hbut.edu.cn