Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心


# 基于深度学习的二进制函数相似性分析：深入探究两大主流研究方向

硕士研究生 高玺凯

2024年02月04日

# 问题回溯

- 相关内容
  - 2022.11.27 沈宇辉《二进制函数相似性分析》
  - 2022.06.26 邢继媛《二进制程序开源成分分析》
  - 2021.11.07 邢继媛《基于汇编指令嵌入的漏洞同源性判别》
  - 2021.04.24 邢继媛《基于图神经网络的二进制函数相似性检测》

# 内容提要

- 预期收获
- 内涵解析与研究目标
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - DiEmph
  - Asteria-Pro
- 特点总结与未来展望
- 参考文献

# 预期收获

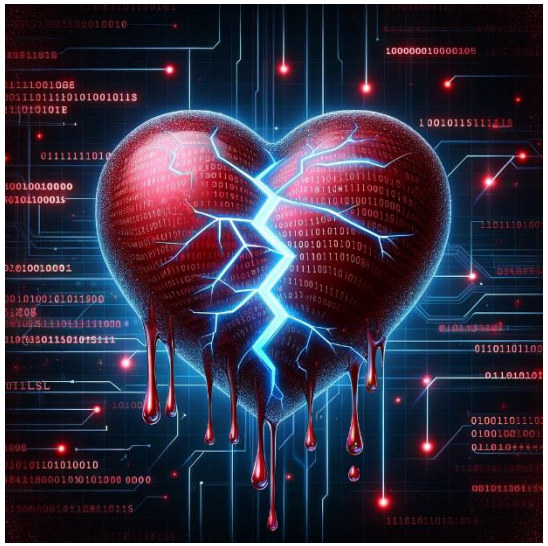
- 预期收获
  - 掌握二进制函数相似性分析任务的基本概念、应用和研究现状
  - 了解一种目前最先进的单架构二进制函数相似性检测方法
  - 了解一种目前最先进的跨架构二进制函数相似性检测方法

# 内涵解析与研究目标

- 内涵解析
  - 二进制函数(Binary Function)：指的是由高级编程语言源代码函数经过编译过程生成的机器码函数（仅包含0和1），是二进制可执行程序的一部分，是计算机硬件能直接识别和执行的最基本形式
  - 二进制函数相似性分析(Binary Function Similarity Analysis)：在给定一个可执行二进制查询函数情况下（没有源代码或任何符号信息），从大量候选函数池中确定一组与查询函数相似的函数
    - “相似的函数”在二进制函数相似性分析背景下指的是由相同源代码使用不同编译器、不同优化选项，针对不同目标架构编译得到的二进制函数，由于编译过程的差异，它们的二进制表示会有所不同
- 研究目标
  - 结合深度学习、二进制程序分析等理论
  - 开发在跨编译器、跨优化选项和跨架构场景下有效执行相似性分析的方法

# 研究背景

- 研究背景
  - 为提升软件开发效率、降低开发成本，开发人员越来越广泛地复用开源项目和第三方库中的代码
    - 在一定程度上提高了生产力
    - 带来了潜在的安全隐患，使得漏洞在大量计算机和物联网固件设备中迅速传播
  - 在现实世界中，由于软件知识产权保护等原因，厂商通常只提供软件的被剥离二进制文件版本（没有任何符号信息），并且其源代码通常也难以获取



心脏滴血漏洞(Heartbleed)

破壳漏洞(Shellshock)


Log4Shell漏洞

# 研究意义

- 研究意义
  - 二进制函数相似性分析在1-Day漏洞检测、代码克隆检测、恶意软件检测、软件剽窃检测和自动软件修复等多个应用领域中具有广泛的应用
  - 研究二进制代码相似性分析技术能够在无源码条件下自动化分析二进制程序中的漏洞，以有效监控和分析漏洞代码的传播情况
  - 同一源代码在不同架构、编译环境以及代码混淆条件下编译得到的二进制代码之间存在显著差异。现有二进制函数相似性分析方法根本无法在复杂多变的实际场景下应用，亟需开发更高效且准确的方法以适应实际需求！
    - HUAWEI公司2023年7月17日于黄大年茶思屋网站发布的技术难题和技术诉求
      - **基于二进制相似度匹配的二进制漏洞扫描：**预训练二进制大语言模型[4]，具备漏洞补丁的识别能力，能够**部分识别漏洞修复补丁**是否打入并提升漏洞扫描准确率(约为80%)
        - 该技术严重受到**编译选项优化等级**影响，在部分恶劣条件下，准确率下降到不足10%

**技术诉求**

**二进制漏洞扫描算法：**在Linux或Windows的二进制数据集上准确识别二进制文件中的漏洞，漏洞识别准确率90%以上。

# 研究历史

2017 Xu等人提出Gemini方法，首次将深度学习技术引入该领域。该方法基于基本块统计特征生成属性控制流图，使用孪生神经网络将其嵌入至向量空间，将向量之间的余弦距离作为函数相似性度量

2019 Massarelli等人提出使用Word2vec模型生成函数的汇编指令指令及基本块嵌入的GraphEmb方法。与Gemini相比实现了无监督的基本块嵌入生成

2019 Ding等人提出一种汇编代码无监督表示学习模型Asm2Vec。该方法仅需将汇编代码作为输入，即可自动整合其中丰富的语义信息，克服了汇编指令词汇表外问题

2019 Zuo等人提出InnerEye方法。受神经机器翻译技术的启发，该方法首先利用Word2vec模型生成指令嵌入，然后利用LSTM网络对指令嵌入和指令依赖关系进行编码，可初步编码跨架构指令间的共性语义特征

2020 Yu等人提出Order Matters方法。使用预训练的BERT模型将生成指令嵌入；使用消息传递神经网络生成属性控制流图嵌入；使用ResNet网络提取基本块顺序信息，取得了十分优异的性能

2021 Pei等人提出Trex方法。该方法使用Transformer在预训练阶段从微轨迹（受约束的动态轨迹）中自动学习指令序列的执行语义

2021 Yang等人提出Asteria方法。该方法将二进制函数反编译为伪代码，提取其AST作为函数语法和语义信息表征，使用Tree-LSTM网络生成AST的嵌入向量表示

2022 Wang等人提出jTrans方法。该方法使用Transformer模型将控制流信息嵌入到函数向量表示中，根据二进制函数的指令语义和汇编代码的跳转关系进行相似性检测

2023 Yang等人提出Asteria-Pro方法

2023 Yang等人提出DiEmph方法

二进制函数相似性分析

- 利用汇编指令序列嵌入
- 利用函数结构化信息表征嵌入

# 基础概念 编译设置

- 编译器类型
  - 不同类型的编译器（如Clang、MSVC、gcc、icc等）具有不同的特性和固有约定，对源代码的解释和处理也各不相同
- 编译器优化级别
  - O0：不启用优化
  - O1：启用基本优化
  - O2：启用更多优化
  - O3：启用几乎所有的优化技术
  - Os：优化代码的空间占用
- 架构
  - 不同架构（如ARM、x86、MIPS等）都有其特定的指令集、优化方式和执行模式

# 研究现状

- 利用汇编指令序列嵌入的方法
  - 将二进制函数反汇编得到的汇编指令序列视为自然语言，利用NLP领域的代码表示学习技术将指令嵌入到向量空间，以捕获指令间的语义信息，实现二进制函数相似性检测
  - 难以精确捕获二进制函数跨编译环境和跨架构的共性语义特征
- 利用函数结构化信息表征嵌入的方法
  - 在二进制函数的汇编代码或中间表示层级上提取它的各类结构化信息表征，通过深度学习模型将其嵌入至向量空间，生成函数的嵌入向量表示；最后计算两个向量的相似度并与阈值比较，实现相似性检测
  - 提取了函数在跨编译环境或跨架构场景下的部分共性结构特征，对相应变化更加鲁棒，但难以精确捕获函数内部的语义特征

# DiEmph

## Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis

# DiEmph TIPO

| | | |
|---|---|---|
| T | 目标 | 改进基于Transformer的二进制函数相似性模型，提升模型的泛化性 |
| I | 输入 | 基于Transformer的二进制函数相似性模型*1，训练（微调）数据集*1 |
| P | 处理 | 1.指令分类重要性分析<br>2.指令语义重要性分析<br>3.移除分类重要性高但语义上不重要的指令，并重新运行微调过程 |
| O | 输出 | 改进后的基于Transformer的二进制函数相似性模型*1 |

| | | |
|---|---|---|
| P | 问题 | 由于编译器的确定性和固有约定，数据集中存在不希望的指令分布偏差，影响了基于Transformer的二进制函数相似性模型的准确率 |
| C | 条件 | 数据集中的二进制程序可被正常反汇编且未经过代码混淆技术处理 |
| D | 难点 | 如何准确识别并抑制数据集中因编译器的确定性和固有约定引入的不希望的指令分布偏差 |
| L | 水平 | ISSTA 2023 CCF A |

# DiEmph 必要知识

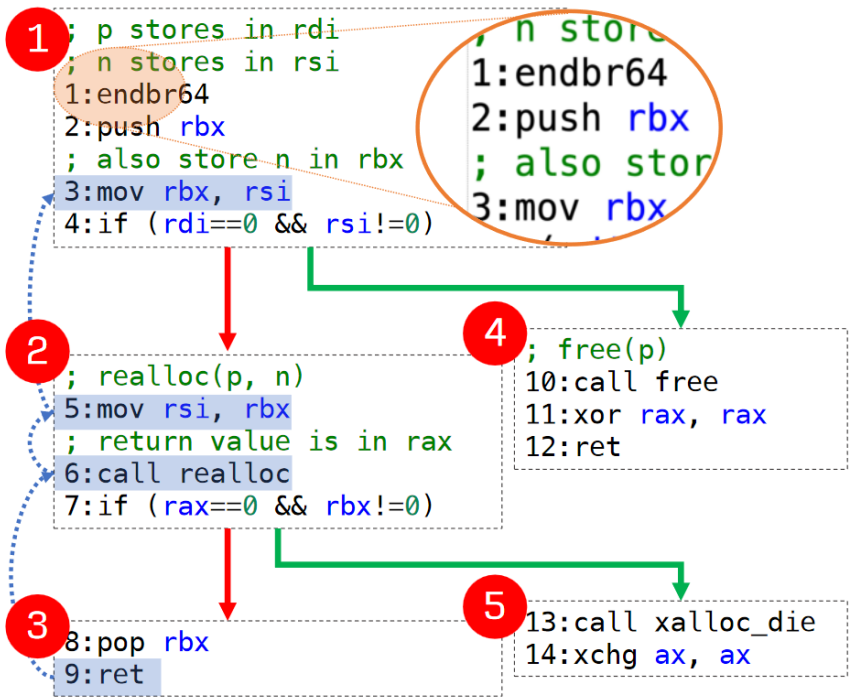
- 稳定变量(Stable Variables)
  - 定义：对编译器和优化选项的变化不敏感的变量
  - 特点：稳定变量在相似函数中往往具有一致的表示
  - 分类：全局变量、堆变量、函数返回变量、作为函数实际参数传递的变量
  - 挑战：处理的对象是可执行二进制文件，变量的符号信息和函数签名不可用
- 程序切片(Program Slicing)
  - 定义：一种程序分析技术，用于识别程序中影响到某个特定点（例如变量或计算结果）的代码集合，又分为静态切片和动态切片
  - 后向切片(Backward Slicing)
    - 程序切片的一种特殊类型
    - 仅从某个特定的程序点开始逆向追踪，仅关注该点之前的指令、决策或数据的影响

# DiEmph 动机（Motivation）

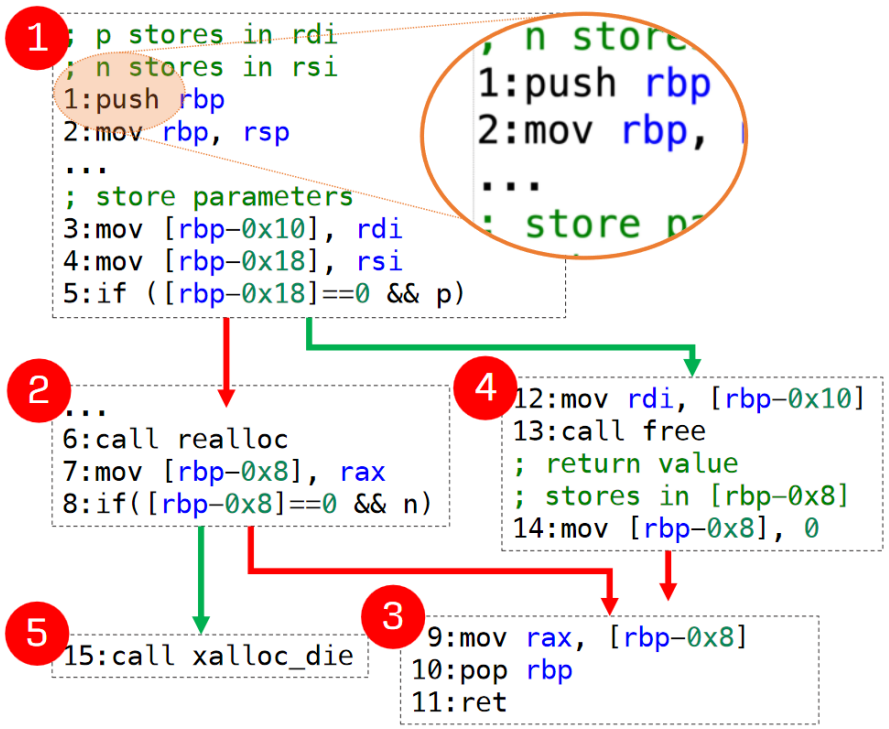
- 示例

```
1   void xalloc_die(){
2       ...
3       exit(-1);
4   }
5   void*xrealloc(void*p,
6                 size_t n){
7     if (!n && p){
8       free (p);
9       return NULL;
10    }
11    p = realloc (p, n);
12    if (!p && n)
          xalloc_die ();
13    return p;
14  }
```

(a) Source Code




(b) CFG (GCC -O3)




(c) CFG (Clang -O0)

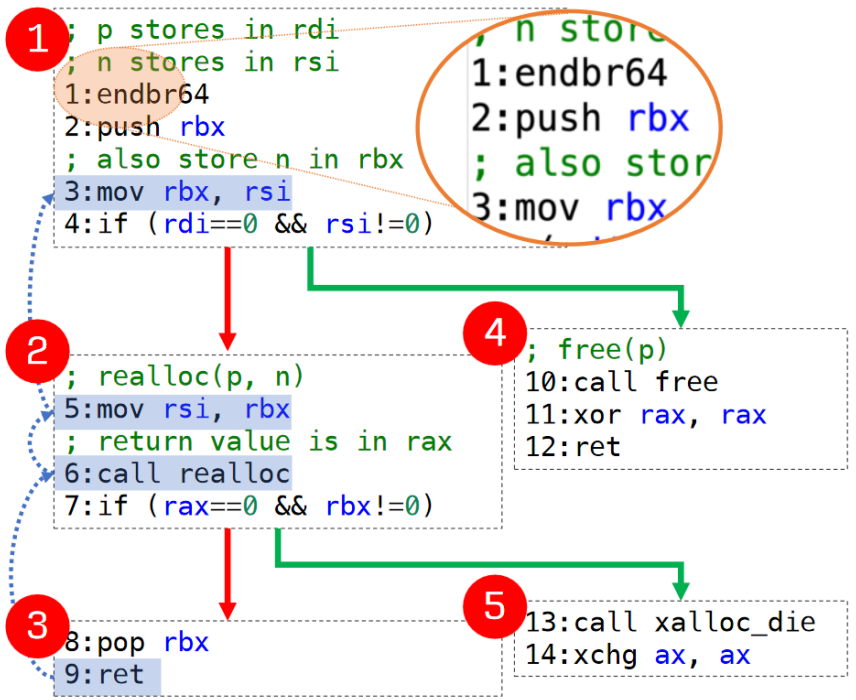
- 两个控制流图具有相似的结构，并且包含许多对应的指令
- 根据jTrans模型的计算结果，两个函数的相似性得分为0.3

# DiEmph 动机（Motivation）

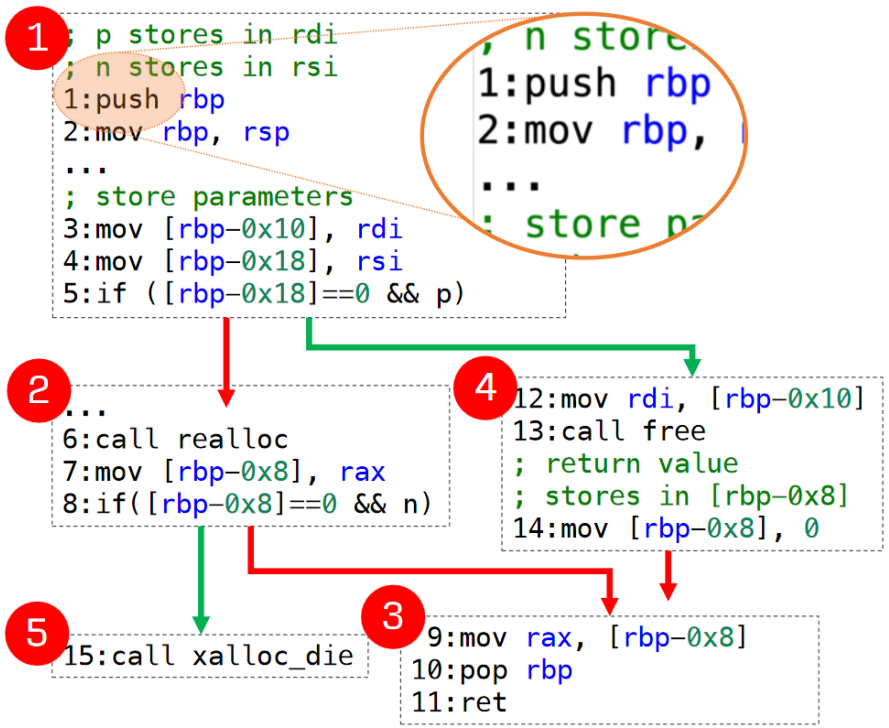
- 示例

```
1   void xalloc_die(){
2       ...
3       exit(-1);
4   }
5   void*xrealloc(void*p,
6                 size_t n){
7     if (!n && p){
8       free (p);
9       return NULL;
10    }
11    p = realloc (p, n);
12    if (!p && n)
          xalloc_die ();
13    return p;
14  }
```

(a) Source Code




(b) CFG (GCC -O3)



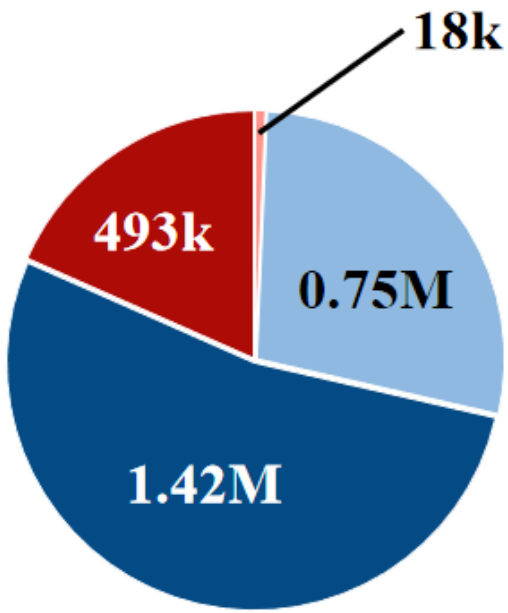

(c) CFG (Clang -O0)

- jTrans模型对函数序言中的endbr64指令过度敏感，而它与程序功能和语义无关
- 通过手动添加或删除该指令，能够将相似性得分提升至0.6或0.52

# DiEmph 动机（Motivation）

- 进一步研究jTrans的训练数据集
  - 相似函数对与不相似函数对的比例为1:2
  - 当函数对中仅有一个函数包含endbr64指令时，相似和不相似函数对的数量比例为1:27
  - 由于编译器的确定性和固有约定， endbr64指令在分布上存在不希望的偏差

$$\frac{18k}{493k} \approx \frac{1}{27}$$

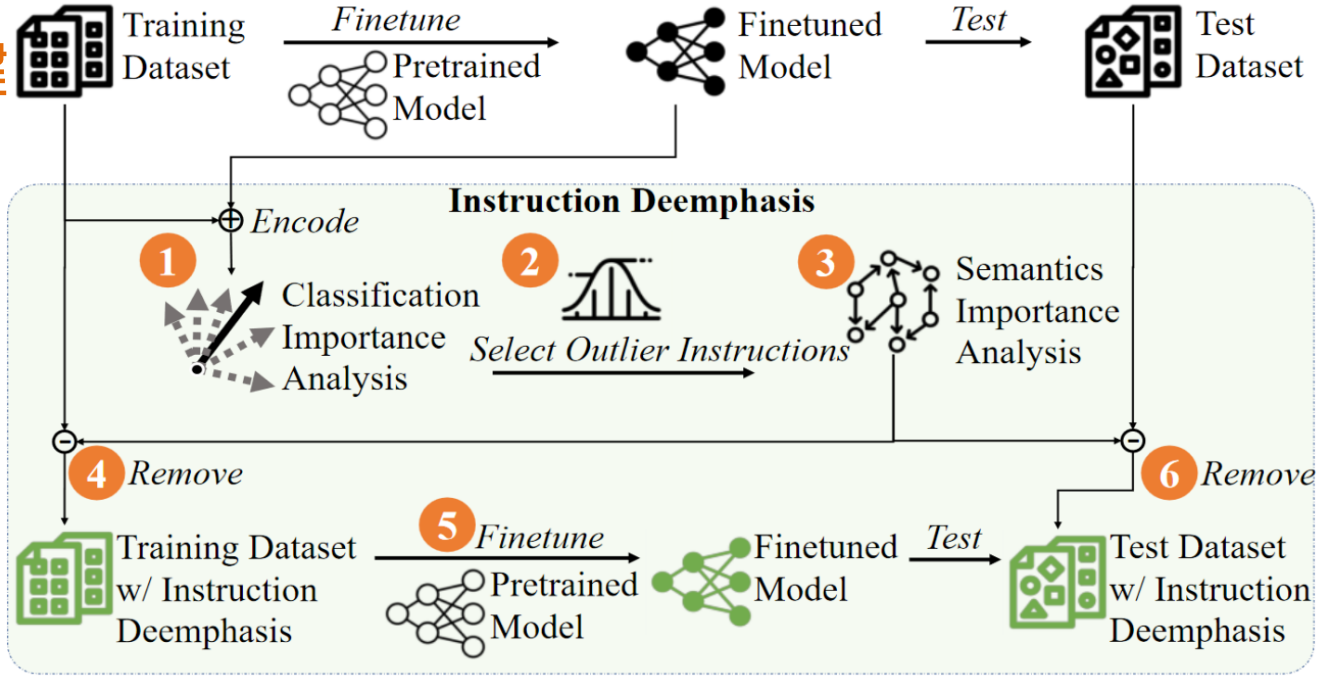



- Dissimilar Function Pairs with endbr64 in **one** of the functions
- Similar Function Pairs with endbr64 in **one** of the functions
- Dissimilar Function Pairs with endbr64 in **both/neither** of the functions
- Similar Function Pairs with endbr64 in **both/neither** of the functions


为什么不能简单地移除数据集中所有的指令分布偏差?

# DiEmph 算法原理

- 核心思想：降低那些不表示程序关键语义的指令的重要性
- 算法原理图
  - 步骤1：指令的分类重要性分析
  - 步骤2：选择异常值指令
  - 步骤3：指令的语义重要性分析
  - 步骤4：从微调数据集中移除分类重要性高但语义上不重要的指令
  - 步骤5：重新进行微调过程，得到改进后的模型
  - 步骤6：移除测试数据集中的问题指令，确保训练和测试空间的一致性




# DiEmph 指令分类重要性分析

- 指令分类重要性分析
  - 将模型视为黑盒
  - 指令$i$关于函数$f$的分类重要性$I_c(i)$，定义为从$f$中移除指令$i$导致的函数嵌入变化

$$I_c(i) = 1 - \cos(emb(f), emb(f \backslash \{i\}))$$

  - 具体算法
    - 从微调数据集中采样$N$个函数
    - 使用核密度估计（KDE）找出异常值，即对于每个函数，分类重要性异常大的指令
    - 记录每条指令被认为是异常指令的次数
    - 返回被认为是异常值频率最高的前K个指令列表

**Algorithm 1:** Selecting Instructions with High Classification Importance

```
1  Function select(model, functions)
       // importantInstr is a map from instructions to integers.
       // It counts how may times an instruction is considered
          important. Initially, all instructions are mapped to 0.
2      importantInstr = {}
3      for f ∈functions do
4          allInstrs = []
5          for i ∈ f do
               // emb(f) denotes using model to encode f
6              I_c(i) = cos(emb(f), emb(f\{i}))
7              allInstrs.append(i, I_c(i) )
8          outliers = KDE(allInstrs)
9          for i, _ ∈ outliers do
10             importantInstr[i] += 1
11     return largestK(importantInstr)
```

# DiEmph 指令语义重要性分析

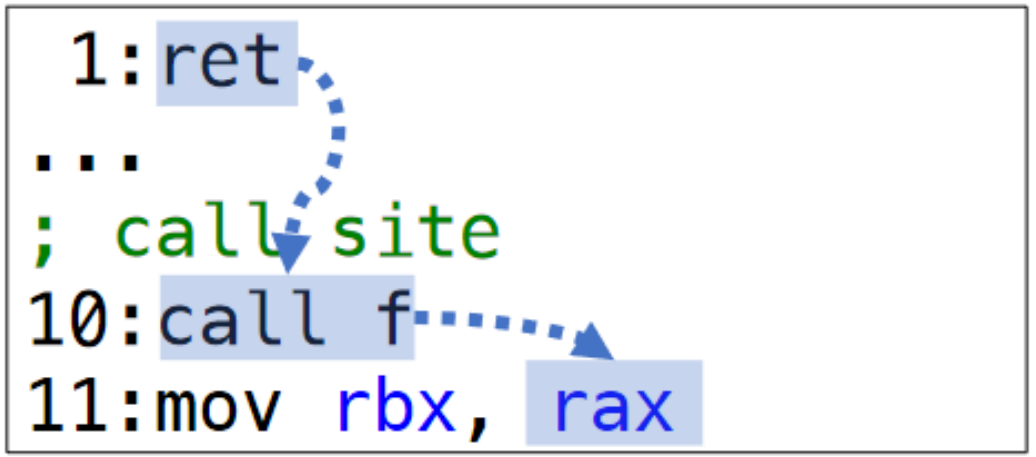
- 若指令$i$存在于函数$f$的任何稳定变量的后向切片中，认为$i$对函数$f$具有语义重要性
- 如果$i$对数据集中至少一定数量的函数都具有语义重要性，那么认为它对整个数据集具有语义重要性
- 具体步骤
  - 识别函数中的稳定变量
  - 计算稳定变量的后向程序切片
- 识别函数中的稳定变量
  - 识别全局变量和堆变量
    - 通过识别内存访问指令是否访问全局地址或堆地址

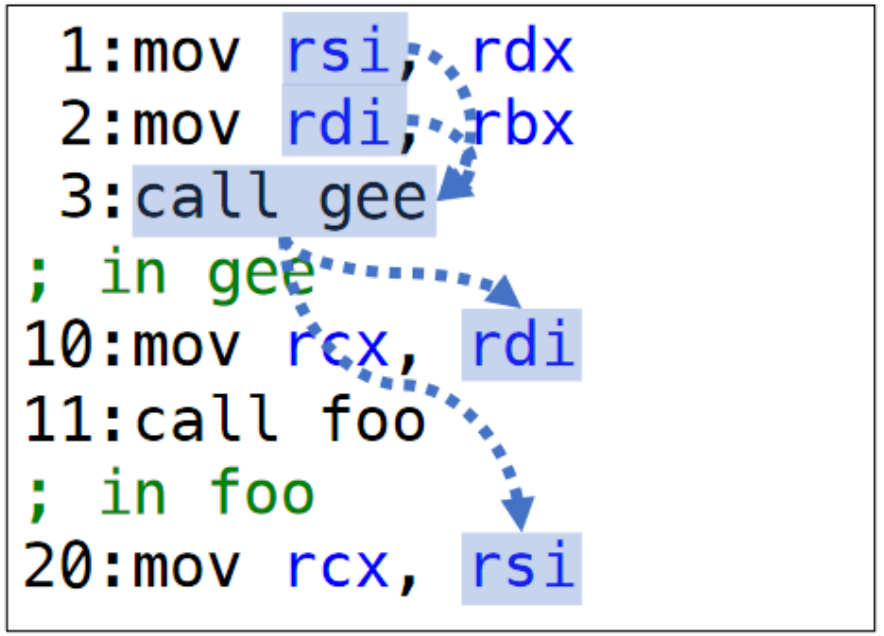
```
 1: lea rdi, GLOBAL_VAR
 2: ...
10: mov [rdi], 0
```

```
 1: call malloc
 2: mov rbx, rax
...
10: mov [rbx+0x10], 0
```

# DiEmph 指令语义重要性分析

- 识别函数返回变量
  - 判断一个函数是否存在返回值
    - 搜索在ret指令前是否对rax进行的任何写入操作的方法不可行
- 作为函数实际参数传递的变量
  - 利用编译器的惯例，即前六个实际参数是通过寄存器从调用者传递给被调用者的
  - 通过检查跨函数调用边界的寄存器数据流，可以识别出作为函数参数传递的变量



# DiEmph 指令语义重要性分析

- 计算稳定变量的后向程序切片
  - 排除栈地址依赖：如果涉及内存访问的地址表示一个栈地址，会排除由地址计算引入的依赖关系
  - 基址指针寄存器rbp与栈指针寄存器rsp之间的区域表示整个栈帧

```
1   int g;
2   int foo(){
3     int x;
4     ...
5     g = x;
6   }
```

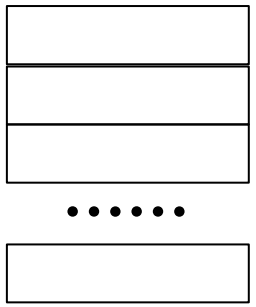
(a) Source Code of foo()

```
 1:push rbp
 2:push rbx
 3:mov rbp, rsp
 4:sub rsp, 0x20
...
 9:mov [rbp-0x10], rcx
10:mov STABLE_VAR,[rbp-0x10]
```

(b) Optimized Version

```
 1:push rbp
 2:push rbx
 3:mov rbp, rsp
 4:sub rsp, 0x50
...
 9:mov [rbp-0x20], rcx
10:mov STABLE_VAR,[rbp-0x20]
```

(c) Un-optimized Version



栈的内存分配是从高地址到低地址！

# 实验设计 数据资源

- 训练数据集（微调数据集）

| 数据集名称 | BinaryCorp-3M(BC) | BinKit(BK) | HowSolve(HS) |
|---|---|---|---|
| 样本数量 | 300万个二进制函数<br>（对应60万个源代码函数） | 450万个二进制函数<br>（对应12.6万个源代码函数） | 440万个二进制函数<br>（对应11万个源代码函数） |
| 编译器 | GCC v9.2.1 | GCC v4.9.4/v5.5.0/v6.4.0/v7.3.0/v8.2.0<br>Clang v4.0.0/v5.0.0/v6.0.0/v7.0.0 | GCC v4.8.5/v5.5.0/v7.4.0/v9.2.1<br>Clang v3.5.2/v5.0.1/v7.0.0/v9.0.0 |
| 优化选项 | O0/O1/O2/O3/Os | O0/O1/O2/O3 | O0/O1/O2/O3/Os |
| 目标架构 | x64 | 仅使用针对x64目标架构编译的程序 | |

- 测试数据集

| 数据集名称 | 数据集-I（分布内测试数据集） | 数据集-II（分布外测试数据集） |
|---|---|---|
| 包含项目 | Curl、Coreutils、Binutils、ImageMagick、SQLite、OpenSSL、Putty | |
| 编译设置 | GCC v7.5 –O0 x64<br>GCC v7.5 –O3 x64 | GCC v9.4 –O3 x64<br>Clang v10.0.0 –O0 x64 |

确保测试数据集和训练数据集之间没有重叠的函数

# 实验设计 数据资源

- 对比方法（基线模型）
  - jTrans(2022，SOTA)：$jTrans_{BC}$、$jTrans_{BK}$、$jTrans_{HS}$
  - Trex(2020)：$Trex_{BC}$、$Trex_{BK}$、$Trex_{HS}$
- 评价指标
  - PR@1(Precision at 1)：正确函数被识别为与查询函数最相似的函数的次数占总查询次数的比例（每个查询函数在候选函数池中有且仅有1个相似函数）
  - PR@5(Precision at 5)
  - PR@10(Precision at 10)
  - 平均倒数排名(MRR)：$MRR = \frac{1}{N}\sum_{1}^{N}\frac{1}{p_i}$

    $N$：查询样本总数；$p_i$：第$i$个查询样本对应的相似函数的排名结果
- 超参数
  - 采样函数数量$N = 200$、移除指令数量$K = 4$、候选函数池函数数量$S = 500$

# 实验设计 对比实验

- **RQ1：DiEmph在分布外测试数据集上对于基线模型的性能提升**

| Programs | $JTrans_{BC}$ | | | $JTrans_{BK}$ | | | $JTrans_{HS}$ | | | $Trex_{BC}$ | | | $Trex_{BK}$ | | | $Trex_{HS}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. |
| Curl | 42.6 | 61.8 | 19.2 | 54.0 | 55.6 | 1.6 | 50.6 | 50.6 | 0.0 | 48.8 | 50.2 | 1.4 | 35.6 | 40.2 | 4.6 | 25.8 | 33.8 | 8.0 |
| Binutils | 37.6 | 53.2 | 15.6 | 49.8 | 54.4 | 4.6 | 36.4 | 40.4 | 4.0 | 37.6 | 44.0 | 6.4 | 33.8 | 34.8 | 1.0 | 21.4 | 27.0 | 5.6 |
| Coreutils | 31.4 | 42.6 | 11.2 | 37.4 | 41.8 | 4.4 | 32.6 | 34.0 | 1.4 | 33.4 | 40.2 | 6.8 | 28.0 | 31.6 | 3.6 | 22.4 | 28.2 | 5.8 |
| ImageMagick | 22.6 | 39.0 | 16.4 | 42.2 | 46.2 | 4.0 | 30.6 | 43.4 | 12.8 | 39.6 | 45.8 | 6.2 | 27.6 | 33.8 | 6.2 | 22.8 | 29.6 | 6.8 |
| SQLite | 42.8 | 60.0 | 17.2 | 51.4 | 56.8 | 5.4 | 42.4 | 56.0 | 13.6 | 65.6 | 66.8 | 1.2 | 44.0 | 48.2 | 4.2 | 32.6 | 40.2 | 7.6 |
| OpenSSL | 47.6 | 53.8 | 6.2 | 46.2 | 52.0 | 5.8 | 54.8 | 61.0 | 6.2 | 46.2 | 50.2 | 4.0 | 32.8 | 38.4 | 5.6 | 28.4 | 33.0 | 4.6 |
| Putty | 35.8 | 50.6 | 14.8 | 36.8 | 39.2 | 2.4 | 42.0 | 45.4 | 3.4 | 39.2 | 38.6 | -0.6 | 34.4 | 37.2 | 2.8 | 27.0 | 33.8 | 6.8 |
| **Average** | **37.2** | **51.6** | **14.4** | **45.4** | **49.4** | **4.0** | **41.3** | **47.3** | **6.0** | **44.3** | **48.0** | **3.7** | **33.7** | **37.7** | **4.0** | **25.7** | **32.2** | **6.5** |

PR@1

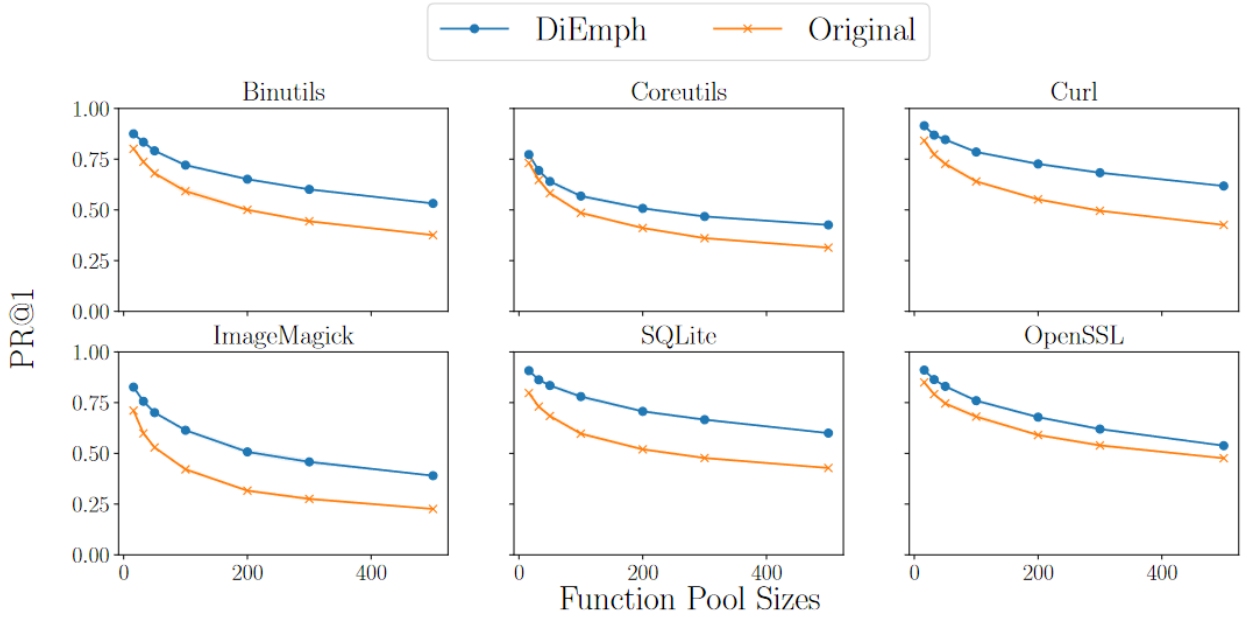
| Programs | $JTrans_{BC}$ | | | $JTrans_{BK}$ | | | $JTrans_{HS}$ | | | $Trex_{BC}$ | | | $Trex_{BK}$ | | | $Trex_{HS}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. | Ori. | DiEmph | Impr. |
| Curl | 55.3 | 71.9 | 16.6 | 63.9 | 65.9 | 2.0 | 60.8 | 62.4 | 1.6 | 60.0 | 61.5 | 1.5 | 47.1 | 50.7 | 3.6 | 38.4 | 46.0 | 7.6 |
| Binutils | 50.5 | 64.6 | 14.1 | 60.5 | 64.3 | 3.8 | 48.8 | 52.5 | 3.7 | 49.8 | 55.1 | 5.2 | 44.0 | 47.1 | 3.2 | 33.7 | 39.6 | 6.0 |
| Coreutils | 42.1 | 51.7 | 9.5 | 48.0 | 50.4 | 2.4 | 42.7 | 45.3 | 2.6 | 42.7 | 49.6 | 6.8 | 35.4 | 39.7 | 4.3 | 32.2 | 37.6 | 5.4 |
| ImageMagick | 34.0 | 51.8 | 17.8 | 53.6 | 57.5 | 3.9 | 41.1 | 54.6 | 13.5 | 51.7 | 57.7 | 6.0 | 36.3 | 44.1 | 7.9 | 32.9 | 41.1 | 8.2 |
| SQLite | 53.1 | 70.6 | 17.5 | 61.8 | 66.8 | 5.1 | 52.7 | 65.9 | 13.2 | 74.0 | 74.2 | 0.3 | 53.1 | 58.3 | 5.1 | 44.9 | 52.5 | 7.6 |
| OpenSSL | 58.4 | 64.8 | 6.4 | 55.9 | 62.0 | 6.0 | 65.5 | 69.2 | 3.7 | 57.3 | 60.3 | 3.0 | 42.3 | 47.7 | 5.4 | 40.0 | 45.2 | 5.2 |
| Putty | 46.8 | 59.2 | 12.4 | 47.7 | 49.4 | 1.7 | 52.5 | 55.3 | 2.8 | 50.3 | 48.1 | -2.2 | 43.7 | 48.1 | 4.3 | 37.3 | 44.1 | 6.7 |
| **Average** | **48.6** | **62.1** | **13.5** | **55.9** | **59.5** | **3.6** | **52.0** | **57.9** | **5.9** | **55.1** | **58.1** | **2.9** | **43.1** | **48.0** | **4.8** | **37.1** | **43.7** | **6.7** |

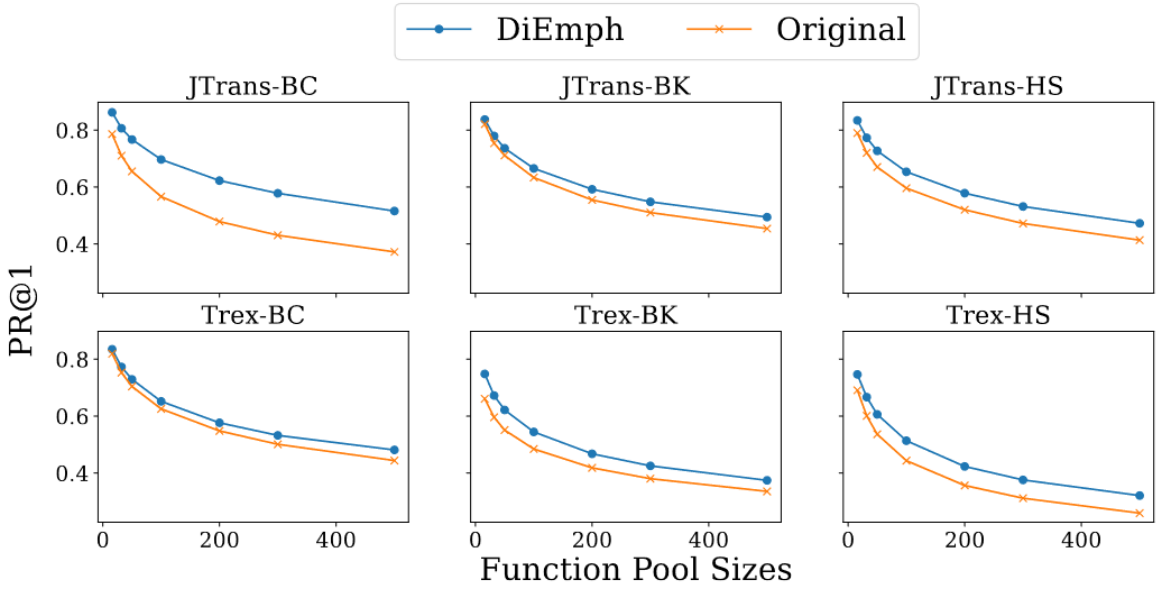
MRR

- **实验结论**
  - **DiEmph将基线模型的PR@1提升了3.7-14.4%，MRR提升了2.9-13.5%**
  - **对于使用BC数据集训练的JTrans模型，提升最为显著（新的SOTA）**

# 实验设计 对比实验

- **RQ2：DiEmph在不同候选函数池大小下的有效性**
  - **实验条件：分布外测试数据集、7个不同函数池大小、实验重复10次取平均值**




- **实验结论**
  - **DiEmph能够有效提升所有基线模型在不同候选函数池大小上的性能**
  - **当候选函数池的大小超过100时，性能提升更加显著**

# 实验设计 对比实验

- **RQ3：DiEmph在分布内测试数据集上对于基线模型的性能影响** PR@1

| Programs | $JTrans_{BC}$ | | | $JTrans_{BK}$ | | | $JTrans_{HS}$ | | | $Trex_{BC}$ | | | $Trex_{BK}$ | | | $Trex_{HS}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ori. | DiEmph | Diff. | Ori. | DiEmph | Diff. | Ori. | DiEmph | Diff. | Ori. | DiEmph | Diff. | Ori. | DiEmph | Diff. | Ori. | DiEmph | Diff. |
| Curl | 68.8 | 70.4 | 1.6 | 46.8 | 51.4 | 4.6 | 66.4 | 67.8 | 1.4 | 57.2 | 56.0 | -1.2 | 27.0 | 36.0 | 9.0 | 43.4 | 43.8 | 0.4 |
| Binutils | 64.2 | 67.6 | 3.4 | 49.8 | 55.0 | 5.2 | 62.6 | 63.0 | 0.4 | 57.4 | 55.4 | -2.0 | 26.2 | 36.2 | 10.0 | 36.8 | 43.6 | 6.8 |
| Coreutils | 34.6 | 44.0 | 9.4 | 31.6 | 44.6 | 13.0 | 36.4 | 40.4 | 4.0 | 38.0 | 38.2 | 0.2 | 15.2 | 24.8 | 9.6 | 20.0 | 25.4 | 5.4 |
| ImageMagick | 43.8 | 47.8 | 4.0 | 41.0 | 43.4 | 2.4 | 50.2 | 49.4 | -0.8 | 44.8 | 43.6 | -1.2 | 26.4 | 35.0 | 8.6 | 28.2 | 31.6 | 3.4 |
| SQLite | 66.6 | 70.2 | 3.6 | 40.6 | 46.2 | 5.6 | 58.0 | 55.8 | -2.2 | 56.0 | 53.0 | -3.0 | 26.0 | 30.4 | 4.4 | 39.2 | 41.8 | 2.6 |
| OpenSSL | 64.4 | 71.2 | 6.8 | 46.8 | 58.2 | 11.4 | 69.0 | 72.8 | 3.8 | 56.2 | 59.4 | 3.2 | 30.4 | 40.6 | 10.2 | 43.2 | 45.0 | 1.8 |
| Putty | 65.0 | 60.2 | -4.8 | 51.4 | 50.2 | -1.2 | 62.4 | 62.8 | 0.4 | 55.8 | 55.0 | -0.8 | 24.4 | 35.8 | 11.4 | 42.8 | 47.0 | 4.2 |
| **Average** | **58.2** | **61.6** | **3.4** | **44.0** | **49.9** | **5.9** | **57.9** | **58.9** | **1.0** | **52.2** | **51.5** | **-0.7** | **25.1** | **34.1** | **9.0** | **36.2** | **39.7** | **3.5** |

- **实验结论**
  - **DiEmph对于基线模型在分布内测试数据集上的提升较小，对于PR@1的提升幅度为1.0-5.9%**
  - **对于$Trex_{BC}$模型，出现了负提升**

**在BK数据集上出现了过拟合！**

| $JTrans_{BK}$ | | |
|---|---|---|
| Ori. | DiEmph | Impr. |
| 54.0 | 55.6 | 1.6 |
| 49.8 | 54.4 | 4.6 |
| 37.4 | 41.8 | 4.4 |
| 42.2 | 46.2 | 4.0 |
| 51.4 | 56.8 | 5.4 |
| 46.2 | 52.0 | 5.8 |
| 36.8 | 39.2 | 2.4 |
| **45.4** | **49.4** | **4.0** |

| $Trex_{BK}$ | | |
|---|---|---|
| Ori. | DiEmph | Impr. |
| 35.6 | 40.2 | 4.6 |
| 33.8 | 34.8 | 1.0 |
| 28.0 | 31.6 | 3.6 |
| 27.6 | 33.8 | 6.2 |
| 44.0 | 48.2 | 4.2 |
| 32.8 | 38.4 | 5.6 |
| 34.4 | 37.2 | 2.8 |
| **33.7** | **37.7** | **4.0** |

# 实验设计 运行时间效率实验

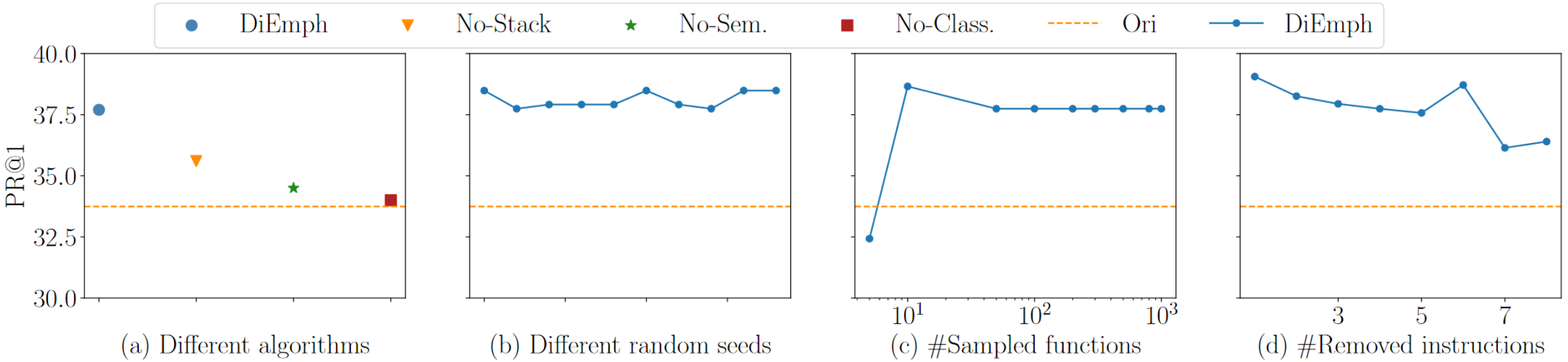
- **RQ4：DiEmph的运行时间效率（分钟）**
  - **采样函数数量$N = 200$、移除指令数量$K = 4$**

| Model | KDE | Class. Imp. | All |
|---|---|---|---|
| $JTrans_{BC}$ | 4 | 9 | 15 |
| $JTrans_{BK}$ | 10 | 22 | 33 |
| $JTrans_{HS}$ | 13 | 25 | 39 |
| $Trex_{BC}$ | 13 | 14 | 29 |
| $Trex_{BK}$ | 14 | 16 | 29 |
| $Trex_{HS}$ | 14 | 17 | 31 |
| Average | **11** | **17** | **29** |

- **实验结论**
  - **由于使用DiEmph改进模型是一次性工作，所以它的时间消耗是可接受的**
  - **总消耗时间 ≈ 指令分类重要性分析消耗的时间 + 选择出异常指令消耗的时间**

# 实验设计 消融实验

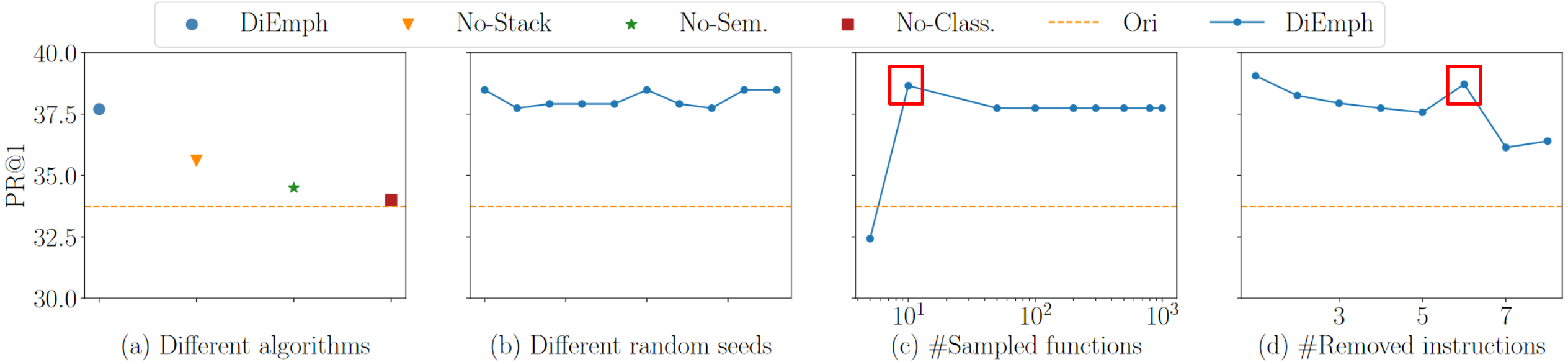
- 消融实验：验证排除栈地址依赖、语义重要性分析、指令分类重要性分析的影响
  - 实验条件：$Trex_{BK}$基线模型、分布外测试数据集




- 实验结论
  - No-Stack变体能够改进基线模型的性能，但提升幅度不如DiEmph显著
  - No-Sem.变体也略微提升了基线模型的性能
  - No-Class.变体相较于原始模型几乎没有表现出任何改进

# 实验设计 消融实验

- 消融实验：验证随机种子、采样函数数量$N$、移除指令数量$K$对DiEmph的影响
  - 实验条件：$\text{Trex}_{\text{BK}}$基线模型、分布外测试数据集




- 实验结论
  - 在使用不同随机种子的情况下，DiEmph能够一致地提升基线模型的性能
  - 随着采样函数数量$N$的增大， DiEmph的效果逐渐变好后趋于稳定
  - 随着移除指令数量$K$的增大，DiEmph的效果逐渐变差，最后急剧下降

# 算法总结 DiEmph

- 算法贡献
  - 结合深度学习模型分析技术和二进制程序分析技术，首次提出了一种识别指令分类重要性和语义重要性的方法
  - 算法可有效防止模型学习由编译器引入的指令分布偏差，有效提升模型在分布外数据上的泛化能力
- 算法不足
  - 不适用于其他模型，例如基于GNN的模型
  - 反汇编工具（IDA Pro）生成反汇编代码的质量会影响方法的性能
  - 只能实现单架构二进制函数相似性分析

# Asteria-Pro: Enhancing Deep-Learning Based Binary Code Similarity Detection by Incorporating Domain Knowledge

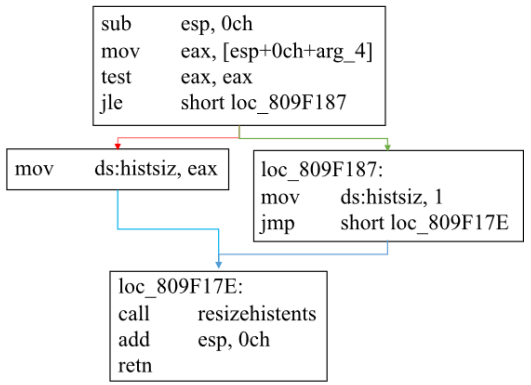
# Asteria-Pro TIPO

| | | |
|---|---|---|
| T | 目标 | 实现高效、准确的跨架构大规模二进制函数相似性检测 |
| I | 输入 | 目标二进制函数*1，候选二进制函数*N |
| P | 处理 | 1.基于领域知识的预过滤：高效地从候选函数中过滤掉不相似的函数<br>2.基于深度学习的相似性计算：计算目标函数和所有候选函数的相似度得分，并输出与目标函数相似度最高的前K个候选同源函数列表<br>3.基于领域知识的重排序：使用函数调用关系对候选同源函数重排序 |
| O | 输出 | 与目标二进制函数相似度最高的前K个候选同源二进制函数列表*1 |

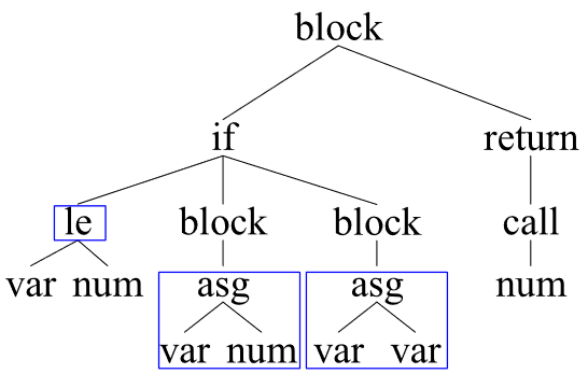
| | | |
|---|---|---|
| P | 问题 | 现有方法需对所有候选函数（其中绝大多数是非同源函数）进行特征提取和编码，时间成本较高；难以区分相似但非同源的函数，准确率不足 |
| C | 条件 | 数据集中的二进制程序可被正常反汇编且未经过代码混淆技术处理 |
| D | 难点 | 如何在深度学习模型进行特征提取和编码前尽可能多地排除非同源函数的同时保留所有同源函数；如何增强方法区分相似但非同源函数的能力 |
| L | 水平 | TOSEM 2023（SCI一区） |

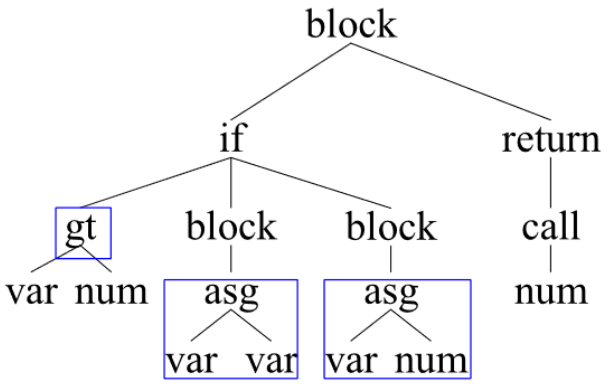
# Asteria-Pro 必要知识

- 控制流图(Control Flow Graph, CFG)
  - 二进制函数的重要表征之一，包含丰富的代码控制流信息
  - 节点表示基本块（不包含分支的一段语句序列），边表示跳转关系
- 抽象语法树(Abstract Syntax Code, AST)
  - 函数抽象语法结构的树结构化表示，包含丰富的代码语法和语义信息
  - 每个节点对应于源代码中的一个语句或表达式
  - 使用IDA Pro反汇编工具和Hexray Decompiler反编译插件对二进制代码执行反编译，生成C伪代码，并提取AST
  - AST基于更高层次中间表示的伪代码生成，能够在不同架构之间保持较高的一致性




```
LDR     R3, =histsiz
CMP     R1, #0
MOVLER2, #1
STRGT   R1, [R3]
STRLE   R2, [R3]
B       resizehistents
```

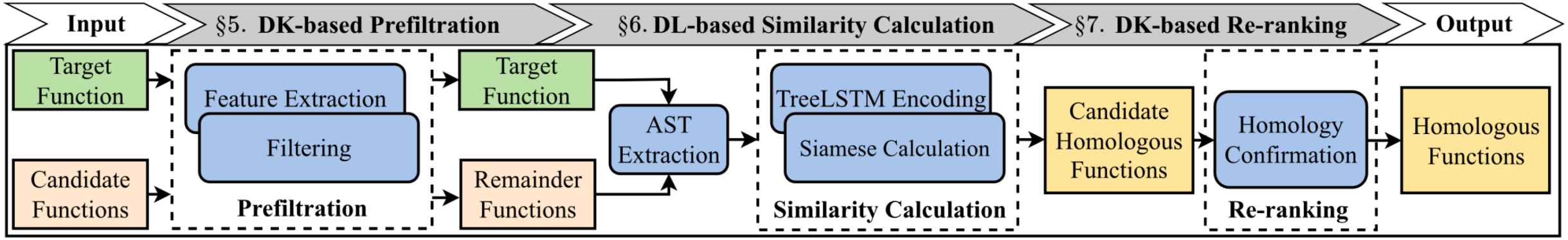


- 动态符号表(Dynamic Symbol Table, DST)
  - 程序运行时用于解析动态链接库中函数和变量地址的数据结构
  - 为了便于动态链接，DST中的函数（通常是导入或导出函数）的名称将被保留，而不是在剥离后被移除
- 函数内联(Function inline)
  - 将被调用函数的代码直接插入到调用函数中，而不进行单独的函数调用
  - 通过减少函数调用的开销和提高缓存利用率来提高程序性能
  - 是否内联一个函数通常由编译器基于各种因素决定，例如函数大小、调用频率和可用的寄存器空间
- 内建函数(Intrinsic function)
  - 由编译器自身实现的特殊函数，提供了对硬件的低级访问，并用于实现各种低级操作，如算术、位操作和内存访问，也称为内置函数(built-in functions)

# Asteria-Pro 算法原理

- 算法原理图
  - 基于领域知识的预过滤
    - 利用轻量级函数特征，高效地从候选函数中过滤掉不相似的函数
  - 基于深度学习的相似性计算
    - 提取目标函数和剩余候选函数的AST，使用Tree-LSTM将其编码为向量，并使用孪生神经网络确定目标函数和所有候选函数的相似度得分，输出与目标函数相似度最高的前K个候选同源函数列表
  - 基于领域知识的重排序
    - 使用函数调用关系对上一个模块产生的候选同源函数进行重排序



# Asteria-Pro 基于领域知识的预过滤

- 目标：在尽可能有效地过滤掉非同源函数的同时保留所有同源函数
- 函数特征选择
  - CFG类特征
    - 指令数量(No. Instruction)、算术指令数量(No. Arithmetic)、调用指令数量(No. Callee)、逻辑指令数量(No. Logic)、字符串常量(String Constant)、数值常量(Numeric Constant)、命名被调用函数列表(Named Callee List, NCL)
  - AST类特征
    - AST节点数量(No. AST Nodes)、AST节点聚类(AST Node Cluster)、AST模糊哈希(AST Fuzzy Hash)
  - 命名被调用函数列表(Named Callee List, NCL)
    - 基于调用图(Call Graph, CG)构建
    - $CG = \{V, E\}$，$V = \{v | v\ is\ a\ function\}$，$E = \{(u, v) | u\ calls\ v\}$
    - $NCL_f = \{v | v \in V, v \in DST, (f, v) \in E\}$，其中$v$按其调用指令的地址排序

# Asteria-Pro 基于领域知识的预过滤

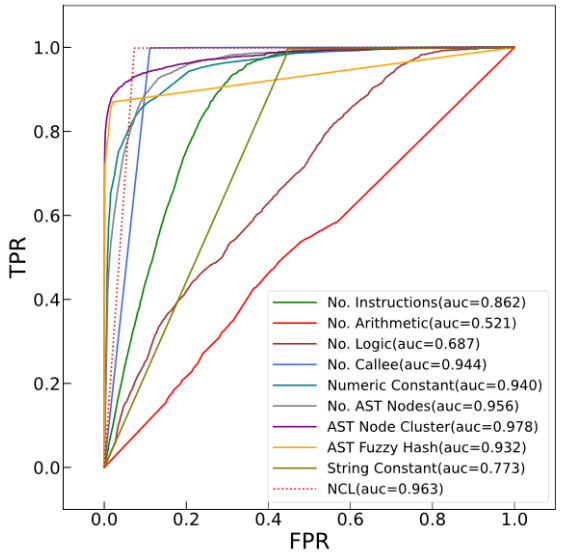
- 函数特征评估：识别出最高效和最有效的过滤特征
  - 评估数据集：针对x86、x64、ARM、PowerPC四种架构，分别对应40111个函数
  - 评估方法：随机选择$n$个目标函数，分别构建包含$M$个随机函数和3个同源函数的候选函数池。分别计算目标函数与所有候选函数的特征相似度得分，低于阈值T的函数被过滤。剩余函数中同源函数被视为真正例，非同源函数对被视为假正例
  - 特征相似度得分计算
    - 数值类型特征：计算相对差异比(Relative Difference Ratio, RDR)

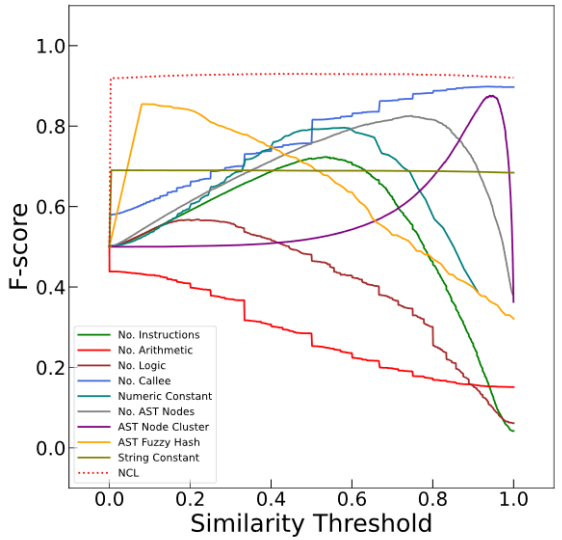
$$RDR(V_1, V_2) = 1 - \frac{abs(V_1 - V_2)}{max(V_1, V_2)}$$

    - 序列类型特征：基于最长公共子序列(Longest Common Sequence, LCS)计算公共序列比(Common Sequence Ratio, CSR)

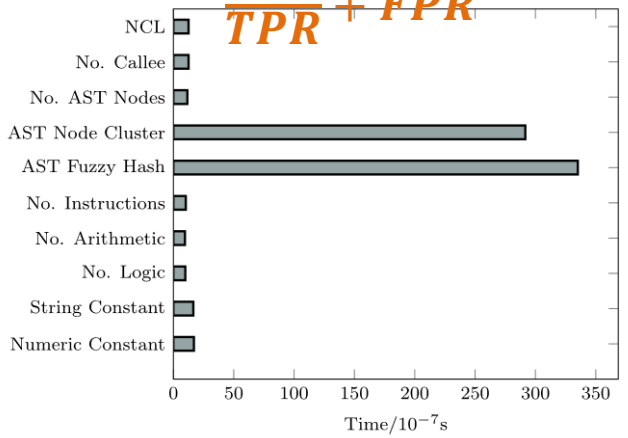
$$CSR(S_1, S_2) = \frac{2 \times LCS(S_1, S_2)}{len(S_1) + len(S_2)}$$

- 指标
  - 真正例率(TPR)：$TPR = \frac{\sum_{i=1}^{n} TP_i^p}{3 \times n}$，表示特征保留同源函数的能力
  - 假正例率(FPR)：$FPR = \frac{\sum_{i=1}^{n} FP_i}{n \times (M \times 4 - 4)}$，表示特征排除非同源函数的能力
  - AUC
- 评估结果($n = 1000, M = 20000$)
  - 在预过滤设计中使用NCL以及No. Callee和String Constant这三个特征



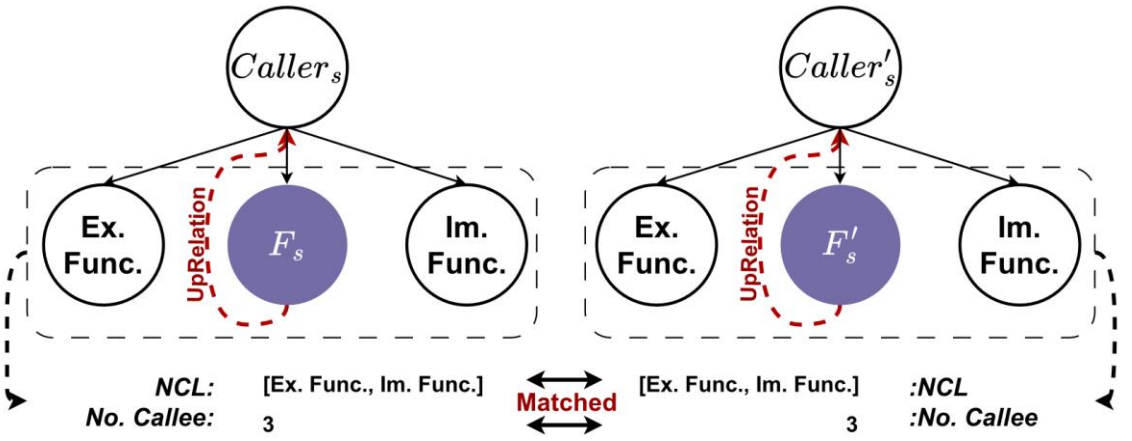

$$F_{score} = \frac{1}{\frac{1}{TPR} + FPR}$$



# Asteria-Pro 基于领域知识的预过滤

- 人工分析被NCL、No. Callee过滤导致的同源函数假阴性，确定了3个开发挑战
  - C1：被修饰的被调用函数名称（编译器的名称重整）
    - 使用恢复工具cxxfilt来恢复函数名（这在逆向工程中是一个著名的研究领域）
  - C2：调用图中的叶节点函数不存在被调用函数
    - 提出一种新的算法UpRelation
  - C3：二进制函数中的函数调用并不总是与源代码一致（函数内联、内建函数替换、用于优化的指令替换等）
    - 最后使用String Constant特征辅助过滤

- UpRelation算法
  - 对于非叶节点函数，逐一计算NCL、No. Callee和String Constant的特征相似度，并结合阈值进行过滤
  - 对于叶节点函数利用父节点来匹配相似的叶节点



# Asteria-Pro 基于领域知识的预过滤

- **UpRelation算法具体流程**
  - 判断$NCL_{fv}$是否为空，若不为空，计算$NCL_{fv}$与所有候选函数的$NCL_f$的共同序列比，并过滤结果低于阈值$T_{NCL}$的候选函数
  - 同样地，若$Callee_{fv}$不为空，计算$Callee_{fv}$与所有候选函数的$Callee_f$的相对差异比并过滤结果低于阈值$T_{Callee}$的候选函数

**ALGORITHM 1:** UpRelation

**Input**: Vulnerable Function $fv$, Target Function List $TFL$, Thresholds $T_{NCL}, T_{callee}, T_{string}$
**Output**: Vulnerable Candidate Function List $VFL$

```
1  VFL ← TFL;
2  if NCL_fv is not null then
3      for f ∈ TFL do
4          s = CSR(NCL_fv, NCL_f);
5          if s < T_NCL then VFL.pop(f);
6      end
7  else if Callee_fv > 0 then
8      for f ∈ TFL do
9          s = RDR(Callee_fv, Callee_f);
10         if s < T_callee then VFL.pop(f);
11     end
12 else
13     FL' = ∅;
14     for caller ∈ GetCallers(fv) do
15         for caller' ∈ UpRelation(caller, VFL) do
16             FL'.add(GetCallees(caller'));
17         end
18     end
19     VFL = FL';
20 if StrCons_fv is not null then
21     for f ∈ VFL do
22         s = CSR(StrCons_fv, StrCons_f);
23         if s < T_string then VFL.pop(f);
24     end
25 else
26     return
27 end
28 VFL;
```

# Asteria-Pro 基于领域知识的预过滤

- 若$NCL_{fv}$与$Callee_{fv}$均为空，表明目标函数$f_v$在调用图中是一个叶结点
  - 确定其所有父节点，即调用叶子节点的函数
  - 使用UpRelation算法找到所有与调用函数相似的函数
  - 这些所有候选函数调用的函数（即它的子节点）被认为是该叶子节点的候选函数
- 最后通过计算目标函数与剩余候选函数的字符串常量的共同序列比进行最终过滤

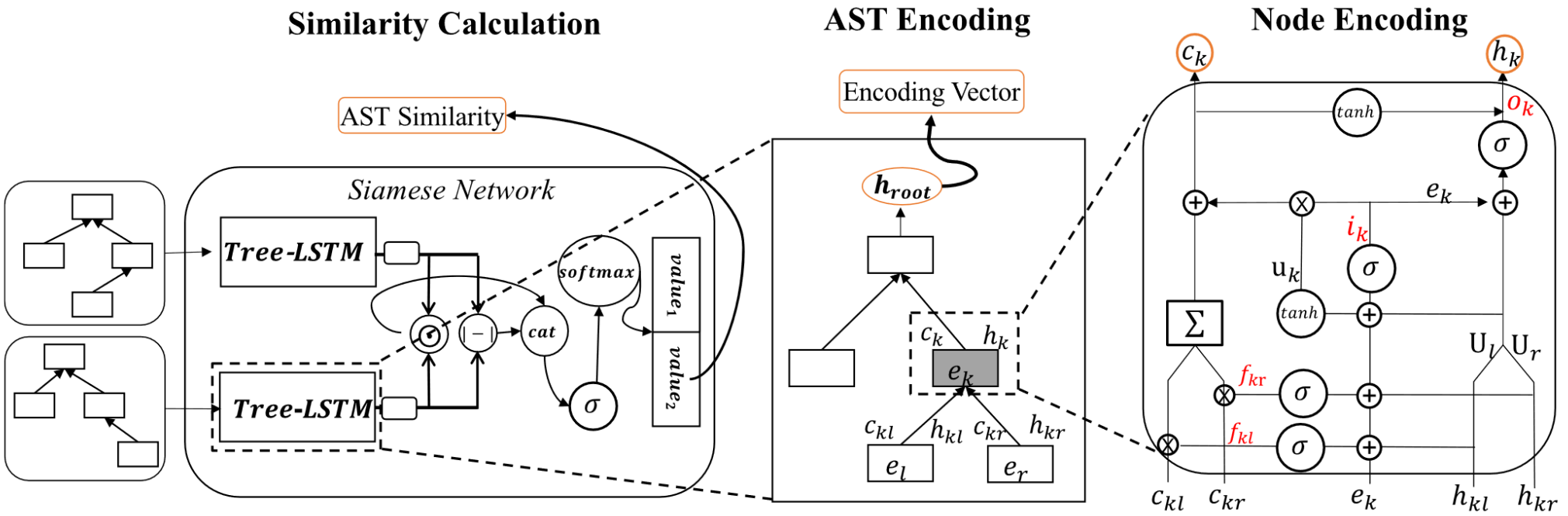
**ALGORITHM 1:** UpRelation

**Input**: Vulnerable Function $fv$, Target Function List $TFL$, Thresholds $T_{NCL}, T_{callee}, T_{string}$
**Output**: Vulnerable Candidate Function List $VFL$

```
1  VFL ← TFL;
2  if NCL_fv is not null then
3      for f ∈ TFL do
4          s = CSR(NCL_fv, NCL_f);
5          if s < T_NCL then VFL.pop(f);
6      end
7  else if Callee_fv > 0 then
8      for f ∈ TFL do
9          s = RDR(Callee_fv, Callee_f);
10         if s < T_callee then VFL.pop(f);
11     end
12 else
13     FL' = ∅;
14     for caller ∈ GetCallers(fv) do
15         for caller' ∈ UpRelation(caller, VFL) do
16             FL'.add(GetCallees(caller'));
17         end
18     end
19     VFL = FL';
20 if StrCons_fv is not null then
21     for f ∈ VFL do
22         s = CSR(StrCons_fv, StrCons_f);
23         if s < T_string then VFL.pop(f);
24     end
25 else
26     return
27 end
28 VFL;
```

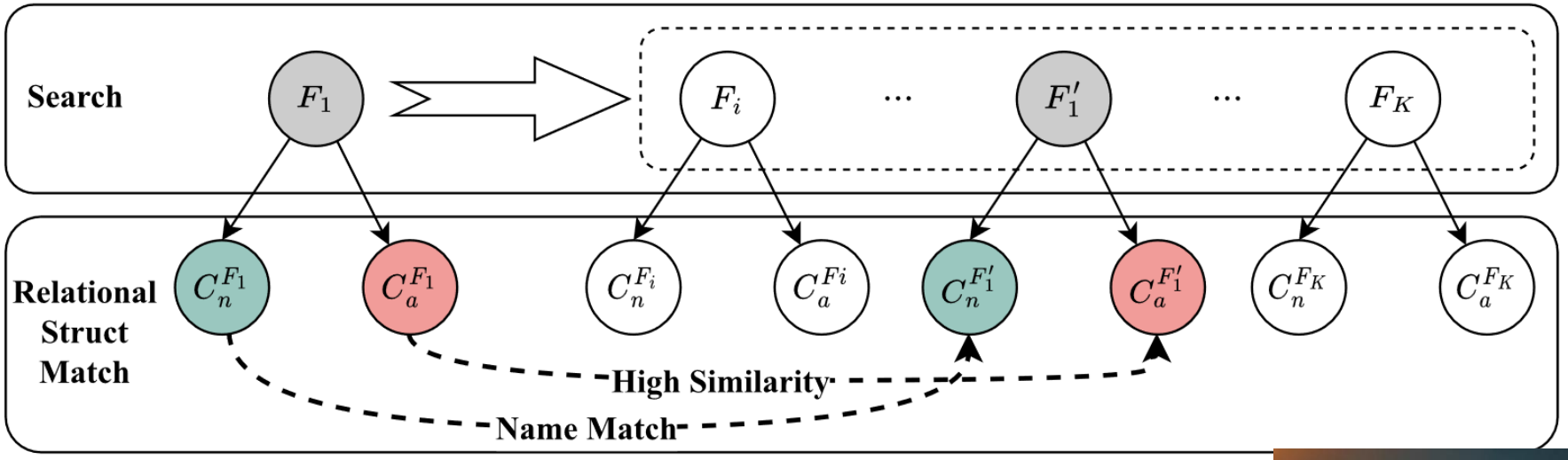
# Asteria-Pro 基于深度学习的相似性计算

- 输入：目标函数、剩余候选函数列表
- 输出：与目标函数相似度最高的前$K$个候选同源函数列表
- 提取目标函数和剩余候选函数的AST，使用Binary Tree-LSTM将其编码为向量，并使用孪生神经网络确定目标函数和所有候选函数的相似性得分，输出与目标函数相似度最高的前K个候选同源函数列表

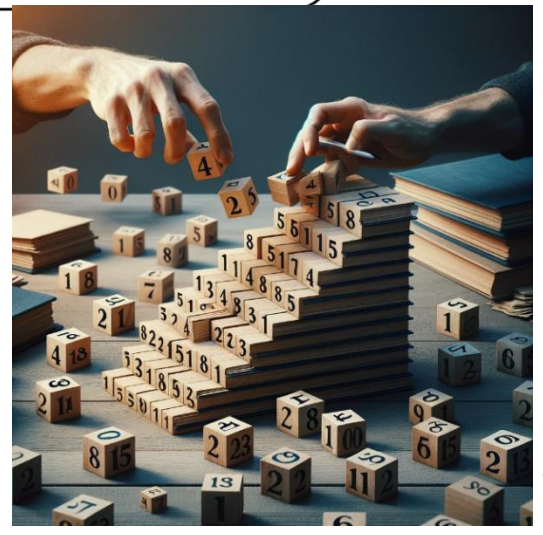


# Asteria-Pro 基于领域知识的重排序

- 目标：对相似性计算模块输出的前$K$个候选同源函数进行重排序
- 动机：Tree-LSTM在提取函数间信息上存在不足




- 计算新的匹配得分
  - 被调用函数分为：命名被调用函数和匿名被调用函数
  - 当目标函数$F_1$无被调用函数时
    - 直接移除所有具有一个或多个被调用函数的候选函数
    - 剩余候选函数根据它们的原始相似度得分进行重排序



- 当目标函数$F_1$存在被调用函数时
  - 命名被调用函数匹配得分计算
    - 基于函数名称匹配$F_1$与每个候选函数$F_i$的命名被调用函数，成功匹配数量记为$N_n^{F_i}$
  - 匿名被调用函数匹配得分计算
    - 使用基于深度学习的相似性计算模块计算$F_1$与每个候选函数$F_i$的匿名被调用函数之间的相似性得分。对于每个候选函数$F_i$，取其中最大的相似性得分表示为$S_{aj}^{F_i}$
  - 目标函数$F_1$与候选函数$F_i$之间新的匹配得分

$$M_{F_i} = N_n^{F_i} + \sum S_{aj}^{F_i}$$

- 最终匹配得分

原始相似度得分

$$S_{F_i}^{re-rank} = \alpha \times m_{F_i} + \beta \times M_{F_i}$$

$$(\alpha + \beta = 1)$$

# 实验设计 数据资源

两个数据集中均不包含参与预过滤测试的函数！

- **模型数据集**
  - 目标架构：x86、x64、ARM、PowerPC
  - 用途：用于Tree-LSTM模型训练和评估
  - 样本数量：314852对同源函数、314852对非同源函数，按8:2划分为训练集和验证集
- **评估数据集**
  - 目标架构：x86、x64、ARM、PowerPC
  - g-dataset
    - 用途：分类测试任务(Task-C)，评估模型分类同源和非同源函数对的能力
    - 形式：$(F,(F_h,F_n))$，$F$表示目标函数，$F_h$表示它的同源函数，$F_n$表示它的非同源函数
  - v-dataset
    - 用途：漏洞搜索测试任务(Task-V)，评估模型在更大的候选池中识别同源函数的能力
    - 形式：$(F,(F_h,F_{n1},\dots,F_{ni},\dots,F_{n10000}))$，$F$表示目标函数，$F_h$表示它的同源函数，$F_{n1},\dots,F_{ni},\dots,F_{n10000}$表示它的非同源函数

# 实验设计 数据资源

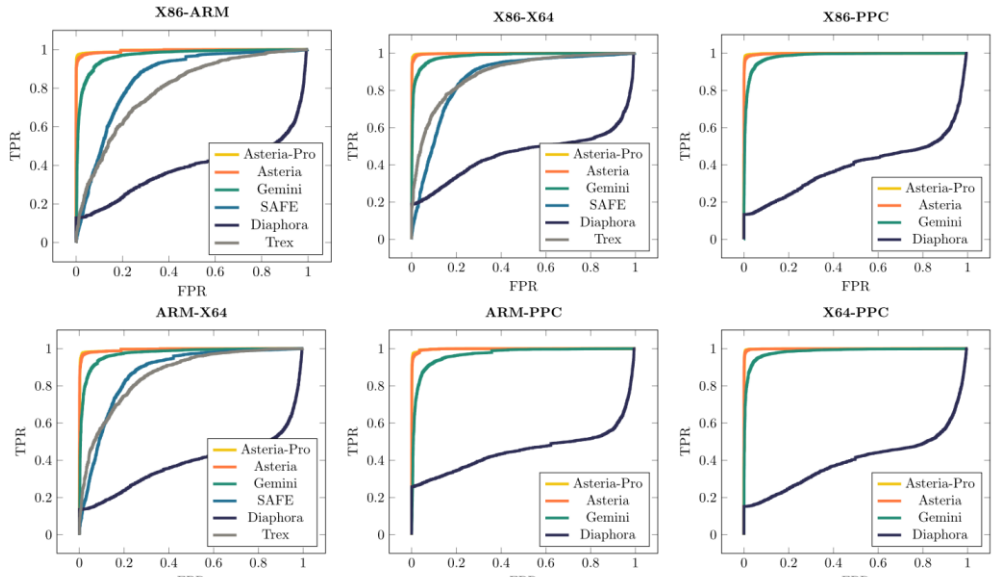
- 对比方法
  - Diaphora(2022)、Trex(2020)、SAFE(2019)、Gemini(2017)
- 评价指标
  - AUC（用于Task-C）
  - MRR（用于Task-V）
  - Recall@Top-1（用于Task-V）
  - Recall@Top-10（用于Task-V）

$$Recall@Top-K = \frac{\text{前}K\text{个结果中正样本的总数}}{\text{正样本总数}}$$

- 超参数设置
  - $T_{NCL} = 0.1$、$T_{callee} = 0.8$、$T_{string} = 0.8$
  - $\alpha = 0.1$、$\beta = 0.9$

# 实验设计 对比实验

- **RQ1：Asteria-Pro在跨架构和跨编译器检测方面的性能表现**
  - **跨架构检测**




| Metrics | Methods | X86-X64 | X86-ARM | X86-PPC | X64-ARM | X64-PPC | ARM-PPC | Avg |
|---|---|---|---|---|---|---|---|---|
| MRR | **ASTERIA-PRO** | **0.934** | **0.887** | **0.931** | **0.879** | **0.919** | **0.903** | **0.908** |
| | Asteria | 0.776 | 0.724 | 0.731 | 0.708 | 0.713 | 0.750 | 0.734 |
| | Trex | 0.414 | 0.206 | — | 0.309 | — | — | 0.310 |
| | Gemini | 0.478 | 0.250 | 0.325 | 0.336 | 0.357 | 0.256 | 0.334 |
| | Safe | 0.029 | 0.007 | — | 0.009 | — | — | 0.015 |
| | Diaphora | 0.023 | 0.019 | 0.020 | 0.019 | 0.020 | 0.021 | 0.020 |
| Recall@Top-1 | **ASTERIA-PRO** | **0.917** | **0.868** | **0.912** | **0.879** | **0.899** | **0.903** | **0.896** |
| | Asteria | 0.706 | 0.648 | 0.652 | 0.627 | 0.631 | 0.675 | 0.657 |
| | Trex | 0.274 | 0.110 | — | 0.192 | — | — | 0.192 |
| | Gemini | 0.405 | 0.180 | 0.242 | 0.261 | 0.279 | 0.229 | 0.266 |
| | Safe | 0.004 | 0.002 | — | 0.002 | — | — | 0.003 |
| | Diaphora | 0.021 | 0.016 | 0.017 | 0.016 | 0.017 | 0.018 | 0.018 |
| Recall@Top-10 | **ASTERIA-PRO** | **0.961** | **0.921** | **0.962** | **0.913** | **0.952** | **0.932** | **0.940** |
| | Asteria | 0.902 | 0.867 | 0.882 | 0.857 | 0.866 | 0.890 | 0.877 |
| | Trex | 0.710 | 0.452 | — | 0.575 | — | — | 0.579 |
| | Gemini | 0.615 | 0.383 | 0.482 | 0.478 | 0.502 | 0.468 | 0.488 |
| | Safe | 0.022 | 0.010 | — | 0.014 | — | — | 0.015 |
| | Diaphora | 0.029 | 0.024 | 0.026 | 0.026 | 0.025 | 0.027 | 0.026 |

| Methods | X86-ARM | X86-X64 | X86-PPC | ARM-X64 | ARM-PPC | X64-PPC | Average |
|---|---|---|---|---|---|---|---|
| **ASTERIA-PRO** | **0.996** | **0.998** | **0.995** | **0.998** | **0.998** | **0.999** | **0.997** |
| Asteria | 0.995 | 0.998 | 0.998 | 0.995 | 0.998 | 0.999 | 0.997 |
| Gemini | 0.969 | 0.984 | 0.984 | 0.973 | 0.968 | 0.984 | 0.977 |
| SAFE | 0.851 | 0.867 | — | 0.872 | — | — | 0.863 |
| Trex | 0.794 | 0.891 | — | 0.861 | — | — | 0.849 |
| Diaphora | 0.389 | 0.461 | 0.397 | 0.388 | 0.455 | 0.400 | 0.415 |

- **Task-C**
  - **Asteria-Pro实现最优性能**
  - **Asteria-Pro与Asteria性能相当**
- **Task-V**
  - **Asteria-Pro实现最优性能**

**同一方法在不同评估任务中的表现可能会有显著差异！**

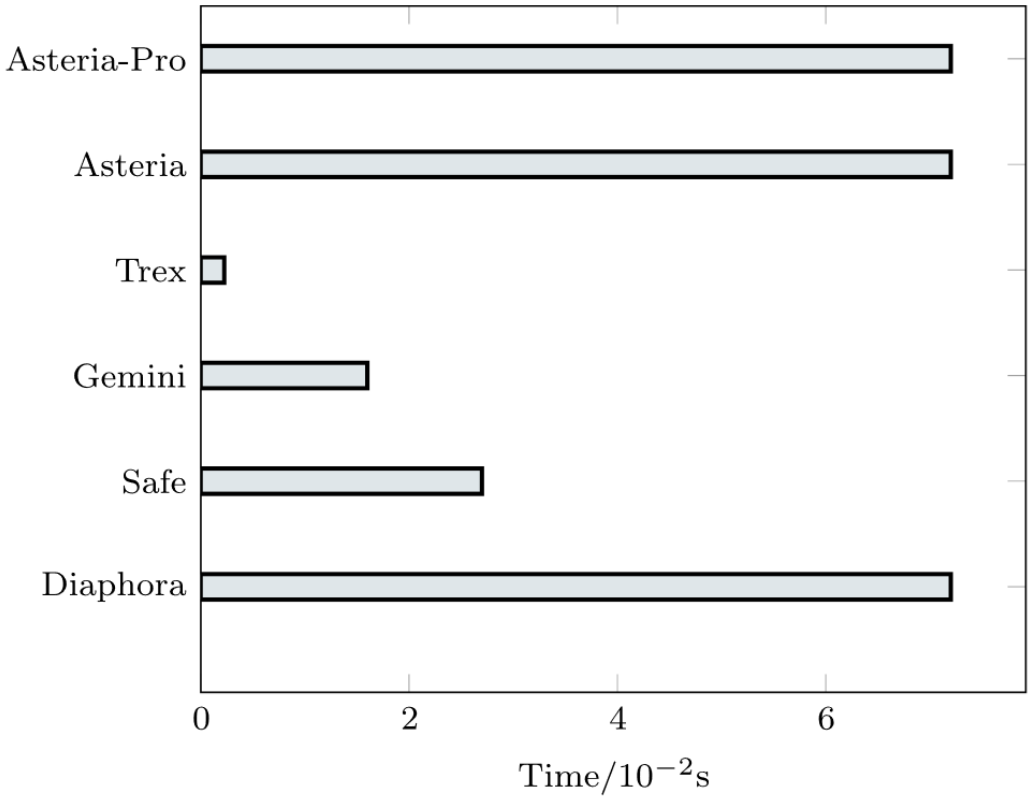
# 实验设计 对比实验

- **RQ1：Asteria-Pro在跨架构和跨编译器检测方面的性能表现**
  - **跨编译器检测**
    - **编译器版本：**
      - **gcc v5.4.0**
      - **icc 2021.1**
      - **clang 10.0.0**
    - **Asteria-Pro实现最优性能**
    - **所有方法均在gcc-clang组合下实现了最佳性能（除Safe）**
    - **icc编译器采用了更激进的优化策略，生成了高度优化的代码**

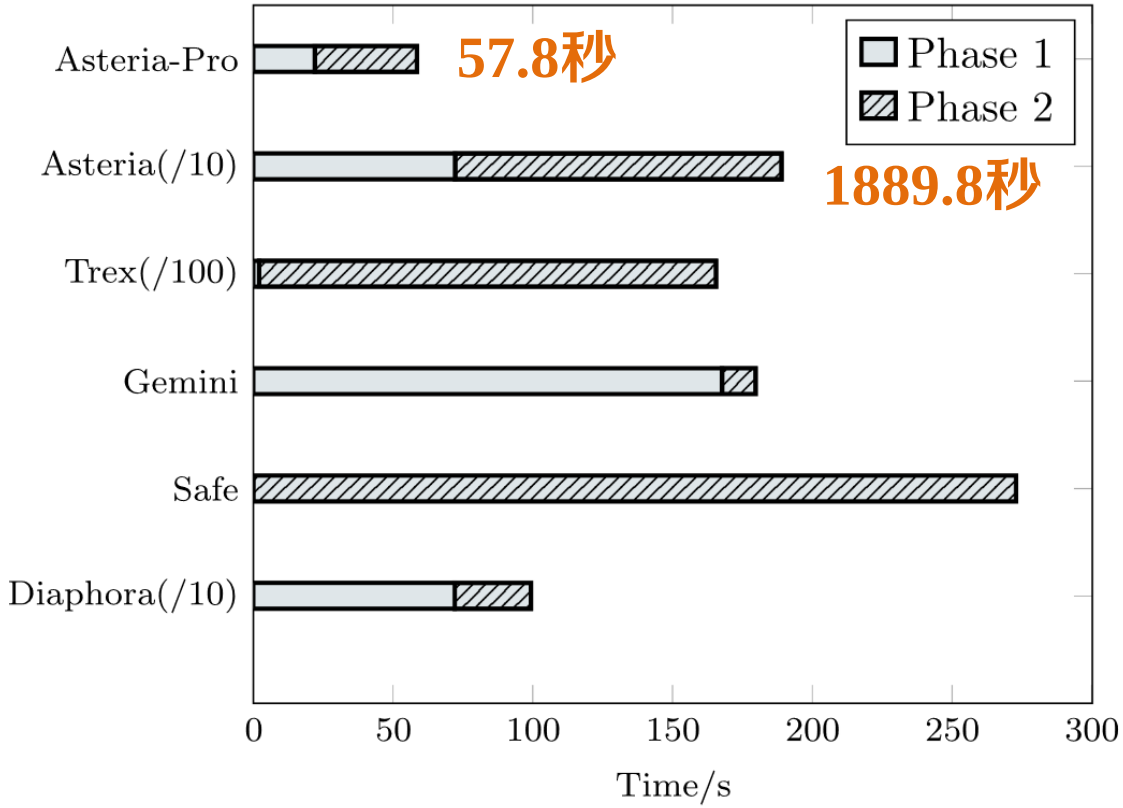
| Metrics | Methods | gcc-clang | gcc-icc | clang-icc | Avg. |
|---|---|---|---|---|---|
| MRR | **Asteria-Pro** | **0.755** | **0.560** | **0.564** | **0.626** |
| | Asteria | 0.624 | 0.319 | 0.328 | 0.424 |
| | Trex | 0.148 | 0.063 | 0.093 | 0.101 |
| | Gemini | 0.234 | 0.121 | 0.080 | 0.145 |
| | Safe | 0.058 | 0.187 | 0.076 | 0.107 |
| | Diaphora | 0.727 | 0.370 | 0.384 | 0.494 |
| Recall@Top-1 | **Asteria-Pro** | **0.694** | **0.479** | **0.486** | **0.553** |
| | Asteria | 0.541 | 0.244 | 0.256 | 0.347 |
| | Trex | 0.099 | 0.031 | 0.040 | 0.057 |
| | Gemini | 0.164 | 0.079 | 0.048 | 0.097 |
| | Safe | 0.027 | 0.152 | 0.031 | 0.070 |
| | Diaphora | 0.662 | 0.312 | 0.330 | 0.435 |
| Recall@Top-10 | **Asteria-Pro** | **0.864** | **0.706** | **0.711** | **0.760** |
| | Asteria | 0.783 | 0.466 | 0.469 | 0.573 |
| | Trex | 0.257 | 0.124 | 0.075 | 0.152 |
| | Gemini | 0.368 | 0.196 | 0.137 | 0.234 |
| | Safe | 0.101 | 0.239 | 0.149 | 0.163 |
| | Diaphora | 0.844 | 0.476 | 0.497 | 0.606 |

# 实验设计 对比实验

- **RQ2：Asteria-Pro在Task-V上的时间开销**
  - **Asteria-Pro的搜索时间最短**
  - **与Asteria相比，Asteria-Pro通过引入预过滤模块将搜索时间缩短了96.90%**




每个函数的平均特征提取时间




完成搜索任务的总时间

# 实验设计 消融实验

- 验证预过滤模块和重排序模块对方法在Task-V上的性能改进

| Module Combination | MRR | Recall@Top-1 | Recall@Top-10 | Average Time (s) |
|---|---|---|---|---|
| Pre-filtering + Asteria | 0.824 | 0.764 | 0.929 | 57.8 |
| Asteria + Reranking | 0.882 | 0.864 | 0.910 | 1889.8 |

- 验证预过滤模块和重排序模块是否可以与其他基线方法集成
  - 所有基线方法在Task-V上的性能均有较大的改进！

| Methods | Trex | **Trex-I** | Gemini | **Gemini-I** | Safe | **Safe-I** | Diaphora | **Diaphora-I** | Asteria | **ASTERIA-PRO** |
|---|---|---|---|---|---|---|---|---|---|---|
| MRR | 0.310 | **0.547** | 0.334 | **0.775** | 0.015 | **0.533** | 0.020 | **0.772** | 0.734 | **0.908** |
| Recall@Top-1 | 0.192 | **0.377** | 0.266 | **0.722** | 0.003 | **0.484** | 0.018 | **0.711** | 0.657 | **0.896** |
| Recall@Top-10 | 0.579 | **0.881** | 0.488 | **0.865** | 0.015 | **0.603** | 0.027 | **0.878** | 0.877 | **0.940** |

# 实验设计 超参数实验

- 预过滤中不同的过滤阈值$T_{NCL}$
  - 随着$T_{NCL}$增大，被过滤掉的函数数量增加，召回率下降
  - $T_{NCL}=0.1$，召回率最高，且过滤掉96.67%的非同源函数
- 重排序中的权重$\alpha$和$\beta$
  - 随着$\alpha$增大，所有指标持续下降
  - $\alpha=0.1$和$\beta=0.9$时，Asteria-Pro实现最佳性能
- 超参数实验并不充分！

| $T_{NCL}$ | # Filtered Function | Recall |
|---|---|---|
| 0.1 | 9666.7 | 0.9813 |
| 0.2 | 9734.1 | 0.9808 |
| 0.3 | 9777.4 | 0.9791 |
| 0.4 | 9793.5 | 0.9773 |
| 0.5 | 9805.5 | 0.9737 |

| $\alpha$ | $\beta$ | MRR | Recall@Top-1 | Recall@Top-10 |
|---|---|---|---|---|
| 0.0 | 1.0 | 0.901 | 0.890 | 0.930 |
| **0.1** | **0.9** | **0.908** | **0.896** | **0.940** |
| 0.2 | 0.8 | 0.905 | 0.893 | 0.938 |
| 0.3 | 0.7 | 0.902 | 0.890 | 0.937 |
| 0.4 | 0.6 | 0.900 | 0.889 | 0.935 |
| 0.5 | 0.5 | 0.899 | 0.887 | 0.934 |
| 1.0 | 0.0 | 0.824 | 0.764 | 0.929 |

# 算法总结 Asteria-Pro

- 算法贡献
  - 首次提出一种结合领域知识的预过滤和重排序增强基于深度学习模型的二进制函数相似性检测方法
  - 基于领域知识的预过滤模块显著减少了时间开销
  - 基于领域知识的重排序模块显著提升了检测准确率
- 算法不足
  - 依赖于IDA Pro反汇编工具和Hexray Decompiler反编译插件对二进制代码执行反编译并提取AST，其生成反编译代码与AST的质量会影响方法性能
  - 未对模型在跨编译优化选项场景下的泛化能力进行评估

# 特点总结与未来展望

# 特点总结与未来展望

- 特点总结
  - DiEmph
    - 结合深度学习模型分析技术和二进制程序分析技术，首次提出了一种识别指令分类重要性和语义重要性的方法，可有效提升基于Transformer模型的泛化性
    - 只适用于单架构二进制函数相似性检测
  - Asteria-Pro
    - 结合领域知识，首次提出一种使用预过滤和重排序增强基于深度学习模型的二进制函数相似性检测方法，显著降低了算法时间开销，提升了检测准确率
    - 适用于跨架构二进制函数相似性检测
- 未来发展
  - 研究如何提取函数在跨编译环境和跨架构下的共性信息表征
  - 研究如何充分利用函数内部丰富的各类信息（控制流信息、数据依赖信息、语法信息和语义信息等）
  - 结合更多的二进制程序分析技术与领域相关知识针对性地提升方法的性能

# 参考文献


[1] Xu X, Feng S, Ye Y, et al. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. Proceedings of the 32st ACM SIGSOFT International Symposium on Software Testing and Analysis[C]. New York, NY, United States: ACM, 2023: 164-175.

[2] Yang S, Dong C, 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(TOSEM), 2023, 31(3): 1-27.

[3] Wang H, Qu W, Katz G, et al. Jtrans: Jump-aware transformer for binary code similarity detection. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis[C]. New York, NY: ACM, 2022: 1-13.

[4] Xu X, Liu C, Feng Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection. Proceedings of the 2017 ACM SIGSAC conference on computer and communications security[C]. New York, NY: ACM, 2017: 363-376.

# 道德经

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！