Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization论文翻译& 笔记

Abstract

逆向工程是一项需要大量人工分析但必不可少的技术，用于了解新的恶意软件的内部工作原理，发现现有系统中的漏洞以及检测已发布软件中的专利侵权技术。汇编克隆搜索引擎通过识别那些重复或已知的块来促进逆向工程师的工作。 但是，要想设计一个健壮的克隆搜索引擎是一项挑战，因为存在各种编译器优化选项和代码混淆技术，它们会使逻辑上相似的汇编函数看起来非常不同。

实用的克隆搜索引擎依赖于汇编代码的强大矢量表示形式。 但是，现有的克隆搜索方法依赖于手动的特征工程过程来为汇编函数形成特征向量，却**无法考虑特征之间的关系**并**无法识别那些可以在统计学上区分汇编函数的独特模式**。 为了解决这个问题，我们建议基于汇编代码共同学习汇编程序的词汇语义关系和向量表示。 我们已经开发了汇编代码表示学习模型**Asm2Vec**。 它只需要汇编代码作为输入，不需要任何先验知识，例如汇编函数之间的正确映射。 **它可以在汇编代码中出现的标记之间找到并合并丰富的语义关系**。 我们进行了广泛的实验，并使用最新的静态和动态克隆搜索方法对学习模型进行了基准测试。 我们表明，学习的表示形式更鲁棒，并且相对于混淆和优化所引入的变化，其性能明显优于现有方法。

1. Introduction

软件开发大多不是从零开始。 由于在软件开发过程中普遍且通常会不受控制地重用各种源代码，因此在底层汇编代码中也存在大量克隆。一个有效的程序集克隆搜索引擎可以大大减少逆向工程中涉及的手动分析过程的负担。 通过利用现有的大量二进制数据，它可以满足逆向工程师的信息需求。

汇编代码克隆搜索作为一种信息检索（IR）技术正在兴起，该技术可帮助解决与安全性相关的问题。 它已用于不同的二进制文件以查找更改的部分，识别已知的库函数（例如加密库等等），在现有软件或物联网（IoT）设备、固件中搜索已知的编程错误或零日漏洞，并可以在源代码不可用时检测软件窃或GNU许可证侵权。 但是，由于编译器优化和混淆技术的多样性，使逻辑上相似的汇编函数看起来截然不同，因此设计有效的搜索引擎非常困难。 图1显示了一个示例。 优化或模糊的汇编函数破坏了控制流程和基本块的完整性。 将这些语义相似但结构和句法不同的汇编功能识别为克隆是具有挑战性的。

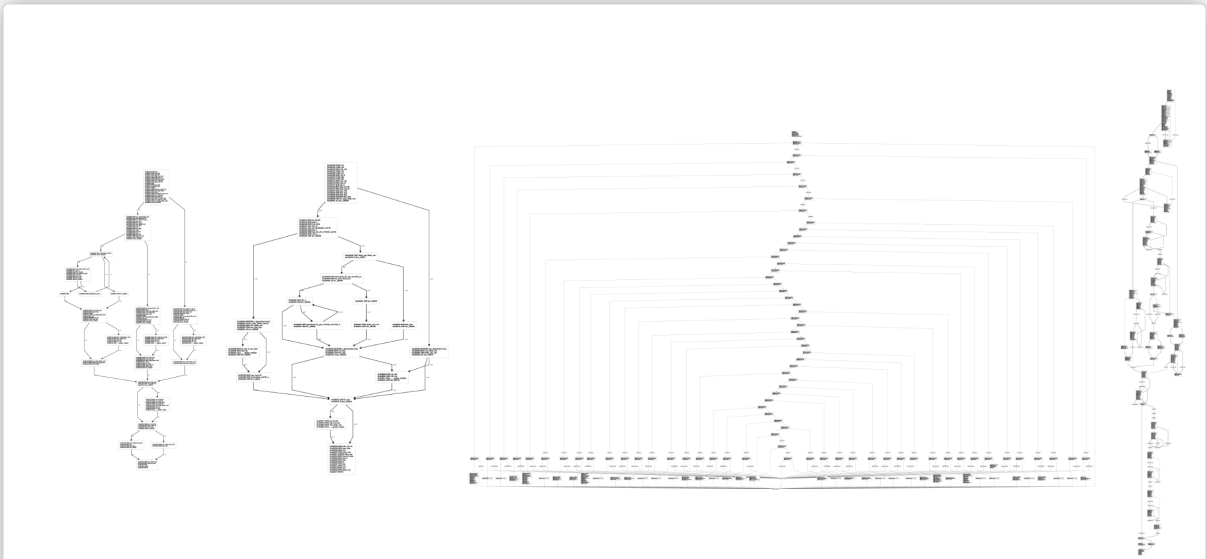


Figure 1: Different assembly functions compiled from the same source code of gmpz_tdiv_r_2exp in libgmp. From left to right, the assembly functions are compiled with gcc O0 option, gcc O3 option, LLVM obfuscator Control Flow Graph Flattening option, and LLVM obfuscator Bogus Control Flow Graph option. Asm2Vec can statically identify them as clones.

图一：从libgmp中的gmp_tdiv_r_2exp的相同源代码编译的不同汇编函数。 从左到右，使用gcc的 O0选项，gcc的O3选项，LLVM混淆器控制流图展平选项和LLVM混淆器伪造控制流图选项来编译汇编函数。 但是Asm2Vec依然可以将其静态标识为克隆代码。

开发一个克隆搜索解决方案需要一个强大的汇编代码矢量表示，通过它可以测量查询和索引函数之间的相似性。 基于手动设计的功能，可以将相关研究分为静态或动态方法。 动态方法**通过动态分析汇编代码的I/O行为来建模语义相似性**。静态方法通过针对**汇编代码之间的语法**或**描述性统计数据**两个方面寻找静态差异来对汇编代码之间的相似性进行建模。 静态方法比动态方法更具可伸缩性，并且提供了更好的覆盖范围。 动态方法对语法更改更健壮，但可伸缩性较差。 我们确定了两个可以缓解的问题，可以提高静态特征的语义丰富性和鲁棒性。 我们表明，考虑到这两个因素，**静态方法甚至可以比最新的动态方法获得更好的性能**。

P1：现有的最新静态方法无法考虑特征之间的关系。 LSH-S，n-gram，n-perm，BinClone和Kam1n0模型汇编代码片段作为**操作**和**分类操作数**的频率值。 Tracelet将汇编代码建模为指令序列之间的编辑距离。 Discovre和Genius构建了描述性功能，例如算术汇编指令的比例，传输指令的数量，基本块的数量等。 所有这些方法都假定每个函数或类别都是一个独立的维度。 但是，xmm0 Streaming SIMD Extensions (SSE) 寄存器是与SSE操作(如movaps)相关的。 一个fclose(libc函数)和其他文件相关libc函数比如fopen是相关的。 strcpy函数可以替换为memcpy。**这些关系提供了更具有语义性质的单个标记或描述性的统计**。

为了解决这个问题，我**们建议将词汇语义关系纳入特征工程过程**。 如果从汇编语言的先验知识中手动指定所有潜在的关系这样会既耗时又不可行。 相反，我们可以**直接从简单的汇编代码中学习这些关系**。 Asm2Vec探索 token 之间的共现关系，并发现token之间的丰富词汇语义关系（参见图2）。 例如，memcpy，strcpy，memncpy和mempcpy在语义上似乎彼此相似， **SSE寄存器与SSE操作数有关**。 **Asm2Vec在训练过程中不需要任何先验知识**。

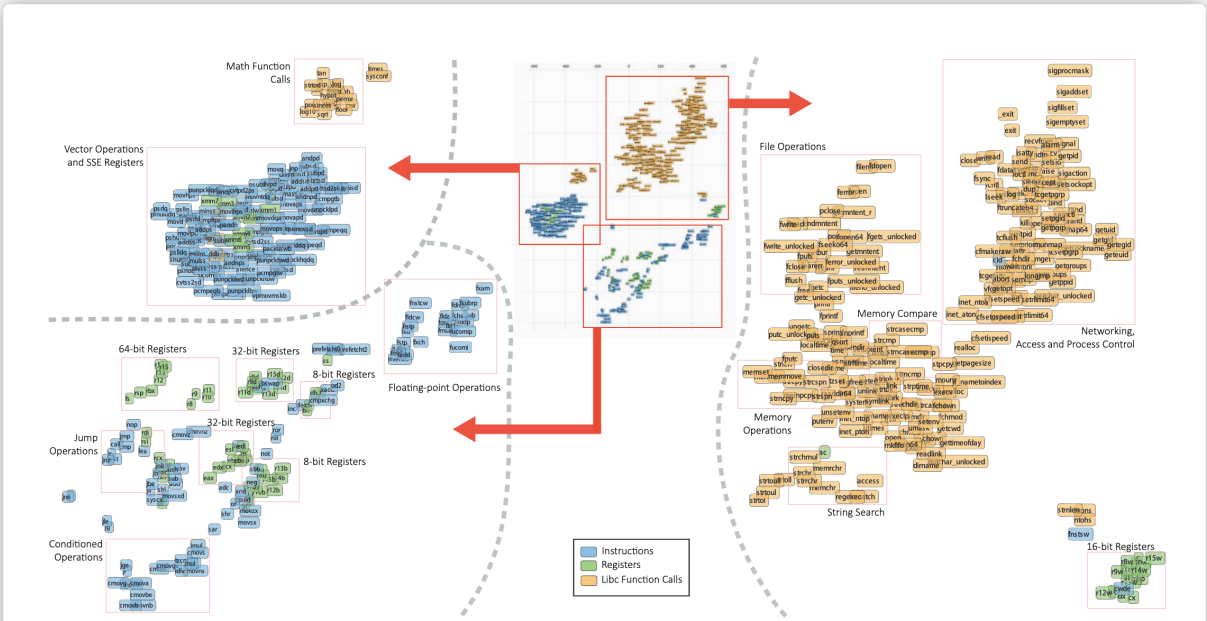


Figure 2: *T-SNE* clustering visualization of tokens appearing in assembly code. There are three categories of tokens: operation, operand, and *libc* function call. Each token is represented as a 200-dimensional numeric vector. They are learned by *Asm2Vec* on plain assembly code without any prior knowledge of the assembly language. The training assembly code does not contain the *libc* callee functions' content. For visualization, *T-SNE* reduces the vectors to two dimensions by nearest neighbor approximation. A smaller geometric distance indicates a higher lexical semantic similarity.

图二：汇编代码中出现的token的T-SNE集群可视化。 token分为三类：操作，操作数和libc函数调用。 每个标记都表示为200维数字矢量。 它们是由Asm2Vec在简单的汇编代码上学习的，而无需任何汇编语言的先验知识。 程序训练集代码不包含libc被调用者函数的内容。 为了可视化，T-SNE通过最近邻近似将向量缩减为二维。 较小的几何距离表示较高的词汇语义相似度。

P2：现有的静态方法假设特征同等重要，或者需要映射等效的汇编函数来学习权重。 权重的选择可能不能体现一个汇编函数和别的汇编函数区别模式的重要性和多元性。 经验丰富的逆向工程师不会通过均等地浏览整个内容或逻辑来识别已知功能，而是会根据过去的二进制分析经验来确定可识别特定功能的关键点和重要模式，不需要相似汇编代码映射。

为了解决这个问题，我们发现可以模拟经验丰富的逆向工程师的工作方式。 得到启发，我们提出训练一个神经网络模型，读取汇编代码，输出最佳向量表示（能够和其他函数有区分的为最佳）。这篇论文贡献如下：

- **我们提出了一种用于汇编克隆检测的新方法。** 这是使用表示学习为汇编代码构造特征向量的第一项工作，以减轻当前手工特征中的问题P1和P2。 以前有关汇编克隆搜索的所有研究都需要手动进行特征工程过程。 克隆搜索引擎是开源平台1的一部分。
- **我们为汇编代码语法和控制流图开发了表示学习模型，即Asm2Vec。** 该模型学习token之间的潜在词汇语义，并把汇编函数表示成搜集到的语义的加权混合。学习过程不需要关于汇编代码的先验知识，比如编译器优化参数或者汇编函数的正确映射。只需要汇编代码作为输入。
- **我们展示了Asm2Vec 比静态或动态方法在代码混淆和编译器优化更具有弹性。** 我们的实验涵盖了编译器的不同配置以及强大的混淆器，该混淆器可替代指令，拆分基本块，添加伪逻辑以及完全破坏原始控制流图。 我们还对公开的漏洞数据集进行了漏洞搜索案例研究，其中Asm2Vec的误报率为零，召回率为100％。 它优于动态的最新漏洞搜索方法。

Asm2Vec作为静态方法无法完全克服代码混淆。 但是，与最新的静态功能相比，它在代码混淆方面更具弹性。 本文的组织如下：第2节正式定义了搜索问题。 第三部分将表示学习系统地集成到克隆

搜索过程中。 第4节介绍了该模型。 第5节介绍了我们的实验。 第六部分讨论文献。 第7节讨论了局限性并总结了本文。

2. Problem Definition

在汇编克隆搜索文献中，有四种类型的克隆：1. 字面相同，2. 语法上等价，3. 稍作修改，和4. 在语义上相似。 我们关注第四种克隆，其中汇编函数在语法上可能看起来不同，但是在其源代码中共享相似的函数逻辑。 例如，有和没有混淆的相同源代码，或不同发行版之间的修补源代码。 我们使用以下概念：function denotes an assembly function; 函数代表汇编函数； 原函数代表源代码的函数，比如C++函数；repository function stands for the assembly function that is indexed inside the repository;库函数代表库中有索引的汇编函数； 给定一个汇编功能，我们的目标是从存储库RP中搜索其语义克隆。 我们正式定义搜索问题如下：

Definition 1. (Assembly function clone search) Given a target function ft, the search problem is to retrieve the top-k repository functions $f_s \in RP$, ranked by their semantic similarity, so they can be considered as Type IV clones.

定义1.（汇编函数克隆搜索）给定目标函数 f_s，搜索问题是检索按语义相似性排序的top-k存储库函数 $f_s \in RP$，因此可以将它们视为 第四种克隆（语义上相似）。

3. Overall Workflow

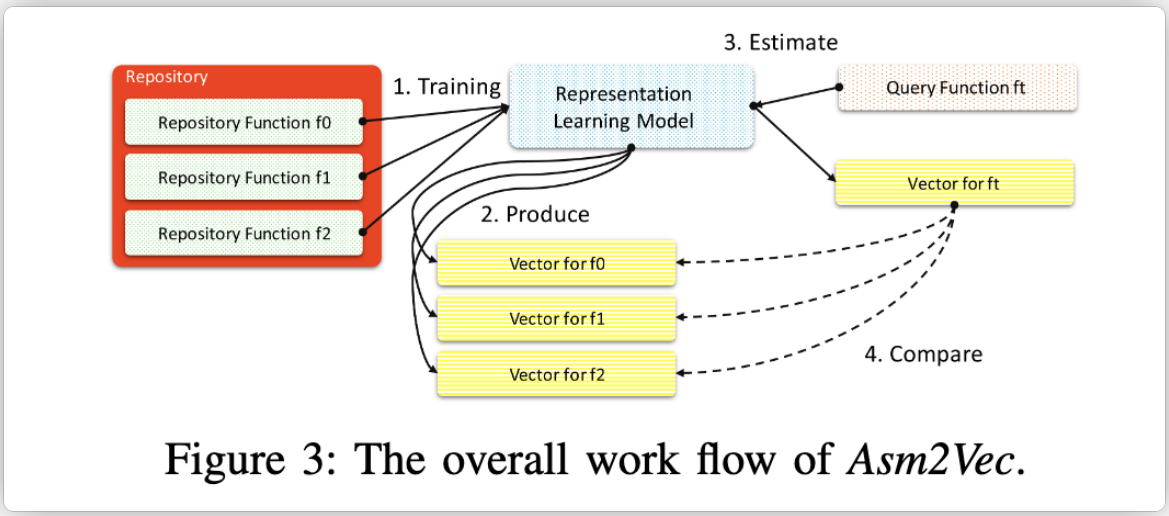


Figure 3: The overall work flow of *Asm2Vec*.

图3显示了整个工作流程。 共有四个步骤：步骤1：给定汇编函数的存储库，我们首先为这些函数构建神经网络模型。 我们只需要它们的汇编代码作为训练数据，而无需任何先验知识。步骤2：在训练阶段之后，模型为每个存储库的函数生成向量表示形式;步骤3：鉴于未使用此模型训练的目标函数函数，我们使用模型来估计其向量表示形式;步骤4：我们比较向量通过使用余弦相似度来检索排名最高的候选者作为结果，从而与存储库中的其他向量进行对抗。训练过程是一次性的工作，可以有效地学习查询的表示形式。 如果将新的程序集功能添加到存储库，则我们将按照步骤3中的相同步骤来估计其向量表示形式。 可以定期重新训练模型以确保向量的质量。

4. Assembly Code Representation Learning

在本节中，我们提出了汇编代码的表示学习模型。 具体来说，我们的设计基于PV-DM模型。 PV-DM模型基于文档中的token来学习如何转换为文档表示向量。 然而，文档是按顺序排列的，这与汇编代码不同，因为汇编代码可以表示为图形并具有特定的语法。 首先，我们描述原始的PV-DM神经网络，该网络学习文本段落的矢量化表示。 然后，我们公式化我们的Asm2Vec模型，并描述如何在给定函数的指令序列上对其进行训练。 之后，我们详细说明如何将控制流图建模为多个序列。

4.1. Preliminaries

PV-DM模型设计用于文本数据。 它是原始word2vec模型的扩展。 它可以共同学习每个单词和每个段落的矢量表示。 图4展示了PV-DM模型。

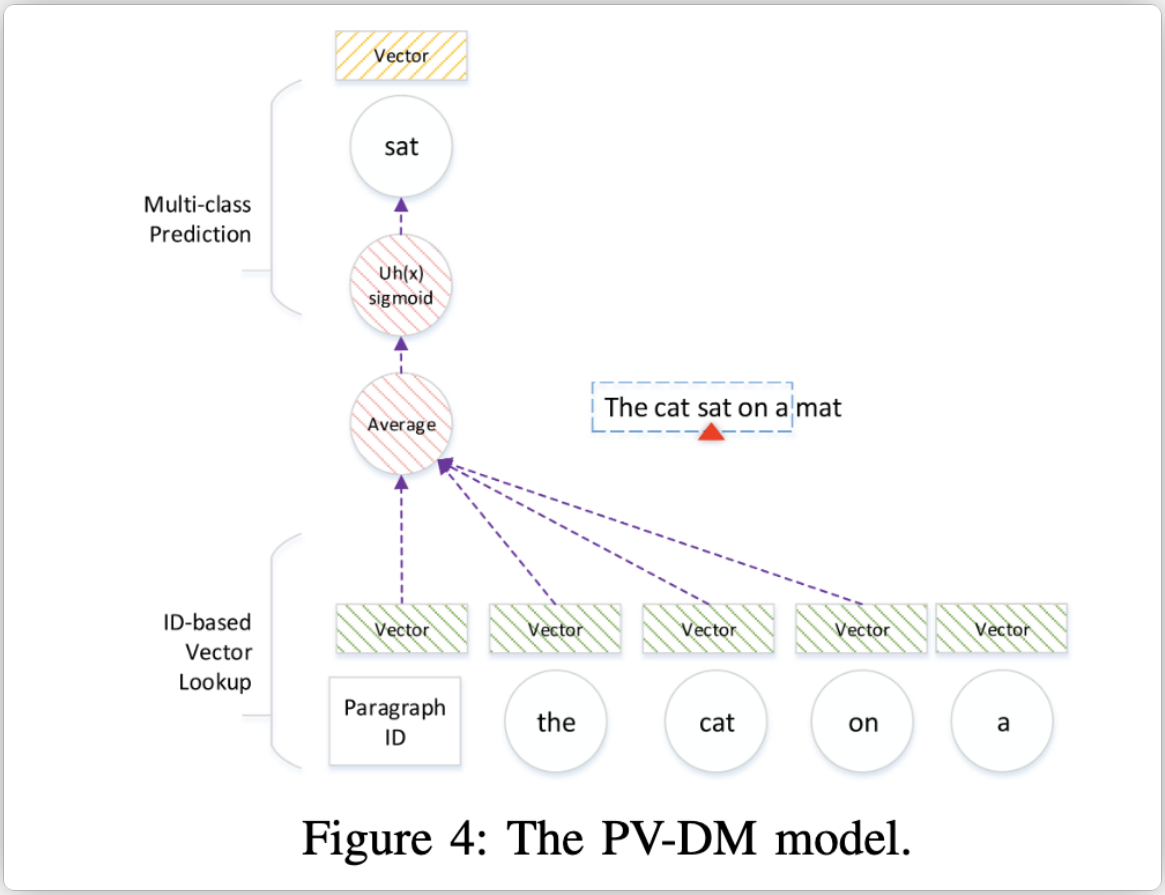


Figure 4: The PV-DM model.

给定包括多语句的文本段落，PV-DM利用每个语句滑动窗口。滑动窗口始于语句的开始，一步一个单词走到结尾。例如，图4中的滑动窗口大小为5。第一步滑动窗口包括5个词，‘the’, ‘cat’, ‘sat’, ‘on’ and ‘a’，中间的‘sat’被当成目标，周围的单词被当成上下文；第二步，滑动窗口往前走一步，包括‘cat’, ‘sat’, ‘on’, ‘a’ and ‘mat’，‘on’ 成为目标。

每一步，PV-DM模型完成一个多类预测任务。它把当前段落基于段落ID映射成一个向量，把每个上下文中的单词基于单词ID映射成一个向量。这些向量平均化这些向量，通过softmax分类预测词库中的目标词汇。back-propagated分类错误用来更新这些向量。 形式上，给定一个包含段落$p \in T$的文本语料库T，每个段落p包含一个句子$s \in p$列表，每个句子里面的单词$w_t \in s$是$|s|$的有序序列。 PV-DM使对数log概率最大化：

$$\sum_{p}^{T}\sum_{s}^{p}\sum_{t=k}^{|s|-k} log\ \mathbf{P}(w_t|p, w_{t-k}, ..., w_{t+k}) \qquad (1)$$

滑动窗口大小为2k + 1。段落向量捕获到了上下文丢失的信息来预测目标。这个被解释为话题（topics）。PV-DM是为了顺序布局的文本数据设计的。然而，汇编代码比文本数据承载了丰富的语法，它包括操作符、操作数，控制流，和文本数据有结构化的不同。这些不同之处需要一个不同的模型架构设计，解决PV-DM无法完成的任务。其次，我们展示了一个表示学习模型集成了汇编代码语法。

4.2 The Asm2Vec Model

汇编函数可以表示为控制流图（CFG）。 我们建议将控制流图建模分为多个序列，每个序列对应于一个潜在的执行轨迹，该轨迹包含线性排列的汇编指令。 给定一个二进制文件，我们使用IDA Pro反汇编程序来提取汇编函数，其基本块和控制流图的列表。

本部分对应于图3中的步骤1和2。在这些步骤中，我们训练表示模型，并为每个存储库函数$f_s \in RP$生成数值向量。 图5显示了该模型的神经网络结构，它不同于原始的PV-DM模型。

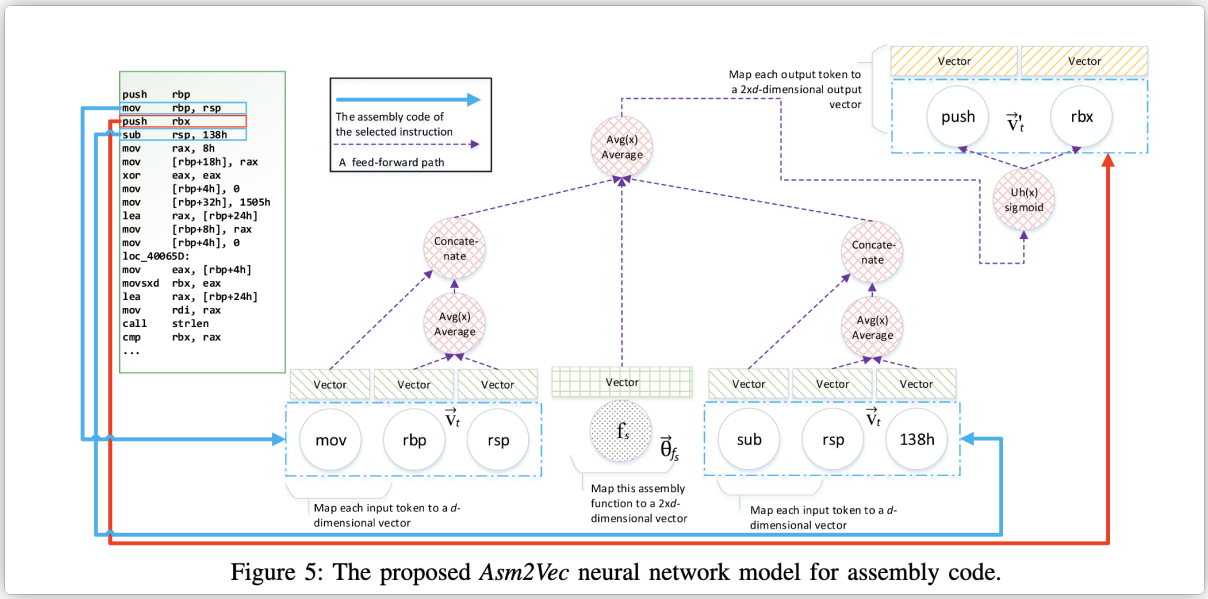


Figure 5: The proposed *Asm2Vec* neural network model for assembly code.

首先，我们将每个存储库函数f_s映射到一个矢量$\theta_{f_s}^{->} \in R^{2*d}$。$\theta_{f_s}^{->}$是函数$f_s$训练中将要学习到的向量表示，$d$是用户选择的参数。同样的，我们会在存储库RP中收集所有单独的token。我们将汇编代码标记中的操作数和操作符视为token。我们将每个token t映射到一个数值向量$v_t^{->} \in R^d$，以及另外一个数值向量$v_t'^{->} \in R^{2*d}$。$v_t^{->}$为token t的向量表示。经过训练后，它代表了token的词汇语义。 图2中使用$v_t^{->}$ vector可视化token之间的关系。$v_t'^{->}$使用token的预测。 将所有的$\theta_{f_s}^{->}$和$v_t^{->}$初始化为零附近的较小随机值。 将所有$v_t'^{->}$初始化为零。 我们使用$2 * d$来表示f_s，因为我们将操作向量和操作数连接起来以表示一条指令。

我们将每个存储库函数$f_s \in RP$视为多个序列$S(f_s) = seq[1:i]$，其中seq_i是其中之一。 我们假设序列的顺序是随机的。 序列表示为指令列表$L(seqi) = in[1:j]$，其中in_j是其中之一。 指令in_j包含操作数$A(in_j)$和一个操作符$P(in_j)$的列表。 它们拼接起来形成token列表$T(in_j) = P(in_j)||A(in_j)$，其中|| 表示串联。 常量token被标准化为十六进制形式。

对于函数f_s中的每个序列seq_i，神经网络从一开始就遍历所有指令。 我们收集当前指令in_j，其前序指令in_{j-1}和后序指令in_{j+1}， 忽略边界外的指令。 提出的模型试图在存储库RP中最大化以下对数概率：

$$\sum_{f_s}^{\mathrm{RP}} \sum_{seq_i}^{\mathcal{S}(f_s)} \sum_{in_j}^{\mathcal{I}(seq_i)} \sum_{t_c}^{\mathcal{T}(in_j)} log\ \mathbf{P}(t_c | f_s, in_{j-1}, in_{j+1}) \quad (2)$$

给定当前的汇编函数f_s和邻居指令，**它将在当前指令中看到tokent_c的对数概率最大化**。 直观想法是使用当前函数的向量和邻居指令列表提供的上下文来预测当前指令。 邻居指令列表提供的向量捕获词汇语义关系。 函数的向量会记住在上下文中无法预测的内容，把指令建模成互相可区分的模型（可区分当前函数和其他函数的指令进行建模）。

对于给定的函数f_s，我们首先通过先前构建的字典查找其向量表示$\theta_{f_s}^{->}$。 为了将邻居指令建模为$CT(in) \in R^{2\times d}$，我们对它的操作数$(\in R^d)$的向量表示取平均，并将平均向量$(\in R^d)$与操作的向量表示进行连接。 它可以表示为：

$$\mathcal{CT}(in) = \vec{v}_{\mathcal{P}(in)} || \frac{1}{|\mathcal{A}(in)|} \sum_{t}^{\mathcal{A}(in)} \vec{v}_{t_b} \quad (3)$$

回想一下，$P(*)$表示一个运算，并且它是单个token。 通过用$CT(in_{j-1})$和$CT(in_{j+1})$对f_s求平均值，$\delta(in, f_s)$对邻居指令的联合存储进行建模：

$$\delta(in_j, f_s) = \frac{1}{3}(\vec{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1})) \quad (4)$$

示例1.考虑一个简单的汇编代码函数f_s及其图5中的序列之一。以第三个指令当$j = 3$为例，$T(in_3) = \{'push', 'rbx'\}. A(in_{3-1}) = \{'rbp', 'rsp'\}. P(in_{3-1}) = \{'mov'\}$。我分别收集了他们各自的向量 $v_{rbp}^{->}$ 、 $v_{rsp}^{->}$、 $v_{mov}^{->}$ ， 然后计算了$CT(in_{3-1}) = v_{mov}^{->} || (v_{rbp}^{->} + v_{rsp}^{->})/2$。紧跟着相同的步骤，我们计算了$CT(in_{3+1})$。利用等式4和$\theta_{f_s}^{->}$ 我们有$\delta(in_3, f_s)$。

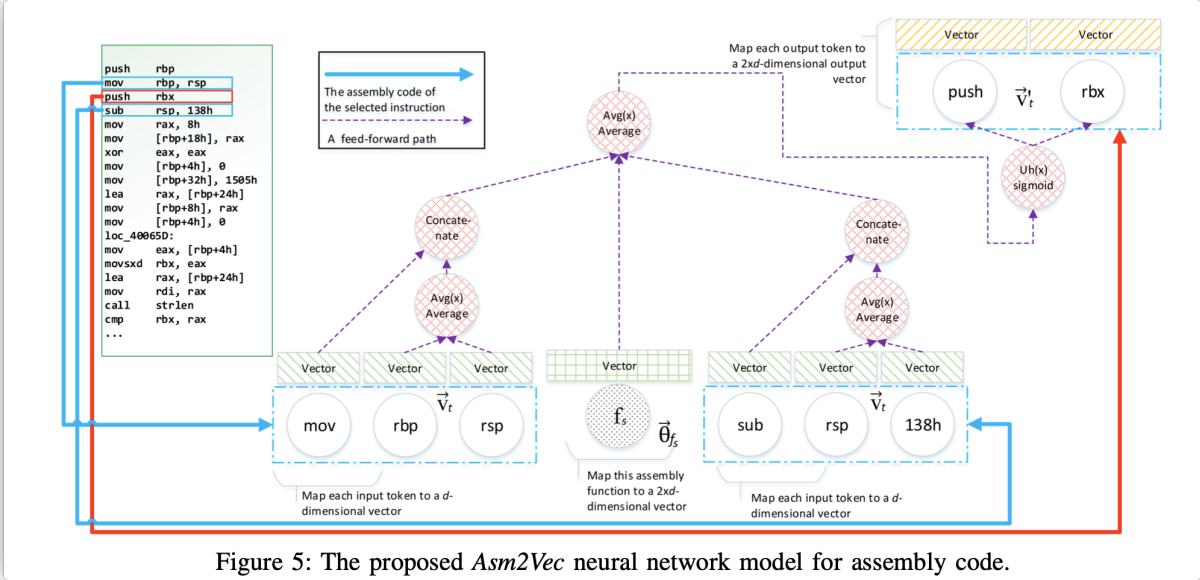


Figure 5: The proposed *Asm2Vec* neural network model for assembly code.

给定$\delta(in, f_s)$，在等式二的长概率可以写重写为如下：

$$\mathbf{P}(t_c|f_s, in_{j-1}, in_{j+1}) = \mathbf{P}(t_c|\delta(in_j, f_s)) \quad (5)$$

回想起我们把token映射为两个向量：$v^{->}$和$v'^{->}$。对于每一个目标token $t_c \in T(in_j)$是属于当前指令的。我们可以看到他的输出向量为：$v'^{->}_{t_c}$。公式5中的概率可以建模为softmax多类回归问题：

$$\begin{aligned}\mathbf{P}(t_c|\delta(in_j, f_s)) &= \mathbf{P}(\vec{v'}_{t_c}|\delta(in_j, f_s)) \\ &= \frac{f(\vec{v'}_{t_c}, \delta(in_j, f_s))}{\sum_d^D f(\vec{v'}_{t_d}, \delta(in_j, f_s))} \\ f(\vec{v'}_{t_c}, \delta(in_j, f_s)) &= Uh((\vec{v'}_{t_c})^T \times \delta(in_j, f_s))\end{aligned}$$

Sigmoid函数常被用作神经网络的激活函数，将变量映射到0,1之间

D表示根据存储库RP构建的整个词汇表。 $Uh(\cdot)$表示应用于向量的每个值的sigmoid型函数。 每次经过隐藏层，要估计的**参数总数为**$(|D| + 1) \times 2 \times d$。 $|D|$ 对于softmax分类太大。 根据[20]，[21]，我们使用k个负采样方法将对数概率近似为：

$$log\ \mathbf{P}(t_c|\delta(in_j, f_s)) \approx log\ f(\vec{v'}_{t_c}|\delta(in_j, f_s))$$
$$+\sum_{i=1}^{k} \mathbb{E}_{t_d \backsim P_n(t_c)} \left([\![t_d \neq t_c]\!] log\ f(-1 \times \vec{v'}_{t_d},\ \delta(in_j, f_s)) \right) \tag{6}$$

[[·]]是一个标识函数。 如 果此函数内部的表达式的计算结果为true，则输出1；否则，结果为1。 否则为0。例如，[[1+ 2 =3]]= 1和[[1+ 1 =3]]= 0。 负采样算法使用k + 1 logistic回归将正确的猜测t_c与k个随机选择的负采样 $\{td|td\ /=\ tc\}$(td不等于tc)区分开。

$E_{t_d} -> P_n(t_c)$是一种采样函数，它根据由D构成的噪声分布$P_n(t_c)$，从词汇表D中抽取一个token t_d。通过分别对v'_t->和θ->f_s求导数，我们可以计算出梯度如下。

$$\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) = \frac{1}{3} \sum_{i}^{k} \mathbb{E}_{t_b \backsim P_n(t_c)} \left([\![t_b = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \right) \times \vec{v'}_t$$
$$\frac{\partial}{\partial \vec{v'}_t} J(\theta) = [\![t = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \times \delta(in_j, f_s) \tag{7}$$

通过分别取导于$v^{->}_{P(in_{j+1})}$和$\{v^{->}_{t_b} | t_b \in A(in_{j+1})\}$的导数，我们可以如下计算它们的梯度。 通过将$in_{j+1}$替换为$in_{j-1}$，它将与先前的指令$in_{j-1}$有着相同的方程式。

$$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}} J(\theta) = \left(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \right) [0 : d-1]$$
$$\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta) = \frac{1}{|\mathcal{A}(in_{j+1})|} \times \left(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \right) [d : 2d-1] \tag{8}$$
$$t_b \in \mathcal{A}(in_{j+1})$$

之后，我们使用反向传播(back propagation)来更新所涉及向量的值。 具体来说，我们根据学习率(learning rate)更新$\theta^{->}_{f_s}$，根据梯度涉及到所有的v_t和v'_t。

示例2：从示例1继续，示例1中的目标是“push”。 接下来，我们使用负采样（公式A）计算$P(v'^{->}_{push}|\delta(in_j\ ,\ f_s))$。 之后，我们使用公式7和8计算梯度。我们根据两个实例各自的梯度、学习速

率更新这两个示例中所有涉及的向量。

4.3. Modeling Assembly Functions

在本节中，我们将汇编函数建模为多个序列。 形式上，我们将每个存储库函数$f_s \in RP$视为多个序列$S(f_s) = seq[1:i]$。 控制流图的原始线性布局覆盖了一些无效的执行路径。 我们不能直接将其用作训练序列。 相反，我们将控制流图建模为边缘覆盖序列和Random walks。

4.3.1. Selective Callee Expansion.

函数内联是一种编译器优化技术，可将函数调用指令替换为被调用函数的主体。 它扩展了原始的汇编功能，并通过消除了调用开销来提高了其性能。 它显着修改了控制流图，并且是程序集克隆搜索中的主要挑战。

BinGo[12]提出了动态分析过程中有选择性挑选内联函数到被调函数。我们在静态分析中采用了这个技术。函数调用指令被有选择性的扩展成被调用函数。BinGo为了语义正确内联所有标准库。我们不内联所有标准库，因为库调用token词汇语义已经被模型很好的描绘了。BinGo递归内联被调用函数，我们只扩展第一层调用函数。递归扩展调用将会导致调用函数体包括太多被调用函数，使得调用函数静态更像被调用函数。

BinGo使用的分离指标关注了每个被调用函数f_c的入度与出度之比：

$$\alpha(f_c) = \text{outdegree}(f_c)/(\text{outdegree}(f_c)+\text{indegree}(f_c)) \quad (9)$$

我们采用相同的方程式和相同的阈值0.01，来挑选被调用函数去扩展。 另外，我们发现如果被调用函数比调用函数长度更长或相当，被调用函数将会占有调用函数的一大部分。扩展后的函数和被调用函数相似。这样我们增加了额外的指标来筛选冗长的被调用函数：

$$\delta(f_s, f_c) = \text{length}(f_c)/\text{length}(f_s) \quad (10)$$

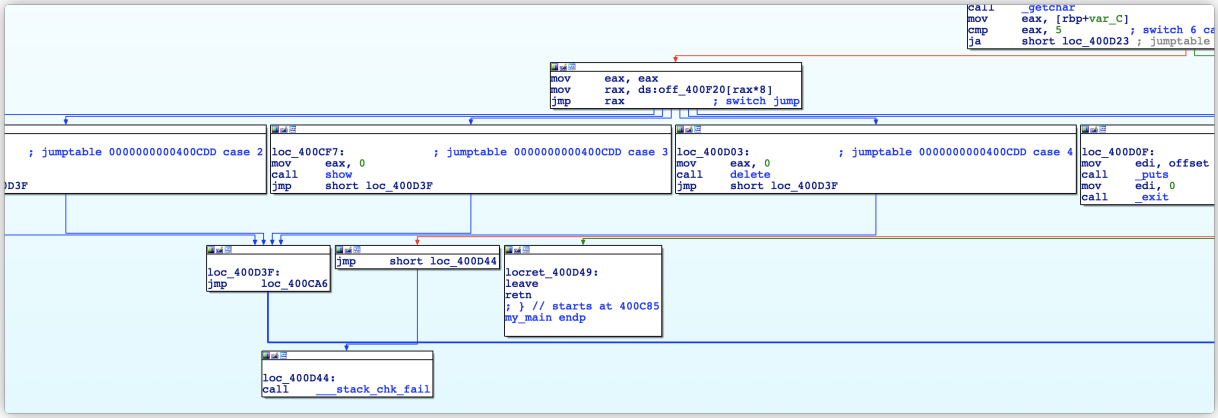
如果δ小于0.6或f_s短于10行指令，我们将扩展被调用方。 第二个条件是容纳wrpper功能。

> 提出只扩展被调用函数的第一层，防止被调用函数比调用函数的特征更多，导致调用函数的特征被淡化；同时还要比较被调用函数的长度与调用函数的长度（以一个比值来进行判断）。

4.3.2. Edge Coverage

为了为一个汇编函数生成多个序列，我们从被调用者展开的控制流图中随机采样所有边缘，直到覆盖原始图中的所有边缘。 对于每个采样边，我们将它们的汇编代码连接起来以形成一个新序列。 这样，我们确保控制流图被完全覆盖。 即使控制流图中的基本块被分割或合并，该模型仍然可以产生相似的序列。

随机采样的意思应该是指如果存在if语句、switch语句等等jmp语句进行随机选择跳转来进行采样



4.3.3. Random Walk.

CACompare[13]使用随机输入序列来分析汇编函数的I/O行为。一个随机输入模拟在真实执行流上的随机游走。我们通过增加扩展后的控制流图上的多随机游走来扩展汇编函数的汇编序列。这样生成的序列比边取样生成的序列要长的多。

Dominator是控制流分析和编译器优化中常用概念。一个基本块A 控制另外一个基本块B，如果B必须通过A才能到达其他块。多Random walk将给控制的基本块更高的概率。这些popular基本块可以作为循环结构的指标或者覆盖重要分支条件。使用Random可以被考虑成自然方法去使得负责控制的基本块优先级更高。

4.4. Tranining, Estimating and Searching

训练过程对应算法1。对于任意库中函数，通过边取样和随机游走生成序列。对于每个序列，通过了每个指令，并应用Asm2Vec来更新向量（10-19行）。正如算法1所示，训练过程不需要相同汇编代码间的ground-truth映射。

Algorithm 1 Training the Asm2Vec model for one epoch

function TRAIN(Repository RP)
    shuffle(RP)
    for each $f_s \in \text{RP}$ **do**
        for each $seq_i \in \mathcal{S}(f_s)$ **do**
            for $j = 1 \rightarrow (|seq_i| - 1)$ **do**
            ▷ Going through each instruction.
                lookup f_s's representation $\vec{\theta}_{f_s}$
                calculate $\mathcal{CT}(in_{j-1})$ by Equ. 3
                calculate $\mathcal{CT}(in_{j+1})$ by Equ. 3
                calculate $\delta(in_j, f_s)$ by Equ. 4
                for each $tkn \in in_j$ **do**
                ▷ Going through each token
                    $targets \leftarrow \mathbb{E}_{t_b \backsim P_n(tkn)} \cup \{tkn\}$
                    ▷ Sample tokens from $P_n(tkn)$
                    calculate and cumulate gradient for $\vec{\theta}_{f_s}$ (Equ. 7)
                    calculate gradient for $\vec{v'}_t$ (Equ. 7)
                    update $\vec{v'}_t$
                    calculate and cumulate gradient for in_{j-1} (Equ. 8)
                    calculate and cumulate gradient for in_{j+1} (Equ. 8)
                update vectors for tokens of in_{j-1}
                update vectors for tokens of in_{j+1}
                update $\vec{\theta}_{f_s}$

function $\mathcal{S}$(Function f_s)
    $graph \leftarrow \text{CFG}(f_s)$
    $graph \leftarrow \text{ExpandSellectiveCallee}(graph)$
    $sequences \leftarrow \{\}$
    for each $edg \in \text{SampleEdge}(graph)$ **do**
        $seq \leftarrow \text{source}(edg) \,||\, \text{target}(edg)$
        ▷ Concatenate the source and the target blocks
        $sequences \leftarrow sequences \cup \{seq\}$
    for $i \leftarrow$ numRandomWalk **do**
        $seq \leftarrow \text{RandomWalk}(graph)$
        $sequences \leftarrow sequences \cup \{seq\}$
    return $sequences$

估计步骤对应于图3中的步骤3.对于不属于训练汇编函数集f_t的查询$f_t / \in RP$(f_t不属于RP)的未知汇编函数，我们首先将其与向量$\theta_{f_t}^{->} \in R^{2\times d}$关联，并将其初始化为 小值接近零。 然后，我们在训练过程中遵循相同的过程，其中神经网络遍历每个序列，并从该序列的每个指令中进行训练。 在每个预测步骤中，我们固定训练模型的所有$v_t^{->}$和$v_t'^{->}$，仅将误差传播到$\theta_{f_t}^{->}$。 最后，我们使所有$f_s \in RP$和$\{v_t^{->}, v'^{->}_t | t \in D\}$的向量保持相同。 为了搜索匹配，将向量展平并使用余弦相似度进行比较。

对于不属于训练集合RP的未知汇编函数，我们会将他与一个函数关联，初始值接近零，在训练过程中会将误差传播到该函数f。

可伸缩性对于二进制克隆搜索至关重要，因为存储库中可能有数百万个汇编函数。 在大规模汇编代码上训练Asm2Vec是实用的。 一种类似的文本模型已被证明可扩展到数十亿个文本样本进行训练[21]。 在这项研究中，我们仅使用成对相似性进行最近邻居搜索。 低维固定长度向量之间的成对搜索可能很快。 在5.3节的实验中，有139,936个函数。 每个函数的平均训练时间为49毫秒。 平均查询响应时间少于300毫秒。

5. Experiments

我们将Asm2Vec与现有的最新动态和静态程序集克隆搜索方法进行了比较。 **所有实验均使用具有32G内存的Intel Xeon 6核心3.60GHz CPU进行。** 为了在相关研究中模拟类似的环境，我们将JVM限制为仅8个线程。 有四个实验。 首先，**我们使用GCC针对不同的编译器优化对基准进行基准测试。** 其次，**我们使用CLANG和OLLVM针对不同的繁重代码混淆评估了克隆搜索质量。** 第三，我们使用前两个二进制文件（两种情况一起训练测试）。 在最后一个中，我们将Asm2Vec应用于公开的漏洞搜索数据集。 克隆搜索之前，将剥离所有二进制文件。 在所有实验中，对于Asm2Vec，我们选择d = 200、25个负样本（k=25），10个随机游动和衰减学习率0.025。 200对应于[20]中使用的建议维度（2d）。

5.1. Searching with Different Compiler Optimization Levels

不同编译优化级别下进行搜索

在此实验中，我们使用**GCC编译器版本5.4.0**针对不同的优化级别对克隆搜索性能进行了基准测试。我们基于表1中的10个广泛使用的实用程序和数值计算库来评估Asm2Vec。它们是根据FOSS库的普遍使用率的内部统计数据选择的。 我们首先使用具有**四个不同编译器优化设置的GCC编译器编译选定的库，这将产生四个不同的二进制文件。 然后，我们测试其中两个的每种组合，它们对应于两个不同的优化级别。** 给定来自同一库但具有不同优化级别的两个二进制文件，我们使用编译器输出的调试符号链接它们的汇编函数，并在函数之间生成克隆映射。 此映射仅用作评估的真实数据。 我们在RP中针对第二个搜索第一个，然后在RP中针对第一个搜索第二个，我们取两者的平均值。仅存储库中的二进制文件用于训练。

较高的优化级别包含较低级别的所有优化策略。 O2和O3之间的比较是最简单的比较（图6）。

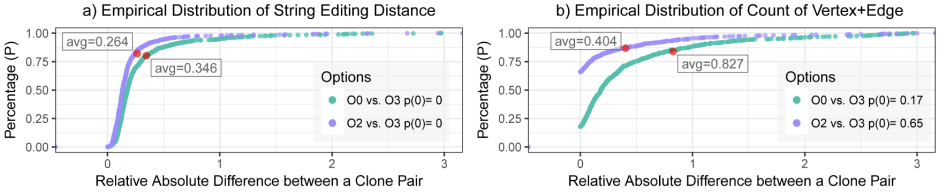


Figure 6: The difference between the O0/O2 optimized and the O3 optimized function. a) Relative string editing distance. 0.264 indicates that around 26.4% percent of bytes are different between two options for the same source code function. b) Relative absolute difference in the count of vertices and edges. 0.404% indicates that one function has 40.4% more vertices and edges than the other.

数据分析图

横坐标：克隆对之间的相对绝对差（也就是表明了这个控制流图修改了多少？编译优化等级不同，对于不同的函数修改的也不同）
纵坐标：克隆对之间的差异比率
比如a图表示的就是克隆对之间的字符串编码距离；而b图则表示克隆对之间的顶点和边的数量的相对绝对差
控制流图修改的越多，相似性越小
从图上点的分布来说，克隆对的相对绝对差大部分处于0-1之间，只有少部分属于1-3，所以他们的平均值才为小于1的数。
图6：优化的O0/O2和优化的O3功能之间的差异。 a）相对字符串编辑距离。 0.264表示同一源代码功能的两个选项之间的字节差异约为26.4%。 b）顶点和边的数量的相对绝对差。
0.404%表示一个函数的顶点和边缘比另一个函数多40.4%。
顶点就是指基本块，边指的是不同的跳转情况为边。

平均来说，26%的函数的字节已修改，所有函数都不相同。 修改了40%的控制流图，而65%的函数对共享相似的图复杂度。可以认为这是两个二进制文件中使用的优化策略相似的最佳情况。 O0和O3之间的比较是最困难的比较。 可以认为这是最差的情况，优化策略之间存在较大差异（图6）。平均，对一个函数的34%字节进行了修改，而没有一个函数是相同的。对控制流程图的82%进行了修改，对函数的17% 对共享相似图的复杂性。 表1列出了这两种情况的结果。 由于案例数量众多，我们仅列出这两个案例的结果以证明最佳和最差的情况。 其他案例的结果位于这两者之间，并且排名相同。

Compiler optimization O2 and O3												
Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p
BinGo†	Ø	.490	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.490	○
Composite	.789	.643	.910	.787	.842	.646	.783	.777	.813	.8IHI 38	.783	○
Constant	.437	.338	.202	.711	.522	.440	.365	.368	.549	.571	.450	●
Graphlet	.309	.268	.355	.262	.321	.297	.212	.313	.406	.148	.289	●
Graphlet-C	.662	.581	.680	.678	.689	.559	.586	.687	.730	.795	.665	●
Graphlet-E	.278	.225	.362	.270	.271	.219	.199	.280	.399	.355	.286	●
MixedGram	.811	.663	.906	.792	.848	.652	.789	.804	.858	.865	.798	●
MixedGraph	.445	.413	.436	.427	.486	.379	.350	.458	.564	.533	.449	●
n-gram	.774	.644	.874	.739	.814	.593	.748	.760	.812	.781	.754	●
n-perm	.803	.654	.912	.788	.848	.646	.785	.799	.850	.855	.793	●
FuncSimSearch	.157	.169	.848	.514	.663	.698	.726	.533	.488	.363	.516	●
PV(DM/DBOW)	.895	.899	.959	.952	.927	.945	.919	.898	.873	.823	.909	●
Asm2Vec*	**.954**	**.929**	**.973**	**.971**	**.951**	**.991**	**.931**	**.926**	**.885**	**.891**	**.940**	○
Compiler optimization O0 and O3												
Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p
BinGo†	Ø	.317	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.317	○
CACompare†	.844	Ø	Ø	**.893**	.794	Ø	**.795**	Ø	Ø	.717	.808	○
Composite	.013	.031	.019	.017	.004	.005	.007	.004	.036	.127	.026	●
Constant	.239	.128	.101	.610	.369	.258	.270	.182	.360	.439	.296	●
Graphlet	.017	.008	.049	.010	.023	.011	.009	.014	.029	.016	.019	●
Graphlet-C	.018	.020	.012	.022	.027	.001	.034	.012	.065	.102	.031	●
Graphlet-E	.021	.011	.075	.019	.017	.003	.018	.019	.051	.058	.029	●
MixedGram	.016	.033	.019	.018	.011	.005	.007	.006	.036	.116	.028	●
MixedGraph	.034	.028	.062	.024	.039	.015	.023	.030	.064	.097	.042	●
n-gram	.011	.029	.012	.021	.011	.010	.005	.003	.036	.129	.027	●
n-perm	.017	.029	.021	.021	.011	.006	.007	.005	.036	.129	.028	●
FuncSimSearch	.008	.019	.323	.039	.036	.030	.220	.011	.054	.040	.078	●
PV(DM/DBOW)	.745	.677	.760	.802	.792	.821	.759	.758	.713	.615	.744	●
Asm2Vec*	**.856**	**.781**	**.763**	.837	**.850**	**.921**	.792	**.776**	**.722**	**.788**	**.809**	○

TABLE 1: Clone search between different compiler optimization options using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of assembly functions that are correctly matched at position 1. In this case, it equals *Recall at Position 1*. *Asm2Vec* is our proposed method. †denotes cited performance. ○ and ● respectively indicate $p > 0.05$ and $p \leq 0.01$ for Wilcoxon signed-rank test between *Asm2Vec* and each baseline.

表1：使用“position 1的精度（Precision @ 1）”度量标准在不同的编译器优化选项之间进行克隆搜索。 它捕获在位置1正确匹配的汇编函数的比率。在这种情况下，它等于在位置1的调用。Asm2Vec是我们提出的方法。†表示所引用的性能。空心点和实心点分别表示Asm2Vec与每个基线之间的Wilcoxon符号秩检验的p> 0.05和p≤0.01。（检验成对观测数据之差是否来自均值为0的总体（产生数据的总体是否具有相同的均值））

在Wilcoxon符号秩检验中，它把观测值和零假设的中心位置之差的绝对值的秩分别按照不同的符号相加作为其检验统计量。它适用于T检验中的成对比较，但并不要求成对数据之差di服从正态分布，

只要求对称分布即可。检验成对观测数据之差是否来自均值为0的总体（产生数据的总体是否具有相同的均值）。

不同二进制文件中进行的Wilcoxon符号秩检验，以查看性能差异是否在统计学上显着

Andriesse指出，**使用监督式机器学习可能会有无效的实验结果的风险。** 例如，**将coreutils二进制文件拆分为训练集和测试集可能会导致无效的良好结果，因为这些二进制文件共享非常相似的代码库。 这个问题不适用于我们的实验。** 首先，我们遵循无监督学习范式，其中真正的克隆映射仅用于评估。 其次，我们的训练数据与测试数据非常不同，如图6和图7所示。例如，coreutils库带有许多二进制文件，但我们将它们静态链接为一个二进制文件。 我们训练O0优化的二进制文件，并匹配O3优化的二进制文件，这两个二进制文件非常不同。

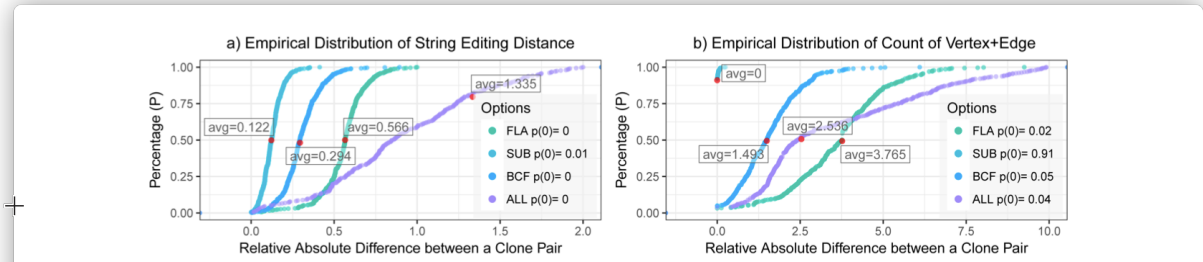


Figure 7: The difference between the original function and the obfuscated function. a) Relative string editing distance. 0.122 indicates that around 12.2% percent of bytes are modified. b) Relative absolute difference in the count of vertices and edge. 1.49% indicates the obfuscated function has 149% more vertices and edges in CFG.

第一个图横坐标是指令差别《体现在指令编辑距离》，第二个是图的距离和边《体现在控制流图》
图7：原始功能和模糊功能之间的区别。 a）相对字符串编辑距离。 0.122表示大约有12.2%的字节被修改。 b）顶点和边的数量的相对绝对差。 1.49%表示混淆函数在CFG中的顶点和边缘增加了149%。

我们使用“position 1（Precision@1）”上的精度来进行度量。 对于每个查询，如果baseline没有返回答案，我们会将改精度计算为零。 因此，Precision@1捕获正确匹配的汇编函数的比率，相当于该比率等于 position 1上的“recall”。

我们在[8]中提出了九种基准特征表示法：助记符n-ngram（表示为n-gram），助记符n-perms（表示为n-perm），Graphlet（表示为Graphlet），Extended Graphlet（表示为Graphlet-E），Colored Graphlet （表示为Graphlet-C），Mixed Graphlet（表示为MixGraph），Mixed n-gram/perms（表示为MixGram），Constants 常量以及n-gram/perms和Graphlets的复合（表示为Composite）。使用Graphlet的想法起源于[23]。 这些基准方法涵盖了从标记到图形子结构的广泛流行功能。 这些基线根据论文中报告的最佳设置进行配置。 我们还包括原始的PV-DM模型和PV-DBOW模型作为基准，其中每个汇编函数都被视为文档。 我们选择最佳结果并将其表示为PV-(DM/DBOW)。 我们仅调整zlib数据集上PV-(DM/DBOW)以及Asm2Vec的配置。 FuncSimSearch是Google最近发布的开源程序集克隆静态搜索工具包。 它具有默认的训练数据集，其中包含等效装配函数的真实映射。 最新的动态方法BinGo [12]和CACompare [13]无法进行评估。 但是，我们使用相同的指标以相同的方式进行实验。 他们报告的结果包括在表1中。我们还包括在不同二进制文件中进行的Wilcoxon符号秩检验，以查看性能差异是否在统计学上显着。

如表1所示，**在最佳情况和最坏情况下**，Asm2Vec的性能均明显优于静态函数。 它还优于**BinGo**（动态），后者是一种涉及**动态函数的最新语义克隆搜索方法。** 它表明，**Asm2Vec可以抵抗繁琐的语法修改和编译器引入的密集内联。** 即使在最坏的情况下，学习的表示形式仍然可以正确

地匹配position 1处超过75%的汇编函数。它甚至可以与针对语义克隆的最新动态方法CACompare相比，具有竞争优势。 由于样本量较小，差异在统计上没有差异。 Asm2Vec在不同的库中稳定运行，并能够以高精度找到克隆。 平均而言，在编译器优化选项O1，O2和O3之间检测克隆时，它可实现93%以上的精度。 随着两个优化级别之间的差异增加，Asm2Vec的性能可能会降低。 但是，它比其他静态函数要敏感得多，这证明了其鲁棒性。

Discovre和Genius是最近两种使用描述性统计信息和图形匹配的静态方法。 两者均不可评估。 已证明CACompare优于Discovre [7]，Genius [6]和Blanket [10]。 我们的方法可达到与CACompare相当的性能，后者可将Asm2Vec的性能与Discovre和Genius进行间接比较。

在最佳情况下，我们在优化级别O2和O3之间进行比较，基线静态函数的性能与原始论文中报告的结果一致，这表明了我们实施的正确性。

> 原始论文指的是什么？其他检测克隆代码的工具实验结果吗

在最坏的情况下，我们注意到，基于汇编指令和图形结构，Constant模型的性能优于其他静态特征。 原因是常量token不会受到汇编指令和子图结构更改的影响。 我们还注意到，在最坏的情况下，BinGo的性能优于静态函数。 但是，在最佳情况下，其性能不如Graphlet-C和n-grams等静态功能好，因为符号逻辑级别的噪声高于汇编代码级别的噪声。 逻辑表达式可促进回忆，并且在语法非常不同时可以发现克隆。 但是，汇编说明可以提供更精确的信息以进行匹配。

最大的二进制文件OpenSSL具有5,000多个函数。 Asm2Vec平均需要153毫秒来训练汇编函数，而平均需要20毫秒来处理查询。 对于OpenSSL，CACompare完成查询平均需要12秒。

5.2. Searching with Code Obfuscation

使用代码混淆进行搜索

混淆器-LLVM（O-LLVM）[24]建立在LLVM框架和CLANG编译器工具链的基础上。 **它在中间语言(IR)级别上运行，并在生成二进制文件之前修改程序的逻辑。** 这增加了二进制代码的复杂性。 O-LLVM使用三种不同的技术及其组合：**伪造控制流图（BCF），控制流平化（FLA）和指令替换（SUB）**。 图7显示了差异统计。

BCF: 通过添加大量不相关的**随机基本块和分支**来修改控制流图，还将**拆分，合并和重新排序原始基本块。** BCF破坏了CFG和基本块的完整性(**均增加了149%的vertices/edges(顶点/边)**)。

FLA : 使用**新条件的复杂层次结构作为开关来重组原始CFG**(请参见图1中的示例)。 **原始说明经过了重大修改，以适应新的输入条件和变量。 线性布局已被完全修改(平均添加了376%的顶点和边)。** 基于图的功能忽略了该技术, 动态方法完全覆盖CFG也是不可扩展的。

SUB : 通过使用预定义的规则遍历函数逻辑，将汇编代码的片段替换为等效形式。 该技术修改了基本块的内容并添加了新的常数。 例如，将加法转换为$a = b - (-c)$。 减法被转换为$r = rand(); a = b - r; a = a - c; a = a + r$。并且运算被转换为$a = (b \wedge \sim c) \& b$.**SUB**不会改变图的大部分结构（**91% 函数的顶点和边数保持相同**）。

$BCF + FLA + SUB$使用上面的所有混淆选项。 O-LLVM大量修改了原始汇编代码。它破坏了CFG和基本块的完整性。 通过设计，大多数静态功能都不会混淆。 **通过将CLANG编译器与O-LLVM结合使用，我们成功地编译了上一个实验中使用的四个库，并使用它们评估了Asm2Vec。**使

用CLANG + O-LLVM 工具链编译其他二进制文件时出现编译错误。根据图7，原始文件与被BCF和FLA混淆的文件之间存在显着差异。 BCF将顶点和边的数量加倍。 FLA几乎是后者的两倍。 使用SUB时，汇编指令的数量显着增加。 除了BinGo和CACompare之外，我们使用与先前实验相同的基线和配置集，因为它们不可用于评估，并且原始论文也未包含此类实验。

我们首先在不应用任何混淆技术的情况下编译选定的库。 之后，我们使用选定的混淆技术再次编译该库，**以具有原始的和混淆的二进制文件**。 我们使用调试符号链接它们的汇编函数，并在汇编函数之间生成一对一的克隆映射。 此映射仅用于评估目的。 剥离二进制文件后，我们针对混淆后的文件搜索原始文件。 然后，我们针对原始内容进行混淆处理。 我们报告平均值。 我们使用Precision @ 1作为评估指标。在这种情况下，Precision @ 1等于Recall @ 1，因为我们将查询的“no-answer”视为零精度。

	O-LLVM - Bogus Control Flow Graph (BCF)					O-LLVM - Instruction Substitution (SUB)				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.226	.224	.312	.246	.252	.620	.675	.600	.766	.665
Constant	.130	.592	.412	.318	.363	.173	.622	.492	.360	.412
Graphlet	.003	.005	.033	.007	.012	.198	.158	.411	.308	.269
Graphlet-C	.112	.118	.165	.124	.130	.626	.572	.539	.585	.581
Graphlet-E	.026	.011	.050	.014	.025	.454	.216	.286	.271	.307
MixedGram	.220	.234	.375	.303	.283	.585	.642	.563	.743	.633
MixedGraph	.011	.007	.049	.014	.020	.356	.325	.495	.488	.416
n-gram	.134	.134	.295	.195	.190	.466	.516	.513	.670	.541
n-perm	.233	.224	.374	.274	.276	.557	.624	.558	.736	.619
FuncSimSearch	.109	.022	.029	.027	.047	.685	.442	.699	.330	.539
PV(DM/DBOW)	.784	.870	.842	.768	.816	.935	.968	.964	.958	.956
Asm2Vec *	**.802**	**.920**	**.933**	**.883**	**.885**	**.940**	**.960**	**.981**	**.961**	**.961**
	O-LLVM - Control Flow Flattening (FLA)					O-LLVM - SUB+FLA+BCF				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.138	.129	.052	.027	.086	.219	.226	.015	.009	.117
Constant	.105	.480	.215	.209	.252	.137	.591	.173	.159	.265
Graphlet	.000	.002	.000	.000	.000	.000	.005	.000	.000	.001
Graphlet-C	.003	.008	.000	.001	.003	.107	.124	.000	.000	.058
Graphlet-E	.001	.002	.000	.000	.001	.020	.012	.000	.000	.008
MixedGram	.148	.143	.075	.036	.101	.221	.234	.018	.010	.121
MixedGraph	.003	.003	.000	.000	.002	.006	.010	.000	.000	.004
n-gram	.095	.093	.059	.030	.069	.154	.144	.013	.007	.079
n-perm	.133	.126	.055	.033	.087	.224	.222	.018	.008	.118
FuncSimSearch	.095	.001	.004	.008	.027	.110	.025	.003	.008	.037
PV(DM/DBOW)	**.852**	**.938**	.786	.763	.835	.780	.873	.639	.595	.722
Asm2Vec *	.772	.920	**.890**	**.795**	**.844**	**.854**	**.880**	**.830**	**.690**	**.814**

TABLE 2: Clone search between the original and obfuscated binary using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of functions that are correctly matched at position 1, which is equal to *Recall at Position 1* (*Recall@1*) in this case. The difference between *Asm2Vec* and each baseline is significant (p <0.01 in a Wilcoxon signed-rank test).

表2：使用“position 1的精度（Precision @ 1）”度量标准在原始二进制文件和模糊二进制文件之间进行克隆搜索。 它捕获在位置1正确匹配的功能的比率，在这种情况下，该比率等于在位置1的调用（Recall @ 1）。 Asm2Vec与每个基线之间的差异是显着的（Wilcoxon符号秩检验中p <0.01）。

表2显示了O-LLVM的结果。 我们发现指令替换会显着降低n-gram的性能。 SUB通过在两者之间添加指令来中断序列。 n-perm的性能优于n-gram，因为它忽略了标记的顺序。 基于图的函数仍然可以恢复超过60％的克隆，因为图的结构没有经过重大修改。 Asm2Vec可以针对汇编指令替换实现96％以上的精度。 指令被其等效形式代替，实际上，它们仍然具有与原始形式相似的词汇语义，Asm2Vec可以很好地捕获此信息。

应用BCF模糊处理后，Asm2Vec仍可以达到88％以上的精度，其中控制流图看上去已经与原始图有很大不同。 它表明Asm2Vec对插入的垃圾代码和伪造的基本块具有弹性。FLA混淆会破坏所有子图结构，子图结构特征的性能下降也反映了这一点。它们中的大多数具有约零的精度值。即使在这种情况下，Asm2Vec仍然可以正确匹配84％的汇编功能克隆。 它表明Asm2Vec对子结构更改和线性布局更改具有弹性。 应用所有混淆技术后，Asm2Vec仍可恢复约81％的汇编函数。

Asm2Vec可以正确地查明并从噪声中识别出关键模式。 插入的垃圾基本块或噪声指令遵循随机汇编代码的一般语法，可以由邻居指令轻松预测。 Asm2Vec中的函数表示捕获了邻居指令无法提供的丢失信息。 它还对这些信息进行加权，以最好地区分一个功能和另一个功能。

5.3. Searching against All Binaries

在此实验中，我们使用前两个实验中的**所有二进制文件。 当候选集很大时，我们评估Asm2Vec是否可以区分不同的汇编函数。** 我们还使用不同的检索阈值来评估其性能，以检查真实阳性的患者是否位居榜首。 **具体来说，总共有60个二进制文件，这些文件是针对不同的编译器选项（O0-O3），不同的编译器（GCC和CLANG）以及不同的O-LLVM模糊配置而编译的库的混合。** 按照Genius [6]和Discovre [7]的实验，我们考虑具有至少5个基本块的汇编函数。 但是，我们不使用抽样。 我们都使用它们。 **总共有139,936个汇编函数。 对于它们中的每一个，我们都会搜索其余的以找到克隆。** 我们对返回的结果进行排序，并依次评估每个结果。 除了FuncSimSearch外，我们使用与上一个实验相同的基线和配置集，因为它在索引所有二进制文件时会引发分段错误。

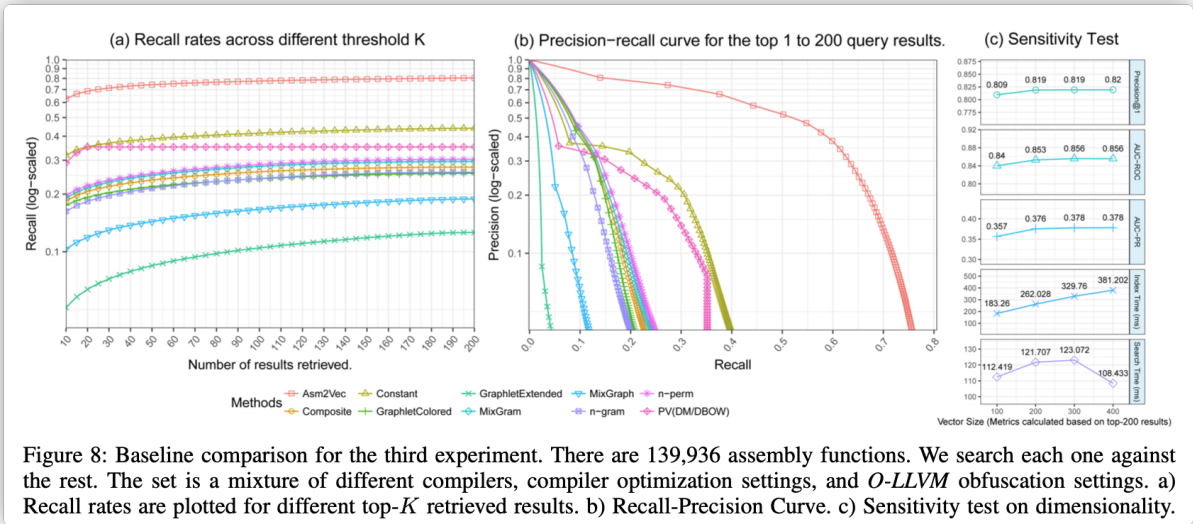


Figure 8: Baseline comparison for the third experiment. There are 139,936 assembly functions. We search each one against the rest. The set is a mixture of different compilers, compiler optimization settings, and *O-LLVM* obfuscation settings. a) Recall rates are plotted for different top-K retrieved results. b) Recall-Precision Curve. c) Sensitivity test on dimensionality.

图8：第三个实验的基线比较。 共有139,936个汇编函数。 我们针对其余每个搜索。 该集合包括不同的编译器，编译器优化设置和O-LLVM模糊设置。 a）为不同的前K个检索结果绘制了召回率。 b）调用精度曲线。 c）对维度的敏感性测试。

我们在前k个不同的位置收集召回率和精度。 我们在图8（a）中针对k绘制召回率。 我们删除图中的Graphlet，因为它的性能不比Graphlet-Extended好。 即使具有很大的汇编函数，Asm2Vec仍可将前20个结果召回70％。 它明显优于其他传统的基于token和基于图形的函数。 此外，我们观察到基于token的方法通常比基于子图的方法表现更好。

我们在图8(b)中针对每个基线绘制了针对召回率的精度。 当更改检索结果的数量时，此曲线针对精确度和查全率之间的折衷评估了克隆搜索引擎。 如图所示，Asm2Vec优于传统的汇编代码表示。 对于k = 1的返回的顶级克隆搜索结果，它达到82％的精度。假阳性平均具有33个基本块(σ= 231)。 另一方面，数据集中的所有函数平均具有47个基本块(σ= 110)。 通过单方面的Kolmogorov-Smirnov检验，我们可以得出结论，假阳性比基本人群具有更少的基本障碍($p < 2.2e^{-16}$)。 我们基于前200个结果进行敏感性测试，以评估向量大小的不同选择。 图8(c)显示具有不同载体大小的Asm2Vec对于功效和效率都是稳定的。 我们尝试合并更多的邻居指令。 但是，这增加了可能学习的模式，并且需要更多数据。 在我们的实验中，我们没有发现这种设计有效。

5.4. Searching Vulnerability Functions

在上述实验中，我们评估了Asm2Vec在匹配常规汇编函数方面的总体性能。 在本案例研究中，我们将Asm2Vec应用于[18]中公开的漏洞数据集4，以评估其在实际恢复函数漏洞的重用方面的性能，数据集包含3015个汇编函数。

对于这8个给定的漏洞中的每一个，任务是从数据集中检索其变体。 这些变体来自不同的源代码版本或由GCC，ICC和CLANG编译器的不同版本生成。 该数据集与现实情况密切相关。

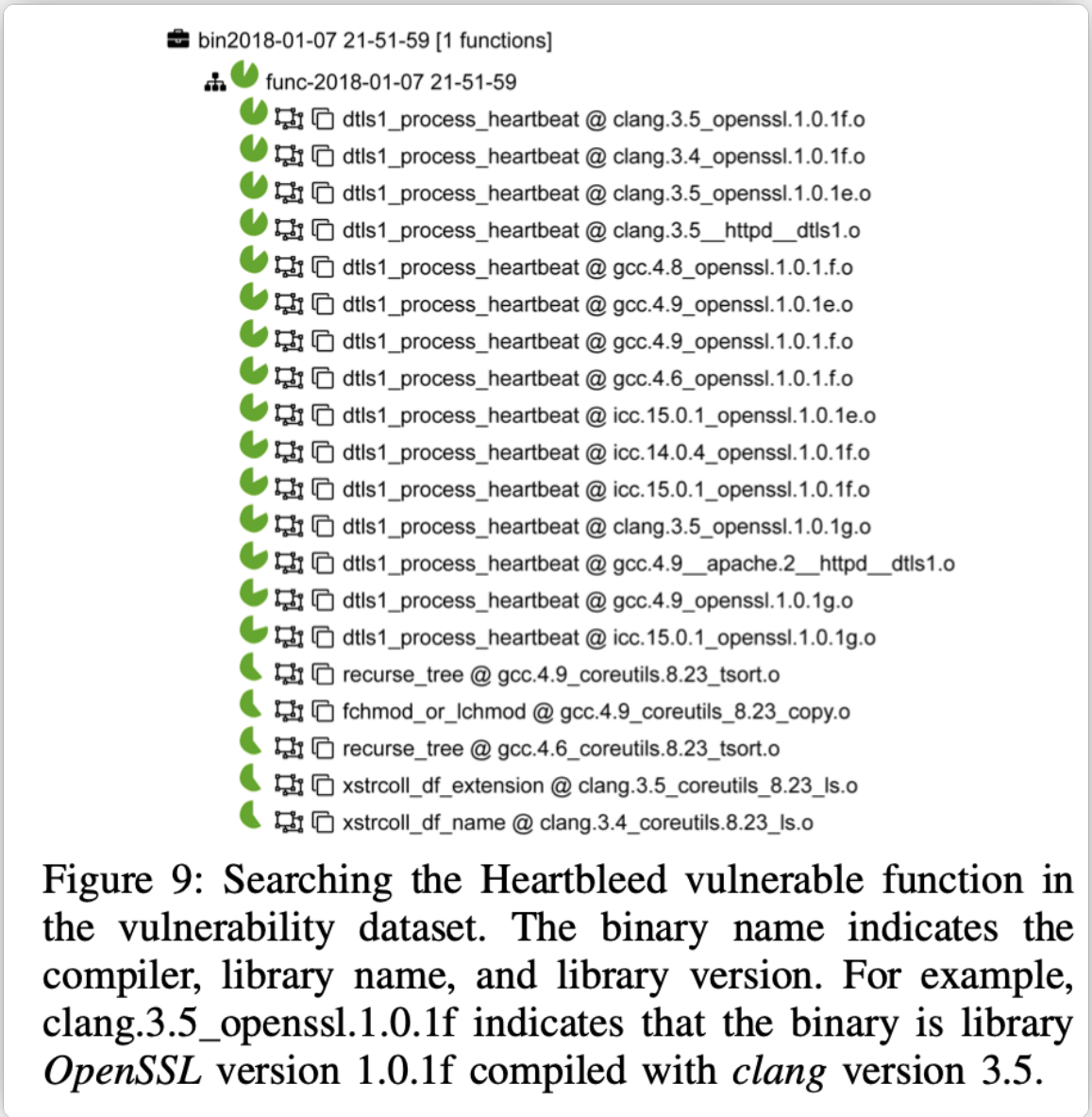


Figure 9: Searching the Heartbleed vulnerable function in the vulnerability dataset. The binary name indicates the compiler, library name, and library version. For example, clang.3.5_openssl.1.0.1f indicates that the binary is library *OpenSSL* version 1.0.1f compiled with *clang* version 3.5.

图9显示了一个使用Asm2Vec在数据集中搜索Heartbleed漏洞的示例。 该查询是一个函数，其中包含用Clang 3.5编译的OpenSSL版本1.0.1f中的Heartbleed漏洞。 共有15个不同函数其对应的二进制文件。 如排名列表所示，Asm2Vec成功检索了前15个结果中的所有15个候选者。 因此，此查询的精度和召回率为1。 第一项对应于与查询相同的功能。 但是，由于查询的表示是估算值，但训练了存储库中的表示，因此它的相似度不为1。 但是，它仍然排名第一。

		ESH [18]			Asm2Vec		
Vulnerability	CVE	FP	ROC	CROC	FP	ROC	CROC
Heartbleed	2014-0160	0	1	1	0	1	1
Shellshock	2014-6271	3	0.999	0.996	0	1	1
Venom	2015-3456	0	1	1	0	1	1
Clobberin' Time	2014-9295	19	0.993	0.956	0	1	1
Shellshock #2	2014-7169	0	1	1	0	1	1
ws-snmp	2011-0444	1	1	0.997	0	1	1
wget	2014-4877	0	1	1	0	1	1
ffmpeg	2015-6826	0	1	1	0	1	1

TABLE 3: Evaluating *Asm2Vec* on the vulnerability dataset [18] using the False Positives (FP), Receiver Operating Characteristic (ROC), and Concentrated ROC (CROC) metrics. For all the cases, *Asm2Vec* retrieves all results without any false positives.

表3中ROC为1表明了分类性极高。

我们将Asm2Vec实施为开源漏洞搜索引擎，并遵循相同的实验协议将其性能与[18]中最先进的漏洞搜索解决方案进行比较。 表3示出了结果。 我们使用与[18]相同的性能指标：误报（FP），接收器工作特性（ROC）和集中式ROC（CROC）。 对于所有漏洞，Asm2Vec的误报率为零，召回率为100％。 因此，它的ROC和CROC为1。它的性能优于[18]。

Tigress[25]是另一种高级混淆器。 它使用虚拟化和Just-In-Time（JIT）执行来转换C Intermediate Language（CIL）中间语言。 由于编译错误，Tigress无法混淆完整的库二进制文件。 因此，我们无法像第5.2节中针对O-LLVM一样评估针对Tigress的Asm2Vec。 我们使用Tigress混淆器增加了漏洞搜索的难度。 在此实验中，对于这8个不同的漏洞，我们使用编码的文字，虚拟化和即时执行来混淆查询功能。 然后，我们尝试从数据集中恢复其原始变体。 编码文字：文字整数被不透明的表达式代替。 文字字符串将替换为在运行时生成它们的函数。

虚拟化(Virtualization)：这种转变将功能转变为具有专用字节码翻译的解释器。 通过设计，静态方法很难检测到受此技术保护的克隆。 JIT：它转换功能以在运行时生成其代码。 几乎每条指令都由函数调用代替。 通过设计，静态方法几乎无法恢复任何变体。 我们的结果表明，使用编码的文字，Asm2Vec仍然能够恢复97.2％，通过虚拟化能够恢复35％，通过JIT执行能够恢复45％（参见表4）。 我们检查了结果，发现Asm2Vec试图匹配混淆器忽略的任何类似信息。 但是，在同时应用三种混淆技术之后，Asm2Vec无法再恢复任何克隆。

Searching with Obfuscation Options in Tigress									
Name	Heartbleed	ShellshocK	Venom	Clobberin' Time	Shellshock #2	ws-snmp	wget	ffmpeg	avg.
CVE	2014-0160	2014-6271	2015-3456	2014-9295	2014-7169	2014-4877	2014-4877	2015-6826	
# of Positives (k)	15	9	6	10	3	7	3	7	
Encode Literal	100%	77.8%	100%	100%	100%	100%	100%	100%	97.2%
Virtualization	0%	0%	100%	20%	100%	0%	66.7%	0%	35.8%
JIT Execution	53.3%	0%	83.3%	30%	33.3%	0%	0%	100%	37.5%

TABLE 4: True Positive Rate (TPR) of the top-k results searching the obfuscated vulnerable function against the dataset in [18]. k is chosen as the number of ground-truth clones in the dataset. For example, *Venom CVE 2015-3456-4877* has 6 variants in the dataset. By inspecting the top-6 results from *Asm2Vec* we recovered 100% (6/6) for the query with literals encoded, 100% (6/6) for the virtualized query, and 83.3% (5/6) for JIT-transformed query. After applying all the options at the same time, *Asm2Vec* cannot recover any true positives.

表4：前k个结果的真实正率（TPR），针对[18]中的数据集搜索模糊的脆弱函数。 选择k作为数据集中真实的克隆数。 例如，Venom CVE 2015-3456-4877在数据集中有6个变体。 通过检查来自Asm2Vec的前6个结果，我们为使用编码的文字的查询恢复了100%（6/6），对于虚拟化查询恢复了100%（6/6），对于JIT转换的查询恢复了83.3%（5/6） 。 同时应用所有选项后，Asm2Vec无法恢复任何真实的正值。

6. Related Work

静态方法，例如k-gram [26]，LSH-S [16]，n-gram [8]，BinClone [15]，ILine [27]和Kam1n0 [17]依赖于操作或分类为静态特征的操作数。 BinSequence [28]和Tracelet [14]将汇编代码建模为指令序列之间的编辑距离(editing distance)。 所有这些函数都无法利用操作或类别之间的语义关系。 TEDEM [29]通过表达式树比较基本块。 但是，即使在语义上相似的指令也会导致不同的表达式和副作用，这使其对指令更加敏感。 ILine [27]，Discovre [7]，Genius [6]，BinSign [30]和BinShape [31]构造了描述性统计特征，例如算术汇编指令的比例，传输指令的比例，基本块的数量和函数调用数量，等等。 基于指令的函数未能考虑指令之间的关系，并受指令替换的影响。 在NLP任务中，通常会通过过滤，二次采样或泛化来惩罚频繁出现的单词。 对于汇编语言，我们发现常用词可以提高表示的鲁棒性。

基于图的函数不会使用CFG。BinDiff [32]和BinSlayer [33]依赖于CFG匹配，它容易受到CFG变化(例如扁率(flattening))的影响。 Gitz [34]是在IR级别使用的另一种静态方法。 但是，它在基本块的边界上运行，并假定基本块的完整性，这很容易分裂。 [35]提出了一种图卷积方法。 它可能能够减轻图形操作。 但是，它依赖于有监督的学习，并且需要对等效的汇编函数进行真实的映射才能进行训练。 Asm2Vec通过考虑汇编代码中出现的标记之间的词汇语义关系来丰富静态功能。 它还避免了直接使用基于图形的功能，并且对CFG的操作更强大。 但是，CFG在某些恶意软件分析方案中很有用，尤其是用于匹配共享相似CFG结构的模板生成功能和marco生成功能。 一种方向是组合Asm2Vec和Tracelet [14]或子图(subgraph)搜索[17]。

动态方法通过动态分析目标汇编代码的行为来测量语义相似性。 BinHunt [36]，iBinHunt [37]和**ESH [18]使用定理证明者来验证两个基本块或链是否等效**。 BinHunt和iBinHunt假定基本块的完整性。 ESH假定链完整性。 它们很容易造成块分裂。 Jiang etc [38]，Blex [10]，Multi-MH [11]和BinGo [12]使用随机采样的值比较I/O值。 随机采样可能无法正确地区分两个逻辑。 如果$v! = 100$，则一个表达式输出1，否则为0。如果$v! = 20$而另一个表达式输出1，否则为0。给定广泛使用的采样范围[-1000,1000]，它们具有较高的等效可能性。 CACompare在[39]，[40]，[41]中使用的类似思想。 除了I / O值，它还记录所有中间执行结果和用于匹配的库函数调用。 在撰写本文时，CACompare使用相似的实验来匹配汇编函数，从而在二进制克隆搜索文献中获得了最佳性能。 但是，它取决于单个输入值，并且仅覆盖一个执行路径。 如作者所述，它很容易受到CFG变化的影

响。 Asm2Vec利用了词法语义而不是符号关系，它具有更高的可扩展性，并且不易受到添加的嘈杂逻辑的影响。 作为静态方法，与CACompare相比，Asm2Vec具有竞争优势。 CryptoHunt是一种用于匹配密码功能的最新动态方法。它可以检测包装的加密API调用。Asm2Vec着重于汇编代码相似性，这与CryptoHunt不同。

源代码克隆是另一个相关领域。 CCFIND-ERX [42]和CP-Miner [43]使用词汇标记作为查找代码克隆的功能。 Baxter等。 [44]和Deckard [45]利用抽象语法树进行克隆检测。 ReDebug [46]是另一个可扩展的源代码搜索引擎。 最近，深度学习已应用于该问题[47]。

7. Limitations and Conclusion

Asm2Vec受到一些限制。 首先，它是为单一汇编代码语言设计的，并且克隆搜索引擎与体系结构无关。 在此阶段，它不适用于跨架构的语义克隆。 将来，我们将通过考虑它们的共享tokens（例如constants和libc调用）来使两种不同的汇编语言之间的词汇语义空间对齐。 其次，当前的选择性被调用者扩展机制无法确定动态跳转，例如跳转表。 第三，作为黑盒静态方法，Asm2Vec无法通过显示克隆的子图或证明符号对等来解释或证明返回的结果。 它的解释性有限。

在本文中，我们提出了一种健壮且准确的汇编克隆搜索方法，称为Asm2Vec，该方法通过将汇编函数与其他函数区别开来学习汇编函数的矢量表示。 Asm2Vec不需要任何先验知识，例如汇编函数之间的正确映射或所使用的编译器优化级别。 它学习汇编代码中出现的标记的词法语义关系，并将汇编函数表示为潜在语义的内部加权混合。 除汇编功能外，它还可以应用于不同的汇编顺序粒度，例如二进制文件，片段，基本块或函数。 我们使用不同的编译器优化选项和混淆技术，对汇编代码克隆搜索进行了广泛的实验。 我们的结果表明，Asm2Vec能够准确，强大地抵抗汇编指令和控制流程图中的严重更改。

参考链接

- https://www.cnblogs.com/Kluas/archive/2004/01/13/10230487.html
-