

研究背景

- ▣ 近年来深度学习在二进制代码分析领域被广泛使用，在准确率和效率上都明显优于传统的方法。
- ▣ 深度学习生成的嵌入可以用于针对二进制程序的各种下游任务，比如克隆检测、相似性检测、恶意代码分类等。
- ▣ 之前的大多数工作学习的是指令、基本块和函数；使用的是有监督或无监督的方法；针对跨平台场景，或者只针对跨架构或系统。

1. 可以做什么

传统的方法：图同构匹配、符号执行等

研究背景

□ 有监督和无监督

- 有监督

数据集：需要相似/不相似或相似度值的标签

例如：DeepSemantic, GESS(EuroS&P'22), VulHawk(NDSS'23)

- 无监督

数据集：不需要标签

例如：jTrans(ISSTA'22), Codee(TSE'22)

2. 现有方法和缺陷

为什么要对齐（标签）：添加监督学习的信息，学习相同语义但语法不同的代码

gess:Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection

研究背景

口有监督和无监督

- 有监督

Gemini用ACFG作为structure2vec孪生网络的输入

InnerEye把每个基本块里的指令序列用word2vec进行嵌入，再将基本块输入LSTM，得到以基本块为粒度的嵌入

- 无监督

DeepBinDiff使用word2vec计算随机游走获得基本块序列的特征向量，再用TADW算法进行嵌入

asm2vec在函数内的控制流图上进行随机游走，再用得到的指令序列训练PV-DM模型

2. 现有方法和缺陷

deepbindiff跟asm2vec和InnerEye都进行了比较

研究背景

□ Innereye(NDSS'19)

作者认为目前工作都无法在跨架构的场景处理跨函数边界时的二进制代码匹配。

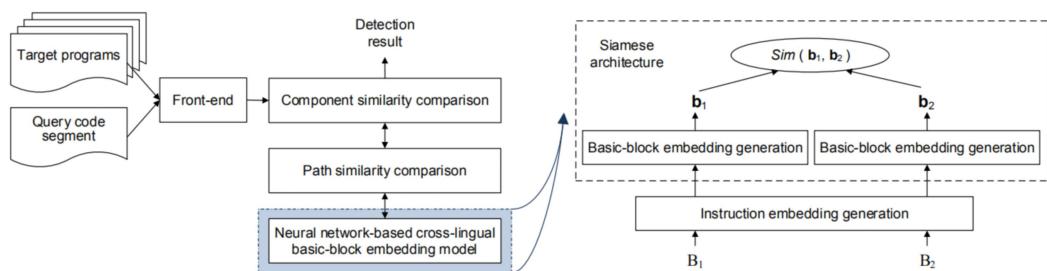


Fig. 1: System architecture.

把指令视为单词 基本块视为句子

文章认为目前都在做：给定一对函数，判断它们的相似性；

作者要做的：给定一个代码组件，可以是函数的一些基本块也可以是多个函数，判断是否被包含在一个程序中，且在跨isa场景可用，用于不同优化级别

输入：要查询的代码组件和已有程序集合。计算代码组件里的基本块序列跟程序集合里基本块序列的相似性分数。

Front-end：反汇编二进制程序和构建控制流图。数据处理：字符串用str；常量改为0，保留符号；函数名改为foo；其他符号label用tag代替

研究背景

□ Innereye-数据处理

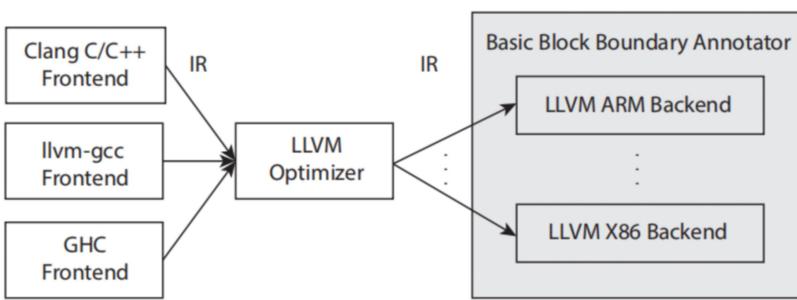


Fig. 6: LLVM architecture. The basic-block boundary annotator is added into the backends of different architectures.

LLVM由左边front-end和右边backend组成，front-end会生成统一的中间表示，在backend增加每个基本块的边界注释器：为每个从相同IR块（同一段源码）编译来的代码片段添加唯一ID。由相同IR编译生成的基本块获得相同的ID（作者在llvm6.0上实现并开源）。

尽管相同的ID总是语义相同，但是不同的ID并不一定语义不同。为了解决这个问题，另外使用N-gram来判断两个由不同代码编译所得的基本块的相似度

□ 在LLVM后端增加一个边界注释器，为同一个IR块编译来的代码片段生成相同的唯一标识

□ 不同的ID对应的代码语义不一定不同，用N-gram来判断不同ID的基本块的相似度

研究背景

□ Innereye-模型设计

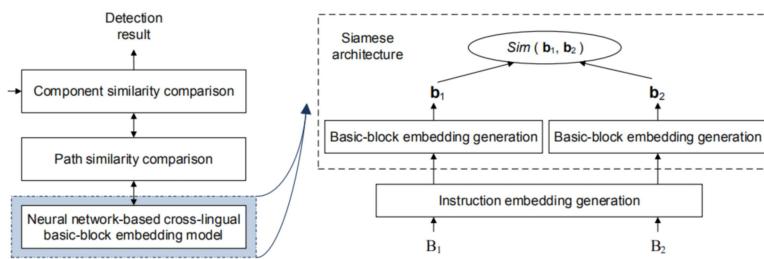


Fig. 1: System architecture.

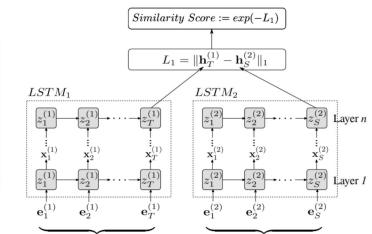


Fig. 5: Neural network-based basic-block embedding model.
Each shaded box is an LSTM cell.

把指令视为单词 基本块视为句子（每个句子中单词数目不一样，最后学习成的函数矩阵维度一样吗？：限制大小在500个单词）

在右边：

指令嵌入：使用word2vec对处理过的数据生成特征向量

基本块的嵌入：在不同平台下的指令嵌入不同，使用完全相同的lstm进行嵌入

判断代码片段是否在程序中：

component similarity: 用待测代码组件，在database里搜索语义相似（llvm编译的时候添加的id相同，id对应代码行号）的代码，得到多个候选基本块（均作为起始基本块）后进行路径探索（path similarity↓）

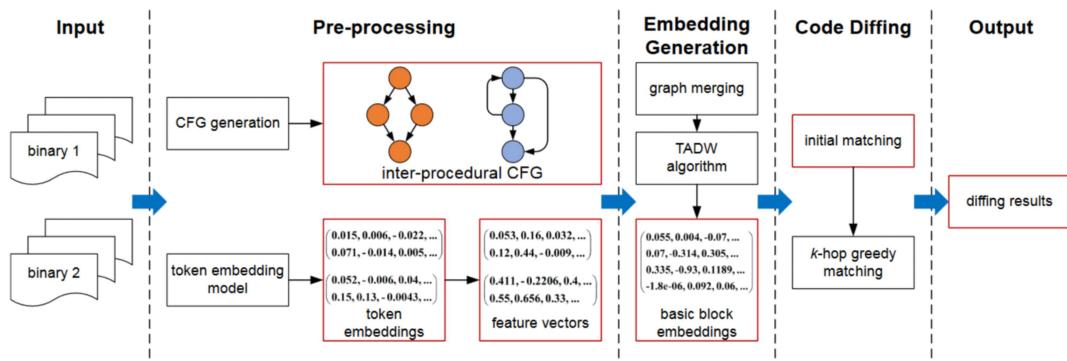
path similarity: 用lcs算法（最长公共子串）比较待测代码组件和目标程序的两条路径的相似度（CoP FSE'14）

basic block embedding model: LSTM+孪生网络，生成的基本块的嵌入保存在局部敏感哈希的database

研究背景

□ DeepBinDiff

□ 解决Innereye的缺陷：每个基本块都要通过有三百万参数的神经网络的计算，分析的时间开销大；把指令视为单词在实际中会导致OOV；对数据集质量要求高



输入：x86下没有加壳和混淆的strip文件

Fig. 1: Overview of DEEPBINDIFF.

预处理->生成基本块的特征向量->用合并后的icfg优化向量，输入tadw算法，生成embedding->比较

生成icfg：结合函数调用图和控制流图，提供程序级别的上下文信息，区分相似代码块在不同上下文位置的语义

在icfg上随机游走：覆盖所有基本块，1.每个基本块至少包含在两个随机游走序列中；2.每条随机游走路径长度包含5个基本块，得到基本块序列

将正则化后的随机游走得到的基本块序列作为训练样本，输入word2vec中的连续词袋模型CBOW（Continuous Bag-of-Words）得到基本块的特征向量

合并两个icfg，使用tadw算法优化基本块的特征向量

研究背景

□ DeepBinDiff-基本块特征

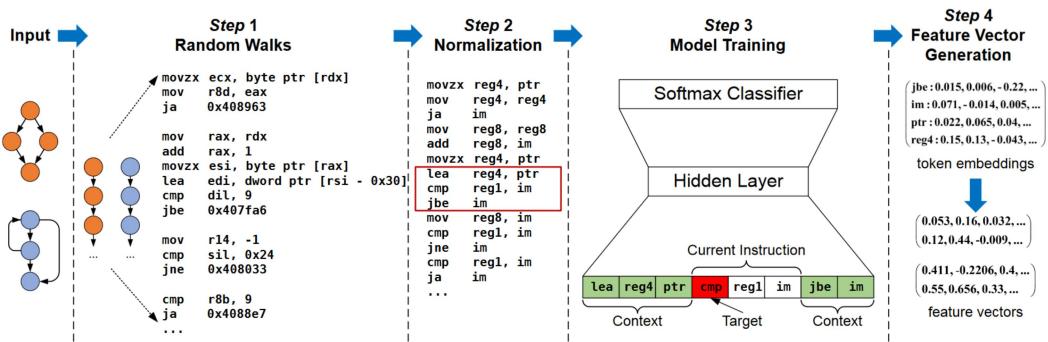


Fig. 2: Basic Block Feature Vector Generation.

正则化处理：数值替换im，指针替换ptr，寄存器根据长度重命名，保留了字符串（在innereye里被替换）

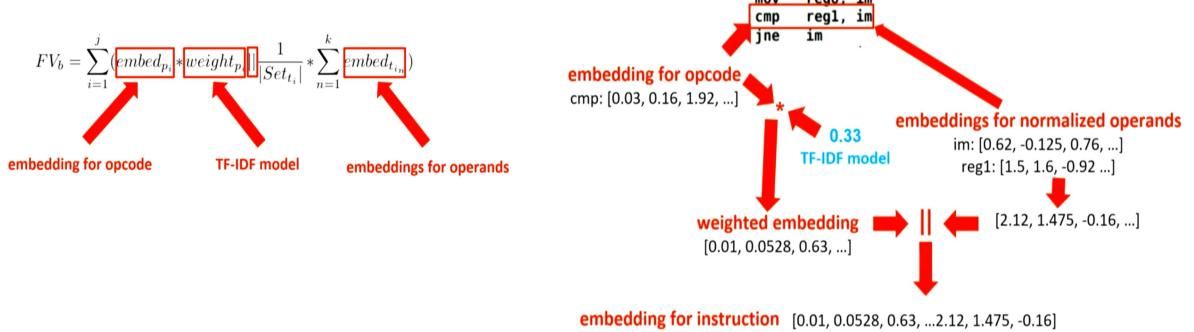
基本块的特征向量：块中所有指令的嵌入的加和，操作码的权重通过在不同优化级别里出现的比例进行调整

将正则化后的随机游走得到的基本块序列 输入word2vec中的连续词袋模型CBOW (Continuous Bag-of-Words) 得到基本块的特征向量

MODEL TRAINING: CBOW, 把紧挨着的指令视为上下文信息

研究背景

□ DeepBinDiff-基本块特征



左边：基本块特征向量的计算方法

TF-IDF:在一个基本块里某个指令出现的频率

右边：指令和操作数特征向量 i.g.

研究背景

□ DeepBinDiff-ICFG合并

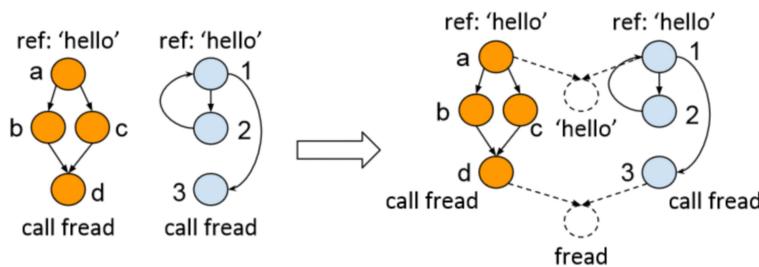


Fig. 4: Graph Merging

把合并的icfg和基本块特征向量送到tadw，获得含有语义信息和上下文信息的嵌入

合并图的原因：在两张图上分别运行TADW算法效率低，且可能忽略了一些相似性。

合并图的方法：提取基本块中的字符串/外部调用/系统调用，为其创造虚拟节点；再通过虚拟节点将两张图合并。通过这种方法，相似节点至少拥有一个共同的邻居节点，提高了相似性。

图4：两个事实上匹配的 ICFG 都有一个调用 `fread` 的基本块和另一个引用字符串 '`hello`' 的基本块。但在实际操作的时候tadw不会为它们生成相似的嵌入，因为块中还有别的指令，结构也不太相似。通过合并，节点 “a” 和 “1” 至少有一个相同的邻居，增加了相似度

研究背景

□ DeepBinDiff-TADW算法

□ DeepWalk算法 (KDD'14)

word2vec通过语料库中的句子序列来描述词与词的共现关系，进而学习到词语的向量表示；DeepWalk的思想类似word2vec，使用图中结点与结点的共现关系来学习节点的向量表示。

对每一个结点使用随机游走在图中进行结点采样，获取足够数量的结点序列后，使用word2vec的skip-gram生成向量表示。

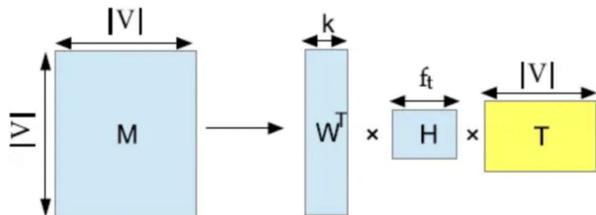
2. 现有方法和缺陷

RandomWalk是一种可重复访问已访问节点的深度优先遍历算法。给定当前访问起始节点，从其邻居中随机采样节点作为下一个访问节点，重复此过程，直到访问序列长度满足预设条件。

对于这些路径我们可以这样理解，它们就好比是自然语言处理（NLP）领域中的语料库中的句子，而节点就好比是一个一个的单词。通过随机游走，DeepWalk相当于针对某个图网络构建了一个语料库

研究背景

- DeepBinDiff-TADW算法 (IJCAI'15)
- 大多数网络表示学习研究仅仅考虑网络结构，而忽略了节点可能包含的丰富的信息，例如节点文本信息。
- 论文中证明了DW相当于对矩阵进行分解，矩阵每个元素 M_{ij} 代表从节点 V_i 经过固定步长的随机游走到达节点 V_j 的平均概率对数。



2. 现有方法和缺陷

TADW：对DeepWalk算法的改进，能够将节点特征包含到表示学习过程中。（论文证明了DeepWalk实质是一个矩阵分解的过程，基于在分解过程中加入对文本特征信息的考虑，提出了新的矩阵分解形式）

k 表示节点向量维度

那么如何在矩阵分解中加入额外信息呢？继续加入一个文本信息矩阵 T ，作为 $2k$ 维的向量表示节点属性。

<https://zhuanlan.zhihu.com/p/39347425>

研究背景

□ 缺点

- 现有的无监督学习方法只能应用在单一平台
- 现有的有监督学习方法只关注了基本块或函数的特征，没有利用过程内和过程间的控制流转移和字符串等信息；对数据集质量要求高

2. 现有方法和缺陷

大部分都存在

deepbindiff: x86

研究背景

口有监督的方法在跨ISA和跨OS的时候还存在以下问题：

1. 编译器生成的指令通常只占指令集的一小部分，还有很多片段是针对不同平台手写的汇编代码；
2. 有些API是针对特定平台的；
3. 编译器对于源码和二进制代码的对应不一定准确。

总的来说，目前的有监督方法都依靠调试信息把源码和二进制函数进行对应，但是源码中针对特定平台的代码行号不同造成不少基本块无法进行对齐。

2. 现有方法和缺陷

第三点作者在后续没有提到

有30~40%的基本块无法对齐；且有很多指令只在未对齐的基本块里出现
因此存在很多需要手动对齐的代码片段，否则难以捕捉跨平台的语义，会并且
导致数据集缩小

研究背景

```
if (mprotect(sh.map_result, pgsz,
    ↪ PROT_NONE) < 0)
    ret = 2;
...
if (VirtualProtect(sh.map_result,
    ↪ pgsz, PAGE_NOACCESS,
    ↪ &f1OldProtect) == FALSE)
    ret = 2;
...
```

(a) C source code for Linux (left) and Windows (right).

```
ecp_nistz256_point_add:
...
por %xmm4, %xmm5
pxor %xmm4, %xmm4
por %xmm0, %xmm1
movq $r_ptr, %xmm0
...
lea $Z2sqr(%rsp), $r_ptr
call __ecp_nistz256_sqr_mont$x
...
ecp_nistz256_point_add:
...
orr $t0,$a0,$a1
orr $t2,$a2,$a3
orr $in2infty,$t0,$t2
cmp $in2infty,#0
csetm $in2infty,ne
add $rp,$sp,#$Z2sqr
bl __ecp_nistz256_sqr_mont
...

```

(b) Handwritten assembly code for x86_64 (left) and AArch64 (right).

Figure 1: Semantically-equivalent code in OpenSSL whose implementation is different across OSs or ISAs.

2. 现有方法和缺陷

比如a、b是两组openssl里语义相同的代码，a、b两组的行号都各不同，依靠现有的方法就只能手动去对齐

研究背景

Table 1: Number of basic blocks that are 1-to-1 aligned (or unaligned) across platforms or compiler optimization levels. We use compiler-kept debug information for this analysis.

Configurations		# of Basic Blocks	
Platform(s) / Opt. Level(s)	1-to-1 Aligned	Unaligned	
libc.so	x86_64 ↔ ARMv7 / O1	29,626	11,531
	x86_64 ↔ ARMv7 / O2	23,274	14,501
	x86_64 ↔ ARMv7 / O3	21,106	14,307
	x86_64 / O1 ↔ O2	31,880	6,315
	x86_64 / O1 ↔ O3	29,166	7,067
	x86_64 / O2 ↔ O3	33,260	1,910
libcrypto.so	x86_64 ↔ ARMv7 / O0	133,282	4,621
	x86_64 ↔ ARMv7 / O1	79,580	16,836
	x86_64 ↔ ARMv7 / O2	67,780	22,595
	x86_64 ↔ ARMv7 / O3	62,404	21,837

Table 2: Number of unique instruction mnemonics in basic blocks that are 1-to-1 aligned (or unaligned) across platforms.

	Platform	1-to-1 Aligned (A)	Unaligned (B)	A ∪ B	B - A
libc.so	x86_64 / O2	125	203	219	94
	ARMv7 / O2	235	216	280	45
libcrypto.so	x86_64 / O2	137	215	235	98
	AArch64 / O2	119	128	138	19
	x86_64 / O0	88	225	244	156
	ARMv7 / O0	68	87	97	29
	x86_64 / O0	87	232	245	158
	AArch64 / O0	89	121	149	60

2. 现有方法和缺陷

table1是统计对齐和无法对齐的情况，每个软件大约有1/3的基本块无法对齐

table2是对齐和非对齐情况下基本块里独特指令的数量

what is unique instruction ? 包括api和汇编指令

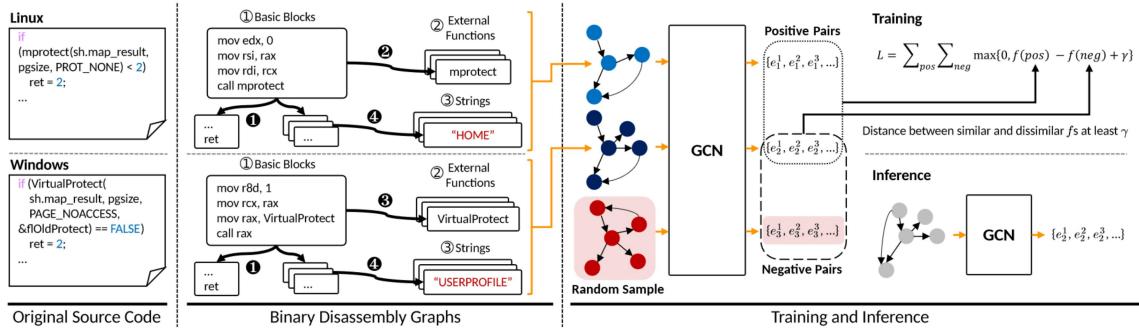
研究背景

□问题和方案

1. 学习数据集中无法对齐的代码对的语义信息——>半监督GCN
2. 让语义相似但是句法差别较大的代码对在嵌入空间里距离接近——>图对齐、损失函数

◦ ◦ ◦

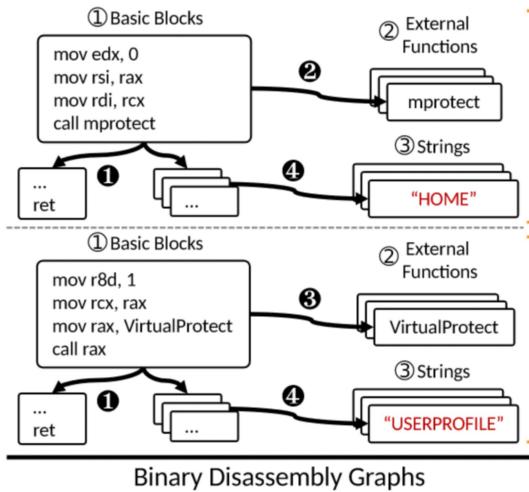
研究方法



1. 结构设计

对于每个文件生成一个二进制反编译图BDG作为输入，包括结点和邻接矩阵

研究方法



2. BDG

左边这个图的指令没有改

把BDG的结点关系表示为邻接矩阵

BDG的结点属性：基本块/外部函数名/字符串

基本块里的指令进行修改：

1. 区分直接调用和间接调用；
2. 寄存器按照位数命名为`reg1,2,4,8,16`；
3. 指针改为`ptr1,2,4,8`；
4. 常量改为“imm”

BDG里的四种边：

1. 过程内的控制流转移；
2. 直接调用；
3. 调用外部函数；
4. 引用字符串

研究方法

□邻接矩阵

表示结点之间的关系， $n \times n$ 矩阵，每个元素代表结点之间的影响和被影响的值的加和。r是边的类型；ei、ej是结点；ifun和func是对应类型的边作为入和出在整个程序里的比例

$$a_{ij} = \sum_{\langle e_i, r, e_j \rangle \in G} ifun(r) + \sum_{\langle e_j, r, e_i \rangle \in G} func(r)$$

2. BDG

研究方法

□模型

孪生结构的GCN（用的是一篇emnlp'18的论文）

两个GCN的结构相同，各自接收一个二进制文件的所有BDG结点和邻接矩阵，输出的嵌入再用距离度量函数进行计算

训练后的模型可以生成别的BDG的嵌入，得到每个BDG结点的向量

$$\mathcal{L} = \sum_{(e_1, e_2) \in M^+} \sum_{(e'_1, e'_2) \in M^-_{(e_1, e_2)}} \max \left\{ 0, f(h(e_1), h(e_2)) - f(h(e'_1), h(e'_2)) + \gamma \right\}$$

3. GCN

gcn: 用的是emnlp'18的一篇论文，bnu

Cross-lingual Knowledge Graph Alignment via Graph Convolutional Networks

gcn: 基于间隔的铰链损失函数。伽马是参数=0.1， $h(e_1), e_2$ 是正样本， e' 是负样本，
 f 用的是曼哈顿距离

M^+ 是正样本， M^- 是负样本集（随机换结点）

实验设置-数据集

Dataset	Platform	Entities	Relations	1-to-1	Alignments
		① / ② / ③	① / ② / ③ / ④	Alignments	
SQLITE3	Linux	37.1k / 0.1k / 1.5k	66.9k / 14.3k / 0.2k / 1.5k	24.0k	
	Windows	36.1k / 0.2k / 1.6k	66.0k / 14.1k / 0.2k / 1.5k		
OPENSSL	Linux	17.0k / 1.3k / 2.2k	33.0k / 10.6k / 0.2k / 2.2k	10.5k	
	Windows	15.8k / 1.1k / 2.1k	31.9k / 9.0k / 0.1k / 2.1k		
CURL	Linux	32.6k / 0.3k / 3.6k	55.0k / 10.0k / 0.1k / 3.6k	18.2k	
	Windows	30.5k / 0.3k / 3.6k	59.3k / 2.4k / 0.0k / 3.6k		
HTTPD	Linux	19.2k / 0.4k / 1.5k	32.7k / 6.0k / 0.2k / 1.4k	9.0k	
	Windows	27.8k / 0.2k / 1.7k	54.1k / 2.1k / 0.0k / 1.7k		
LIBCRYPTO	Linux	95.1k / 0.1k / 9.1k	186.7k / 54.2k / 1.5k / 9.1k	69.2k	
	Windows	93.3k / 0.2k / 9.0k	183.9k / 21.8k / 0.9k / 9.0k		
GLIBC	x86_64	63.7k / 0.0k / 1.1k	106.9k / 9.9k / 0.7k / 1.1k	13.1k	
	AArch64	51.7k / 0.0k / 1.1k	87.6k / 8.4k / 0.6k / 1.1k		
LIBCRYPTO	x86_64	95.1k / 0.1k / 9.1k	186.6k / 54.2k / 1.5k / 9.1k	67.8k	
	AArch64	95.3k / 0.1k / 9.1k	185.1k / 18.3k / 1.1k / 9.1k		

分为跨OS和跨ISA

BDG的结点属性：

1. 基本块
2. 外部函数名
3. 字符串

BDG里的四种边：

1. 过程内的控制流转移；
2. 直接调用；
3. 调用外部函数；
4. 引用字符串

1. 数据集

分为跨OS和跨ISA

- CURL SQLITE OPENSSL LIBCRYPTO HTTP跨OS, ELF和PE32+
- GLIBC LIBCRYPTO跨ISA, x64和aarch64

实验设置-Baseline

- ❑ BOW
- ❑ DeepBinDiff
- ❑ InnerEye
- ❑ 与 BOW+XBA 和 DeepBinDiff+XBA 进行比较

Table 3: Design choices of XBA in comparison with prior basic block embedding approaches.

	InnerEye [65]	DeepBinDiff [18]	XBA (Ours)
Input Format	Blocks	Sequence of Blocks	BDG (§3.1)
Contextual Info.	-	Control-flow only	Full
Cross-Platform	✓	-	✓
Method	NMT [58]	DeepWalk [47]	GCN (§3.3)
Training	Supervised	Unsupervised	Semi-supervised

bow 自然语言处理，没有引入边

deepbindiff 无监督

innereye 有监督，训练了两个，一个跨os一个跨isa

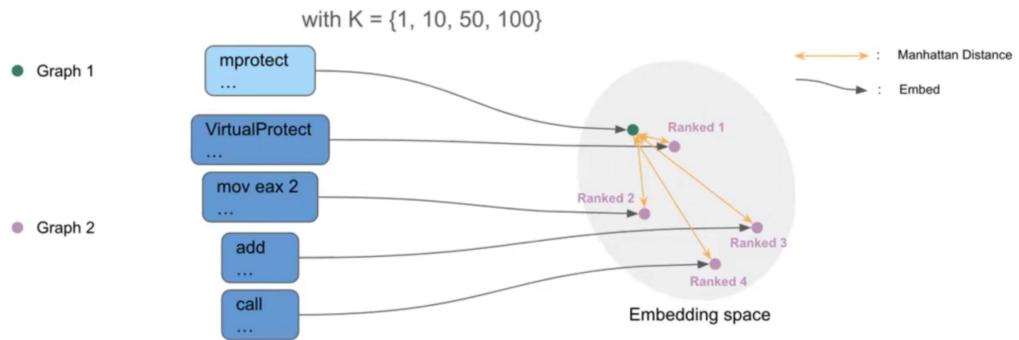
BOW+XBA 用的是 bdg 的结点，通过 bow 编码 bdg 结点的信息后作为 xba 的 gcn 的输入

deepbindiff+XBA 用的是 dbd 训练的结点特征加上 xba 的边作为 xba 的 gcn 的输入

embedding_type: A type of base feature of binary code blocks that will be an input of the first layer of GCN。结点

用 scipy.sparse 和 torch numpy 获得 dbd 和 innereye 生成的特征，生成新的矩阵

实验设置-评估方法



graph1 是待测的函数片段， graph2是训练后生成的嵌入。
在比较的时候列出嵌入空间里所有的结点，然后比较曼哈顿距离

实验结果

- 1. 对于没有预先对齐的基本块是否能进行对齐
- 2. XBA生成嵌入向量是否对真实的跨平台二进制分析有用
- 3. XBA的泛化能力
- 4. 对齐样本的比例对训练效果的影响
- 5. BDG的四种边分别有多大作用

实验结果

□1. 对于没有预先对齐的基本块是否能进行对齐

Cross-OS Configuration	Dataset	Linux→Windows				Windows→Linux			
		Hits@1	Hits@10	Hits@50	Hits@100	Hits@1	Hits@10	Hits@50	Hits@100
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	SQLITE3	70.38	79.27	85.90	90.10	71.52	80.06	86.78	89.76
		84.13	89.60	92.83	96.94	85.80	91.59	94.10	94.57
		97	99.32	99.65	99.71	97.01	99.41	99.75	99.82
		78.14	86.14	90.07	91.45	80.13	87.32	90.80	92.35
		94	98.14	99.31	99.54	94.57	98.64	99.32	99.54
		59.10	70.86	81.04	86.20	63.19	73.80	82.48	85.17
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	OPENSSL	70.47	80.80	87.96	89.28	78.59	85.08	90.66	92.38
		88.17	99.05	99.65	99.79	90.33	99.10	99.69	99.82
		63.48	79.12	86	88.50	84.65	94.66	96.62	97.14
		88.23	97.81	99.22	99.43	90.24	98.46	99.39	99.62
		73.56	81.06	90.20	92.87	74.87	83.15	89.47	91.47
		85.21	90.99	97.17	97.76	87.63	92.15	94.34	98.33
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	CURL	88.59	97.83	99.17	99.35	88.08	96.82	98.22	98.53
		76.57	85.46	89.96	91.84	79.17	87.44	90.95	92.55
		88.27	98.08	99.39	99.62	89.05	97.75	98.68	98.97
		72.27	83.98	95.09	96.27	73.43	84.38	91.57	92.99
		84.02	90.79	97.52	98.24	86.61	91.08	93.60	97.89
		87.95	98.43	99.44	99.64	87.46	97.65	98.88	99.24
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	HTTPD	68.56	83.14	90.60	92.77	65.22	79.78	87.19	90.76
		81.57	95.89	98.19	98.92	81.82	95.22	97.59	98.48
		68.96	73.50	78.26	80.42	68.15	73.01	77.11	79.92
		78.51	83.99	87.02	88.80	77.81	82.75	86.29	88.03
		79.03	94.28	97.01	97.81	81.33	95.06	97.38	97.92
		68.47	79.78	83.71	85.32	67.79	78.80	82.47	84.25
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	LIBCRYPTO	76.34	90.40	94.47	95.65	78.56	91.37	94.81	95.73

把预对齐的数据分成两部分：
一半训练，一半作为测试

Cross-ISA Configuration	Dataset	x86_64→AArch64				AArch64→x86_64			
		Hits@1	Hits@10	Hits@50	Hits@100	Hits@1	Hits@10	Hits@50	Hits@100
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff DeepBinDiff + XBA	GLIBC	1.58	2.37	6.04	9.49	1.52	3.05	6.65	9.89
		11.02	12.50	13.87	15.47	3.64	8.95	10.46	11.75
		49.92	80.92	93.62	99.53	81.70	94.24	97.40	98.18
		2.96	5.41	8.74	11.09	2.74	5.44	8.34	10.26
		29.92	62.77	81.40	87.35	30.63	62.54	81.65	87.56
		4.07	9.15	14	14.52	2.77	3.03	3.51	10.72
InnerEye BoW Encoding BoW Encoding + XBA DeepBinDiff	LIBCRYPTO	6.04	11.34	11.48	11.48	7.46	11.63	11.77	11.77
		55.32	85.41	92.48	94.75	59.78	87.48	94.35	95.86
		1.82	3.45	5.92	7.07	1.85	3.51	4.90	5.84
		28.76	61.70	79.79	85.65	30.71	64.22	81.73	87.27

Q1

把预对齐的数据分成两部分（在BDGs里把边分开），一部分训练，一部分作为测试

bow总体效果会比deepbindiff好，在How Machine Learning Is Solving the Binary Function Similarity Problem里有提到

有监督：innereye表现最差，它只用了基本块内的指令序列，没有引入别的信息
无监督：DeepBinDiff在跨ISA的场景表现不好，因为BDG的结点在不同ISA的差别很大。由于句法的差异导致语义相似性没有被捕捉到。dbd在设计的时候生成的嵌入就没有考虑跨isa

在所有的【】+xba场景下，都有提升；只有在测试openssl的时候deepbindiff大于bow

实验结果

□2. XBA生成嵌入向量是否对真实的跨平台二进制分析有用

		Rank (Top %)	
(A) Linux	(B) Windows	(A)→(B)	(B)→(A)
read	ReadFile	104 (0.53%)	72 (0.32%)
fileno	GetStdHandle	1 (0.0051%)	2 (0.0091%)
strcasecmp	strcmp	2 (0.010%)	5 (0.022%)
gettimeofday	GetSystemTimeAsFileTime	355 (0.88%)	25 (0.060%)
chdir	SetCurrentDirectory	157 (0.39%)	96 (0.23%)
"HOME"	"USERPROFILE"	84 (0.20%)	33 (0.080%)
mmap	VirtualAlloc	110 (0.10%)	2 (0.0018%)
mprotect	VirtualProtect	10 (0.0091%)	9 (0.0081%)
(A) x86_64	(B) AArch64	(A)→(B)	(B)→(A)
aesni_gcm_encrypt	aes_v8_gcm_encrypt	39 (0.035%)	137 (0.12%)
aesni_gcm_decrypt	aes_v8_gcm_decrypt	9 (0.0081%)	14 (0.012%)
gcm_init_avx	gcm_init_v8	10 (0.0090%)	1 (0.00090%)
aesni_gcm_initkey	armv8_aes_gcm_initkey	17 (0.015%)	36 (0.032%)
aesni_set_encrypt_key	aes_v8_set_encrypt_key	260 (0.23%)	105 (0.094%)
sha256_block_data_order_sse2	sha256_block_armv8	348 (0.31%)	38 (0.034%)
Camellia_Ekeygen	Camellia_Ekeygen	77 (0.069%)	89 (0.080%)
ecp_nistz256_point_add	ecp_nistz256_point_add	14 (0.0012%)	107 (0.096%)

训练集全是原本就预对齐的
人工选了其中一些未对齐的基本块进行测试
模拟真实场景

Q2

（重新进行训练）

训练集全是原本就预对齐的，每个项目对齐的个数在8,964至 69,213，用bow编码的结果作为输入。

人工选了表中的几个未对齐的BDG结点进行测试

- 跨OS测试的是api和字符串
- 跨ISA（x64和arm64）： 匹配手写汇编

- 漏洞函数匹配

验证了一个openssl漏洞的匹配，这个函数用汇编写的。表明xba能够帮助进行漏洞函数匹配。（但是没有跟其他方法进行对比）

实验结果

□3. XBA的泛化能力

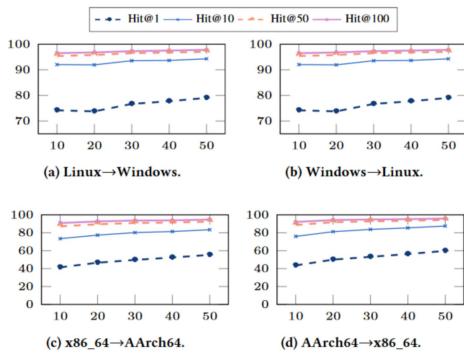
Linux→Windows	Hit@1	Hit@5	Hit@10	Hit@100
BoW Encoding	84.13	89.60	92.83	96.94
BoW Encoding + XBA	91.55	97.40	98.77	99.09
Windows→Linux	Hit@1	Hit@5	Hit@10	Hit@100
BoW Encoding	85.80	91.59	94.10	94.57
BoW Encoding + XBA	91.21	97.23	98.67	99.01

在跨OS的场景下与BOW比较
用其他ELF训练，用sqlite3做预测

检测在未见过的二进制文件上的表现

实验结果

□4. 对齐样本的比例对训练效果的影响



由于xba是半监督的，预对齐的样本比例可能会影响它的效果

跟Q1实验类似，修改了预对齐的比例从10%-50%，横轴就是比例

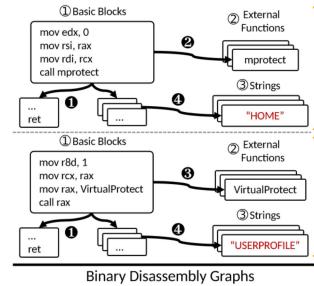
在跨OS和跨ISA场景下对嵌入质量影响都较小，纵轴是hit@k的值

用libcrypto做测试

实验结果

□5. BDG的四种边分别有多大作用

Ablation	Hit@K (Averaged over K = {1, 10, 50, 100})	
	Linux→Windows	Windows→Linux
XBA	96.23	95.42
without ③	96.07	95.50
without ④	95.45	95.09
without ②	92.46	91.25
without ①	73.06	80.06



用跟q1相同的方法，数据集为curl。对于四个边，每一次把邻接矩阵里的对应位置置0

过程内的控制流转移； 直接调用； 间接调用和访问内存地址的操作； 引用字符串

可以看出控制流关系的信息是xba最依赖的

3出现的比较少，但猜测还是有用的，比如函数只有被间接调用过的时候

Q10

- Q1 论文试图解决什么问题?
- Q2 这是否是一个新的问题?
- Q3 这篇文章要验证一个什么科学假设?
- Q4 有哪些相关研究? 如何归类? 谁是这一课题在领域内值得关注的研究员?
- Q5 论文中提到的解决方案之关键是什么?
- Q6 论文中的实验是如何设计的?
- Q7 用于定量评估的数据集是什么? 代码有没有开源?
- Q8 论文中的实验及结果有没有很好地支持需要验证的科学假设?
- Q9 这篇论文到底有什么贡献?
- Q10 下一步呢? 有什么工作可以继续深入?

1. 提升嵌入表示的准确性
2. 不是
3. 没有
4. 除了提到的几种方法外; issta'23的Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis也是提升了jtrans和trex的效果
5. BDG
6. 说过了
7. 公开了, 代码部分开源
8. 不存在
9. BDG的这个结构; 对之前的模型效果的提升
10. 在不同优化级别和编译器、应用在别的模型上



中国科学院 信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

非常感谢您的观看
THANK YOU FOR WATCHING!