面向多样化编译环境的恶意代码同源性分析

刘昕仪[1,2], 彭国军[1,2], 刘思德[1,2], 杨秀璋[1,2], 傅建明[1,2]

[1]空天信息安全与可信计算教育部重点实验室 武汉 中国 430072
[2]武汉大学国家网络安全学院 武汉 中国 430072

摘要 随着恶意样本数量的急剧增加, 为减少人工溯源的工作量, 恶意代码同源性分析研究的重要性日益凸显。然而, 攻击者在复用恶意代码时, 需针对不同的攻击场景设置特定的编译环境, 这会造成同源二进制在语法和结构层面存在很大差异, 降低恶意代码同源性分析的准确率。为解决此问题, 本文通过分析编译环境对二进制生成带来的影响, 实现了一个准确、无监督、高效的恶意代码同源性分析方案。本文采用二进制提升与重优化技术将其统一到中间表示层, 一定程度上消除语法、结构层面的改变。针对传统 CBOW 模型学习代码单词语义的不足, 提出指令级的上下文语义学习方案, 并考虑到出现上下文无关指令的小概率事件, 结合 SIF 模型计算基本块特征向量。此外, 针对恶意代码中库函数和字符串包含敏感信息更丰富的特点, 本文提出基本块初始匹配集合的建立算法, 在 K-Hop 贪心匹配算法和线性匹配算法的基础上, 进一步提高了恶意代码同源性分析的准确率。实验表明, 对于开源恶意代码 Mirai, 本方案相较于现有的无监督模型和预训练模型, 在分析准确性和运行开销两个方面的综合表现更优。同时, 对于其他类型的恶意代码, 本方案输出的同源性指数均高于本文预先设立的同源性判定阈值, 进一步证明其有效性。

关键词 恶意代码同源性; 编译环境; 语义学习
中图法分类号 TP309.5 DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.11.03

Malware Homology Analysis under Diverse Compilation Environments

LIU Xinyi[1,2], PENG Guojun[1,2], LIU Side[1,2], YANG Xiuzhang[1,2], FU Jianming[1,2]

[1] Key Laboratory of Space Information Security and Trusted Computing, Ministry of Education, Wuhan 430072, China
[2] School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Abstract With the sharp increase in the number of malicious software samples, in order to reduce the workload of manual traceability, the importance of malware homology analysis has never been more critical. However, when attackers reuse malicious codes, it is necessary to set up a specific compilation environment for different attack scenarios. This diversity in compilation environments leads to significant variations in the syntax and structure of homologous binaries, thus compromising the accuracy of malware homology analysis. To solve this problem, we implement an accurate, unsupervised and efficient malware homology scheme by analyzing the impact of compilation environments on binary generation. We adopt the binary promotion and re-optimization technologies to unify binaries to the same intermediate representation layer, which eliminates the syntax and structural changes to a certain extent. Aiming at the insufficiency of the traditional Continuous Bag of Words (CBOW) model in token semantics learning, an instruction-level contextual semantics learning scheme is proposed. And considering the small probability events of context-independent instructions, we use the Smooth Inverse Frequency (SIF) model to calculate feature vectors of basic blocks. In addition, in view of the fact that library functions and strings in malwares contain richer sensitive information, we propose an establishment algorithm of the initial matching set of basic blocks, which further improves the accuracy of malware homology analysis based on K-Hop greedy matching algorithm and linear matching algorithm. Experimental results demonstrate the effectiveness of our solution. When applied to the open-source malware Mirai, compared with the existing unsupervised model and pre-trained model, this solution has better overall performance in terms of analysis accuracy and running cost. At the same time, for various other types of malwares, the homology indexes output by this scheme are all higher than the homology judgment threshold we set, further validating its utility in the field of malware homology analysis.

Key words malware homology; compilation environments; semantic learning

通讯作者: 彭国军, 博士, 教授, Email: guojpeng@whu.edu.cn。
本课题得到国家自然科学基金资助项目(No. 62172308, No. U1626107, No. 61972297, No. 62172144)资助。
收稿日期: 2023-02-14; 修改日期: 2023-10-03; 定稿日期: 2024-09-29

1 引言

二进制相似性检测技术能应用于恶意代码同源性分析, 有效减少安全人员在恶意代码溯源时的工作量。然而, 针对 IoT 设备的恶意软件, 采用不同的交叉编译器编译同一份源代码, 会生成运行在不同架构的二进制程序[1]。针对不同攻击场景对恶意软件大小、运行时间的限制, 同一份恶意代码在复用时会采用不同的编译器或编译选项[2]。这两点导致同源二进制在机器码和反汇编代码的表现上存在较大差异, 为相似性检测带来巨大挑战, 因此本文研究面向多样化编译环境的恶意代码同源性分析问题。

二进制相似性分析的对象包括原始字节[3]、反汇编代码[4-23]、中间表示[24-28](Intermediate Representation, IR)三种。针对后两者, 具体的分析方法可分为语法、结构、语义三个层面[29]。语法层面是指代码指令上的相似[4-9,24], 结构层面是指数据流图[9-10]、控制流图[6-8,11-12,25]、过程间控制流图[13-14](Interprocedural Control Flow Graph, ICFG)等图结构的相似性, 语义层面是指二进制代码功能上的相似性[10-28]。现有研究表明, 在面向多样化的编译环境时, 结合结构和语义层面的二进制相似性分析技术具有更强的鲁棒性。

然而在恶意代码同源性分析的应用中, 现有二进制相似性检测技术各自存在一定的局限性, 主要表现为以下三个方面: 1)多样化的编译环境导致汇编指令、图结构信息有很大差异, 从而降低利用此类信息的分析准确率; 2)基于机器学习的语义分析方法依赖大量正确标记的数据集, 而开源恶意代码数量较少, 闭源恶意二进制无法保证标记的正确性, 导致模型泛化能力有限; 3)一些语义分析技术的开销较大、效率较低, 如模拟执行获取代码的输入输出[15,28]、动态执行捕获代码的具体行为[9,27]等方法依赖约束求解器、模拟器等高复杂度算法或工具。

针对上述问题, 本文提出了一种准确、无监督、高效的恶意代码同源性分析方案: 首先, 通过二进制提升技术和重优化技术将二进制统一到相同语法、优化级别的 LLVM IR, 一定程度上消除不同架构、编译器、优化级别带来的干扰, 解决问题 1。接着, 把中间代码看作自然语言, 基于无监督的连续词袋模型[30](Continuous Bag of Words, CBOW)学习指令的上下文语义, 无需大量的开源恶意代码, 避免了问题 2 中方法的数据依赖。然后, 参考 SIF (Smooth Inverse Frequency)句向量生成算法[31], 在指令权重中考虑上下文无关指令的出现概率, 通过求指令向量的加权平均极大简化了基本块特征向量的生成算法, 避免了问题 3 中方法的效率局限性。最后, 充分利用恶意样本中的库函数、字符串特征, 优先匹配语义更丰富的基本块作为初始集合, 采用 K-Hop 贪心匹配算法[13]、线性匹配算法匹配两函数的基本块, 从而得到二进制函数同源的可能性。

本文的主要贡献总结如下:

1) 提出一种无监督的中间代码语义提取模型, 在传统 CBOW 模型生成词向量的基础上, 建立指令级的上下文语义学习模型, 提取 IR 指令间的语义关系, 并结合 SIF 模型, 引入上下文无关指令的出现概率, 生成基本块特征向量。

2) 针对 K-Hop 贪心匹配算法, 提出一种基本块初始匹配集合的建立算法, 优先匹配敏感语义更丰富的基本块, 能有效利用恶意代码中的库函数、字符串特征, 并降低贪心匹配时的错误率。

3) 针对多样化编译环境, 本文结合二进制提升与重优化技术、语义学习与匹配技术, 实现了一个恶意代码同源性分析系统。在单环境变量下, 对比实验的结果表明该系统在分析准确性上与大型预训练模型效果相当, 在运行开销上优于现有无监督模型。在多环境变量下, 本方案 F1 分数平均为 81.33%。

本文余下内容按照如下方式组织: 第 2 节分析多样化编译环境给二进制同源性度量带来的难点, 并总结现有研究技术; 第 3 节介绍本文提出的恶意代码同源性分析方案的设计框架; 第 4 节详细介绍涉及的关键技术与实现方式; 第 5 节对开源恶意代码 Mirai 及其变种进行实验分析, 并扩展分析了其他类型恶意代码、加壳恶意代码的实验结果; 最后总结本文工作。

2 背景

为了抵抗多样化编译环境给同源性分析带来的干扰, 本节分析了由源代码编译生成二进制文件过程中的可变因素, 即目标架构、目标操作系统、编译器、编译选项。然后, 针对这些可变因素在二进制同源性分析时造成的干扰, 介绍了基于语义分析的现有技术及其局限性。

2.1 编译环境的可变因素

由一份源代码生成二进制可执行文件的过程分为 4 个阶段: 预处理、编译、汇编、链接[29], 如图 1 所示。本文关注的是从预编译文件到目标文件的编译、汇编过程, 即对高级语言进行词法分析、语法分析、语义分析, 生成目标汇编代码, 再根据具体 架构生成目标文件。在恶意代码的编译过程中, 图 1 右

侧的 CPU 架构、操作系统、编译器、编译选项都是可变的。攻击者在生成恶意的二进制文件时, 针对不同平台上的受害者, 编译时需要指定不同的 CPU 架构。为满足某执行时间或代码空间的限制条件, 也需要选择不同的优化级别来编译。这些都给二进制的同源性分析带来影响。

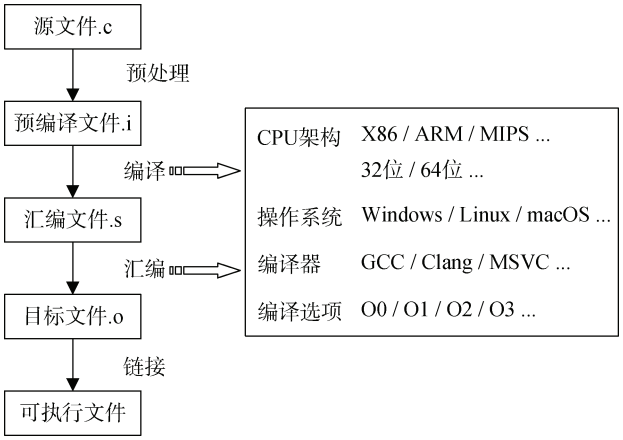


图 1 编译流程扩展

Figure 1 Extended compilation process

CPU 架构 不同处理器的指令集架构不同, 因此生成二进制文件的语法会表现出较大差异。使用交叉编译器编译图 3(a)的函数, 生成目标架构为 X86、ARM、MIPS、X64 的二进制文件, 经过 IDA Pro 反汇编如图 3(b)所示。从反汇编代码可以看出, 不同处理器架构所使用的操作码、寄存器, 以及字符串“Hello World”的寻址方式等等均有所差异。在调用 printf 函数时, 参数存放位置、代码跳转方式、返回值存放位置、栈平衡方式也不尽相同。

操作系统 不同操作系统提供了不同的应用程序二进制接口和应用程序编程接口, 编译生成的二进制文件格式不同, 在中断指令、系统调用号、库函数等方面均存在较大差异。不过, 恶意代码通常是针对特定操作系统量身定制的, 而本文所分析的同源性是指相同的源代码, 因此本文暂时不考虑跨操作系统的恶意代码同源性分析。

编译器和编译选项 在不考虑攻击者加入安全保护等特殊编译选项的条件下, 目标 CPU 架构和操作系统相同时, 不同编译器、编译选项在编译源代码时的差异主要在于优化力度的不同。不同优化类型对二进制的改变范围可以从单个语句到基本块内、循环内、函数内、函数间, 甚至是整个程序。使用 GCC 编译器编译图 4(a)中的 f 函数, 设置目标架构为 X86, 目标操作系统为 Linux, 优化级别为 O0、O1、O2、O3, 生成二进制文件的反汇编代码如图 4(b)所示。可以看到 O1、O2、O3 较 O0, 在出口基本块内删除了死赋值“$x = x + 10$”, 在函数 f 内合并了两个 if 语句的条件。除此之外, O2、O3 的 f 函数被内联到了 main 函数里, 发生了函数间的改变。

通过分析编译过程中的可变因素, 可以看到语法和结构层面的信息在很大程度上受多样化编译环境的影响, 但语义层面的信息不会发生变化。目前主流静态分析工具提取二进制的语法信息(汇编代码)和结构信息(控制流图)都较准确, 然而提取语义信息的相关技术并不成熟。

2.2 研究现状

基于语义层面的二进制相似性检测可分为语义信息提取和语义信息匹配两个阶段。

语义信息提取 源代码在多样化的编译环境下编译后, 其语义信息在二进制代码中保持不变, 它能表示代码的逻辑、行为、功能, 也常与语法特征和结构特征相结合来判断二进制的相似性。语义信息是抽象的, 提取时通常将其转换为符号公式、库函数调用/系统调用、嵌入向量。

提取符号公式的一般流程是, 先将函数依据某些特定值(如 BinGo[15]、Firmup[25]、Esh[26])或行为(如 XMATCH[27])进行切片, 然后将切片作为逻辑独立的最小单元进行规范化处理, 得到统一格式的符号公式。符号公式具有多种表示形式, 最直接的是公式字符串, 但不同的字符串“$x > 3$”和“$3 \leqslant x$”可以表达相同的语义, 于是 XMATCH[27] 将公式转换成等价的抽象语法树。此外, 按值划分的切片还可以通过公式的输入输出对表示其逻辑。

库函数调用和系统调用会执行某些特定操作, 包含特定的语义信息。BinGo[15]根据库函数的功能将其抽象到不同层次, 例如 malloc 函数和 HeapAlloc 函数在功能层面上均为分配内存, 它们和 memcpy 等内存复制函数可进一步抽象为内存相关的库函数。提取函数调用的方式有静态匹配汇编代码中的函数名(如 BinGo[15]、文献[16])、在动态执行或模拟执行过程中记录函数调用的信息。

随着自然语言处理技术(Natural Language Processing, NLP)的发展逐渐成熟, 许多方法将反汇编代码看作自然语言, 学习代码的功能逻辑, 输出包含语义信息的嵌入向量。文献[17]使用 CBOW 模型为每个汇编符号生成嵌入向量, 即反汇编代码中每个操作数和操作码都对应一个特征向量。Asm2Vec[18]建立了 PV-DM 模型[33]输出函数的嵌入向量。文献[12]使用了 BERT 预训练模型[34], 通过自监督的掩码语言模型(Masked Language Model, MLM)得到汇编符

号的特征向量。

语义信息匹配 提取了函数的语义信息后, 二进制函数的相似性转换为语义信息的相似性, 匹配语义信息的相关技术主要分为向量距离、图匹配算法、机器学习的二分类模型。

常见的嵌入向量距离有余弦距离(如 Gemini[7]、VulSeeker[9]、Asm2Vec[18])和欧式距离(如 αDiff[19]、MIRROR[20])。discovRE[4]、Genius[5]中向量的每一维元素都对应控制流图的一个原始特征, 在使用杰卡德距离计算特征相似性时, 为向量的每一维特征赋予了不同的权重。BinDeep[21]中采用汉明距离计算函数向量的相似性。

结构图的匹配通常为求解最大公共子图, 也可以将图序列化后求解最长公共子序列。discovRE[4]在求解最大公共子图问题时, 采用 McGregor 算法[35]匹配基本块。INNEREYE[11]去除了数据流图的回边, 将图匹配转为线性匹配, 并且在处理最长公共子序列问题时允许跳过不匹配的节点, 从而在一定程度上减轻编译环境对图结构带来的影响。

IMF-SIM[22]将两函数的行为特征分别映射到 12 个元素组成的向量后, 通过训练极端随机树模型来预测是同源或非同源。SAFE[23]使用自注意力机制的 RNN 模型生成函数的特征向量, 输入到 Siamese 孪生神经网络得到函数的相似性。该网络通过共享权值处理两个样本的相似性问题, 并且淡化了不同样本的类别差异。

以上技术存在一定的局限性, 例如提取符号公式的输入输出对时, 需采用 Z3 约束求解器[35], 这会导致程序运行时间较长, 降低相似性判断的效率。静态或动态提取库函数信息时, 可能无法覆盖代码所有的执行路径, 导致遗漏关键信息。NLP 模型将代码看作自然语言, 可能忽略了代码语言中的控制流、数据流等结构化信息。有监督的机器学习模型还可能过度依赖数据集。权衡各项技术的利弊后, 本文在 CBOW 模型和 SIF 句向量模型的理论基础上, 提出了一个高准确率、无监督、耗时少的恶意代码同源性分析方案。

3 总体框架

在跨架构、跨编译器、跨优化级别的多样化编译环境下, 对同一恶意的源代码编译生成不同的二进制文件, 本文研究的问题是仅通过生成的二进制代码判断其源代码是否相同。

本文设计了如图 2 所示的恶意代码同源性分析方案。给定两个二进制样本, 指定恶意样本 1 中的函数 a、恶意样本 2 中的函数 b。

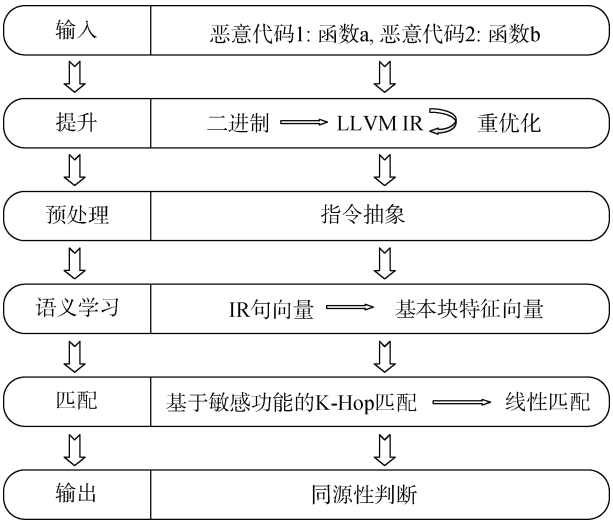


图 2 恶意代码同源性分析框架

Figure 2 Framework of homology analysis

在提升模块中, 将不同的二进制提升到 LLVM IR, 从而使不同架构的语法统一为相同语法的中间表示, 然后针对 LLVM IR 代码, 从基本块内到整个程序范围实施进一步优化, 以减轻编译器、优化级别对代码语法及结构的影响。

在预处理模块中, 对 LLVM IR 进行指令规范化, 初步抽象中间表示层的单词语义, 避免在之后的语义学习模型中出现词表过大、收敛较慢的问题。

在语义学习模块中, 类比无监督的 CBOW 模型, 通过 IR 的上下文预测目标指令, 为每条指令生成包含上下文语义的嵌入向量, 再在 SIF 句向量模型的基础上, 引入上下文无关指令的出现概率, 赋予指令权重, 生成基本块的特征向量。

在匹配模块中, 统计基本块包含的字符串和库函数, 从这些可能包含恶意功能的基本块开始, 使用 K-Hop 贪心匹配算法匹配指定函数的基本块, 剩余小部分的边缘基本块则使用线性匹配, 很大程度上缩短了图结构中节点的匹配时间。

最后根据基本块的匹配占比输出两个恶意二进制函数的同源性指数, 该指数越高代表分析认为同源的可能性越大。

4 模块实现

本节将详细介绍图 2 中提升、预处理、语义学习、匹配四个模块的实现细节。

4.1 提升

4.1.1 二进制提升

本文同源性分析的对象是代码的中间表示, 相

较反汇编代码, 它的语法与具体的运行架构无关, 更贴近源代码。本文采用的中间表示是 LLVM IR, 它是连接编译器 Clang 前端和 LLVM 后端的桥梁。源代码可通过 Clang 翻译为 LLVM IR, 输入到 LLVM 里生成目标架构下的可执行文件。因此, 在不同目标 CPU 架构下, LLVM IR 作为统一的中间表示形式, 可以减弱架构不同带来的影响。二进制提升技术将二进制代码等价转换为中间表示形式, 转换过程主要分为以下三步:

1) 识别可执行文件的运行架构类型, 针对不同架构的二进制指令, 将 Capstone 反汇编的指令转为对应的 LLVM IR 序列。

2) 识别低层次的数据信息, 如栈、全局变量、函数参数和返回值、数据类型等。

3) 提取高层次的结构信息, 如跳转目标、程序入口、调试信息等, 最终生成 LLVM IR 中间语言。

RetDec[37]是基于 LLVM 的开源反汇编工具, 分为前后端, 前端输入二进制文件、输出 LLVM IR 中间语言, 后端输入 LLVM IR、输出 C 等高级语言。本文基于其前端完成对简单指令的翻译, 并为无法翻译的复杂指令定义函数。

将图 3(b)中四种架构的反汇编函数提升到 LLVM IR 层, 它们具有统一的语法, 类似于图 3(c)。main 函数均由 call 语句和 ret 语句组成, 字符串“Hello World”的地址均作为全局变量(形如“@global_var _addr”)来访问, 并且简化了函数调用时参数设置、栈平衡相关的代码, 更贴近源代码。只是, X64 的二进制函数在提升后, 参数及返回值的数据类型仍为 64 位。不过尽管存在因 CPU 位数不同而造成中间代码数据类型的差异, 我们仍能看出将二进制代码提升到 LLVM IR 层能有效减少语法差异, 利于之后语义信息的提取。

```
int main(){
    printf("hello world");
    return 0;
}
```

(a) 源代码

```
@global_var_8408 = constant [12 x i8] c" Hello World \00"
declare i32 @printf(i8*, ...)
define i32 @main(i32 %argc, i8** %argv) {
dec_label_pc_8374:
    %0 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([12 x i8], [12 x i8]* @global_var_8408, i32 0, i32 0))
    ret i32 0
}
```

(c) LLVM IR

X86	ARM	MIPS	X86-64

lea ecx, [esp+4]
and esp, 0FFFFFFF0h
push dword ptr [ecx-4]
push ebp
mov ebp, esp
push ecx
sub esp, 4
sub esp, 0Ch
push offset format
; " Hello World "
call _printf
add esp, 10h
mov eax, 0
mov ecx, [ebp+var_4]
leave
lea esp, [ecx-4]
retn
	push {r11, lr}
	add r11, sp, #4
	ldr r3, =aHelloWorld
	; " Hello World "
	mov r0, r3
	bl printf
	mov r3, #0
	mov r0, r3
	pop {r11, pc}
		addiu $sp, -0x20
		sw $ra, 0x20+var_4($sp)
		sw $fp, 0x20+var_8($sp)
		move $fp, $sp
		li $gp, 0x419010
		sw $gp, 0x20+var_10($sp)
		lui $v0, 0x40
		addiu $a0, $v0, (aHelloWorld - 0x400000)
		# " Hello World "
		la $v0, printf
		move $t9, $v0
		jalr $t9 ; printf
		lw $gp, 0x20+var_10($fp)
		move $v0, $zero
		move $sp, $fp
		lw $ra, 0x20+var_4($sp)
		lw $fp, 0x20+var_8($sp)
		addiu $sp, 0x20
		jr $ra
			push rbp
			mov rbp, rsp
			mov edi, offset format
			; " Hello World "
			mov eax, 0
			call _printf
			mov eax, 0
			pop rbp
			retn

(b) 反汇编代码

图 3 不同目标架构下, 同源二进制的反汇编、提升后的 LLVM IR

Figure 3 Disassembly and promoted LLVM IR of homologous binary under different target architectures

4.1.2 重优化

本文将二进制提升到中间代码后, 在中间表示层进行深度的代码优化, 旨在把不同优化层次的二进制代码统一到相同优化水平的中间代码, 从而削弱编译器和优化级别给二进制生成带来的影响。

重优化的过程主要基于静态单一赋值(Static Single-Assignment, SSA)格式的代码。该格式下每个变量仅被赋值一次, 形成严格的使用-定义链, 有利于基于数据流的优化。因此, 重优化过程可分为将 LLVM IR 转为 SSA 格式、SSA 优化两步。将 LLVM

IR 转为 SSA 格式主要通过重命名每个变量为“变量名+版本号”来实现, 版本号指的是该变量被重新赋值的次数。但是, 当有多条路径可达节点 A, 并且节点 A 使用的变量 v 无法确定其数据流来源时, 需引入 Φ 函数表示变量 v 定义来源的多个候选项。针对 SSA 格式的代码, 可按照优化改变的范围进行以下三类优化:

1) 基本块内部指令的改变。例如使用别名分析法删除公共子表达式, 在 $y = x$ 指令中, y 是 x 的一个别名, 之后用 x 代替 y, 于是产生许多死赋值、死代码从而进一步简化。

2) 函数结构图的改变。例如删除没有前驱节点的基本块, 合并只有一个后继节点的基本块, 删除只有一个前驱节点的 Φ 函数等。

3) 程序范围的改变。例如函数内联用指令数低于恒定阈值的函数主体替换 call 语句, 也可以使用更复杂的启发式方法来权衡时空开销决定是否内联。

图 4(b)中不同优化级别的二进制重优化后, 生成的中间代码均如图 4(c)所示。可以看到, O0 对应的 LLVM IR 重优化后, 也消除了死赋值语句“x=x+10”, 合并了两次 if 语句的比较。至于函数间的改变, 由于 main 函数调用 f 函数时传入的参数为 5, O0 的 main 函数中也直接打印了“x>1”。可见, 对 LLVM IR 重优化能在一定程度上消除编译优化带来的影响, 对同源代码进一步统一。

4.2 预处理

由于二进制提升后将中间语言转换为 SSA 格式进行优化, 中间代码中出现了大量的变量名, 同时还包含大量地址、立即数等常量, 可能导致语言模型的训练过程中出现字典过大、稀有词过多等问题, 因此本文在预处理中规范化各变量、常量名称, 初步抽象单词的语义信息。

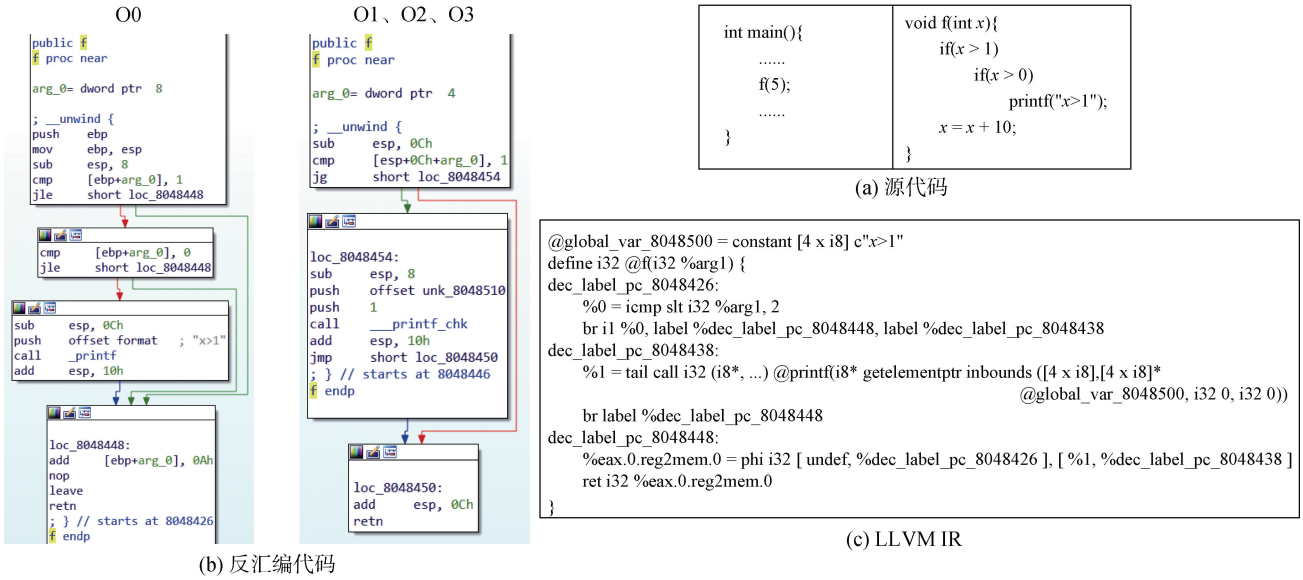
本文在对中间代码进行分词后, 首先识别以“%”开头的局部变量、以“@”开头的全局变量和常量。接着, 对变量和常量的类型进一步细分, 将变量划分为参数变量、全局变量、局部变量、栈变量、寄存器变量, 将常量划分为地址和立即数。然后向前搜索数据类型, 包括整数类型、浮点类型、x86_mmx 类型、void 类型, 以及代码中自定义的结构体类型等。同时, 根据“ptr”关键字识别出指针类型, 根据“[数据类型 x 数字]”识别出数组类型。最后, 使用“_”将数据类型和变量、常量拼接, 如图 5 所示, 其中虚线框中为可选内容。

4.3 语义学习

语义学习模块的核心思想是将中间代码看作自然语言, 生成包含语义信息的基本块特征向量。本文对规范化后的代码分词, 将 call、icmp、i32_ArgVar 等操作码和操作数看作自然语言中的单词, 每条指令看作短语, 基本块看作由短语组成的句子, 采用随机游走算法生成基本块序列, 构成文章语料。在传统 CBOW 模型的基础上, 通过学习 IR 指令的上下文预测目标指令, 形成包含上下文语义的 IR 指令向量, 然后为基本块中的指令赋予相应权重, 从而生成基本块的特征向量。

4.3.1 语料库

DeepWalk[39]在计算网络节点的隐层表示时, 利用随机游走算法将网络中的节点序列化, 从节点的



(a) 源代码

(b) 反汇编代码

(c) LLVM IR

图 4 不同优化级别下, 同源二进制的反汇编、提升后的 LLVM IR

Figure 4 Disassembly and promoted LLVM IR of homologous binary under different optimization levels

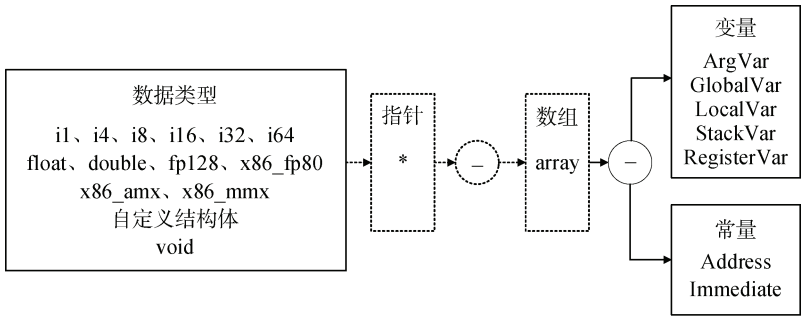


图 5　预处理规则

Figure 5　Preprocessing rules

序列中学习网络局部的结构信息。本文将随机游走的概念应用到中间代码的 ICFG, 从二进制的过程间控制流中截取适当长度的控制流片段, 简化了图结构信息的同时, 保留了部分前驱节点和后继节点间的局部关系。

本文在中间代码层分析基本块间的跳转关系, 为两二进制构建 ICFG: $G_1=(V_1,E_1)$、$G_2=(V_2,E_2)$, 其中 V_1、V_2 表示基本块的集合, E_1、E_2 表示基本块之间跳转边的集合。以基本块 $b\in V_1\cup V_2$ 为中心节点, 定义长度为 M 的随机游走 R_b^M 如下:

$$R_b^M=\{b_1,b_2,\cdots,b_M\},b\in R_b^M \tag{1}$$

其中 $b_1,b_2,...,b_M$ 为 V_1、V_2 中的基本块, 且包含基本块 b, R_b^M 中相邻基本块构成的有向边属于 E_1、E_2 中的跳转边。为了尽可能覆盖 ICFG 的所有可执行路径, 针对任意基本块 b, 构造至少 2 条随机游走, 组成语义模型的语料库 R。

4.3.2　指令向量

Asm2Vec 首次将 CBOW 的扩展模型 PV-DM 应用于汇编代码, 结合单词的上下文指令与函数标签来预测中心词。方磊等人[38]将 PV-DM 模型应用于中间语言 VEX IR, 采用滑动窗口的方式预测中心词。这两种方案的优化目标均为最大化中心词的预测概率, 在优化过程中得到函数、基本块的嵌入向量。然而在预处理后, 汇编代码和 VEX IR 的变量、常量已初步完成语义抽象, 单词之间具有较明确的语义划分。在进行单词级的上下文语义学习时, 侧重于操作码和操作数、操作数之间的语法关系, 而缺乏对指令上下文的语义学习。

因此, 本文建立 LLVM IR 指令级的上下文语义学习模型, 如图 6 所示。针对语料库 R 中的任意一条随机游走 R_b^M, 其包含的基本块 $b_1,b_2,...,b_M$ 中, 所有指令构成的 IR 序列为恶意代码中的一段可执行路径, 任意一条 LLVM IR 指令 I_j 由 L_j 个单词组成, 且可用前一条指令 I_{j-1} 和后一条指令 I_{j+1} 预测。

将指令 I_j 的每个单词映射到 d 维的向量空间 $\boldsymbol{t}_{1,j},\boldsymbol{t}_{2,j},...,\boldsymbol{t}_{L_j,j}\in\mathbf{R}^d$, 这些向量根据正态分布进行随机初始化。同时, 将指令 I_j 也映射到 d 维的向量空间 $\boldsymbol{\theta}_j\in\mathbf{R}^d$, 在模型训练的过程中通过函数 U 计算指令向量, 函数 U 的定义如公式(2)。其中, $idf_{i,j}$ 为指令 I_j 中第 i 个单词的权重。为了突出稀有单词的重要性, 本文参考了 NLP 中单词逆文档频率(Inverse Document Frequency, IDF)的定义, 即文档频率的倒数。文档总数对应这里的基本块总数, 包含单词 $\boldsymbol{t}_{i,j}$ 的文件数对应这里包含单词 $\boldsymbol{t}_{i,j}$ 的基本块数。

$$U\left(\boldsymbol{t}_{1,j},\boldsymbol{t}_{2,j},...,\boldsymbol{t}_{L_j,j}\right)=\frac{1}{L_j}\sum_{i=1}^{L_j}idf_{i,j}\cdot\boldsymbol{t}_{i,j} \tag{2}$$

同时, 指令向量 $\overrightarrow{\boldsymbol{\theta}_j}$ 可用上下文指令 $\overrightarrow{\boldsymbol{\theta}_{j-1}}$、$\overrightarrow{\boldsymbol{\theta}_{j+1}}$ 来预测, 具体计算方式如下:

$$\overrightarrow{\boldsymbol{\theta}_j}=\frac{1}{2}\left(\overrightarrow{\boldsymbol{\theta}_{j-1}}+\overrightarrow{\boldsymbol{\theta}_{j+1}}\right) \tag{3}$$

本文的优化目标是对任意指令 I_j, 最小化由上下文预测的向量 $\overrightarrow{\boldsymbol{\theta}_j}$ 和由指令单词组成的向量 $\boldsymbol{\theta}_j$ 间的绝对误差 $loss_j$。$loss_j$ 定义如下:

$$loss_j=\mathrm{MAE}(\overrightarrow{\boldsymbol{\theta}_j},\boldsymbol{\theta}_j) \tag{4}$$

此外, 在训练过程中采用 Adam 优化器[40]自动调节参数的学习率, 更新词向量来提高指令向量的预测能力。本文使用其默认参数便能快速收敛。

4.3.3　基本块向量

在 NLP 中, SIF 句向量模型是一种由词向量生成句向量的无监督方法。该方法认为, 句子 s 中单词 w 出现的可能性主要由句子 s 的语境 $\boldsymbol{c}_s\in\mathbf{R}^d$ 与词向量 $\boldsymbol{v}_w\in\mathbf{R}^d$ 的相关性决定。除此之外, 它引入了与当前语境相关性小的两类单词, 一类是仍以一定概率出现的低频词, 一类是在任何语境下均能出现的语法

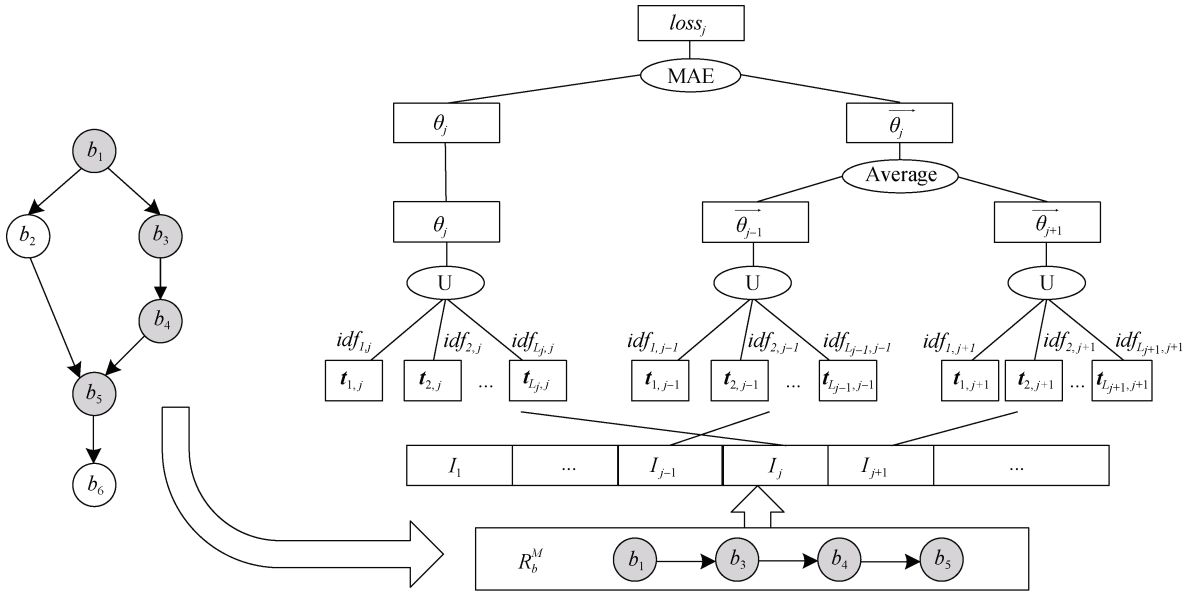


图 6　LLVM IR 指令向量模型

Figure 6　LLVM IR instruction model

类单词。于是, 该模型引入两个平滑项 $\alpha p(w)\in\mathbf{R}$、$\beta\boldsymbol{c}_0\in\mathbf{R}^d$。其中, α、β 为标量, $p(w)$ 为词频, $\boldsymbol{c}_0$ 为通用语境向量。因此, 句子 s 中单词 w 出现的概率定义如下:

$$\Pr[w\mid\boldsymbol{c}_s]=\alpha p(w)+(1-\alpha)\frac{\exp\left(\langle\tilde{\boldsymbol{c}}_s,\boldsymbol{v}_w\rangle\right)}{Z_{\tilde{c}_s}} \tag{5}$$

$$\tilde{\boldsymbol{c}}_s=\beta\boldsymbol{c}_0+(1-\beta)\boldsymbol{c}_s,\quad \boldsymbol{c}_0\perp\boldsymbol{c}_s$$

$$Z_{\tilde{c}_s}=\sum\nolimits_{w\in W}\exp\left(\langle\tilde{\boldsymbol{c}}_s,\boldsymbol{v}_w\rangle\right)$$

将句子的嵌入向量定义为 $\boldsymbol{c}_s$ 的极大似然估计, 似然函数如公式(6), 即句子 s 出现的概率为句子中所有单词的联合概率密度, 并用常量 Z 替代同一句子中的 $Z_{\tilde{c}_s}$。求解极大似然方程得到, 句向量为词向量的加权平均, 如公式(7)。

$$\Pr[s\mid\boldsymbol{c}_s]=\prod_{w\in s}\Pr[w\mid\boldsymbol{c}_s] \tag{6}$$

$$=\prod_{w\in s}\left[\alpha p(w)+(1-\alpha)\frac{\exp\left(\langle\tilde{\boldsymbol{c}}_s,\boldsymbol{v}_w\rangle\right)}{Z}\right]$$

$$\operatorname{argmax}\left\{\log\Pr[s\mid\boldsymbol{c}_s]\right\} \tag{7}$$

$$=\operatorname{argmax}\left\{\sum_{w\in s}\log\Pr[w\mid\boldsymbol{c}_s]\right\}$$

$$\propto\sum_{w\in s}\frac{a}{p(w)+a}\boldsymbol{v}_w,\quad a=\frac{1-\alpha}{\alpha Z}$$

将上述定义的句子 s 与单词 w 的关系迁移到基本块 b 与指令 I 的关系中。基本块内某指令的出现概率除了由上下文语境和自身指令向量决定外, 也需考虑两个平滑项 $\alpha p(w)$、$\beta\boldsymbol{c}_0$。前者能对恶意代码中的敏感行为给予重视, 例如 Mirai 使用一长串算术指令、逻辑指令对进程名加密后, 调用 prctl 函数隐藏自身进程, 该系统调用指令与加密指令的相关性很小, 但是也以一定的概率出现在代码中。对于后者, 能矫正一些基本块常用指令的出现概率, 例如基本块结尾的跳转指令可能与基本块内部实现的功能无关, 但仍以较大概率出现在基本块的出口代码中。因此, 定义基本块 b 的特征向量 $\boldsymbol{\mu}_b$ 如下:

$$\boldsymbol{\mu}_b=\sum_{I_j\in b}\frac{a}{p(I_j)+a}\boldsymbol{\theta}_j \tag{8}$$

其中, a 参考 SIF 句向量模型取 0.0001。本文将词频 $tf_{i,j}$ 定义为出现单词 $t_{i,j}$ 的指令数占总指令数的比例, $p(I_j)$ 为句子 I_j 中最大的词频 $tf_{i,j}$, 定义如下:

$$p(I_j)=\max_{t_{i,j}}tf_{i,j} \tag{9}$$

4.4　匹配

ICFG 中的基本块转为向量形式后, 通过向量的余弦距离可得到任意两基本块的相似度。本模块结合基本块间的相似度、代码中敏感的字符串和库函数、ICFG 的结构信息三者, 对两个函数的基本块进行匹配, 从而得到它们的同源性指数。基本块的匹配过程主要分为两步: 基于敏感功能的 K-Hop 匹配、基于匈牙利算法的线性匹配。下面将重点介绍本文对 K-Hop 贪心匹配算法的改进。

基本块匹配算法的主体是 DeepBinDiff[13]提出的

K-Hop 贪心匹配算法, 具体如算法 1 所示。输入中, 基本块初始匹配集合 *InitMatchedPairs* 是根据基本块使用的库函数和字符串来设立的(将在算法 2 中详细介绍), 初始化时需判断是否属于待匹配函数的基本块集合 *FV* 。接着, 该算法遍历待分析队列 *CurrentPairs* 中每个匹配对, 找到K个最近邻居中相似度最高的一对未匹配对 *newPair* , 若这对基本块均属于待匹配函数中的基本块集合 *FV* , 并且相似度大于基本块相似度阈值 *VSimTH* , 那么就加入匹配对集合 *MatchedPairs* 和待分析队列 *CurrentPairs* 。不断遍历待分析的基本块队列, 直到队列为空。

算法 1.K-Hop 贪心匹配.

输入: 初始匹配对集合 *InitMatchedPairs*
函数基本块集合 *FV*
基本块相似度矩阵 **SIM**
基本块相似度阈值 *VsimTH*
输出: 匹配对集合 *MatchedPaira*
过程:

```
1: CurrentPairs = InitMatchedPairs
2: FOR pair IN InitMatchedPairs DO
3:     IF pair⊆FV THEN
4:         MatchedPairs = MatchedPairs∪pair
5:         FV= FV– pair
6: WHILE CurrentPairs! = NULL DO
7:   ( b1,b2)= CurrentPairs pop()
8:   nb_b1 = GetKHopNeighbors(b1)
9:   nb_b2 = GetKHopNeighbors(b2)
10: newPair = FindMaxUnmatched(mb_b1, nb_b2, SIM)
11: IF newPair ⊆ FV and
12:   IF SIM[newPair] > VSimTH THEN
13:     MatchedPairs = MatchedPairs∪nenwPair
14:     CurrentPairs = CurrentPairs∪newPair
15:     FV= FV–newiPair
16: RETURN MatchedPairs
```

至于初始匹配集合 *InitMatchedPairs* , 本文根据基本块的敏感语义采用算法 2 设立。匹配原则是, 按基本块调用库函数和使用字符串的次数从大到小进行匹配, 即语义较丰富的基本块优先匹配, 并且要求两基本块使用恶意代码敏感库函数、字符串的种类和次数相同。算法 2 中, 输入两二进制的基本块集合 V_1、V_2 , *CalAPIStrNum* 函数统计每个基本块调用库函数和使用字符串的次数, 返回的字典 *NumVDict* 记录了不同次数对应的基本块列表。最外层循环按总次数从大到小遍历。 *CheckSameAPIStr* 函数检查基本块 b_1 、 b_2 中调用不同库函数、字符串的次数是否对应相同。

本文采用的基于敏感功能的 K-Hop 匹配除了对对敏感的库函数、字符串加以重视外, 还具有以下三个特点:

算法 2.初始匹配对集合设立.

输入: 基本块集合 V_1 和 V_2
输出: 初始匹配对集合 *InitMatchedPairs*
过程:

```
1: InitMatchedPairs = ø
2 : NumVDict1= CalAPIStrNum(V_1)
3: NumVDict2 = CalAPIStrNum(V_2)
4: NumList=list(NumVDict1.kevs∪NumVDict2.keYs)
5: NumList.sort(Reverse = True).remove(0)
6: FOR mum IN NumList DO
7:    FOR b_1 IN NumVDict1[mum] DO
8:       FOR b_2 IN NumVDict2[mum] DO
9:          IF CheckSameAPIStr(b_1, b_2) THEN
10:            InitMatchedPairs.append([b_1,b_2])
11:            Num Vdict1[mum].remove(b_1)
12:            NumVDict2[num].remove(b_2)
13:            RETURN InitMatchedPcirs
```

1) 贪心算法的时间复杂度低于线性匹配, 因此优先使用该算法能减少线性匹配时处理的基本块数量, 从而缩短匹配时长。

2) 设立的初始匹配集合相较于 DeepBinDiff 需满足更严格的匹配条件, 能有效防止基本块初始匹配时发生错位, 导致 K-Hop 匹配遗漏更多语义相似的基本块。

3) 该算法从全局 ICFG 中的基本块出发开始匹配, 能在一定程度上解决函数内联的问题。例如在匹配图 4(a)中的 main 函数时, 假设重优化算法未将 O0 对应二进制的 *f* 函数内联到 main 函数中, 仍然可以从 *f* 函数中的字符串“$x > 1$”, 通过 2 跳邻居到 main 函数的基本块中。

5 实验评估

本文对开源僵尸蠕虫Mirai家族进行同源性分析测试。针对跨优化级别、编译器、架构三个环境变量, 与现有的无监督方案 DeepBinDiff、预训练模型 PalmTree[41]对比, 本文方案具备准确性高、运行开销少的优势。同时, 针对 32 种编译环境的两两任意组合, 本文同源性分析的 F1 分数为 81.33%, 说明在面向多样化编译环境带来不同程度的干扰时, 本文方案均具备一定的抵抗力。除此之外, 本文也对其他类型的恶意代码进行了实验, 表明本文采用的二进制同源性分析方法能广泛应用于不同种类的恶意代码

分析场景。最后, 本节讨论了对加密、混淆、加壳恶意代码的解决方法。

5.1 实验数据

本实验采用的数据集主要为僵尸蠕虫Mirai家族及其变种, 其 BOT 模块可运行在不同架构的 IoT 设备上, 对目标发起 DDoS 攻击。同时, 本实验设置了 3 个环境变量, 分别是架构(X86、ARM、MIPS、X64)、编译器(GCC、Clang)、优化级别(O0、O1、O2、O3), 可组成 32 种编译环境。在不同的编译环境下, 对 50 个 BOT 模块的源码进行编译, 得到二进制文件共 1600 个、函数共 51953 个, 如表 1 所示。

表 1 实验数据设置

Table 1 Dataset settings of experiments

数据类型	数量
BOT 模块源码	50
编译环境	32
二进制文件	1600
二进制函数	51953

5.2 评价指标

现有的二进制同源性分析技术中, 无监督方案 DeepBinDiff 采用传统 CBOW 模型和 TADW 算法[42]生成基本块的语义嵌入向量, 并结合 K-Hop 贪心匹配算法得到匹配的基本块对集合, 在跨编译器、跨优化级别、跨版本方面取得了较优成果。预训练模型 PalmTree 搜集 22.5 亿条指令作为数据集, 基于 BERT 生成汇编指令的语义嵌入向量, 能在下游应用中取得较好的效果。因此, 针对跨优化级别和跨编译器的实验组(O0-O1、O0-O2、O0-O3、GCC-Clang), 本文选取 DeepBinDiff、PalmTree-KHop 的实验结果作为基准, 与采用原始 K-Hop 算法的本文方案(本文-KHop)、本文方案(本文-KHop*)进行对比实验。

至于跨架构方面, 目前大部分跨架构的语义学习模型均在中间表示层学习代码语义, 于是本实验将二进制提升到 LLVM IR 并重新编译后再输入 DeepBinDiff 模型(ReC-DeepBinDiff)、PalmTree 模型(ReC-PalmTree-KHop)中, 使其能对跨架构二进制代码进行同源性分析。因此, 针对跨架构的实验组(X86-ARM、X86-MIPS、ARM- MIPS、X86-X64), 本文选取 ReC-DeepBinDiff、ReC-PalmTree-KHop 的实验结果作为基准, 与本文-KHop、本文-KHop*进行对比实验。

本文认为, 由同一源码生成且名称相同的二进制函数是同源的, 反之则是不同源的。同源性分析的结果可分为: 同源函数被预测为同源的个数 TP、同源函数被预测为非同源的个数 FN、非同源函数被预测为非同源的个数 TN、非同源函数被预测为同源的个数 FP。为验证本文方案在面向多样化编译环境时的有效性, 将同源性分析的精确率(Precision)、召回率(Recall)、F1 分数(F1)作为评价指标, 计算如下:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{10}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{11}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{12}$$

5.3 参数选择

在句向量模型的训练过程中, 训练轮数根据模型的收敛情况确定, 这里仅测试句向量维度对模型准确率的影响。当句向量维度取 16、32、64、128 时(K 为 4, 基本块相似度阈值为 0.6), 得到 ROC 曲线如图 7。在句向量维度达到 32 时, ROC 曲线下与坐标轴围成的面积(Area Under Curve, AUC)便不再增长, 为避免增加模型的分析时长, 本文向量维度选取 32。

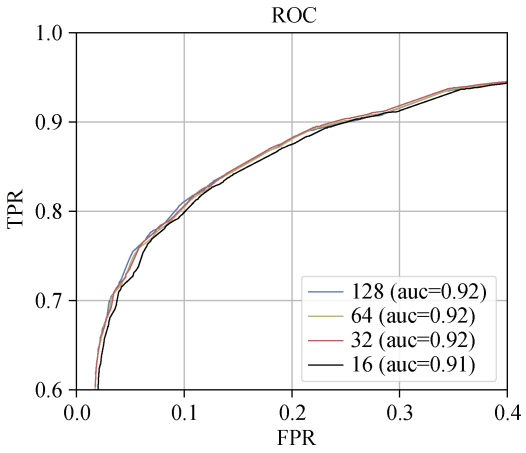


图 7 不同大小 IR 指令嵌入向量对应的 ROC

Figure 7 ROC among different sizes of the IR instruction embeddings

本文方案在基本块匹配过程中涉及两个参数, 其一是 K-Hop 匹配的 K 跳邻居, 决定了贪心搜索基本块邻居时的广度, 其二是匹配基本块的相似度阈值, 限定了匹配集合中基本块对的相似度最小值。除此之外, 在根据同源性指数判定两函数为同源和非同源时, 还需设置两函数的同源性判定阈值。

本文通过多次实验得出匹配过程的三个参数(K 值、基本块相似度阈值、函数同源性判定阈值)如表 2 所示。给定两个恶意样本, 我们可以根据二进制文

表 2 本文匹配过程的参数设置

Table 2 Parameter settings of our matching process

	K 值	基本块相似度阈值	函数同源性判定阈值
X86	2	0.5	0.67
ARM	5	0.5	0.67
MIPS	2	0.5	0.62
X64	2	0.6	0.64
X86-X64	4	0.5	0.58
X86-ARM	4	0.6	0.49
X86-MIPS	4	0.5	0.60
X64-ARM	3	0.5	0.60
X64-MIPS	5	0.6	0.50
ARM-MIPS	4	0.5	0.55

件头获取其运行架构的信息, 而难以得知其编译器类型和优化级别。因此, 针对同架构恶意代码的同源性分析, 为 X86、ARM、MIPS、X64 4 种架构分别设置了实验参数, 针对不同架构恶意代码的同源性分析, 为 X86-X64、X86-ARM、X86-MIPS、X64-ARM、X64-MIPS、ARM-MIPS 6 种跨架构组合分别设置了实验参数。

5.4 单环境变量实验分析

针对优化级别、编译器、架构这 3 个编译环境的可变因素, 本文进行了 3 组对比实验, 每组实验仅改变一个环境变量, 对比 4 个同源性分析方案的实验结果。

5.4.1 精确率、召回率、F1 分数

在跨优化级别的同源性分析实验中, 设立 O0-O1、O0-O2、O0-O3 的同源实验组和非同源实验组, 并且控制二进制函数的编译器和架构相同, 编译器均为 GCC, 架构均为 X86。在跨编译器的同源性分析实验中, 设立 GCC-Clang 的同源实验组和非同源实验组, 并且控制二进制函数的优化级别和架构相同, 优化级别均为 O0, 架构均为 X86。对比实验结果如图 8 所示。当采用原始 K-Hop 算法时, 本文方案的精确率、召回率、F1 分数平均为 92.08%、93.99%、93.02%, 优于 DeepBindiff 与 PalmTree-KHop。尤其在 O0-O3 实验组中, 本文的召回效果显著, 召回率达 90.99%。这说明, 本文采用二进制提升与重优化技术能有效消除编译器、编译优化对代码语法、ICFG 结构的影响, 同时本文建立的语义学习模型能有效提取指令上下文信息。此外, 当采用改进的基本块初始匹配集合的建立算法后, 精确率、召回率、F1 分数整体上分别提升了 0.37%、1.03%、0.69%, 说明采用更严格的匹配规则、优先匹配语义更丰富的基本块能进一步提高 K-Hop 贪心匹配时的准确性, 从而提高恶意代码同源性分析的准确性。

在跨架构的同源性分析实验中, 设立 X86-ARM、X86-MIPS、ARM-MIPS、X86-X64 的同源实验组和非同源实验组, 并且控制二进制函数的编译器和优化级别相同, 编译器均为 GCC, 优化级别均为 O0。对比实验结果如图 9 所示。当采用原始 K-Hop 算法时, 本文方案的精确率、召回率、F1 分数平均为 79.25%、81.26%、80.24%, 相较 ReC-DeepBinDiff 分别提高了 13.22%、13.88%、13.64%。可见, 尽管 ReC-DeepBinDiff 使用二进制提升技术和重优化技术在一定程度上能使跨架构代码统一到相同语法、相同优化级别, 但仅利用 CBOW 模型生成汇编词向量并不能充分学习指令的语义。而本文模型在已抽象的 IR 代码上, 通过指令的上下文预测目标指令, 能有效学习指令间的结构化信息, 弥补自然语言与代码语言之间的语义鸿沟。此外, 相较 ReC-PalmTree-KHop, 本文方案在 X86-MIPS 和 ARM-MIPS 实验组效果更佳, 说明预训练模型在跨架构同源性分析时的泛化能力有限, 而本文方案不依赖大量数据集, 在处理新数据时更准确、可靠。

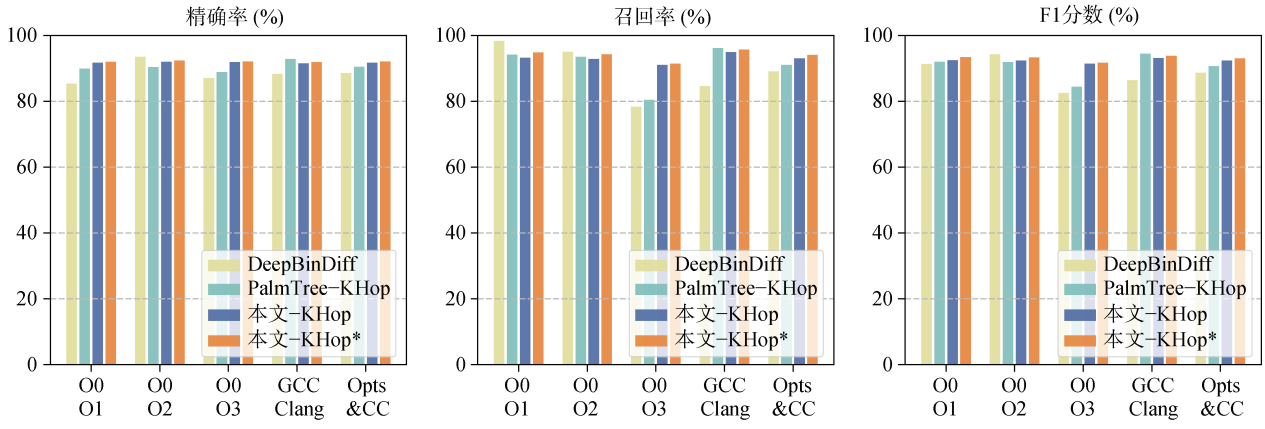


图 8 不同优化级别和编译器下, 同源性分析的精确率、召回率、F1 分数

Figure 8 Precision, recall and F1-score of homology analysis under different optimization levels and compilers

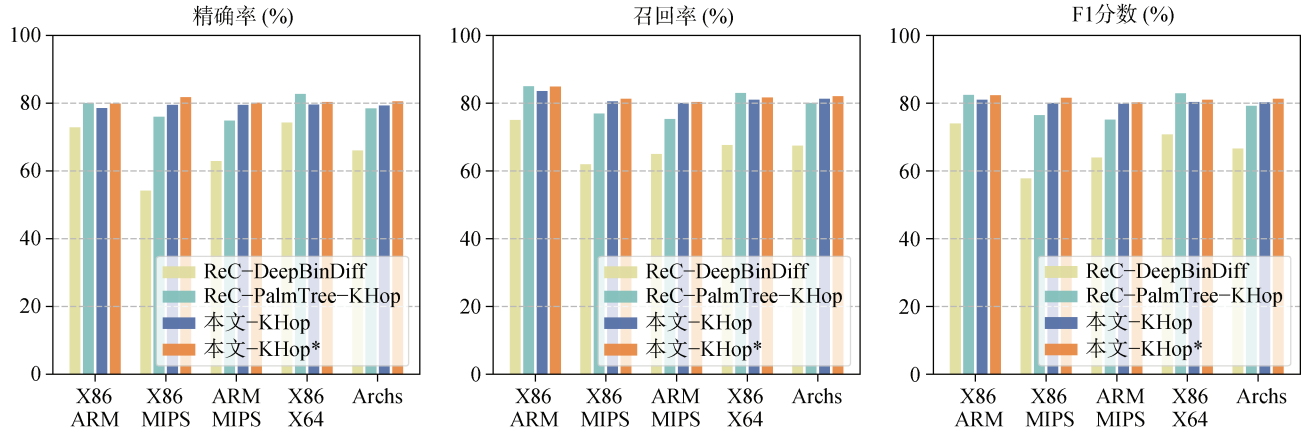


图 9　不同目标架构下，同源性分析的精确率、召回率、F1 分数

Figure 9　Precision, recall and F1-score of homology analysis under different target architectures

5.4.2　运行开销

在 Windows 10 操作系统、8GB 的内存环境中，将本文方案与 DeepBinDiff 的总运行时间对比，如图 10 所示。可以看到，当样本大小在 50～180kB 时，DeepBinDiff 的运行时间普遍大于本文方案。这是因为本文在确保精确率、召回率和 F1 分数维持较高水平的条件下，在基本块特征向量生成阶段采用加权平均的方式，很大程度上缩减了对二进制基本块特征向量的学习时间。除此之外，本文方案比 DeepBinDiff 在同源性分析的时长上表现得更稳定，这是因为在基本块匹配阶段，本文设立初始基本块对的条件更严格，要求初始基本块对调用 API 的顺序、使用字符串的种类相同，而 DeepBinDiff 是按照地址顺序，匹配了调用相同 API 或使用相同字符串的基本块，容易造成误报，从而导致 K-Hop 匹配时间更长。

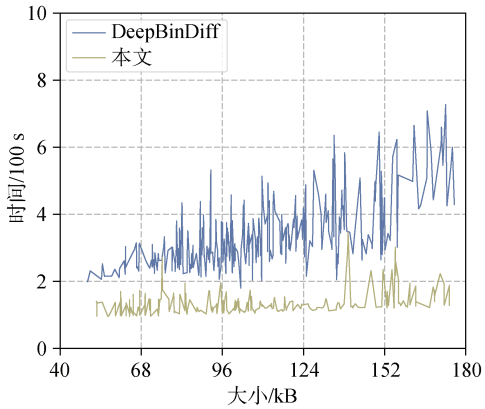


图 10　同源性分析的运行时间

Figure 10　Running time of homology analysis

5.5　多环境变量实验分析

如表 1 所示，本文考虑的三个环境变量可组成 32 种编译环境，因此对任意两个编译环境可设置 528 种组合。本文选取了 840 个函数，分别设置同源实验组和非同源实验组各 528 对，共进行了 887040 组实验。采用 5.2 中的参数设置，实验结果如图 11。总体上，对不同编译环境的全部组合，本文同源性分析的精确率、召回率、F1 分数分别为 79.91%、82.80%、81.33%。其中 66%的编译环境组合，本文同源性分析的 F1 分数高于 80%，说明对不同编译环境的大部分组合，本文能有效提取、匹配两份同源恶意代码的相同语义，具备较高的恶意代码同源性分析能力。对单个环境变量不同的组合、两个环境变量不同的组合、三个环境变量均不同的组合，本文同源性分析的 F1 分数分别为 83.45%、80.04%、79.29%。该实验结果表明，尽管在编译环境发生较大程度的改变时，本文恶意代码同源性分析方案的性能仍保持稳定。

5.6　其他类型恶意代码

针对其他类型的恶意代码，本文在 Github 上搜集了 Backdoor(Double Dragon)、Infector(Dataseg)、Ransomware(Randomware)、Trojans(Kaal Bhairav)和 Botnet 其他家族(BASHLITE)的源码。前 4 种恶意类型的源码进行跨优化级别、跨编译器编译，Botnet 类型的 IoT 恶意代码进行跨架构编译。采用本文方案对不同实验组进行同源性分析测试，输出同源性指数的均值如表 3。可以看到，针对不同类型的恶意代码，在面向多样化的编译环境时，本文方案输出的同源性指数均在 76.99%以上，均高于对应的函数同源性判定阈值。结果说明，在针对同架构不同编译选项的恶意代码与跨架构的僵尸蠕虫时，本文方案均具有较强的同源性分析能力。

5.7　加壳恶意代码

当恶意代码采用加密、混淆、加壳等手段增加

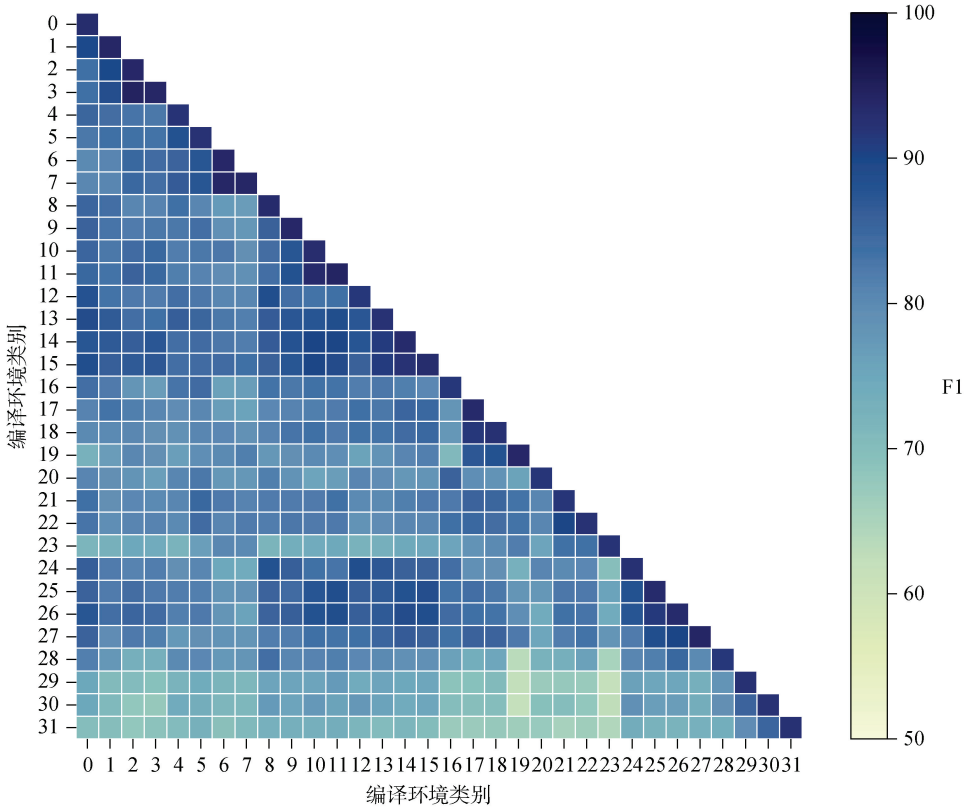


图 11　任意两个编译环境下, 同源性分析的 F1 分数

Figure 11　F1-score of homology analysis under any two compilation environments

表 3　其他类型恶意代码同源性指数

Table 3　Homology indexes of other malware types

同源性指数(%)	O0 O1	O0 O2	O0 O3	GCC Clang
Backdoor	88.35	88.35	86.11	88.89
Infector	99.07	90.49	84.24	89.10
Ransomware	98.00	98.00	96.00	85.36
Trojans	76.99	79.15	79.49	83.16
	X86 ARM	X86 MIPS	ARM MIPS	X86 X64
Botnet	81.77	83.79	84.09	88.99

其被分析的难度时, 处理后的代码在语法和结构层面完全不同, 语义层面也可能存在许多垃圾指令和无用操作, 导致采用本文方案直接对加壳恶意代码进行同源性分析时具有一定的局限性。根据现有研究, 基于语法特征、结构特征的静态分析方法均难以直接识别加壳恶意代码的同源性, 大多方法先脱壳再同源性分析。此外, 基于语义特征的机器学习方法在模型训练时, 学习的往往是壳, 而非恶意代码的真实语义逻辑[43]。

因此, 针对加壳恶意代码, 我们先脱壳再采用本文方法进行同源性分析。本课题组先前已提出 BinUnpack[44]、API-Xray[45]等方法, 能有效解决二进制脱壳及 IAT 表重建的问题。通过实验发现, 对于使用无损壳(例如 UPX 等)的恶意代码, 经过脱壳工具进行脱壳后, 本文方法所提取的 IR 指令语义和控制流图与未加壳的恶意代码完全相同, 因此同源性分析的效果也与未加壳的恶意代码相同。对于使用有损壳(如 Themida、VMProtect 等)的恶意代码, 即使经过脱壳工具进行脱壳, 也很难完全还原出原始程序, 从而在一定程度上限制了本文方法在同源性分析时的适用性。

6　结论

本文将二进制相似性检测技术应用于恶意代码同源性分析, 首先将不同编译环境生成的二进制提升到 LLVM IR 并重优化, 接着采用无监督模型学习上下文语义得到指令的嵌入向量, 再利用 SIF 句向量模型生成基本块的特征向量, 最后基于代码使用的敏感字符串和库函数匹配两函数的基本块。在系统实现方面, 该方案具备无需大量数据集、算法高效的优点。在实验结果方面, 该方案在面向跨架构、跨编译器、跨优化级别的编译环境时具有较强的鲁棒性, 能有效解决跨编译环境恶意代码复用、跨架构 IoT 蠕虫的同源性分析问题。

致 谢 感谢本文的对照基准 DeepBinDiff 原型系统、PalmTree 原型系统, 感谢审稿专家和编辑老师的指导建议。

参考文献

[1] Alhanahnah M, Lin Q C, Yan Q B, et al. Efficient Signature Generation for Classifying Cross-Architecture IoT Malware[C]. *2018 IEEE Conference on Communications and Network Security*, 2018: 1-9.

[2] Why hackers reuse malware. https://www.helpnetsecurity.com/2017/11/20/hackers-reuse-malware. Nov. 2017.

[3] Guo H, Huang S G, Huang C, et al. A Lightweight Cross-Version Binary Code Similarity Detection Based on Similarity and Correlation Coefficient Features[J]. *IEEE Access*, 2020, 8: 120501-120512.

[4] Eschweiler S, Yakdan K, Gerhards-Padilla E. DiscovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 58-79.

[5] Feng Q, Zhou R D, Xu C C, et al. Scalable Graph-Based Bug Search for Firmware Images[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 480-491.

[6] Nouh L, Rahimian A, Mouheb D, et al. BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables[M]. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2017: 341-355.

[7] Xu X J, Liu C, Feng Q, et al. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 363-376.

[8] Alrabaee S, Shirani P, Wang L Y, et al. *FOSSIL*[J]. *ACM Transactions on Privacy and Security*, 2018, 21(2): 1-34.

[9] Gao J, Yang X, Fu Y, et al. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary[C]. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018: 896-899.

[10] Alrabaee S, Debbabi M, Wang L Y. CPA: Accurate Cross-Platform Binary Authorship Characterization Using LDA[J]. *IEEE Transactions on Information Forensics and Security*, 2020, 15: 3051-3066.

[11] Zuo F, Li X P, Young P, et al. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs [EB/OL]. 2018: 1808.04706. https://arxiv.org/abs/1808.04706v2.

[12] Yu Z P, Cao R, Tang Q Y, et al. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection[J]. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020, 34(1): 1145-1152.

[13] Duan Y, Li X, Wang J H, et al. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing[C]. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[14] Yang J, Fu C, Liu X Y, et al. Codee: A Tensor Embedding Scheme for Binary Code Search[J]. *IEEE Transactions on Software Engineering*, 2022, 48(7): 2224-2244.

[15] Chandramohan M, Xue Y X, Xu Z Z, et al. BinGo: Cross-Architecture Cross-OS Binary Search[C]. *The 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016: 678-689.

[16] Li Y C, Wang B Y, Hu B J. Semantically Find Similar Binary Codes with Mixed Key Instruction Sequence[J]. *Information and Software Technology*, 2020, 125: 106320.

[17] Redmond K, Luo L N, Zeng Q. A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis[EB/OL]. 2018: 1812.09652. https://arxiv.org/abs/1812.09652v1.

[18] Ding S H H, Fung B C M, Charland P. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search Against Code Obfuscation and Compiler Optimization[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 472-489.

[19] Liu B C, Huo W, Zhang C, et al. ADiff: Cross-Version Binary Code Similarity Detection with DNN[C]. *The 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018: 667-678.

[20] Zhang X C, Sun W J, Pang J M, et al. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture[C]. *Proceedings 2020 Workshop on Binary Analysis Research*, 2020.

[21] Tian D H, Jia X Q, Ma R, et al. BinDeep: A Deep Learning Approach to Binary Code Similarity Detection[J]. *Expert Systems with Applications*, 2021, 168: 114348.

[22] Wang S, Wu D H. In-Memory Fuzzing for Binary Code Similarity Analysis[C]. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017: 319-330.

[23] Massarelli L, Di Luna G A, Petroni F, et al. SAFE: Self-Attentive Function Embeddings for Binary Similarity[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019: 309-329.

[24] Hu Y K, Zhang Y Y, Li J R, et al. Cross-Architecture Binary Semantics Understanding *via* Similar Code Comparison[C]. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016: 57-67.

[25] David Y, Partush N, Yahav E. Firmup: Precise static detection of common vulnerabilities in firmware[J]. *ACM SIGPLAN Notices*, 2018, 53(2): 392-404.

[26] David Y, Partush N, Yahav E. Statistical Similarity of Binaries[J]. *ACM SIGPLAN Notices*, 2016, 51(6): 266-280.

[27] Feng Q, Wang M H, Zhang M, et al. Extracting Conditional Formulas for Cross-Platform Bug Search[C]. *The 2017 ACM on Asia Conference on Computer and Communications Security*, 2017: 346-359.

[28] Hu Y K, Zhang Y Y, Li J R, et al. Binary Code Clone Detection across Architectures and Compiling Configurations[C]. *2017 IEEE/ACM 25th International Conference on Program Comprehension*, 2017: 88-98.

[29] Haq I U, Caballero J. A Survey of Binary Code Similarity[J]. *ACM Computing Surveys*, 2022, 54(3): 1-38.

[30] Mikolov T, Chen K, Corrado G, et al. Efficient Estimation of Word Representations in Vector Space[EB/OL]. 2013: 1301.3781.

https://arxiv.org/abs/1301.3781v3.

[31] Arora S, Liang Y, Ma T. A simple but tough-to-beat baseline for sentence embeddings[C]. *International conference on learning representations*, 2017.

[32] New SysJoker Backdoor Targets Windows, Linux, and macOS. https://www.intezer.com/blog/incident-response/new-backdoor-sysjoker. Jan. 2022

[33] Le Q, Mikolov T. Distributed representations of sentences and documents[C]. *International conference on machine learning*, 2014: 1188-1196.

[34] Devlin J, Chang M W, Lee K, et al. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding[EB/OL]. 2018: 1810.04805. https://arxiv.org/abs/1810.04805v2.

[35] McGregor J J. Backtrack Search Algorithms and the Maximal Common Subgraph Problem[J]. *Software*: *Practice and Experience*, 1982, 12(1): 23-34.

[36] Moura L, Bjørner N. Z3: An efficient SMT solver[C]. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008: 337-340.

[37] Retdec: An open-source machine-code decompiler. https://www.botconf. eu/wp-content/uploads/2017/12/2017-KroustekMatulaZemek-retdec-slides-botconf-2017.pdf. Dec. 2017.

[38] Fang L, Wei Q, Wu Z H, et al. Neural Network-Based Binary Function Similarity Detection[J]. *Computer Science*, 2021, 48(10): 286-293.
(方磊, 魏强, 武泽慧, 等. 基于神经网络的二进制函数相似性检测技术[J]. *计算机科学*, 2021, 48(10): 286-293.)

[39] Perozzi B, Al-Rfou R, Skiena S. DeepWalk: Online Learning of Social Representations[C]. *The 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014: 701-710.

[40] Kingma D P, Ba J, Hammad M M. Adam: A Method for Stochastic Optimization[EB/OL]. 2014: 1412.6980. https://arxiv.org/abs/1412.6980v9.

[41] Li X, Qu Y, Yin H. PalmTree: Learning an Assembly Language Model for Instruction Embedding[C]. *The 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[42] Yang C, Liu Z, Zhao D, et al. Network representation learning with rich text information[C]. *Twenty-fourth international joint conference on artificial intelligence*, 2015.

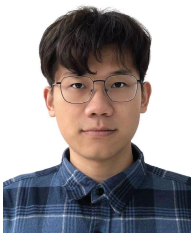
[43] Aghakhani H, Gritti F, Mecca F, et al. When Malware Is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features[C]. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[44] Cheng B L, Ming J, Fu J M, et al. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[45] Cheng B, Ming J, Leal E, et al. Obfuscation-resilient executable payload extraction from packed malware[C]. *30th Usenix Security Sympoisum*, 2021.

刘昕仪 于 2021 年在武汉大学信息安全专业获得学士学位。现在武汉大学网络空间安全专业攻读硕士学位。研究领域为软件安全、恶意代码分析等。Email: xinyiliu@whu.edu.cn。

彭国军 于 2008 年在武汉大学信息专业获得博士学位。现任武汉大学教授。研究领域为网络与信息系统安全。Email: guojpeng@whu.edu.cn。



刘思德 于 2019 年在武汉大学信息安全专业获得学士学位。现在武汉大学网络空间安全专业攻读博士学位。研究领域为恶意代码检测与溯源、二进制逆向等。Email: sidelao@whu.edu.cn。

杨秀璋 于 2016 年在北京理工大学软件工程专业获得硕士学位。现在武汉大学网络空间安全专业攻读博士学位。研究领域为网络与信息系统安全。研究兴趣包括: 恶意代码检测、网络安全。Email: yangxiuzhang@whu.edu.cn。

傅建明 于 2000 年在武汉大学信息专业获得博士学位。现任武汉大学教授。研究领域为恶意代码分析, 软件漏洞挖掘与利用。Email: jmfu@whu.edu.cn。