

基于函数拆分对抗二进制代码相似性检测

魏 然, 舒 辉

(信息工程大学, 河南郑州 450001)

摘要: 二进制代码相似性检测能够识别软件中的同源函数, 进而定位关键模块。攻击者将其应用于逆向工程, 给软件安全带来了风险。代码混淆可以增加软件的复杂程度, 增强其抗检测能力。传统代码混淆主要在函数内部实现, 存在空间开销和运行时开销大的问题。为此, 提出了基于函数拆分的方法进行代码混淆, 进而对抗二进制代码相似性检测。该方法在程序的中间表示层实现。首先, 基于支配树将原函数的基本块集合划分为多个子集。其次, 通过“定义—使用”链识别子集间的数据依赖关系。接着, 引入控制变量以修复子集间的控制流。最后, 通过代码重写生成子函数。实验结果表明, 经过函数拆分混淆后的程序, 使二进制相似性检测模型的召回率降至 1.3% (函数池大小为 1 024), 代码膨胀率为原始模型的 2.06 倍。

关键词: 代码混淆; 二进制代码相似性检测; 函数拆分; 支配树

中图分类号: TP309.5

文献标识码: A

文章编号: 1671-0673(2026)01-0064-08

Function Splitting-Based Adversarial Attacks on Binary Code Similarity Detection

WEI Ran, SHU Hui

(Information Engineering University, Zhengzhou 450001, China)

Abstract: Binary code similarity detection can identify homologous functions in software, thereby locating critical modules. When attackers apply this technology to reverse engineering, posing significant risks to software security. Code obfuscation can increase software complexity and enhance its resistance to detection. Traditional code obfuscation techniques are primarily implemented within a single function, often resulting in substantial space and runtime overhead. To address this, a code obfuscation method based on function splitting to counteract binary code similarity detection is proposed in this paper. The proposed method is operated at the intermediate representation level of the program. First, the basic block set of the original function is partitioned into multiple subsets based on the dominator tree. Second, data dependencies between subsets are identified through def-use chains. Subsequently, control variables are introduced to repair the control flow among these subsets. Finally, sub-functions are generated via code rewriting. Experimental results demonstrate that after applying function-splitting obfuscation, the recall rate of binary similarity detection models drops to as low as 1.3% (with a function pool size of 1 024), while the code bloating ratio is 2.06 times that of the original model.

Key words: code obfuscation; binary code similarity detection; function splitting; dominator tree

随着信息技术的飞速发展, 软件已成为现代社会中不可或缺的基础设施, 广泛支撑着从日常生活到关键行业的各个领域。二进制代码作为软

件的主要表现形式, 在发布后即成为公开的分析目标。攻击者能够借助逆向工程等手段对其进行分析, 对软件安全构成了严峻挑战。

二进制代码相似性检测旨在量化不同二进制代码之间的相似程度,在漏洞挖掘^[1-2]、恶意代码分析^[3-5]、代码剽窃检测^[6-7]、补丁生成与分析^[8-9]等多个领域具有重要应用价值。近年来,随着人工智能技术的不断进步,基于深度学习的二进制代码相似性检测模型在识别精度方面取得了显著提升。例如,基于Transformer架构的jTrans模型在跨优化级别的函数匹配任务中,召回率达到了94.9%。然而,该技术亦是一把“双刃剑”:攻击者同样可以借助相似性检测方法,识别加密、权限验证等关键函数,定位相关模块,从而实施软件破解。

代码混淆技术可以增加代码的理解难度,被广泛用于增强软件的抗逆向分析能力。常见的混淆手段包括指令替换^[10]、控制流平坦化^[11]及虚假控制流插入^[12]等。但传统混淆方法通常向函数中插入冗余代码,以改变其指令序列或控制流结构,这会带来代码体积膨胀和执行效率下降等问题^[13]。因此,为有效防御二进制代码相似性检测,亟需一种轻量化的混淆方案。本文提出一种基于函数拆分的混淆方法,将原始函数的代码逻辑拆分至多个子函数中,并通过函数调用维持原有功能,从而实现对抗二进制代码相似性检测的目标。

1 基于支配树划分的函数拆分方法

本文研究的函数拆分方法在程序的中间表示

层实现。程序的二进制代码通常由源码经过编译生成。编译过程中,编译器会生成程序的中间表示,比如,Clang的中间表示是LLVM IR(Low Level Virtual Machine Intermediate Representation)^[14]。本文基于LLVM框架,在中间表示层实现函数拆分。LLVM IR是一种采用静态单赋值(Static Single Assignment, SSA)形式的中间语言。整个编译流程中,前端借助Clang将源代码编译为LLVM IR;中端则在此IR上实施函数拆分操作;后端负责将处理后的IR生成目标机器代码。LLVM前端对C、C++等多种语言的支持,以及后端对不同处理器架构的广泛适配,使得本文方法具备良好的跨语言与跨架构应用潜力。

在程序的中间表示层,函数由基本块构成。基本块是具有单一入口和出口的指令序列,其最后一条指令决定了控制流走向,通常是分支、跳转或返回指令。本文提出的函数拆分方法,是对原函数的基本块集合进行重新划分,并将划分后的子集分别重构为独立的子函数。整个流程可分为5个主要步骤,如图1所示:首先,对原函数的基本块进行规范化处理;其次,基于支配树分析筛选合适的拆分点,将基本块集合划分为两个子集;接着,分析这两个子集间的数据依赖关系;然后,修复因划分而被割裂的控制流;最后,执行代码重写,生成子函数的代码,并创建可供后端编译的中间表示文件。

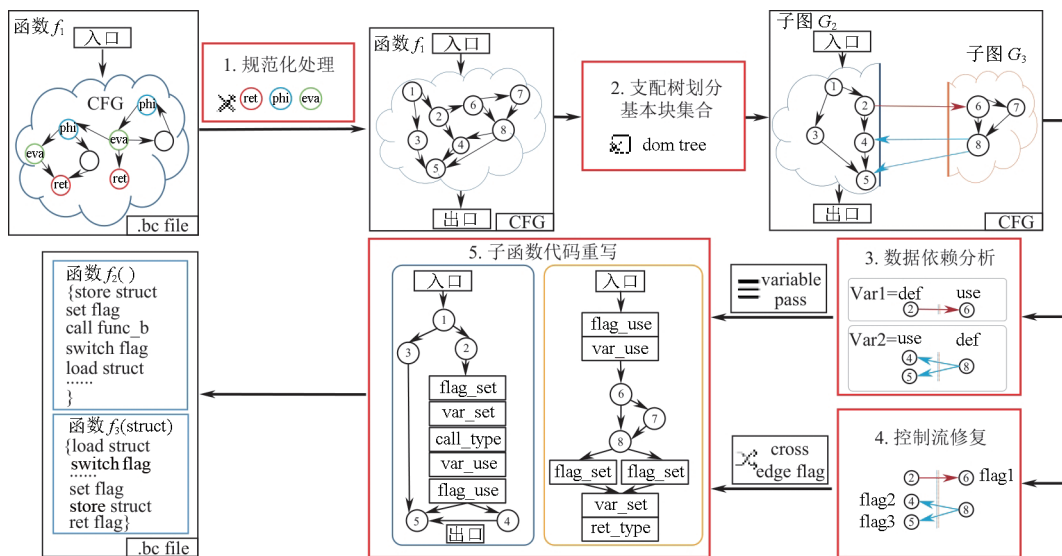


图1 函数拆分的技术路线

1.1 基本块集合的规范化处理

本文中,函数建模为基本块的集合,基本块是函数拆分操作的最小单元。函数内基本块之间常

存在紧密的耦合关系,这会限制拆分的灵活性。比如,基本块中若存在Phi指令,Phi指令定义变量的取值依赖于其前驱基本块,如果将存在耦合关

系的基本块分别拆分到两个子函数中,将造成程序错误。与此同时,若基本块中包含返回指令或者特殊变量,也会影响拆分过程。为此,本文方法首先对基本块集合进行以下规范化处理。

1)合并包含返回指令的基本块(本文中称这种基本块为返回基本块)。返回指令会使控制流从当前函数跳转至调用函数,直接将返回基本块拆分至子函数,会导致子函数提前退出。函数中可能存在多个返回基本块,需要对其进行合并处理。为此,本文在函数中新建一个返回基本块,并引入一个返回值变量,令各返回基本块对该变量赋值后跳转至新增的返回基本块。在新增返回基本块中加载返回值变量,而后最终返回。

2)消除Phi指令。如前所述,Phi指令是SSA形式中用于合并来自不同控制流路径的值的特殊指令,包含Phi指令的基本块与其前驱基本块,不能拆分到两个子函数中。本文采用为Phi指令的目标变量分配栈上内存的方法消除Phi指令。各前驱基本块在跳转前将对应值写入该内存位置,而Phi指令所在的基本块则通过从内存加载该值来替代原Phi操作。

3)处理逃逸变量。本文中逃逸变量是指在一个基本块中定义(排除入口基本块),但在另一个基本块中使用的变量。从函数入口到达使用逃逸变量的基本块,必然要经过定义该变量的基本块。但是函数拆分过程中可能会破坏这种关系。为此,本文将逃逸变量的定义转移到函数的入口基本块中,将原定义该变量的指令替换为对该内存的写入,将原使用该变量的指令替换为对内存的加载。

1.2 基于支配树划分基本块集合

函数拆分过程中,需要将原函数的基本块集合拆分为多个子集,每一个子集重构为一个子函数。理论上,拆分后的子函数越多,混淆效果越强,但由此引入的函数调用、上下文保存及数据流修复开销也越大。出于对程序性能和混淆效果之间平衡的考虑,本文探讨将原函数基本块集合拆分为两个子集的情况。将原函数记为 f_1 ,拆分后的子函数记为 f_2 和 f_3 ,原函数的基本块集合记为 B_1 ,两个子集分别记为 B_2 和 B_3 。

在函数的控制流图中,基本块集合 B_2 和 B_3 分别对应着一个子图,子图之间存在多条有向边。函数拆分后,每一个子图独立为一个子函数的控制流图。因此,子图之间的边将被割断,其对应的

控制流将被破坏,进而数据流也会受影响。修复控制流和数据流需要增加额外的代码,带来空间和运行时间方面的开销。因此,子图间边的数量越少,越利于降低混淆后程序的性能开销。

为了减少子图之间边的数量,本文基于支配树对函数的基本块集合进行划分。函数的控制流图是有向图,记为 $G = (B, E, R_{Entry}, R_{Exit})$,其中: B 为基本块集合; $E \subseteq B \times B$ 为控制流边集合; $R_{Entry} \in B$ 为入口块; $R_{Exit} \subseteq B$ 为出口块。

对于任意两个基本块 b_i 与 b_j ,若从 R_{Entry} 到 b_j 的每条路径均经过 b_i ,则称 b_i 支配 b_j ,记作 $b_i \text{ dom } b_j$ 。

若基本块 b_i 直接支配基本块 b_j ,需满足以下条件:首先, $b_i \text{ dom } b_j$;其次,不存在基本块 b_k ,使得 $b_i \text{ dom } b_k$ 且 $b_k \text{ dom } b_j$ 。基于直接支配关系,可以将控制流图构建为一棵支配树,记为 $T_{\text{dom}} = (B, E_{\text{dom}})$,其中边 $(b_i, b_j) \in E_{\text{dom}}$ 当且仅当 b_i 是 b_j 的直接支配结点。如图2所示,图2(b)是图2(a)所示控制流图对应的支配树。

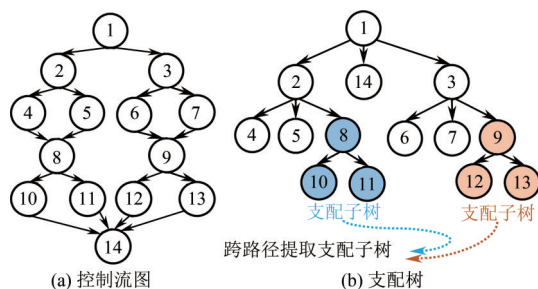


图2 基于支配树的划分示意图

从支配树中选择一颗子树,子树中的节点组成一个基本块集合,作为 B_3 ,支配树的其他节点构成基本块集合,作为 B_2 。从函数的入口基本块到达子树中的每一个节点都必然经过子树的根节点,所以控制流从 B_2 对应的子图转移到 B_3 对应的子图只有一个入口节点,即支配子树的根。因此,基于支配树划分基本块集合,有利于降低混淆后程序的性能开销。

然而,结合程序代码分析支配树的构建过程,会发现支配子树节点间存在比较强的逻辑关系。比如,if-else判断语句的每一个分支都可以构成一棵支配子树。通常,if-else的各分支会有相对独立的执行逻辑。基于一颗支配子树划分原函数的基本块集合,生成的子函数会保留原函数的较多局部特征。为应对这一问题,本文中将支配树划分为多棵子树,根据子树根节点在程序中的逻辑地址交错选择支配子树,以此构造集合 B_2 和 B_3 。子

函数由多棵支配子树的节点重构而成,从而弱化其体现出来的局部特征。

通过以上分析,本文设计了一种基于支配树划分基本块集合的算法,输入包括原函数 f_1 的基本块集合 B_1 、入口块 R_{Entry} 和出口块 R_{Exit} ,输出为基本块集合 B_2 和 B_3 。算法执行如下步骤。

步骤1:构建支配树。在LLVM IR层面,利用 f_1 的控制流图 G ,基于Lengauer-Tarjan算法^[15]构建支配树 T_{dom} 。

步骤2:划分支子树。从根节点先序遍历支配树,统计以当前节点为根的子树中的节点数量。若子树包含的基本块数量达到预设阈值,则将其

标记为候选支配子树。阈值采用自适应策略,根据 B_1 的基本块数量动态调整。

步骤3:在LLVM IR中按照地址的顺序管理函数的基本块,令每棵候选支配子树所包含基本块中地址最小值作为该候选支配子树的编号,随后按照编号大小顺序排列每棵支配子树。

步骤4:将排序后的支配子树序列按奇偶位置交错划分,奇数位置子树划入 B_2 ,偶数位置子树划入 B_3 ,如图3所示。该策略具有以下优势:相邻子树被划分至不同集合,破坏原始控制流的空间局部性;形成非连续的代码切片,对抗基于指令序列匹配的相似性检测;子树内部结构完整,有助于控制性能开销。

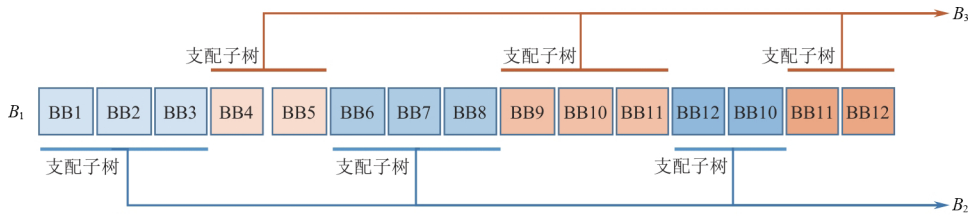


图3 基于支配树的交叉划分示意图

1.3 基本块集合同的数据依赖分析

在将原函数 f_1 的基本块集合 B_1 划分为 B_2 和 B_3 后,原控制流图 G_1 被划分为两个子图 G_2 和 G_3 。如图4所示,连接 G_2 与 G_3 的交叉边 e_{cross} 表示在 f_2 与 f_3 之间的数据流路径,该边在函数拆分后将断开。为保证程序功能正确,需要通过函数调用并合理设置参数来修复被破坏的数据流。

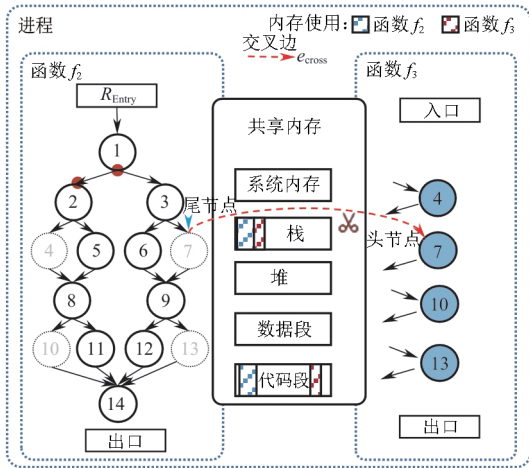


图4 子函数 f_2 和 f_3 的程序状态示意图

程序状态定义为程序点上所有变量到其当前取值的映射^[16]。如图4所示,交叉边 $e_{\text{cross}} = (u, v)$ 。 p_{tail} 表示 u 执行后的程序点, p_{head} 表示 v 执行前的程序点。变量分布于不同内存区域:代码段、数据段、堆、栈和全局内存。由于 f_2 与 f_3 在同一进程内

执行,除栈上局部变量外,其余内存区域均为共享。因此, p_{head} 与 p_{tail} 之间的状态差异仅来源于栈上局部变量,精确识别此类差异是确保语义等价性的关键。

在LLVM IR中,每个变量仅被赋值一次,这使得变量的定义和使用关系清晰,便于进行数据分析。假设函数入口基本块位于 B_2 ,则局部变量在 B_2 和 B_3 中的使用情况可分为多种类型。针对每种类型,表1总结了变量传递策略。

表1 基本块集合同变量传递策略

定义位置	在 B_2 中使用	在 B_3 中使用	传递策略
B_2	是	是	需跨函数传递
B_2	否	是	在 B_3 中重新定义
B_2	是	否	无需传递
B_2	否	否	无需传递

数据依赖分析的具体流程可概括为:首先,识别两个基本块集合 B_2 与 B_3 中定义和使用的变量,建立对应的定义集与使用集。本文中, $\text{def}(\cdot)$ 返回给定基本块集合中所有被定义的变量, $\text{use}(\cdot)$ 返回给定基本块集合中所有被使用的变量。随后,通过集合运算精确计算出需跨函数传递的变量,即 $V_{f_2 \rightarrow f_3} = \text{def}(B_2) \cap \text{use}(B_3)$ 与 $V_{f_3 \rightarrow f_2} = \text{def}(B_3) \cap \text{use}(B_2)$ 。

1.4 基本块集间控制流修复

原函数 f_1 的基本块集合 B_1 被划分为 B_2 和 B_3 。在控制流图中,边可分为内部边和交叉边。内部边是位于同一子图内的基本块之间的边,在拆分后仍保留在子函数内部。交叉边是连接不同子图的基本块的边,这类边在函数拆分后将断开,其表示的控制流需要进行修复。

为便于描述控制流修复过程,本文定义入口节点和出口节点。入口节点指交叉边的头节点,出口节点则为交叉边的尾节点。如图5(a)所示,一个子图可能包含多个入口节点。但是,通常情况下函数只能有一个入口基本块。针对这一问题,本文引入分支选择机制。如图5(b)所示,首先为每个入口节点分配唯一标识;然后,引入控制变量 V_{flag} ,在出口节点设置控制变量 V_{flag} ,控制变量的值由出口节点对应入口节点的标识决定;最后,在子函数入口引入分支选择基本块,通过解析控制变量 V_{flag} 的值,引导控制流指向正确的入口节点。该机制实现了多入口节点、多出口节点情况下的控制流修复。

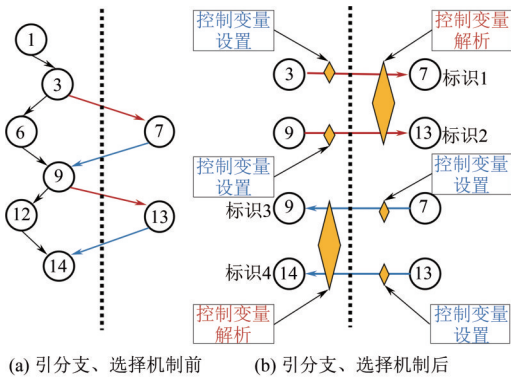


图5 控制流修复机制示意图

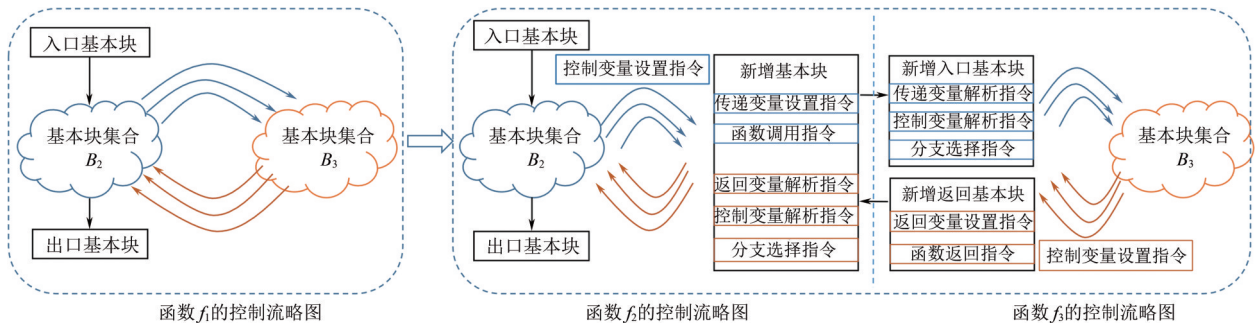


图6 子函数结构示意图

1.5 子函数的代码重写

在前述内容中,数据流和控制流修复的过程主要进行了程序分析,在程序分析的基础上对代码进行重写才能获得混淆后的二进制代码。代码重写过程中,首先设计子函数的结构,而后在其中插入具体指令,最终生成中间表示文件,作为编译器后端的输入。

1)子函数结构设计。原函数 f_1 拆分为 f_2 和 f_3 , f_2 通过调用 f_3 共同实现 f_1 的功能。如图6所示, f_1 的基本块集合 B_1 划分为 B_2 和 B_3 。 f_1 的基本块集合 B_1 规范化处理后,存在唯一入口基本块和返回基本块,将其划分到基本块集合 B_2 中。为了保证程序正确, f_2 中新增一个基本块,用于调用函数 f_3 ,并根据函数 f_3 的返回值选择执行路径的分支。具体包含调用函数 f_3 的指令、 $V_{f_2 \rightarrow f_3}$ 变量设置指令,以及返回值解析指令。 f_3 新增入口基本块和返回基本块。入口基本块用于解析 f_2 传来的变量 $V_{f_2 \rightarrow f_3}$ 和控制变量,并选择正确的控制流分支。返回基本块用于设置返回变量,而后返回至调用函数。

2)具体指令生成。在控制流图中,基本块集合 B_2 和 B_3 对应的子图之间存在交叉边,在函数拆分后交叉边断开,但可以通过指令维系其功能。每一条交叉边需要以下指令:控制变量赋值、传递参数赋值、函数调用、传递参数解析、控制变量解析以及执行分支选择指令。具体而言,控制变量赋值指令在交叉边的尾节点设置。由于交叉边存在多条,传递变量设置指令和函数调用指令统一在 f_2 的新增块中实现。传递参数解析、控制变量解析以及分发指令在 f_3 的入口基本块中实现。同理,可以生成 f_3 返回 f_2 的指令,不同点是分支选择指令需要置于 f_2 的新增基本块中。

2 实验评估

实验在一台配备 Intel Xeon Gold 6330 处理器、

128 GB 内存及 NVIDIA RTX 3090 GPU 的工作站上进行,操作系统为 Ubuntu 22.04 LTS。本文围绕函数拆分方法开展了以下实验评估。

2.1 函数拆分方法正确性测试

为验证函数拆分方法的正确性,本文选用开源软件 coreutils 工具集作为测试用例,对其包含的 7 998 个函数进行拆分混淆。实验中,使用 Clang 编译器在不同优化级别(O0~O3)下分别编译原始及混淆后代码,生成对应的可执行文件。测试结果表明,混淆后程序在所有优化级别下均能正确运行,输出与原始程序一致,验证了函数拆分方法的正确性。

1)代码体积影响分析。为评估函数拆分对代码体积的影响,引入代码膨胀率作为评价指标,记为 R_{size} 。通过 objdump 工具提取各函数的大小信息,分别统计原函数 f_1 与拆分后子函数 f_2 和 f_3 的大小。代码膨胀率计算方法如式(1)所示:

$$R_{size} = \frac{S_{f_2} + S_{f_3}}{S_{f_1}} \quad (1)$$

式中: S_{f_1} 代表原函数的大小; S_{f_2} 和 S_{f_3} 代表子函数的大小。

对全部函数的代码膨胀率进行统计分析,结果如图7所示。实验测得在 O0 优化级别下平均代码膨胀率为 2.06,而在 O1~O3 级别下约为 3.52。代码膨胀率与原函数规模呈负相关:原函数越大,代码膨胀率越低。与 O1、O2、O3 编译选项相比,由于 O0 编译选项生成的原函数体积较大,而混淆引入的控制流与数据流修复代码在不同优化级别下体积基本恒定,因此 O0 级别的膨胀率小于其他优化级别。

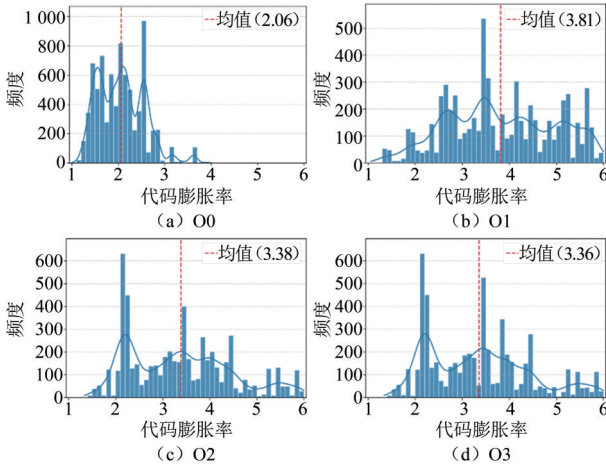


图7 代码膨胀率分布

2)程序性能影响分析。为进一步评估函数拆分对程序性能的影响,从 coreutils 工具集中选择了 15 个程序作为测试用例。将程序混淆后执行时间与混淆前执行时间的比值作为评价指标。利用 time 工具测量程序的执行时间,每个测试用例执行 1 000 次。实验结果显示,测试用例的平均执行时间比值为 1.034,整体分布如表2所示。

表2 函数拆分对执行时间的影响(O0优化级别)

测试用例	执行时间比值	测试用例	执行时间比值
b2sum	1.02	tty	0.99
csplit	0.99	base32	1.14
factor	1.02	cut	1.1
ls	0.89	false	1.24
paste	0.81	md5sum	1.14
seq	1.08	pathchk	1.1
stty	1.11	sha1sum	0.93
tty	0.99	sum	0.95
stty	1.11		

混淆后程序执行时间增加,一方面由于函数调用次数上升导致出栈入栈操作增加,出入栈操作需要访问内存,而执行访问内存指令通常比其他指令需要更长时间。另一方面,子函数间进行函数调用时,调用者需要设置参数,被调用者需要读取参数,设置和读取参数也需要访问内存。与之相反,实验中少量测试程序(csplitt等)在混淆后执行时间比混淆前有所减少,是由于混淆后的程序运行过程中需要调用 libc 等动态库的函数。动态库并没有混淆,如果对库函数的调用较多,则会降低混淆函数对程序整体执行时间的影响。

综上所述,在应用函数拆分混淆后,测试用例的功能均被保留,程序正确运行。混淆后程序的代码膨胀率为 2.06,执行时间比值为 1.034。

2.2 对抗二进制代码相似性检测的有效性实验

为评估对抗二进制代码相似性检测模型的效果,本文选取 6 类基于人工智能方法的模型作为对抗目标,具体包括:jTrans^[17]、SAFE^[18]、Asm2Vec^[19]、Gemini^[20]、GraphEmb^[21]、Genius^[22]。前 3 个模型基于指令序列进行相似性检测,后 3 个模型基于控制流图进行相似性检测。

实验以 coreutils 作为基准测试集。coreutils 软件包中的代码包含于所选模型的训练集中,可确保模型对测试函数具备基本识别能力,从而将识别准确性的变化归因于函数拆分带来的混淆。

1)对抗基于 AI 的检测模型的实验设计。实验采用平均倒数排名(Mean Reciprocal Rank, MRR)和召回率(Recall@1)作为评价指标,分别表示为 R_{MRR} 和 $R_{Recall@1}$ 。构建原函数池 $F = \{f_1, f_2, \dots, f_n\}$,拆分后函数池 $H = \{h_1, h'_1, h_2, h'_2, \dots, h_n, h'_n\}$,记 R_F 表示集合 F 中元素的个数。设查询函数为 f_i ,则对应的拆分函数为 h_i 和 h'_i 。在本实验中,将查询函数 f_i 与 H 中每一个函数进行相似性分析,并按照相似性分数进行排名。 $Rank(h_i)$ 表示函数 h_i 在查询结果中

的排名。定义识别函数为

$$\mathbb{I}(f_i) = \begin{cases} 1, & \text{Rank}(h_i) = 1 \text{ 或 } \text{Rank}(h'_i) = 1; \\ 0, & \text{其他.} \end{cases} \quad (2)$$

在此基础上计算 $R_{\text{Recall}@1}$ 、 R_{MRR} , 计算公式为

$$R_{\text{Recall}@1} = \frac{1}{R_F} \sum_{f_i \in F} \mathbb{I}(f_i) \quad (3)$$

$$R_{\text{MRR}} = \frac{1}{R_F} \max \left(\frac{1}{\text{Rank}(h_i)}, \frac{1}{\text{Rank}(h'_i)} \right) \quad (4)$$

如表3所示,在较小搜索池(函数池大小为32)

中,函数拆分已对各类模型产生明显干扰,平均 R_{MRR} 为0.387, $R_{\text{Recall}@1}$ 为0.148。 R_{MRR} 较高而 $R_{\text{Recall}@1}$ 较低,表明尽管正样本未被准确检索至首位,其排名仍相对靠前,这与小规模池中正负样本比例较高(约1:15)有关。

当池规模扩大至1 024时(表4),所有模型的识别性能进一步下降, R_{MRR} 均值降至0.072, $R_{\text{Recall}@1}$ 均值降至0.013,充分验证了函数拆分在真实场景下的有效对抗能力。

表3 跨优化级别检测性能对比(函数池大小为32)

对抗模型	O0vsO0split		O1vsO1split		O2vsO2split		O3vsO3split		O0vsO3split		平均值	
	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$
jTrans	0.826	0.343	0.563	0.185	0.569	0.183	0.566	0.179	0.502	0.168	0.605	0.212
SAFE	0.734	0.277	0.448	0.127	0.451	0.127	0.468	0.136	0.396	0.110	0.499	0.155
Asm2Vec	0.663	0.210	0.385	0.155	0.402	0.163	0.378	0.142	0.325	0.130	0.431	0.160
GraphEmb	0.312	0.142	0.285	0.128	0.294	0.133	0.276	0.121	0.243	0.115	0.282	0.128
Gemini	0.287	0.135	0.261	0.122	0.273	0.127	0.254	0.116	0.228	0.105	0.261	0.121
Genius	0.265	0.127	0.239	0.115	0.251	0.120	0.237	0.108	0.214	0.098	0.241	0.114

表4 跨优化级别检测性能对比(函数池大小为1 024)

对抗模型	O0vsO0split		O1vsO1split		O2vsO2split		O3vsO3split		O0vsO3split		平均值	
	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$	R_{MRR}	$R_{\text{Recall}@1}$
jTrans	0.235	0.066	0.095	0.018	0.088	0.016	0.091	0.018	0.090	0.018	0.120	0.027
SAFE	0.174	0.045	0.053	0.007	0.053	0.007	0.055	0.008	0.043	0.006	0.076	0.015
Asm2Vec	0.128	0.032	0.042	0.005	0.038	0.004	0.036	0.004	0.031	0.003	0.055	0.010
GraphEmb	0.155	0.039	0.048	0.007	0.047	0.005	0.049	0.004	0.040	0.005	0.068	0.012
Gemini	0.142	0.025	0.047	0.005	0.041	0.006	0.039	0.003	0.032	0.003	0.060	0.008
Genius	0.128	0.021	0.041	0.003	0.035	0.004	0.033	0.003	0.028	0.003	0.053	0.007

2)对抗传统检测模型的实验分析。针对基于传统图匹配方法的BinDiff工具,本文评估了函数拆分对其相似性分析的影响。BinDiff通过基本块哈希、调用图匹配与交叉引用分析计算相似性,输出结果为文件相似度与置信度。实验测试了5种对比情况:O0vsO0split、O1vsO1split、O2vsO2split、O3vsO3split以及跨优化级别的O0vsO3split,分别统计其平均相似度与置信度,结果如图8所示。

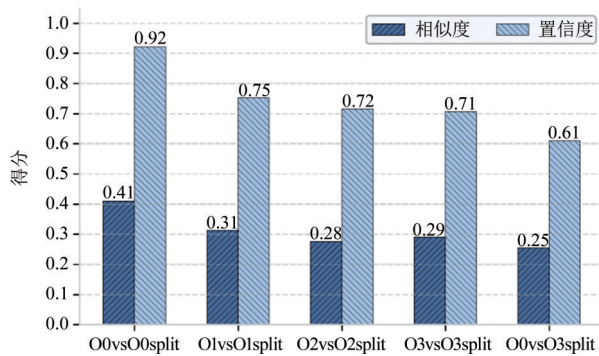


图8 对抗BinDiff检测的相似度与置信度

实验数据显示,在O0同级别混淆下,平均相似度由基准值约1.0降至0.41,降幅达59%。在O1、O2和O3优化级别下,相似度进一步分别下降至0.31、0.28和0.29。在跨优化级别(O0vsO3split)场景中,相似度降至0.25,置信度亦由O0同级下的0.92下降至0.61,降幅为33.7%。结果表明,函数拆分在不同优化级别及跨级别设置下均能有效干扰BinDiff的识别能力,验证了该方法对传统结构匹配型检测模型同样具备良好的对抗效果。

3 结束语

为了对抗二进制代码相似性检测,本文提出了一种基于支配树划分的函数拆分方法,该方法可以对程序进行混淆。传统混淆方法主要是在程序中人为添加冗余的指令,实现对控制流或者指令序列的扰动。本方法则瞄准函数的基本块,通过在程序的中间表示层,将函数的基本块在多个子函数中进行重构,来实现改变函数控制流图结

构和指令序列的目标。实验结果显示,该方法降低了混淆前后函数的相似度,实现了对抗二进制代码相似性检测的目标。未来工作将进一步探索函数融合方法,通过合并来自不同函数的拆分片段以构造更复杂度的函数,进一步提高对抗效果。

参考文献:

- [1] CHANDRAMOHAN M, XUE Y X, XU Z Z, et al. BinGo: cross-architecture cross-OS binary search [C] // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, USA: ACM, 2016:678-689.
- [2] DAVID Y, PARTUSH N, YAHAV E. Statistical similarity of binaries[C]//Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2016:266-280.
- [3] HU X, CHIUEH T C, SHIN K G. Large-scale malware indexing using function-call graphs[C]//Proceedings of the 16th ACM Conference on Computer and Communications Security. New York, USA: ACM, 2009:611-620.
- [4] POLYCHRONAKIS M, ANAGNOSTAKIS K, MARKATOS E. Detection of intrusions and malware & vulnerability assessment [C] // Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Cham, Switzerland: Springer, 2006:54-73.
- [5] WANG Z, PIERCE K, MCFARLING S. Bmat: a binary matching tool for stale profile propagation [J]. The Journal of Instruction-Level Parallelism, 2000, 2: 1-20.
- [6] ESCHWEILER S, YAKDAN K, GERHARDS-PADILLA E. DiscovRE: efficient cross-architecture identification of bugs in binary code [C] // Proceedings of the 2016 Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2016:58-79.
- [7] FENG Q, WANG M H, ZHANG M, et al. Extracting conditional formulas for cross-platform bug search [C] // Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. New York, USA: ACM, 2017:346-359.
- [8] KARGÉN U, SHAHMEHRI N. Towards robust instruction-level trace alignment of binary code [C] // Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. Piscataway, USA: IEEE, 2017:342-352.
- [9] HUANG H, YOUSSEF A M, DEBBABI M. BinSequence: fast, accurate and scalable binary code reuse detection [C] // Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. New York, USA: ACM, 2017:155-166.
- [10] 李明亮, 庞建民, 岳峰. 基于地址重用的二进制翻译本地代码替换[J]. 信息工程大学学报, 2022, 23(1): 38-44.
- [11] WANG C X, HILL J, KNIGHT J, et al. Software tamper resistance: obstructing static analysis of programs [R]. Charlottesville, USA: University of Virginia, 2000: 1-18.
- [12] JUNOD P, RINALDINI J, WEHRLI J, et al. Obfuscator-LLVM: software protection for the masses [C] // Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection. Piscataway, USA: IEEE, 2015: 3-9.
- [13] HAQ I U, CABALLERO J. A survey of binary code similarity [J]. ACM Computing Surveys, 2021, 54(3): 1-38.
- [14] LATTNER C, ADVE V. LLVM: a compilation framework for lifelong program analysis & transformation [C] // Proceedings of the International Symposium on Code Generation and Optimization, 2004. Piscataway, USA: IEEE, 2004: 75-86.
- [15] GRAHAM S L, LENGAUER T, TARJAN R E. A fast algorithm for finding dominators in a flowgraph [J]. ACM Transactions on Programming Languages and Systems, 1979, 1(1): 121-141.
- [16] AHO A V, LAM M S, SETHI R, et al. Compilers [M]. Boston, USA: Addison Wesley, 2006: 597.
- [17] WANG H, QU W J, KATZ G, et al. jTrans: jump-aware transformer for binary code similarity detection [C] // Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. New York, USA: ACM, 2022: 1-13.
- [18] MASSARELLI L, DI LUNA G A, PETRONI F, et al. SAFE: self-attentive function embeddings for binary similarity [C] // International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Cham, Switzerland: Springer, 2019: 309-329.
- [19] 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] // Proceedings of the 2019 IEEE Symposium on Security and Privacy. Piscataway, USA: IEEE, 2019: 472-489.
- [20] XU X J, LIU C, FENG Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection [C] // Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2017: 363-376.
- [21] MASSARELLI L, DI LUNA G A, PETRONI F, et al. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis [C] // Proceedings of the 2019 Workshop on Binary Analysis Research. CA, USA: Internet Society, 2019. DOI: 10.14722/bar.2019.23020.
- [22] FENG Q, ZHOU R D, XU C C, et al. Scalable graph-based bug search for firmware images [C] // Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2016: 480-491.

(编辑:邵 华)