二进制比对技术：场景、方法与挑战

胡梦莹[1,2], 王笑克[1,2], 赵 磊[1,2]

[1]武汉大学国家网络安全学院 武汉 中国 430072

[2]武汉大学空天信息安全与可信计算教育部重点实验室 武汉 中国 430072

摘要 二进制比对技术通过比较两段二进制代码片段的特征来识别它们之间的相似度和差异性, 其在安全领域应用广泛, 包括漏洞搜索、补丁分析、恶意软件检测等, 在各个应用场景下也伴随着不同的技术挑战。尽管已有研究对二进制比对技术进行了调研分类, 然而现有研究无法准确描述二进制比对技术的特点、不同挑战对二进制代码特征的具体影响以及二进制比对技术的比较基准。为弥补上述缺失, 对二进制比对工作进行了大规模的调研, 发现目前以应用场景对二进制比对技术进行分类的方式不足以精确描述二进制比对技术的特点, 并且大部分工作没有明确其应用场景, 因此提出了二进制比对的通用描述模型, 该模型由二进制比对的比较对象、预期目标、技术挑战和方法特征 4 个维度构成, 通过该模型可以更精确描述二进制比对技术。进而, 论述了各技术挑战对二进制代码特征的影响, 具体包括编译配置、语义修改以及代码混淆对二进制代码的句法特征、结构特征和语义特征的影响。与此同时, 提出了一种二进制比对技术的比较基准并通过实验进行了验证, 实验结果表明, 在选择比较基准时, 应考虑不同方法的比较对象、预期目标、解决的挑战是否一致。当比较对象、预期目标、解决的挑战不一致时, 对它们之间的对比没有意义; 当比较对象、预期目标、解决的挑战一致时, 对它们之间的对比更有意义。最后, 结合研究发现给出了下一步的建议研究方向。

关键词 二进制比对; 软件安全; 比较实验

中图法分类号 TP311 DOI 号 10.19363/J.cnki.cn10-1380/tn.2025.03.04

Binary Comparison Techniques: Applications, Approaches, and Challenges

HU Mengying[1,2], WANG Xiaoke[1,2], ZHAO Lei[1,2]

[1] School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

[2] Key Laboratory of Aerospace Information Security and Trusted Computing Ministry of Education, Wuhan University, Wuhan 430072, China

Abstract Binary comparison technology identifies the similarities and differences between two binary code fragments by comparing their features. It is widely used in the field of security, including bug search, patch analysis and malware detection, and it has different technical challenges in various application scenarios. Although studies have been conducted to investigate and classify binary comparison techniques, they are unable to accurately describe the characteristics of binary comparison techniques, the specific impact of different challenges on binary code features, and the benchmark of binary comparison techniques. In order to make up for the above shortcomings, a large-scale investigation was conducted on binary comparison work. It was found that the current method of classifying binary comparison technology based on application scenarios is not sufficient to accurately describe the characteristics of binary comparison technology, and most of the work has not clearly declared its application scenarios. Therefore, a generic descriptive model for binary comparison technology was proposed, which consists of the comparison object, expected target, technical challenges and the characteristics of binary comparison technology. This model can more accurately describe binary comparison technology. Furthermore, the impact of various technical challenges on the characteristics of binary code was discussed, including the impact of compilation configuration, semantic modification, and code confusion on the syntactic, structural, and semantic features of binary code. At the same time, a benchmark for binary comparison technology was proposed and verified through experiments. The experimental results showed that when selecting a comparison benchmark, it is necessary to consider whether the comparison objects, expected goals, and challenges solved by different methods are consistent. When the comparison objects, expected goals, and challenges to be solved are inconsistent, the comparison between them is meaningless; When the comparison objects, expected goals, and challenges solved are consistent, the comparison between them is more meaningful. Finally, based on the research findings, suggestions for further research directions were proposed.

Key words binary comparision; software security; comparative experiment

通讯作者： 赵磊, 博士, 教授, Email: leizhao@whu.edu.cn。

本课题得到国家自然科学基金(No. 62172305)资助。

收稿日期: 2023-05-14; 修改日期: 2023-08-07; 定稿日期: 2025-01-22

1 引言

考虑到应用场景, 二进制代码比较的主要目标一方面包含代码的相似性分析, 另一方面又包括了代码的差异性检测, 因此, 不同参考文献中对二进制代码的比较技术称谓不同, 例如, 中文参考文献[1]使用二进制文件比对技术表示二进制文件的相似性与差异性检测, 文献[2]使用二进制比对表示二进制文件的相似性检测, 并使用 Binary Comparision 作为二进制比对的翻译; 英文文献中, BinHunt[3]使用 Binary difference analysis 表示二进制代码差异性检测, BinSim[4]使用 Binary diffing 表示二进制代码差异性检测, 文献[5]中使用 Binary code similarity approaches 表示二进制代码相似性和差异性检测。

为了统一上述不同参考文献中的说法, 本文中统一使用二进制比对技术(Binary comparison)来表示二进制代码的相似性和差异性检测, 将其定义为: 给定两个或多个二进制代码片段, 二进制比对的目标是在整个程序、函数或者基本块粒度上来分析它们之间的相似性或差异性。

出于对软件知识版权的保护等原因, 软件的源代码往往是不可获得的, 软件发行商通常会发行由源代码编译得到的二进制程序, 因此二进制比对技术在许多实际应用场景中得到了广泛关注, 包括漏洞搜索[6-8]、补丁分析[9-11]、恶意软件检测[12-14]、恶意软件谱系[15]和软件盗版检测[16]等。

伴随着广泛的应用场景, 二进制比对技术也面临许多难点与挑战, 具体包括编译配置、语义修改、代码混淆等。其中, 编译配置指的是在程序编译过程中, 使用不同的架构、编译器和编译优化选项, 导致二进制代码的表现形式不同; 程序语义修改指的是在代码基本结构相似的情况下, 代码的实际功能发生了变化; 代码混淆指的是对代码经过了一系列特定的转换, 如增加冗余、变换编码乃至层层封装以增加识别实际代码的复杂度。

尽管目前已有相关综述[5,17]对二进制代码相似性检测领域进行了详细的总结, 介绍了该领域各方法的应用场景、方法特征、方法实现以及评估基准, 然而, 我们发现, 结合二进制比对技术的广泛应用场景和诸多技术挑战, 当前二进制比对领域仍存在一些问题需要解决:

首先, 二进制比对技术的分类不清晰。研究人员通常会从应用场景角度来区分二进制比对技术, 但是二进制比对技术在相同场景下所面临的挑战并不一定相同, 而在不同场景下也可能面临着相同的挑战, 并且多项相关工作甚至没有声明其应用场景。因此, 仅从应用场景来区分二进制比对技术是不准确的, 需结合二进制比对技术的各项属性进行综合考虑。

其次, 不同技术挑战与代码特征之间的关系不明确。现有方法通常利用程序切片、自然语言处理等技术来提取二进制代码特征, 然而并没有解释这些特征提取方法的必要性, 为评估所提取的特征是否能够有效应对不同技术挑战, 需进一步分析各项技术挑战对代码特征带来的影响及二进制比对技术在各项挑战下的鲁棒性。

最后, 不同二进制比对技术之间难以比较。如上所述, 以应用场景为维度来对各类二进制比对技术进行比较是不准确的, 且现有方法所提取的特征与不同挑战之间的影响关系也并不明确。在此情况下, 需针对二进制比对技术设计一套公平的比较基准, 以便后续研究进行对比实验, 并帮助用户在实际任务中选择合适的二进制比对技术。

针对上述研究发现, 本文对近 10 年二进制比对代表性技术进行了广泛调研与分析, 发现在不同的应用场景下, 二进制比对的比较对象和预期目标往往并不相同, 与此同时, 即使在相同应用场景下, 其比较对象和预期目标也不一定相同。并且, 比较对象的不同导致二进制比对的预期目标与面临的技术挑战不同, 二进制比对技术通过提取不同的二进制代码特征来解决各技术挑战。换言之, 比较对象在涵盖应用场景的基础上, 可以更加细粒度地体现二进制比对技术的目的。因此, 本文从二进制比对的比较对象出发, 结合二进制比对的预期目标、面临的挑战以及二进制比对技术提取的特征, 构建了一个二进制比对的通用描述模型。通过该模型, 我们明确了区分二进制比对技术的主要因素。其次, 我们分别讨论了不同挑战对二进制代码的语法特征、结构特征以及语义特征的具体影响。此外, 论述了二进制比对技术在基准技术比较方面的观点, 并设计实验验证, 我们发现, 当二进制比对技术的比较对象、预期目标以及解决的挑战都一致时, 对它们之间的比较更有意义。最后给出下一步的建议研究方向。

本文的主要贡献为以下 3 点:

(1) 提出了二进制比对的通用描述模型, 通过该模型可以区分不同的二进制比对技术。

(2) 论述了不同的技术挑战对二进制代码特征的具体影响。

(3) 提出了一种二进制比对技术的比较基准, 利用该比较基准可以更公平地对二进制比对方法进行比较。

本文第 2 节介绍二进制比对的相关概念、基本思路、应用场景。第 3 节阐述本文的研究问题、研究内容以及研究范围。第 4 节提出了二进制比对技术的通用描述模型。第 5 节讨论了不同技术挑战对二进制代码特征带来的影响。第 6 节提出二进制比对方法之间的比较基准并进行实验验证。第 7 节分析二进制比对面临的挑战以及可行的研究方向。第 8 节总结本文的内容。

2 二进制比对背景介绍

2.1 二进制比对相关概念

给定两个或多个二进制代码片段, 二进制比对的目标是在整个程序、函数或者基本块粒度上来分析它们之间的相似性或差异性。由于比较多个二进制代码片段可以转化为多次比较两个二进制代码片段, 因此本文用 P 和 P1 表示二进制比对的比较对象。

分析代码片段之间的相似性或差异性在方法上有相似之处, 例如, 二进制比对技术通常将比较对象表示为易于比较的特征向量, 通过计算向量的相似度得到相似性分数来实现代码相似性检测, 或是通过向量对齐凸显出比较对象之间的差异, 并对这些差异进行下一步分析。

2.2 二进制比对技术基本思路与方法

一般地, 二进制比对的基本流程可分为三个阶段, 分别是: 二进制代码预处理、特征提取和相似性/差异性检测。下面对这三个阶段进行详细解释。

二进制代码预处理的目的是统一和规范比较对象格式, 剔除与代码相似性无关的或影响较小的信息。二进制比对方法通常利用二进制分析工具(例如 IDA Pro)对二进制文件进行反汇编来获得汇编代码, 并对汇编代码进行预处理, 例如删除反汇编过程中产生的空格、注释以及多余的语句等信息。

特征提取的目的是在尽量不丢失代码原始信息的前提下, 从二进制代码中提取特征(包括语法、结构和语义特征), 并将其转化成易于处理和进行相似性比较的形式。考虑到二进制比对方法所面临的技术挑战, 通常需要根据实际情况选取能够抵抗技术挑战带来的影响的特征表现形式。

相似性/差异性检测的目的是根据所提取的代码特征, 计算两段代码的相似度或定位出两段代码的差异。其中, 相似性检测对应的应用场景是漏洞搜索、恶意软件检测、恶意软件谱系等, 二进制比对方法通过选择相似性度量算法, 如子图匹配、计算向量距离等来衡量两段代码的相似性; 差异性检测对应的应用场景是补丁分析, 重点关注两段代码的差异, 这些差异体现了补丁对程序做出的改动。

2.3 二进制比对的应用场景

二进制比对技术常见的应用场景包括:

补丁分析: 各大软件厂商在发布漏洞和相应的补丁时, 往往不会公布详细的补丁细节, 而二进制比对技术可用于识别补丁前后的软件版本之间的差异性, 从而为定位补丁位置、揭示补丁信息提供支持。

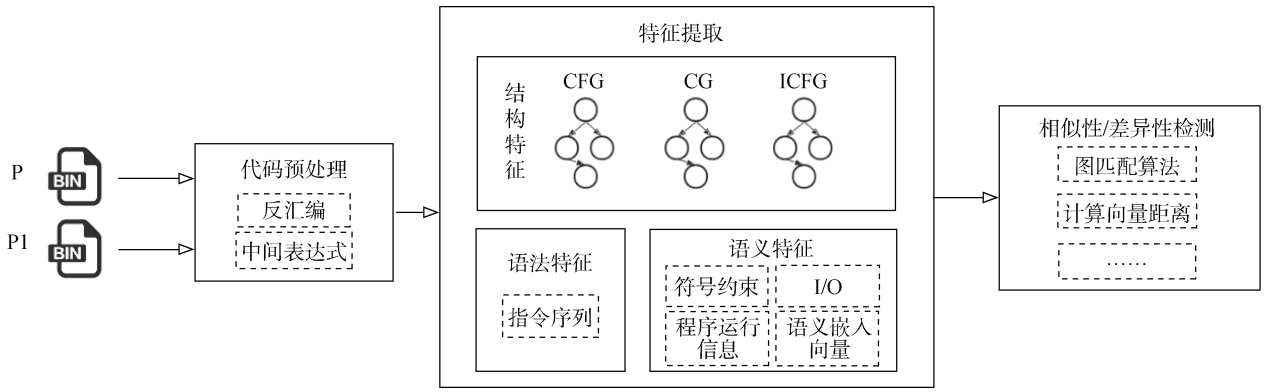

图 1 二进制比对基本流程

Figure 1 The basic process of binary comparison

漏洞搜索: 在进行已知软件漏洞挖掘时, 可以将包含某个漏洞的二进制函数或二进制代码段作为查询代码, 随后通过二进制比对技术来比较目标代码与漏洞代码的相似性, 如果某段代码与漏洞代码高度相似, 那么这段代码极有可能也包含相同的漏洞。

恶意软件检测: 由于恶意软件家族衍生出的恶意代码样本共享相似的指令序列及编码风格, 因此, 通过二进制比对技术来比较目标样本与已知的恶意代码样本之间的相似度, 可辅助判断该比较对象样本是否为已知恶意软件家族的变体。

恶意软件谱系: 恶意软件谱系是指原始家族与

其后代、版本或变种之间的家族演化关系。谱系可以帮助恶意软件分析师了解在一段时间内的恶意软件发展趋势, 并帮助分析人员决定首先分析哪个恶意软件。

软件盗版检测: 当软件的代码在未经过许可被剽窃使用时, 可利用二进制比对技术将其与目标软件进行比较, 来识别目标软件中是否存在代码剽窃的行为。

3 研究问题与研究内容

3.1 研究问题

本文重点关注的三个研究问题如下:

研究问题 1: 如何区分二进制比对方法? 对二进制比对方法的分类应该考虑哪些方面?

通常情况下, 现有研究通常根据应用场景来对二进制比对技术进行分类, 然而, 二进制比对领域的一部分相关工作是针对具体的应用场景提出的, 另一部分工作没有明确具体的应用场景, 旨在提出通用的二进制比对能力。即, 目前不能仅通过应用场景的不同来难以区分不同的二进制比对方法。那么, 是否存在其他因素可以区分不同的二进制比对方法?

研究问题 2: 二进制比对面临的主要挑战对其方法所提取特征的具体影响体现在哪些方面?

为了解决各技术挑战带来的问题, 诸多方法所设计的代码特征不同, 包括二进制代码的结构特征、语法特征、语义特征等。然而, 现有方法仅关注其方法的有效性, 并未解释所提取特征的必要性。例如, 通常情况下, 二进制比对方法假设编译配置、代码混淆会严重影响二进制代码的语法以及结构, 却并未说明这些挑战对二进制代码的语法以及结构特征带来的影响体现在哪些方面。

研究问题 3: 二进制比对方法之间的比较, 其比较基准的设定应该从哪些维度来考虑?

现有方法通常会选取应用场景相同的方法进行比较, 然而, 相同场景下二进制比对技术解决的挑战不一定相同, 例如, Gemini[18]和 Gitz[19]的应用场景都是漏洞搜索, 然而 Gemini 解决的技术挑战为跨架构和跨编译优化选项的二进制比对, Gitz 解决的技术挑战为跨架构和跨编译器的二进制比对, 导致它们之间的比较不够公平。此外, 一些二进制比对方法提出通用的二进制比对能力, 适用于多种场景, 这些通用方法与应用场景明确的方法之间的比较也很困难。那么, 应该以哪些维度为比较基准, 对二进制比对方法进行比较?

3.2 研究内容

围绕上述研究问题, 本文将从以下几个方面展开:

(1) 二进制比对技术的分类问题。通过对二进制比对相关文献的分析, 我们发现二进制比对的应用场景与其比较对象以及预期目标存在映射关系, 并且比较对象的不同决定了二进制比对面临的技术挑战不同, 为了解决各技术挑战, 二进制比对方法设计了不同的代码特征。因此, 我们从二进制比对的比较对象、预期目标、技术挑战和方法特征 4 个维度建立了二进制比对的通用描述模型。通过这 4 个维度便可以对二进制比对技术进行准确的分类。

(2) 各挑战对二进制代码特征的影响。在上述二进制比对的通用描述模型基础上, 为了进一步探究不同挑战对二进制代码特征的影响, 我们首先梳理总结了当前各二进制比对方法所提取的具体的语法特征(指令的操作码等)、结构特征(控制流图、函数调用图等)以及语义特征(API/系统调用等), 并分析了不同挑战(架构、编译器、编译优化选项、语义修改和代码混淆)对语法特征、语义特征、结构特征的影响体现在哪些方面, 最后分析不同挑战下二进制比对技术的鲁棒性。

(3) 二进制比对技术的比较基准。基于本文提出的二进制比对通用描述模型, 我们认为当前以应用场景为维度来进行方法之间的比较是不公平的, 在选择比较基准时, 应考虑不同方法的比较对象、预期目标、解决的挑战是否一致。为了验证这一观点, 我们设计了相关实验, 选取了比较对象、预期目标、解决的挑战不尽相同的 5 种二进制比对代表性方法, 并对这 5 种方法进行对比实验, 最终通过实验结果验证该观点的有效性。

3.3 研究范围

二进制比对是网络与信息安全领域以及软件工程领域的重要研究方向。本文的研究范围力图覆盖 2012—2022 年发表在国际重要会议及期刊的符合“二进制比对”的研究工作。

具体来说, 本文使用以下方式来进行相关文献的检索和筛选:

(1) 检索目标: 网络与信息安全领域和软件工程领域的重要国际会议(IEEE S&P, ACM CCS, USENIX Security, NDSS, ACSAC, ASIACCS, DIMVA, ICSE, FSE, ISSTA, ASE 等) 及期刊(IEEE TDSC, IEEE TSE 等)。

(2) 检索关键词: “Binary diffing”、“Binary code similarity analysis”等。

(3) 检索起止时间: 2012-2022 年之间的文献。

(4) 文献审查: 首先对检索出来的文献进行人工筛选, 去除不符合研究主题的文献; 然后检查选中

论文中的参考文献列表补充未检索到的文献。

经过文献筛选与汇总, 本文收集了针对二进制比对技术展开研究的 57 篇学术论文。表 1 列出了二进制比对的相关论文的发表情况, 在 CCF 划分的网络与信息安全领域的重点会议上有 18 篇论文, 在软件工程领域的重点会议上有 20 篇论文, 包括顶级会议 CCS 论文 3 篇, USENIX Security 论文 3 篇, NDSS 论文 3 篇, ASE 论文 6 篇, ISSTA 论文 3 篇, PLDI 论文 3 篇。

表 1 二进制比对相关文献列表

Tablel 1 The reference lists of binary comparison techniques

类型	出版物缩写及引用列表
国际会议	CCS[18,20,25], USENIX Security[4,33,39], NDSS[10,24,44], ACSAC[22,32,54], ASIACCS[27,41], DIMVA[28,46], ICISC [58], SecureComm[40], FSE[6,16], ASE[8,42,43,57,62,64], ICSE[9], ISSTA[7,53,55], PLDI[19,26,38], MSR[37], COMPSAC[65], SANER[11,66,68], 其他[23,29,30,31,36,45,48,51,59,60]
国际期刊	TDSC[12], TSE[47,49,52,63], 其他[21,35,50,67]

4 二进制比对的通用描述模型

本文通过对近 10 年二进制比对文献的调研发现, 57 篇文献中, 22 篇明确了它们的应用场景, 如表 2 所示。

表 2 二进制比对技术的应用场景

Table 2 The application of binary comparison techniques

应用场景	相关工作
补丁分析	SPAIN[9], PatchScope[20], BinXray[7], EnBinDiff[21]
漏洞搜索	TEDEM[22], Multi-MH[23], discovRE[24], Genius[25], Esh[26], Xmatch[27], Gemini, GitZ, BinArm[28], FirmUp[29], VulSeeker[8]
恶意软件检测	MBC[30], BinClone[31], CXZ2014[12]
恶意软件谱系	Beagle[32], iLine[33]
软件盗版检测	CoP[34], MKIS[35]

可以看出, 大部分方法没有明确其应用场景, 导致难以区分这些工作的差别, 并且难以确定这些工作在不同应用场景下的有效性。换言之, 目前以应用场景分类的方式不足以精确描述二进制比对技术的特点。

通过对现有工作的进一步分析, 我们发现应用场景的不同, 主要体现在二进制比对的比较对象和预期目标不同, 并且, 二进制比对的比较对象决定了其预期目标与面临的技术挑战, 为了解决各技术挑战, 二进制比对技术提取了不同的代码特征。因此, 我们在本工作中提出从二进制比对的比较对象出发, 结合二进制比对的预期目标、技术挑战和方法特征, 建立二进制比对的通用描述模型。围绕该模型, 本节分别讨论了二进制比对的比较对象与预期目标之间的联系、比较对象与技术挑战之间的联系, 并对二进制代码特征进行了分类。

4.1 比较对象与预期目标的联系

我们发现现有方法的比较对象(P 和 P1)可以分为 3 种, 如表 3 所示, 分别是: (1) P 和 P1 源代码相同; (2) P 和 P1 源代码不同, 此时 P 和 P1 可能来自同一个程序(例如 P1 是 P 应用安全补丁或者经过版本更新得到的), 也可能来自不同程序; (3) P1 是 P 经过一定的转换规则转换而来, 转换规则包括代码混淆、加壳等, 此时 P 和 P1 的源代码可能相同, 也可能不同。

当 P 和 P1 的源代码相同时, 二进制比对的预期目标为比较它们之间的相似性, 应用场景为漏洞搜索(Multi-MH、discovRE、Genius、Esh、Xmatch、GitZ、Gemini、VulSeeker、BinArm、FirmUp、TEDEM、Tracy)等。

当 P 和 P1 的源代码不同时, 若 P 和 P1 来自同一个程序的不同版本, 那么二进制比对的预期目标为定位它们之间的差异, 此时二进制比对适用于补丁分析(SPAIN、KS2017、PatchScope、EnBinDiff)、恶意软件谱系(Beagle、iLine)等; 若 P 和 P1 来自不同程序, 那么二进制比对的预期目标为比较它们之间的相似性, 应用场景包括漏洞搜索、恶意软件检测(MBC、BinClone、CXZ2014)、软件盗版检测(CoP)等。

当 P1 从 P 转换而来时, P 和 P1 的源代码可能相同, 也可能不同, 此时二进制比对的预期目标为比较它们之间的相似性或差异性。

4.2 比较对象与技术挑战的联系

比较对象的差别, 导致二进制比对面临的挑战有所不同。当 P 和 P1 源代码相同时, 面临的主要挑战是由编译配置引起的, 这是因为近年来开源社区规模逐渐扩大, 为了提高开发效率, 商业软件/嵌入式设备制造商会复用第三方组件的开源代码, 并且在代码编译过程中会使用不同的指令集架构、编译器以及编译优化选项, 导致二进制代码的表现形式不同, 为二进制比对带来了挑战; 当 P 和 P1 源代码不同时, 即 P1 在 P 的基础上, 对 P 的源代码进行增加、删除、修改来修复漏洞或修改功能, 此时 P 和 P1 的语义不同, 二进制比对面临的主要挑战是程序语义修改; 当 P1 是 P 转换而来时, 面临的主要挑战是代码混淆, 这是因为在恶意软件开发过程中, 开发人员会使用代码混淆技术使得程序更加难以被理解和分析, 增加了二进制比对的难度。

表 3 二进制比对技术的分类

Table 3 The classification of binary comparison techniques

P 和 P1 的关系		相关工作
源代码不同	同一程序的不同版本	BinHash[36], Rendezvous[37], TEDEM, Tracy[38], BLEX[39], Multi-MH, discovRE, Genius, Esh, BinGo[6], MockingBird[11], BinDNN[40], Xmatch, Gemini, GitZ, BinSim, BinSequence[41], IMF-sim[42], BinArm, FirmUp, αDiff[43], VulSeeker, InnerEye[44], Asm2Vec[45], SAFE[46], BinGo-E[47], DeepBinDiff[10], IMOPT[48], Codee[49], BinDeep[50], Asteria[51], BinUSE[52], XBA[53], BinShot[54], jTrans[55], WLMXJ2020[56], MKIS, L2021[57]
		iBinHunt[58], Beagle, BinHash, BinJuice[59], BinSlayer[60], Rendezvous, iLine, TEDEM, Tracy, BLEX, Multi-MH, discovRE, Genius, Esh, BinGo, MockingBird, Kam1n0[61], BinDNN, Xmatch, Gemini, GitZ, BinSim, BinSequence, SPAIN, KS2017[62], IMF-sim, BinArm, FirmUp, αDiff, VulSeeker, InnerEye, Asm2Vec, SAFE, BinGo-E, PatchScope, DeepBinDiff, BinXray, BinDiff$_{NN}$[63], EnBinDiff, QBinDiff[64], L2021
	不同程序	MBC, BinHash, BinJuice, Rendezvous, Exposé[65], TEDEM, Tracy, CoP, BLEX, BinClone, CXZ2014, Multi-MH, QSM2015[66], discovRE, Genius, Esh, BinGo, MockingBird, Kam1n0, BinDNN, LIBV[67], Xmatch, Gemini, GitZ, BinSim, BinSequence, IMF-sim, BinArm, FirmUp, αDiff, VulSeeker, InnerEye, Asm2Vec, SAFE, BinGo-E, WLMXJ2020, IMOPT, LibDX[68], MKIS, Asteria, L2021, BinUSE, BinShot
P1 从 P 转换而来		iBinHunt, CoP, CXZ2014, LIBV, BinSim, KS2017, Asm2Vec, IMOPT, BinUSE

4.3 二进制比对的特征分类

在二进制比对基本方法的三个步骤中, 特征提取是最困难的一步, 也是最重要的一步。为了解决各技术挑战带来的影响, 二进制比对技术通常提取二进制代码的语法特征、结构特征以及语义特征, 并将其转化成易于比较的形式, 从而进行相似性/差异性的比较。

二进制代码的**语法特征**体现为具体的指令序列。基于语法的二进制比对通常通过匹配特定的指令序列来衡量代码片段的相似性, 例如, BinClone 使用基于签名的语法分析, 对目标指令序列进行哈希, 如果两段指令序列的哈希值相同, 则这两段指令序列相似; 此外, 还可以通过对齐两个指令序列, 来比较两个序列的相似性, Tracy 利用控制流图将函数分解为执行轨迹的指令片段, 首先对齐指令片段, 并定义指令重写规则, 通过计算重写次数来比较二进制代码的相似性。

结构特征体现了代码的控制流信息, 包括控制流图(Control Flow Graph, CFG)、过程间控制流图(Interprocedural Control Flow Graph, ICFG)、函数调用图(Call Graph, CG)等。计算图结构的相似性可以通过图同构算法实现, 或者将图结构表示为数字特征向量, 然后计算数字向量的距离来度量图结构的相似性。例如, Beagle 使用子图同构算法, 通过子图的匹配数量来度量代码相似性。Genius 将图结构表示为特征向量来计算相似性。

语义特征体现了代码的功能, 如果两段二进制代码的功能完全相同, 那么它们在语义上是等价的。常见的语义特征形式包括符号约束、输入/输出对(I/O)、程序行为、语义嵌入向量等。其中, 符号约束指的是将基本块、程序片段等表示为符号公式, 之后使用约束求解器(如 STP)来比较它们之间的等价性。例如, BinSim 从程序运行的路径中提取符号公式, 进而获取系统调用的参数。输入/输出对指的是通过为程序提供输入, 比较它们的输出状态来判断二进制代码之间是否相似。例如 Multi-MH 通过为代码片段随机提供输入, 来捕获代码的输出信息, 将输入/输出对作为语义特征比较代码相似性。程序行为包含丰富的程序功能信息(如系统调用等), 例如, BLEX 通过重复执行二进制程序的相同函数, 记录函数执行过程中所有的特征, 包括内存读/写操作、系统调用、返回值等, 之后利用这些信息匹配相似的函数。语义嵌入向量是指利用自然语言处理、深度学习等技术对程序行为进行学习, 将语义特征表示为向量形式。例如, InnerEye 利用基于神经机器翻译的方法, 将指令看作单词, 基本块看作句子, 并使用循环神经网络聚合指令语义, 产生基本块的语义嵌入向量。

4.4 二进制比对的通用描述模型

由上述可知, 当前按照应用场景对二进制比对技术进行分类的方法不足以准确描述二进制比对技术, 本文提出的二进制比对的通用描述模型由二进制比对的比较对象、预期目标、技术挑战和方法特征 4 个维度构成。如图 2 所示, 通过这 4 个维度便可以精确描述二进制比对技术。

5 不同挑战对二进制代码特征的影响

二进制比对技术存在泛化能力弱的局限性, 这是因为在不同场景下, 二进制比对技术面临的挑战不同, 为了解决各挑战带来的影响, 二进制比对技术通常利用程序静态分析、自然语言处理等技术来提取二进制代码的特征, 然而各挑战对所提取的特征影响不同, 从而导致二进制比对技术的鲁棒性受到影响。本节首先梳理了 57 种二进制比对技术所提取的特征, 然后分析了各挑战对二进制代码特征带来的具体影响, 最后论述二进制比对技术的鲁棒性问题。

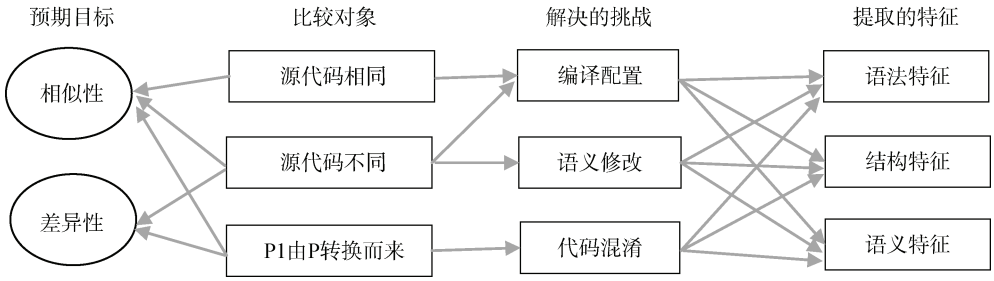

图 2　二进制比对的通用描述模型

Figure 2　The description model of binary comparison

5.1　二进制比对中的特征提取

为了说明各挑战对二进制代码特征带来的影响, 我们首先统计了 57 篇论文所提取的具体的语法特征、结构特征以及语义特征。

表 4　二进制比对技术的挑战与特征

Table 4　The challenges and characteristics of binary comparison techniques

相关工作	解决的挑战					提取的特征		
	编译			语义修改	代码混淆	语法特征	结构特征	语义特征
	跨架构	跨编译器	跨编译优化选项					
MBC[30]						指令的操作码序列		
iBinHunt[58]				√	√		ICFG	基本块的符号表达式
Beagle[32]				√			CFG	系统调用, API 调用
BinHash[36]								基本块的输入/输出样例
BinJuice[59]				√				基本块的符号表达式
BinSlayer[60]				√			CFG, CG	
Rendezvous[37]		√	√			指令操作码, 常量	CFG	
Exposé[65]						函数参数个数, 函数大小, 循环复杂度		函数内 trace 的符号表达式
iLine[33]				√		代码段的大小, 程序大小, 圈复杂度		
TEDEM[22]							CFG	基本块的符号表达式
Tracy[38]				√		trace 的指令序列		trace 内的数据流
CoP[16]		√	√		√		CFG	基本块的符号表达式
BLEX[39]		√	√					对内存的读/写操作、库函数调用、系统调用和函数返回值
BinClone[31]						指令操作码, 操作数类型		
CXZ2014[12]					√		CFG	
Multi-MH[23]	√	√	√				CFG	基本块的输入/输出样例
QSM2015[66]							自定义的执行依赖图	函数内数据和控制依赖关系
discovRE[24]	√	√	√				CFG	
Genius[25]	√	√	√			字符串常量, 数字常量, 转移指令、函数调用、总指令、算术指令的数目	CFG	
Esh[26]		√						指令序列的符号表达式
BinGo[6]	√	√	√					指令序列的输入/输出样例
MockingBird[11]	√	√	√					系统调用
BinDNN[40]	√	√	√			指令的操作码序列		
LIBV[67]					√	基本块的数目, 指令数目等	CFG	

续表

相关工作	解决的挑战					提取的特征		
	编译			语义修改	代码混淆	语法特征	结构特征	语义特征
	跨架构	跨编译器	跨编译优化选项					
Xmatch[27]	√	√						函数的符号表达式
Gemini[18]	√		√			字符串常量、数值常量、转移指令数目、函数调用数目、指令总数和算术指令数目	CFG	
GitZ[19]	√	√						基本块内的数据流
BinSim[4]					√			系统调用, 基本块的符号表达式
BinSequence[41]							CFG	
Spain[9]				√				trace 的输入/输出样例
KS2017[62]		√	√	√				trace 的输入/输出样例
IMF-sim[42]		√	√					对 trace 的内存读/写, 堆读/写, 系统调用
BinArm[28]			√			指令的操作码, 字符串常量	CFG	
FirmUp[29]	√							基本块内的数据流
αDiff[43]	√	√	√	√		函数的字节流	CG	
VulSeeker[8]	√		√				CFG	
InnerEye[44]	√		√					基本块的嵌入向量
Asm2Vec[45]			√		√			trace 内指令序列的嵌入向量
SAFE[46]		√	√					函数内指令序列的嵌入向量
BinGo-E[47]		√	√			局部变量, 指令操作码		系统调用, 函数调用序列, 函数参数
PatchScope[20]				√				trace 内的内存对象
DeepBinDiff[10]			√	√			ICFG	基本块的嵌入向量
BinXray[7]				√		操作码序列	CFG	函数调用
IMOPT[48]			√				CFG	
LibDX[68]	√					字符串常量		
MKIS[35]								API 调用, 算术指令中寄存器和内存读/写
Codee[49]	√		√			指令的操作码和操作数	ICFG, CFG	
$BinDiff_{NN}$[63]				√		指令的操作码和操作数		
EnBinDiff[21]				√		函数内的常量	CFG	
BinDeep[50]	√	√	√	√		指令的操作码和操作数		
Asteria[51]	√							抽象语法树
QBinDiff[64]				√			CG	
L2021[57]								指令的符号表达式
BinUSE[52]	√	√	√		√			trace 的符号表达式
jTrans[55]			√			指令的操作码和操作数		
XBA[53]	√					指令的操作码和操作数, 字符串常量		
BinShot[54]	√	√				指令的操作码和操作数		

5.2 各挑战对二进制代码特征的影响

从表 4 可以看出, 针对不同的技术挑战, 二进制比对方法所提取的特征不同。这是因为各技术挑战对二进制代码特征的影响体现在不同方面, 具体地, 各挑战对二进制代码的语法特征的影响主要体现在指令序列上, 对结构特征的影响主要体现在 CFG、CG 中, 对语义特征的影响主要体现在代码功能上。本小节将分别讨论各技术挑战对二进制代码的语

法、结构、语义特征的具体影响。

5.2.1 编译配置对代码特征的影响

从程序源代码到最终可执行文件(即二进制代码)会经历 4 个步骤: 预编译、编译、汇编和链接。其中编译是主要部分, 编译过程如下图所示:

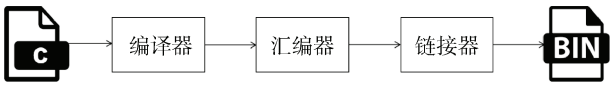

图 3 程序编译过程

Figure 3 The compilation process of programs

影响编译结果的因素有很多, 例如: 指令集架构、编译器以及编译优化选项。这些因素对于二进制代码相似性分析技术带来了挑战, 这是因为, 针对相同的源代码, 在不同的指令架构平台上, 使用不同的编译器、编译优化选项编译得到的二进制代码在语义上是相似的, 但是在语法、结构上存在差异。下面介绍以上因素对二进制代码的差异带来的具体影响。

指令集架构的影响。当前流行的三大指令集架构为 x86、ARM 与 MIPS。不同指令集架构下编译的二进制代码差别很大, 这是由指令集、寻址方式以及寄存器使用等引起的。具体的影响如下:

指令集: 指令集类型可分为精简指令集(如 ARM、MIPS)和复杂指令集(如 x86), 精简指令集的指令长度是固定的, 反之, 复杂指令集的指令长度是可变的。这种差别会直接影响二进制代码的长度。

寻址方式: 根据操作数的来源不同分类, 寻址方式一般可以分为存储器方式和非存储器方式。非存储器寻址一般有寄存器寻址、立即数寻址, 对于 x86, 还有隐含寻址, IO 寻址等; x86 处理器有多种存储器寻址方式, 包括直接寻址、间接寻址、基址寻址、变址寻址以及基址加变址寻址, MIPS 和 ARM 通过 load/store 指令进行寻址。寻址方式不同, 导致二进制代码的具体指令存在差异。

寄存器: 通用寄存器的设计, 直接决定了逻辑处理的流程(例如出入栈, 数据 load 和 store)以及算术处理的过程。不同指令集架构拥有的通用寄存器数量并不一致, 一些寄存器还可以用于特殊用途。不同架构对寄存器的使用, 在二进制代码层面, 体现为具体指令不同。

编译器的影响。编译器基于编程语言的规则、目标机器的指令集和操作系统遵循的惯例, 经过一系列的阶段生成机器代码。编译器通常分为两个部分, 前端和后端。编译器前端负责产生机器无关的中间代码, 编译器后端将中间代码转换成目标机器代码。

二进制文件的特性在很大程度上取决于底层编译器的后端, 而 GCC、Clang 和 MSVC 三大编译器采用不同的后端, 导致不同的编译器生成的二进制代码存在差别, 这种差别具体体现在所生成的二进制代码的语法、结构不同, 包括具体的指令不同、函数和基本块的数量不同。例如, 当指令集架构为 ARM 或 MIPS、编译优化选项为 O0 时, 用 Clang 编译的二进制文件中的基本块的数量明显大于用 GCC 编译的二进制文件[69]; 此外, 即使 GCC 和 Clang 的编译优化选项种类相同(O0、O1、O2、O3 和 Os), 但针对某一具体优化选项的实现技术不同, 例如, GCC 在优化选项为 O0 时采取函数内联优化技术, 而 Clang 在优化选项为 O1 时采取该技术, 最终导致在优化选项相同时, 两个编译器生成的函数数量显著不同。

编译优化选项的影响。为了最小化或最大化计算机可执行程序的某些属性, 例如, 最小化程序的执行时间、内存占用、存储大小和功耗, 编译器通常使用一系列算法对代码进行优化。GCC 编译器默认的优化选项包含 O0、O1、O2、O3 和 Os, 下表列出了每个优化选项对应的优化目的。

表 5 GCC 编译优化选项及其目的

Table 5 GCC compilation optimization options and purpose

优化选项	优化目的
O0	无优化
O1	降低代码大小和加快代码运行的速度
O2	在 O1 的基础上, 降低代码大小和加快代码运行的速度
O3	在 O2 的基础上, 降低代码大小和加快代码运行的速度
Os	在 O2 的基础上, 减少目标代码的大小

不同优化选项导致的二进制代码差异主要是由不同的寄存器分配、指令替换、基本块分割和组合以及函数内联和外联引起的。并且, 除了 O0 之外, 其他优化选项都包含了大量具体的优化 flag, 如: -funroll-loops(循环展开优化)、-fpeephole2(尾调用优化)、-finline-functions(函数内联优化)等, 导致使用不同的编译优化选项生成的二进制代码语法、结构不同。例如, 在编译优化选项为 O3 时, GCC-4.1.0 版本会开启-finline-functions 优化, 将一些短小、逻辑简单的函数体插入并取代每一处调用该函数的地方, 从而节省了每次调用函数带来的额外时间开支, 函数内联优化会导致函数数目减少, 同时影响调用者函数内部的指令发生变化。

5.2.2 语义修改对代码特征的影响

在二进制代码层面, 程序语义的修改会改变程序的基本块、函数以及控制流图, 为二进制比对带来挑战。

程序语义修改有两种方式, 其中一种是在相近版本的同一程序之中, 新版本会通过增添、删除、修改代码的方式来修改旧版本程序功能或修复漏洞(即对旧版本的程序打补丁); 另一种是在不同程序中, 第三方开发商通常在开源代码的基础上修改代码以扩展功能, 例如, 智能手机开发商移植安卓内核来适应自己的设备。

补丁通常引入微小的变化来修复漏洞[70], Bscout[71]分析了 194 个真实世界的 Android 和 Java 第三方库的安全补丁, 结果表明近 80%的补丁发生在函数内部; PatchScope 对 2205 个内存损坏漏洞的安全补丁进行人工分析, 根据安全补丁的代码改动将其归纳为 9 种类型, 分别是增加判断条件(43.5%)、修改判断条件(25.1%)、增加数据结构(6.1%)、修改数据结构定义(6.5%)、修改数据结构引用(22.3%)、修改函数传入参数(10.9%)、增加或修改函数调用(15.3%)、增加函数定义(4.7%)和修改函数定义(7.6%)。其中, 71%的补丁明显改变了控制流图/函数调用图的结构(增加了新分支、基本块或函数), 其余的补丁没有破坏控制流程图/函数调用图的完整性, 只是造成了基本块内部的差异。

除了修复漏洞之外, 第三方代码定制也会改变程序语义, Pdiff[72]调研了基于 Linux 内核的第三方开发商(包括 Google、Samsung、小米、华为等)的 406 个 image 源代码, 发现与 Linux 内核补丁相关的 6027 个函数中, 4383 个函数的代码发生了改变。

5.2.3 代码混淆对代码特征的影响

二进制比对技术可以用于恶意软件检测、恶意软件谱系等。恶意软件是指在未经用户许可的情况下收集敏感信息、控制用户设备、严重侵犯用户个人权益的软件。大多数恶意软件都是少部分恶意软件家族的变体, 因此共享相似的指令序列。然而, 恶意代码编写者经常使用代码混淆技术, 使得恶意软件更难被检测或分析, 例如, 通过在程序中增加花指令, 使得程序难以被理解, 阻碍了对程序的静态分析。

Collberg[73]将混淆技术分为 4 类: 布局混淆、数据混淆、控制流混淆和预防性混淆。布局混淆的重要特点是保持程序语义不变, 在程序源代码级别进行混淆, 例如删除代码注释信息或替换变量名、函数名、类名等标识符; 数据混淆是对程序中的数据结构或数据域进行混淆, 主要包括数组结构变换、修改类与类之间的继承关系、合并和分裂标量变量、排序变换、编码变换; 控制混淆的作用是在不更改程序原有功能的前提下, 通过插入垃圾代码、更改循环语句的顺序、扩展循环条件等方法来改变程序的控制流, 使混淆后的程序更加难以被破解; 预防性混淆指针对特定的反编译器, 找出其存在的缺陷或漏洞, 针对这些漏洞来设计相应的混淆方案。

通过综合使用这些混淆手段, 混淆器在保持程序语义不变的前提下, 改变程序的语法或结构信息, 对代码相似性检测带来了挑战。例如, O-LLVM 混淆器[74]利用控制流扁平化(Control flow flattening)、指令替换(Instruction substitution)、虚假控制流程(Bogus control flow graph)三种手段在中间语言层面进行代码混淆, 破坏了程序 CFG 和基本块的完整性, Asm2Vec 利用 O-LLVM 混淆器对函数进行混淆, 结果表明, 相较于混淆前的函数 CFG, 虚假控制流程平均增加了 149%的顶点和边, 控制流扁平化平均增加了 376%的顶点和边, 指令替换不会改变 CFG 的结构(91%函数的 CFG 保持相同数量的顶点和边)。此外, 恶意代码编写者还会利用加壳器(如 UPX)对恶意代码进行压缩、加密或者其他转换, 据估计[75], 超过 80%的恶意软件都被加了壳, 加壳的两个主要目的是缩减程序的大小、阻碍对加壳程序的分析。对加壳后的程序进行分析之前必须将其脱壳, 才能进行反汇编, 获得汇编代码, 从而进行代码相似性比较。脱壳技术可分为三种: 自动静态脱壳、自动动态脱壳及手动动态脱壳。

5.3 各二进制比对技术的鲁棒性分析

本文从技术层面对近 10 年的二进制比对方法进行了对比, 如表 6 所示, 对比的维度包括各方法比较对象、预期目标、解决的挑战以及提取的特征类型。从表中可以发现, 二进制比对的比较对象(P 和 P1)、预期目标以及面临的挑战存在多种组合, 并且, 由于现有方法所提取的特征都会受到各挑战的影响, 因此导致二进制比对技术的泛化能力弱, 即并不存在一种方法可以解决二进制比对领域内的所有挑战。

6 二进制比对的比较基准

二进制比对方法之间的对比是否有意义体现在它们的输出结果是否具有可比性, 具有可比性意味着输出结果在形式上是一致的并且输出结果的评价指标在同一量级, 在此基础上, 它们表示的含义也应是一致的。通过对二进制比对领域相关文献的调

表 6　二进制比对方法的对比

Table 6　The comparison of binary comparison techniques

相关工作	P 和 P1 的关系			预期目标		解决的挑战				提取的特征		
	源代码相同	源代码不同	P1 从 P 转换而来	相似性	差异性	跨架构	跨编译器	跨编译优化选项	语义修改	代码混淆	语法特征	结构特征	语义特征
MBC[30]		√		√							√		
iBinHunt[58]		√	√	√	√				√	√		√	√
Beagle[32]		√		√					√			√	√
BinHash[36]	√	√		√									√
BinJuice[59]		√		√					√				√
BinSlayer[60]		√		√					√			√	
Rendezvous[37]	√	√		√			√	√			√	√	
Exposé[65]		√		√							√		
iLine[33]		√		√					√		√		
TEDEM[22]	√	√		√								√	√
Tracy[38]	√	√		√					√		√		√
CoP[16]		√	√	√			√	√		√		√	√
BLEX[39]	√	√		√			√	√					√
BinClone[31]		√		√							√		
CXZ2014[12]		√	√	√						√		√	
Multi-MH[23]	√	√		√		√	√	√				√	√
QSM2015[66]		√		√								√	√
discovRE[24]	√	√		√		√	√	√				√	
Genius[25]	√	√		√		√	√	√			√	√	
Esh[26]	√	√		√			√						√
BinGo[6]	√	√		√		√	√	√					√
MockingBird[11]		√		√		√	√	√					√
BinDNN[40]	√	√		√		√	√	√			√		
LIBV[67]		√	√	√						√	√	√	
Xmatch[27]	√	√		√		√	√						√
Gemini[18]	√	√		√		√		√			√	√	
GitZ[19]	√	√		√		√	√						√
BinSim[4]		√	√	√						√			√
BinSequence[41]	√	√		√							√	√	
Spain[9]		√			√				√				√
KS2017[62]		√			√		√	√	√				√
IMF-sim[42]	√	√		√			√	√					√
BinArm[28]	√	√		√				√			√	√	
FirmUp[29]	√	√		√		√							√
αDiff[43]	√	√		√		√	√	√	√		√	√	
VulSeeker[8]	√	√		√		√		√				√	
InnerEye[44]	√	√		√		√		√					√
Asm2Vec[45]	√	√	√	√				√		√			√
SAFE[46]	√	√		√			√	√					√
BinGo-E[47]	√	√		√			√	√			√		√
PatchScope[20]		√			√				√				√
DeepBinDiff[10]	√	√		√				√	√			√	√
BinXray[7]		√		√					√		√	√	√

续表

相关工作	P 和 P1 的关系			预期目标		解决的挑战					提取的特征		
	源代码相同	源代码不同	P1 从 P 转换而来	相似性	差异性	跨架构	跨编译器	跨编译优化选项	语义修改	代码混淆	语法特征	结构特征	语义特征
IMOPT[48]	√	√		√				√				√	
LibDX[68]	√	√		√		√					√		
MKIS[35]	√	√		√									√
Codee[49]	√	√		√		√		√			√	√	
$BinDiff_{NN}$[63]	√	√		√					√		√		
EnBinDiff[21]		√			√				√		√	√	
BinDeep[50]	√	√		√		√	√	√	√		√		
Asteria[51]	√	√		√		√							√
QBinDiff[64]	√	√		√					√			√	
L2021[57]	√	√		√									√
BinUSE[52]	√	√	√	√		√	√	√		√			√
jTrans[55]	√	√		√				√			√		
XBA[53]	√	√		√		√					√		
BinShot[54]	√	√		√		√	√				√		

研与分析, 我们发现现有方法大多以应用场景为维度来选择基准方法进行比较, 然而大部分二进制比对方法并没有明确其应用场景, 从而导致的问题是难以选择基准方法进行对比实验。结合本文提出的二进制比对通用描述模型, 我们发现应用场景的不同, 主要体现在二进制比对的比较对象和预期目标不同, 并且, 二进制比对的比较对象决定了其预期目标与面临的技术挑战。因此, 我们认为, 在选择比较基准时, 应考虑不同方法的比较对象、预期目标、解决的挑战是否一致。当比较对象、预期目标、解决的挑战不一致时, 对它们之间的对比没有意义; 当比较对象、预期目标、解决的挑战一致时, 对它们之间的对比更有意义。

为了验证“当比较对象、预期目标、解决的挑战不一致时, 对它们之间的对比没有意义。”这一观点, 我们选取了比较对象、预期目标、解决的挑战不尽相同的 5 种二进制比对代表性方法, 并对这 5 种方法进行对比实验, 并通过回答以下 2 个研究问题验证该观点的有效性:

问题 1: 当比较对象不同、预期目标不同、解决的挑战不同时, 二进制比对方法之间的对比是否有意义?

问题 2: 当比较对象相同、预期目标相同、解决的挑战不同时, 二进制比对方法之间的对比是否有意义?

此外, 我们在 6.5 节通过一个案例分析, 来验证“当二进制比对技术的比较对象、预期目标以及解决的挑战一致时, 对它们之间的比较更有意义。”这一观点。

6.1 实验方法选取

为了验证以上两个研究问题, 本文选取了 5 种二进制比对代表性方法进行实验分析, 如表 7 所示, 它们的预期目标、比较对象之间的关系、解决的挑战不尽相同。5 种方法的原理如下:

VulSeeker。首先获取程序的控制流图, 并提取基本块特征将其表示为数值向量。然后, 通过将基本块的数值向量输入到 DNN 模型中, 生成整个二进制函数的嵌入向量。最后, 根据余弦距离测量两个函数之间的相似性。

DeepBinDiff。提出了一种无监督的二进制程序代码表示学习技术, 依靠代码语义信息和程序内部的控制流信息来生成基本块的嵌入向量, 并利用 k- hop 贪婪匹配算法来寻找最优的程序基本块比对结果。

Gemini。利用改进的 Structure2Vec 模型将函数的 ACFG 嵌入高维向量, 以向量之间的距离来度量函数间的相似性。

Blex。通过重复执行函数, 在执行期间收集动态行为, 例如内存访问、系统调用、返回值等, 然后计算函数的相似度。

PatchScope。提出了一种内存对象访问序列语义感知的逆向分析方法, 其中, 内存对象访问序列展示了程序在执行过程中是如何访问各种内存对象来操作输入的。之后通过差异比对内存对象访问序列来识别安全补丁引入的代码改动, 并为补丁细节修复的漏洞提供丰富的语义信息。

表 7 5 种二进制比对方法

Table 7 Five binary comparison techniques

相关工作	比较相似性/差异性	P 和 P1 的关系	解决的挑战	比较粒度
VulSeeker	相似	源代码相同或不同	跨架构、跨编译优化选项、语义修改	函数
DeepBinDiff	相似	源代码相同或不同	跨编译优化选项、语义修改	基本块
Gemini	相似	源代码相同或不同	跨架构、跨编译优化选项、语义修改	函数
Blex	相似	源代码相同或不同	跨编译器、跨编译优化选项、语义修改	函数
PatchScope	差异	同一程序的不同版本	语义修改	内存对象访问序列

6.2 实验数据集

数据集 1: 针对问题 1, 本文选取的数据集为存在 CVE 漏洞的真实世界程序, 分析其打补丁前/后的版本。这是因为应用场景为补丁分析时, 二进制比对方法的目标是比较程序之间的差异性, 从而精确定位补丁差异。而在其他应用场景下, 二进制比对方法的目标是比较程序之间的相似性。并且, 预期目标为比较差异性时, 二进制比对的比较对象为源代码不同的程序, 预期目标为比较相似性时, 二进制比对的比较对象为源代码相同或不同的程序。综合考虑两种情况的比较对象, 本文选取源代码不同的程序作为实验数据集。具体程序如表 8 所示。

表 8 应用程序及其 CVE 编号

Table 8 Applications and their CVE number

应用程序	CVE 编号
mcrypt-2.5.8	CVE-2012-4409
binutils-2.12	CVE-2005-4807
mutt-1.4.2.2	CVE-2007-2683
streamripper-1.61.25	CVE-2006-3124
libzip-1.2.0	CVE-2017-12858
nginx-1.4.0	CVE-2013-2028
libpng-1.2.5	CVE-2004-0597
putty-0.66	CVE-2016-2563
libsmi-0.4.8	CVE-2010-2891
gzip-1.2.4	CVE-2001-1228
tiff-4.03	CVE-2013-4232
opendchub	CVE-2010-1147
xrdp-0.4.1	CVE-2008-5904
Apache-1.3.35	CVE-2006-3747
leptonica-1.70.1	CVE-2018-7186

数据集 2: 针对问题 2, 本文选取的数据集为开源软件 Binutils v2.30、Findutils v4.6.0 以及 Coreutils v8.29, 在指令集架构为 x86_64、MIPS64 和 ARM64、编译器为 GCC v4.9.4 和 Clang v4.0、编译优化选项为 O0-O3 时, 对源代码进行交叉编译, 并利用 Ollvm 的 3 种混淆方式(控制流扁平化、指令替换、虚假控制流程)对其进行混淆, 最终得到二进制文件。即, 一个源程序, 在不同的编译和混淆方式的影响下, 可以得到 72 个不同的二进制文件。

6.3 评价指标

针对问题 1, 实验目标是探究当比较对象不同、预期目标不同、解决的挑战不同时, 二进制比对方法之间的对比是否有意义。本文采用的数据集为同一程序打补丁前/后的版本, 比较程序差异性意味着需要准确识别补丁差异, 我们依据补丁的源代码, 并对比程序打补丁前/后的版本, 人工判断这些差异是否是由于安全补丁引起的, 如果是由补丁引起的, 则视为识别成功。最终, 我们将实验的评估指标设置为检测到打补丁前后补丁差异的数量。

针对问题 2, 实验目标是探究当比较对象相同、预期目标相同、解决的挑战不同时, 二进制比对方法之间的对比是否有意义。针对函数级别的二进制比对方法 VulSeeker 和 Gemini, 利用 *top-K* 指数评价其有效性, *top-K* 表示待查询函数在目标二进制文件中所有函数的排名为 K, K 值越小, 该方法效果越好; 针对基本块级别的二进制比对方法 DeepBinDiff, 利用召回率(Recall)和精确度(Precision)评价其有效性。

6.4 实验设计与结果分析

问题 1: 预期目标不同的二进制比对方法之间的对比是否有意义?

为了验证此问题, 本文选取 PatchScope、BLEX 和 DeepBinDiff 进行验证。其中, PatchScope 的预期目标为比较程序的差异性, 比较对象为源代码不同的程序(具体为同一程序打补丁前/后的版本), BLEX 和 DeepBinDiff 的预期目标为比较程序之间的相似性, 比较对象为源代码相同或不同的程序。

实验设计。这 3 种方法的比较粒度各不相同, PatchScope 的比较粒度是自定义的内存对象访问序列, DeepBinDiff 的比较粒度是基本块, BLEX 的比较粒度是函数。由于补丁通常发生在函数内, BLEX 的比较粒度过大, 因此我们选取其中最重要的对内存单元的访问作为比较粒度。通过比较同一程序打补丁前/后的版本, 可以得到各方法识别出的比较粒度的差异, 并以补丁的源代码为标准判定该差异是否由补丁带来, 最后统计出由补丁带来的差异数目。由

上述可知, 三种方法的分析粒度分别为: 内存对象访问序列(PatchScope)、内存单元(BLEX)、基本块(DeepBinDiff)。实验结果如表 9 所示。表中数字代表各方法识别的分析粒度的差异数目, 数字为 0 表示该方法并未识别到补丁差异。

表 9 识别补丁差异的实验结果

Table 9 Experimental results of identifying patch differences

应用程序	PatchScope	BLEX	DeepBinDiff
mcrypt-2.5.8	3	151	4
binutils-2.12	2	92	2
mutt-1.4.2.2	2	148	9
streamripper-1.61.25	1	361	0
libzip-1.2.0	3	514	1
nginx-1.4.0	1	210	8
libpng-1.2.5	5	111	0
putty-0.66	12	4564	33
libsmi-0.4.8	5	1112	4
gzip-1.2.4	2	601	1
tiff-4.03	1	139	0
opendchub	2	266	27
xrdp-0.4.1	5	342	2
Apache-1.3.35	0	158	45
leptonica-1.70.1	1	319	0

从表 9 中可以看出, 在这 15 个程序中, DeepBinDiff 漏报了 3 个程序, 在其余 12 个程序中识别到的平均补丁差异数目(基本块差异数目)为 11.3; PatchScope 漏报了 1 个程序, 识别的平均补丁差异数目(内存对象访问序列差异数目)为 3.2; BLEX 成功识别出 15 个程序的补丁差异, 识别的平均补丁差异数目(内存单元差异数目)为 605.8。实验结果表明 PatchScope、BLEX、DeepBinDiff 的输出结果在形式上不一致, 并且识别的差异数量不在同一量级, 不具备有效比较的条件。

由此可以得出结论, 当二进制比对方法之间的比较对象、预期目标以及解决的挑战不同时, 对它们的比较是没有实际意义的。

问题 2: 当预期目标相同、比较对象相同、解决的挑战不同时, 二进制比对方法间的对比是否有意义?

为了回答该问题, 本文选取 VulSeeker、DeepBinDiff、Gemini 三种二进制比对技术, 分别验证其在不同挑战下的性能表现, 具体挑战包括跨指令集架构、跨编译器、跨编译优化选项以及代码混淆。其中, VulSeeker 与 Gemini 实现函数级别的二进制比对, 支持跨指令集架构、跨编译优化选项。DeepBinDiff 实现基本块级别的二进制比对, 支持跨编译优化选项。

VulSeeker 与 Gemini 的实验设计。在 Binutils v2.30、Findutils v4.6.0 以及 Coreutils v8.29 数据集中, 随机选取一个二进制文件的函数作为待查询函数, 这两种方法会将目标二进制文件中的函数按照与待查询函数的相似度从高到低进行排序, 其排名即为 *top-K* 中的 *K* 值, *K* 值越小, 该方法效果越好。实验结果如表 10 所示。

表 10 Vulseeker 与 Gemini 实验结果

Table 10 Experimental results of Vulseeker and Gemini

数据集		*top-K* 平均值	
		Vulseeker	Gemini
Binutils v2.30	跨指令集架构	3	2
	跨编译器	3	2
	跨编译优化选项	5	5
	混淆	3	12
Coreutils v8.29	跨指令集架构	4	8
	跨编译器	9	18
	跨编译优化选项	8	6
	混淆	3	13
Findutils v4.6.0	跨指令集架构	12	3
	跨编译器	20	2
	跨编译优化选项	8	2
	混淆	10	12

VulSeeker 与 Gemini 的实验结果。VulSeeker 与 Gemini 都支持跨指令集架构和跨编译选项下的函数匹配, 不支持跨编译器和混淆。从实验结果可以看出, VulSeeker 在跨编译器的场景下 *top-K* 值最高, 当数据集为 Findutils v4.6.0 时, 其 *top-K* 值比跨指令集架构下降了 8, 比跨编译优化选项下降了 12。同时, Gemini 在混淆的场景下 *top-K* 值最高, 当数据集为 Binutils v2.30 时, 其 *top-K* 值比跨指令集架构下降了 10, 比跨编译优化选项下降了 7。以上结果说明 VulSeeker 与 Gemini 在跨编译器和混淆的场景下性能表现下降。

DeepBinDiff 的实验设计。DeepBindiff 提出一种基本块级别的二进制比对方法, 支持跨编译优化选项下的基本块匹配。给定两个二进制程序, DeepBinDiff 会返回这两个程序中相匹配的基本块对, 基本块对中的两个基本块被认为是相似的。本文选取 Binutils v2.30、Findutils v4.6.0 和 Coreutils v8.29 的一个编译版本, 分别与其他 71 个编译版本进行对比, 使用召回率与精确度作为评价指标, 召回率和精确

度越高, 该方法效果越好。

DeepBindiff 的实验结果。DeepBinDiff 仅支持跨编译优化选项下的基本块匹配。由实验结果得知, DeepBinDiff 在跨指令集架构的场景下性能明显下降。数据集为 Binutils v2.30 时, 其在跨编译优化选项的场景下平均召回率为 0.66, 平均精确度为 0.66, 此时 DeepBinDiff在跨指令集架构下的召回率相较于其在跨编译优化选项的场景下降了 0.508, 精确度下降了 0.512。此外, DeepBinDiff 在跨编译器的场景下性能也有所下降。

由此可以得出结论, 当预期目标、比较对象的关系确定时, 二进制比对方法仅能解决其方法中声明的挑战, 在其他挑战下该方法的性能会下降, 导致在不同挑战下二进制比对方法之间的比较不够公平。

表 11 DeepBinDiff 实验结果

Table 11 Experimental results of DeepBinDiff

数据集			召回率	精确度
Binutils v2.30	跨指令集架构		0.152	0.148
	跨编译器		0.464	0.531
	跨编译优化选项	O0 – O3	0.317	0.320
		O1 – O3	0.732	0.717
		O2 – O3	0.931	0.943
	混淆		0.721	0.706
Coreutils v8.29	跨指令集架构		0.113	0.127
	跨编译器		0.542	0.542
	跨编译优化选项	O0 – O3	0.291	0.310
		O1 – O3	0.613	0.652
		O2 – O3	0.918	0.904
	混淆		0.543	0.522
Findutils v4.6.0	跨指令集架构		0.118	0.131
	跨编译器		0.625	0.615
	跨编译优化选项	O0 – O3	0.292	0.320
		O1 – O3	0.762	0.801
		O2 – O3	0.957	0.957
	混淆		0.568	0.573

6.5 案例分析

为了验证“当二进制比对技术的比较对象、预期目标以及解决的挑战一致时, 对它们之间的比较更有意义。”这一观点, 我们选取了 Gemini 和 VulSeeker(比较对象、预期目标以及解决的挑战一致)这两种方法进行实验说明。Gemini 和 VulSeeker

都可应用于漏洞搜索这一应用场景, 因此我们利用一个案例分析来给出实验结果。该实验目的为查找 OpenSSLv1.0.1f 中的漏洞函数 tls1_process_heartbeat(CVE 编号为 CVE-2014-0160), 我们分别在不同架构(x86-64, MIPS 和 ARM)、不同编译器(GCC v4.9.4 和 Clang v4.0)、不同编译优化选项(O0-O3)对 OpenSSLv1.0.1f 进行编译, 实验评价指标为 *top-K* 指数, *top-K* 表示待查询函数在目标二进制文件中所有函数的排名为 *K*, *K* 值越小, 该方法效果越好。实验结果如下表所示, 表中第一列为不同的 *top-K* 值, 第二列和第三列为 *top-K* 结果中真实漏洞函数的数量以及真实漏洞函数所占的比例。并且, 我们通过统计发现, 在 OpenSSLv1.0.1f 这一数据集中, Gemini 查找该漏洞函数的平均 *top-K* 值为 97, VulSeeker 查找该漏洞函数的平均 *top-K* 值为 43。这一实验结果说明这两种方法之间的比较结果在同一数量级别, 具备可比性。当比较对象、预期目标以及解决的挑战一致时, 方法之间的对比更能体现出各方法的有效性。

表 12 识别 CVE-2014-0160 的实验结果

Table 12 Experimental results of identifying CVE-2014-0160

top-K	Gemini	VulSeeker
1	1(100%)	1(100%)
5	2(40%)	3(60%)
10	3(30%)	5(50%)
50	26(52%)	32(64%)
100	72(72%)	82(82%)

综上所述, 我们可以得出结论: 当前以应用场景作为比较基准的二进制比对方法之间的比较并没有意义。二进制比对技术比较基准的选择应综合考虑方法之间的比较对象、预期目标和解决的挑战, 当这三个条件一致时, 方法之间的比较更具有实际意义。

7 挑战与方向

虽然二进制比对领域的相关研究已取得较好的成效, 但仍存在一些挑战, 例如并行的二进制比对问题以及二进制比对的证据析取等问题, 并行的二进制比对关注大规模二进制文件比对的效率问题, 二进制比对的证据析取关注二进制代码样本库的构建问题。此外, 本文主要关注的是二进制比对技术在效果层面所面临的难点问题及在效果层面上的比较, 具体挑战包括:

(1) 比较对象关系的确定。现有方法对二进制代码片段之间的比较的前提是已知比较对象之间的关系, 若比较对象之间的关系未知, 那么对它们之间的比较是非常困难的。

(2) 二进制逆向的精确度。现有方法对二进制程序的比较通常依赖于相关逆向工具(如 IDA Pro)来恢复程序信息, 例如函数边界、控制流图、函数调用图等, 并从中提取二进制代码特征, 从而进行二进制比对。然而 IDA Pro 对二进制程序的逆向结果并不一定准确, 导致二进制比对技术所提取的特征不正确。

结合现阶段二进制比对技术存在的挑战, 我们总结了以下几点值得重点关注的研究方向:

(1) 基准模型的选择。现有二进制比对方法大多以特征为维度来选择基准方法进行比较, 并未考虑不同方法的比较对象、预期目标、解决的挑战是否一致, 导致方法之间的比较不够公平。因此本文建议当选择比较的基准模型时, 应考虑方法之间的预期目标、比较对象以及解决的挑战是否一致。

(2) 数据集的构建。二进制比对技术应用广泛, 不同应用场景下各方法使用的数据集不一致, 然而针对相同的应用场景, 现有方法使用的数据集仍不统一, 导致得出的结论具有片面性。因此需要收集全面准确的数据集, 涵盖二进制比对的应用场景以及各应用场景面临的技术挑战, 来增加对比目前研究方法的有效性。目前已有相关工作[71]构建了二进制比对多种挑战下的数据集, 然而针对代码混淆的情况考虑不够全面。

(3) 提升二进制逆向的精确度。二进制逆向的精确度直接影响了二进制比对的效果, 例如函数边界的识别、函数参数的恢复、控制流图的生成以及对混淆的代码进行代码去混淆都会提升二进制比对的准确率。如何提升二进制逆向的精确度可以作为未来的一个研究方向。

8 结束语

本文首先介绍了二进制比对技术的概念、基本流程以及应用场景。结合近 10 年二进制比对代表性技术, 从二进制比对技术的比较对象、预期目标、技术挑战以及方法特征 4 个维度建立二进制比对的通用描述模型。并论述了不同挑战对二进制代码特征的影响。针对二进制比对技术基准选择困难的问题提出观点, 并通过实验进行验证, 实验结果表明, 当二进制比对技术的预期目标、比较对象、解决的挑战一致时, 对它们之间的比较更具有现实意义。

参考文献

[1] Wang J X, Yang F, Han F. Instruction Merging Optimization Algorithm in Binary Files Comparison[J]. *Computer Applications and Software*, 2013, 30(12): 40-42, 126.
(王建新, 杨凡, 韩锋. 二进制文件比对中的指令归并优化算法[J]. *计算机应用与软件*, 2013, 30(12): 40-42, 126.)

[2] Xiao R Q, Fei J L, Zhu Y F, et al. Firmware Binary Comparison Technology Based on Community Detection Algorithm[J]. *Journal of Computer Research and Development*, 2022, 59(1): 209-235.
(肖睿卿, 费金龙, 祝跃飞, 等. 基于社团检测算法的固件二进制比对技术[J]. *计算机研究与发展*, 2022, 59(1): 209-235.)

[3] Gao D B, Reiter M K, Song D. BinHunt: Automatically Finding Semantic Differences in Binary Programs[C]. *Information and Communications Security*, 2008: 238-255.

[4] Ming J, Xu D P, Jiang Y F, et al. BinSim[C]. *The 26th USENIX Conference on Security Symposium*, 2017: 253-270.

[5] Haq I U, Caballero J. A Survey of Binary Code Similarity[J]. *ACM Computing Surveys*, 2021, 54(3): 1-38.

[6] 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.

[7] Xu Y F, Xu Z Z, Chen B H, et al. Patch Based Vulnerability Matching for Binary Programs[C]. *The 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020: 376-387.

[8] 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.

[9] Xu Z Z, Chen B H, Chandramohan M, et al. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills[C]. *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017: 462-472.

[10] 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:.

[11] 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.

[12] Cesare S, Xiang Y, Zhou W L. Control Flow-Based Malware VariantDetection[J]. *IEEE Transactions on Dependable and Secure Computing*, 2014, 11(4): 307-317.

[13] Hu X, Shin K, Bhatkar S, et al. MutantX-S: Scalable Malware Clustering Based on Static Features[C]. *USENIX Annual Technical Conference*, 2018

[14] Farhadi M R, Fung B C M, Fung Y B, et al. Scalable Code Clone Search for Malware Analysis[J]. *Digital Investigation: the International Journal of Digital Forensics & Incident Response*, 2015, 15(C): 46-60.

[15] Ming J, Xu D P, Wu D H. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference[C]. *ICT Systems Security and Privacy Protection*, 2015: 416-430.

[16] Luo L N, Ming J, Wu D H, et al. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection[C]. *IEEE Transactions on Software Engineering*, 2017: 1157-1177.

[17] Fang L, Wu Z H, Wei Q. Summary of Binary Code Similarity Detection Techniques[J]. *Computer Science*, 2021, 48(5): 1-8.
(方磊, 武泽慧, 魏强. 二进制代码相似性检测技术综述[J]. *计算机科学*, 2021, 48(5): 1-8.)

[18] 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.

[19] David Y, Partush N, Yahav E, et al. Similarity of Binaries through re-Optimization[C]. *The 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017: 79-94.

[20] Zhao L, Zhu Y C, Ming J, et al. PatchScope: Memory Object Centric Patch Diffing[C]. *The 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 149-165.

[21] Lin J, Wang D D, Chang R, et al. EnBinDiff: Identifying Data-Only Patches for Binaries[J]. *IEEE Transactions on Dependable and Secure Computing*, 2023, 20(1): 343-359.

[22] Pewny J, Schuster F, Bernhard L, et al. Leveraging Semantic Signatures for Bug Search in Binary Programs[C]. *The 30th Annual Computer Security Applications Conference*, 2014: 406-415.

[23] Pewny J, Garmany B, Gawlik R, et al. Cross-Architecture Bug Search in Binary Executables[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 709-724.

[24] 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.

[25] 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.

[26] David Y, Partush N, Yahav E, et al. Statistical Similarity of Binaries[C]. *The 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016: 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] Shirani P, Collard L, Agba B L, et al. BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices[C]. *Detection of Intrusions and Malware, and Vulnerability Assessment*., 2018: 114-138.

[29] David Y, Partush N, Yahav E. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware[C]. *The Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018: 392-404.

[30] Kang B, Kim T, Kwon H, et al. Malware Classification Method *via* Binary Content Comparison[C]. *The 2012 ACM Research in Applied Computation Symposium*, 2012: 316-321.

[31] Farhadi M R, Fung B C M, Charland P, et al. BinClone: Detecting Code Clones in Malware[C]. *2014 Eighth International Conference on Software Security and Reliability*, 2014: 78-87.

[32] Lindorfer M, Di Federico A, Maggi F, et al. Lines of Malicious Code: Insights into the Malicious Software Industry[C]. *The 28th Annual Computer Security Applications Conference*, 2012: 349-358.

[33] Jang J, Woo M, Brumley D, et al. Towards Automatic Software Lineage Inference[C]. *The 22nd USENIX conference on Security*, 2013: 81-96.

[34] Luo L N, Ming J, Wu D H, et al. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection[J]. *IEEE Transactions on Software Engineering*, 2017, 43(12): 1157-1177.

[35] 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.

[36] Jin W, Chaki S, Cohen C, et al. Binary Function Clustering Using Semantic Hashes[C]. *2012 11th International Conference on Machine Learning and Applications*, 2012: 386-391.

[37] Khoo W M, Mycroft A, Anderson R. Rendezvous: A Search Engine for Binary Code[C]. *2013 10th Working Conference on Mining Software Repositories*, 2013: 329-338.

[38] David Y, Yahav E, David Y, et al. Tracelet-Based Code Search in Executables[C]. *The 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014: 349-360.

[39] Egele M, Woo M, Chapman P, et al. Blanket Execution[C]. *The 23rd USENIX conference on Security Symposium*, 2014: 303-317.

[40] Lageman N, Kilmer E D, Walls R J, et al. BinDNN: Resilient Function Matching Using Deep Learning[C]. *Security and Privacy in Communication Networks*, 2017: 517-537.

[41] HUANG H, YOUSSEF A M, DEBBABI M. BinSequence[C].

Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017: 155-166.

[42] 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.

[43] 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.

[44] Zuo F, Li X P, Young P, et al. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[45] 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.

[46] Massarelli L, Di Luna G A, Petroni F, et al. SAFE: Self-Attentive Function Embeddings for Binary Similarity[C]. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019: 309-329.

[47] Xue Y X, Xu Z Z, Chandramohan M, et al. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation[J]. *IEEE Transactions on Software Engineering*, 2019, 45(11): 1125-1149.

[48] Jiang J G, Li G W, Yu M, et al. Similarity of Binaries Across Optimization Levels and Obfuscation[C]. *Computer Security – ESORICS 2020*, 2020: 295-315.

[49] 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.

[50] 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.

[51] Yang S G, Cheng L, Zeng Y C, et al. Asteria: Deep Learning-Based AST-Encoding for Cross-Platform Binary Code Similarity Detection[C]. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2021: 224-236.

[52] Wang H J, Ma P C, Yuan Y Y, et al. Enhancing DNN-Based Binary Code Function Search with Low-Cost Equivalence Checking[J]. *IEEE Transactions on Software Engineering*, 2023, 49(1): 226-250.

[53] Kim G, Hong S, Franz M, et al. Improving Cross-Platform Binary Analysis Using Representation Learning *via* Graph Alignment[C]. *The 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022: 151-163.

[54] Ahn S, Ahn S, Koo H, et al. Practical Binary Code Similarity Detection with BERT-Based Transferable Similarity Learning[C]. *The 38th Annual Computer Security Applications Conference*, 2022: 361-374.

[55] Wang H, Qu W J, Katz G, et al. JTrans: Jump-Aware Transformer for Binary Code Similarity Detection[C]. *The 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022: 1-13.

[56] Wang W H, Li G, Ma B, et al. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree[C]. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, 2020: 261-271.

[57] Liu Z A. Binary Code Similarity Detection[C]. *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021: 1056-1060.

[58] Ming J, Pan M, Gao D B. IBinHunt: Binary Hunting with Inter-Procedural Control Flow[C]. *Information Security and Cryptology – ICISC 2012*, 2013: 92-109.

[59] Lakhotia A, Dalla Preda M, Giacobazzi R. Fast Location of Similar Code Fragments Using Semantic 'Juice'[C]. *The 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013: 1-6.

[60] Bourquin M, King A, Robbins E. BinSlayer: Accurate Comparison of Binary Executables[C]. *The 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013: 1-10.

[61] Ding S H H, Fung B C M, Charland P. Kam1n0: MapReduce-Based Assembly Clone Search for Reverse Engineering[C]. *The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016: 461-470.

[62] Kargén U, Shahmehri N. Towards Robust Instruction-Level Trace Alignment of Binary Code[C]. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017: 342-352.

[63] Ullah S, Oh H. BinDiffNN: Learning Distributed Representation of Assembly for Robust Binary Diffing Against Semantic Differences[J]. *IEEE Transactions on Software Engineering*, 2022, 48(9): 3442-3466.

[64] Mengin E, Rossi F. Binary Diffing as a Network Alignment Problem *via* Belief Propagation[C]. *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021: 967-978.

[65] Ng B H, Prakash A. Expose: Discovering Potential Binary Code re-Use[C]. *2013 IEEE 37th Annual Computer Software and Applications Conference*, 2013: 492-501.

[66] Qiu J, Su X H, Ma P J. Library Functions Identification in Binary Code by Using Graph Isomorphism Testings[C]. *2015 IEEE 22nd International Conference on Software Analysis*, *Evolution*, *and Reengineering*, 2015: 261-270.

[67] Kim T, Lee Y R, Kang B, et al. Binary Executable File Similarity Calculation Using Function Matching[J]. *The Journal of Supercomputing*, 2019, 75(2): 607-622.

[68] Tang W, Luo P, Fu J L, et al. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code[C]. *2020 IEEE 27th International Conference on Software Analysis*, *Evolution and Reengineering*, 2020: 104-115.

[69] Kim D, Kim E, Cha S K, et al. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned[J]. *IEEE Transactions on Software Engineering*, 2023, 49(4): 1661-1682.

[70] Nguyen H A, Nguyen A T, Nguyen T T, et al. A Study of Repetitiveness of Code Changes in Software Evolution[C]. *2013 28th IEEE/ACM International Conference on Automated Software*

Engineering, 2013: 180-190.

[71] Dai J R, Zhang Y, Jiang Z Y, et al. BScout[C]. *The 29th USENIX Conference on Security Symposium*, 2020: 1147-1164.

[72] Jiang Z Y, Zhang Y, Xu J, et al. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels[C]. *The 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 1149-1163.

[73] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations. Technical Report. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[74] Junod P, Rinaldini J, Wehrli J, et al. Obfuscator-LLVM — Software Protection for the Masses[C]. *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015: 3-9.

[75] Guo F L, Ferrie P, Chiueh T C. A Study of the Packer Problem and Its Solutions[C]. *Recent Advances in Intrusion Detection*, 2008: 98-115.

胡梦莹 于 2017 年在西北工业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。Email: humengying@whu.edu.cn

王笑克 于 2016 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。研究领域为软件安全。Email: xkernel@whu.edu.cn

赵磊 于 2012 年在武汉大学信息安全专业获得博士学位。现为武汉大学国家网络安全学院教授。研究领域为系统及软件安全。Email: leizhao@whu.edu.cn