

## 基于代码相似性的软件漏洞检测研究进展

李晴浩, 刘春玲, 唐永鹤, 董卫宇

(信息工程大学, 河南 郑州 450001)

**摘要:** 软件漏洞是长期困扰网络与信息安全领域的一类关键威胁, 而漏洞检测是缓解此类威胁的首要技术手段。基于代码相似性的漏洞检测具有可规模化、低成本的优势; 特别是与图神经网络、循环神经网络等人工智能技术结合后, 其准确性与适用性进一步提升, 正成为该领域的研究热点。系统梳理了近7年来基于代码相似性的软件漏洞检测进展: 从数据集角度, 剖析了漏洞样本数据集的类别、粒度、规模及其平衡性; 从代码预处理角度, 总结了代码清理与标准化、反汇编与中间语言表示等两类方法; 从代码表征角度, 归纳了基于代码片段、哈希、树、图、指标及混合表征6种方案的检测机制、优缺点与适用场景, 并在统一评测指标下对比其检测性能与开销; 进一步提炼了所用机器学习技术的类别与特点。最后, 总结了该研究面临的挑战, 并展望了未来的研究方向。

**关键词:** 漏洞检测; 代码相似性; 机器学习; 代码表征; 图神经网络; 循环神经网络

**中图分类号:** TP393

**文献标志码:** A

**DOI:** 10.11959/j.issn.2096-109x.2025065

## Research progress on detection of software vulnerabilities based on code similarity

LI Qinghao, LIU Chunling, TANG Yonghe, DONG Weiyu

Information Engineering University, Zhengzhou 450001, China

**Abstract:** Software vulnerabilities have long been a key threat to the field of network and information security, and vulnerability detection is the primary technical means to mitigate such threats. Vulnerability detection based on code similarity offers the advantages of scalability and low cost, particularly when combined with artificial intelligence technologies such as graph neural networks and recurrent neural networks, further enhancing its accuracy and applicability, making it a research hotspot in this field. This paper systematically reviews the progress of software vulnerability detection based on code similarity in the past seven years: from the perspective of datasets, it analyzes the categories, granularity, scale, and balance of vulnerability sample datasets; from the perspective of code preprocessing, it summarizes two types of methods: code cleaning and standardization, and disassembly and intermediate language representation; from the perspective of code representation, it categorizes six schemes of detection mechanisms, advantages and disadvantages, and applicable scenarios based on code snippets, hashes, trees, graphs, metrics, and hybrid representations, and compares their detection performance and overhead under unified evaluation metrics. Furthermore, it distills the categories and characteristics of the machine learning techniques employed. Finally, it summarizes the challenges faced by this research and looks forward to future research directions.

收稿日期: 2025-03-17; 修回日期: 2025-05-17

通信作者: 董卫宇, dongxinbaoer@163.com

基金项目: 河南省重点研发专项计划 (221111210300)

**Foundation Item:** The Key R&D Special Program of Henan Province (221111210300)

**Keywords:** vulnerability detection, code similarity, machine learning, code representation, graph neural network, recurrent neural network

## 0 引言

软件是计算机数据与指令的集合，其在设计、开发、运行与维护等环节的缺陷均可能引入漏洞。网络攻击者利用这些漏洞实施信息窃取、拒绝服务、越权访问等恶意行为，严重威胁国家与公民的信息、财产安全。因此，必须开展软件漏洞检测研究，以便及时、高效地发现并修补漏洞，从而消除安全隐患，提升软件安全性。

对此，研究者提出了模糊测试<sup>[1-2]</sup>、符号执行<sup>[3-4]</sup>、污点分析<sup>[5-6]</sup>及代码相似性分析等多种技术。其中，模糊测试、符号执行和污点分析要求代码完整、可执行，且需适配的软硬件环境，难以应用于大规模或源码残缺的场景。相比之下，代码相似性分析通过传统相似性算法或机器学习框架，计算代码在语法、语义、控制流、数据流及函数调用等特征上与已知漏洞样本的相似度，具备速度快、可扩展、低成本等优势。

基于代码相似性的软件漏洞检测技术发展迅速。自2006年Li等<sup>[7]</sup>提出利用重复代码特定标记检测代码错误的工具CP-Miner以来，相关研究逐年递增；2018年后，图神经网络（graph neural network, GNN）、循环神经网络（recurrent neural network, RNN）、对比学习等人工智能技术被大规模引入，迄今已有数百种有效方法被提出，显著推动了该领域发展。鉴于工业方案普遍闭源，其方法论与学术成果存在显著差异，本文聚焦于学术界公开发表的检测方法。

为系统收集文献，本文以“code similarity”、“machine learning”、“vulnerability detection”等为关键词，在Google Scholar、IEEE Xplore、ACM Digital Library、Springer、arXiv、DBLP、CNKI等数据库检索，获得2 462条高相关记录；经关键词过滤、摘要筛查与人工复核，最终选取2006—2024年发表的447篇核心论文。统计显示，论文数量呈显著上升趋势，如图1所示。进一步分析发现，2018年起，受人工智能第三次浪潮和开源代码生态繁荣的影响，该领域研究出现质性跃迁，涌现出诸多技术新颖、高效可扩展的方

法。因此，本文重点研究2018—2024年发表于CCS、NDSS等会议与期刊的75篇代表性论文。

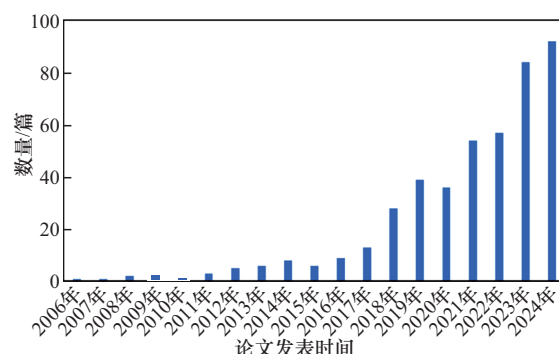


图1 不同年份的论文数量分布

本文从漏洞样本数据集、代码预处理、代码表征与漏洞检测3个方面对现有方法进行技术特征剖析，梳理关键机制，归纳领域挑战，并展望未来研究方向。

### （1）相关术语

#### 1) 代码类型

依据代码形态，软件代码通常划分为源代码与二进制代码两类。

源代码：指C、C++、C#、Java、Python等高级语言编写的原始程序文件，既可以是开源项目，也可以是闭源项目。

二进制代码：指源代码在x86、ARM、MIPS等指令集架构下，经GCC、Clang等特定版本编译器、在不同优化等级（-O0~-O3）下生成的可执行文件或库代码。

#### 2) 代码相似性

又称代码克隆。若两段或多段代码在词法、语法、语义、控制流或数据流层面具有高度一致性，即可认为彼此相似。

#### 3) 代码相似性检测技术

该技术通过文本对比、语法分析、语义分析等手段衡量代码间的相似程度，最早可追溯至1988年Whale<sup>[8]</sup>提出的克隆检测框架。经过三十余年发展，现已形成包含代码预处理、代码表征、相似性判定等环节的完整技术体系。典型工作涵盖跨源代码与二进制两种模态的CodeCMR<sup>[9]</sup>：该

方法提取代码的控制流图 (control flow graph, CFG) 特征, 利用 GNN 同时捕获源代码与二进制代码中的字符串常量与整数字面量特征, 实现跨模态相似代码检测。

源代码侧, DéjàVu<sup>[10]</sup>通过哈希指纹快速定位克隆, CrystalBLEU<sup>[11]</sup>则基于序列相似度精细度量代码接近程度。

二进制侧, MIRROR<sup>[12]</sup>将变量恢复任务转化为 Token 级二分类问题, 并借助概率推导预测变量具体取值, 进而支撑后续相似性判定。

## (2) 基于代码相似性的漏洞检测技术

基于代码相似性的漏洞检测通用流程如图 2 所示: ①构建规模化的漏洞样本数据集; ②依据检测对象 (源代码或二进制代码) 对样本与待测代码进行统一预处理与表征; ③利用相似性检测框架训练得到的漏洞检测模型对待测代码执行比对, 输出检测结果。代码克隆产生的漏洞广泛存在于软件系统中。

Islam 等<sup>[13]</sup>对 34 个软件系统共 870 万行代码进行克隆与漏洞关联分析, 发现 17% 的漏洞由代码克隆引入。随着开源生态繁荣, 开发者频繁复用相似代码实现重复功能, 导致克隆漏洞迅速扩散。Synopsys《开源安全与风险分析报告》指出, 在其检查的 1 250 个商业代码库中, 70% 的代码源自开源项目, 其中 75% 至少存在一个已知开源漏洞<sup>[14]</sup>。可见, 基于代码相似性的漏洞检测已成为保障软件供应链安全的现实需求。

## (3) 相关综述

已有综述工作从不同侧面梳理了相似性与深度学习在漏洞检测中的进展: 2019 年, 李珍等<sup>[15]</sup>系统回顾了基于代码相似性的漏洞检测方法; 2020—2021 年, 李韵等<sup>[16]</sup>、Lin 等<sup>[17]</sup>、Chakraborty 等<sup>[18]</sup>相继从深度学习视角调研通用软件漏洞检测技术; 2022—2023 年, Steenhovek 等<sup>[19]</sup>、Lipp 等<sup>[20]</sup>对多种深度学习漏洞检测模型开展了实证评估, 于颖超等<sup>[21]</sup>专门分析了 2015—2020 年面向二进制代码的 34 种检测方法; Nong 等<sup>[22]</sup>、邓泉等<sup>[23]</sup>进一步比较了 2021 年及之前提出的基于深度学习的源代码漏洞检测方案。

前人综述均聚焦 2022 年及以前的文献, 且大多仅关注深度学习方法; 除李韵等<sup>[16]</sup>外, 其余研究均将源代码与二进制代码方法割裂讨论。本文面向 2018—2024 年最新进展, 系统梳理跨模态、源代码及二进制代码场景下的传统方法与深度学习方法; 从数据集、预处理、代码表征 3 个维度对基于代码相似性的技术路线进行特征分析与总结, 并进一步阐述该领域面临的挑战与未来研究方向。

## 1 漏洞样本数据集

漏洞样本: 包含漏洞代码片段的完整函数或文件。

漏洞样本数据集: 由数量达到一定规模的漏洞样本构成, 涵盖多种漏洞类型及上下文信息,

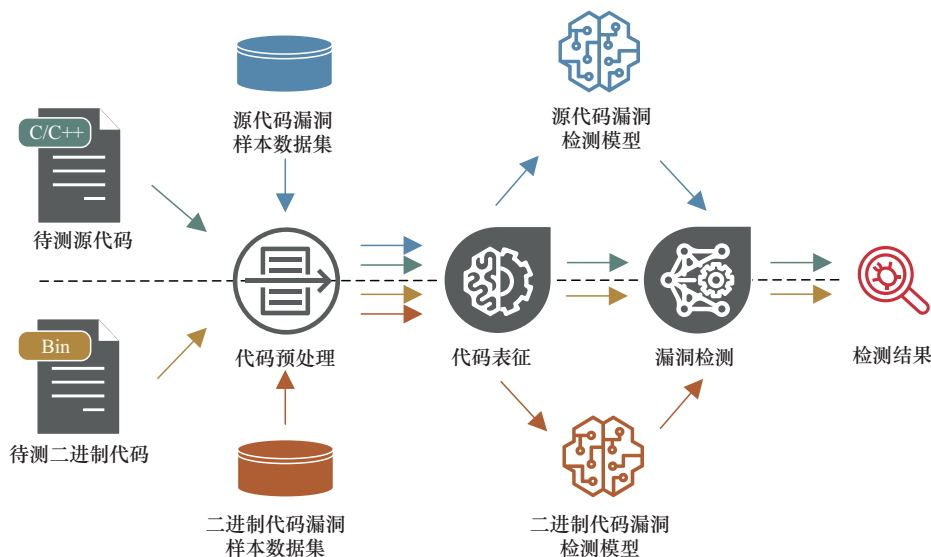


图2 基于代码相似性的漏洞检测通用流程

可用于训练与评估各类漏洞检测模型。

**源代码漏洞样本数据集：**以C、C++、Java等高级语言编写并公开标注的漏洞代码集合，代表数据集有SARD、Big-Vul、Devign等。

**二进制漏洞样本数据集：**受指令集架构(x86/ARM/MIPS)、编译器家族(GCC/Clang/MSVC)及优化等级(O0~O3)多重因素影响，目前尚无统一、公开的通用数据集。学术界普遍采用两类方案：小规模闭源真实环境数据集；将现有源代码漏洞样本按目标平台/编译器/优化等级交叉编译后生成的合成数据集。

### 1.1 数据集的分类

依据来源不同，漏洞样本数据集可分为：基于专家经验的数据集和真实环境的数据集。

#### (1) 基于专家经验的数据集

基于专家经验的数据集通过人工方式构造，典型代表为SARD<sup>[24]</sup>。该测试集由SAMATE项目维护，包含自动生成、人工合成及学术研究提供的带漏洞源代码，涵盖C、C++、C#、Java等高级语言，对应150余种通用弱点枚举(common weakness enumeration, CWE)。每个函数被标注为good(无漏洞)、bad(含漏洞)或mixed(同时给出漏洞与修补版本)。图3展示了源代码漏洞样本示例，SARD来源的漏洞样本CWE121\_01(基于堆栈的缓冲区溢出)示例如图3(a)所示，在CWE121\_Stack\_Based\_Buffer\_Overflow\_01\_bad函数中，字符数组data被初始化为大于字符数组dest长度缓冲区，这将使SNPRINTF函数在执行时产生可能的缓冲区溢出漏洞。在CWE121\_Stack\_Based\_Buffer\_Overflow\_01\_good函数中，字符数组data初始化长度小于字符数组dest，表明缓冲区溢出漏洞被修补。

#### (2) 真实环境的数据集

真实环境数据集主要来源于美国国家漏洞库<sup>[25]</sup>(national vulnerability database, NVD)，NVD是全球最大的公开漏洞情报源，与通用漏洞披露(common vulnerabilities and exposures, CVE)系统<sup>[26]</sup>同步，为每个漏洞分配唯一CVEID。然而，NVD仅提供有限的元数据，往往只有来源于Github<sup>[27]</sup>的开源项目(OSS)漏洞包含漏洞以及修补代码超链接信息。因此，研究者通常利用

爬虫从GitHub获取带CVE ID的开源项目漏洞文件及其对应补丁，构建真实源码漏洞样本。NVD来源的漏洞样本CVE-2020-1967示例如图3(b)所示，CVE-2020-1967是开源软件OpenSSL的一个空指针取消引用漏洞，通常会导致软件进程崩溃或者失败。在tls1\_check\_sig\_alg函数中，对指针sigalg是否为空判断的缺失，将导致此漏洞，增加指针判断语句sigalg!=NULL将修补此漏洞。

```

1 void CWE121_Stack_Based_Buffer_Overflow_01_bad()
2 {
3     ...
4     data = dataBuffer;
5     memset(data, 'A', 100-1);
6     data[100-1] = '\0';
7     {
8         char dest[50] = "";
9         SNPRINTF(dest, strlen(data), "%s", data);
10    ...
11 void CWE121_Stack_Based_Buffer_Overflow_01_good()
12 {
13    ...
14    data = dataBuffer;
15    memset(data, 'A', 100-1);
16    data[50-1] = '\0';
17    {
18        char dest[50] = "";
19        SNPRINTF(dest, strlen(data), "%s", data);
20    ...

```

(a) CWE121\_01

```

1 static int tls1_check_sig_alg(SSL *s, X509 *x, int default_nid)
2 {
3     sigalg = use_pc_sigalgs
4     ? tls1_lookup_sigalg(s->s3.tmp.peer_cert_sigalgs[i])
5     : s->shared_sigalgs[i];
6 - if (sig_nid == sigalg->sigandhash)
7 + if (sigalg != NULL && sig_nid == sigalg->sigandhash)
8     return 1;
9
10

```

(b) CVE-2020-1967

图3 源代码漏洞样本示例

NVD<sup>[25]</sup>样本除标注漏洞行外，还保留其前后上下文代码；对库函数/API调用类漏洞，这些上下文提供了关键的调用关系与数据流信息，可显著提升检测精度。相比之下，SARD<sup>[24]</sup>样本虽然漏洞特征明确，但上下文信息有限，难以满足真实场景下大规模代码的需求，也限制了模型的泛化能力。

### 1.2 数据集粒度与规模

数据集粒度与规模直接影响漏洞检测模型的效能与泛化能力。本文所调研的73.3%文献明确给出了数据集信息，其采用的样本粒度分为文件、函数、切片3类。由于来源与类型差异，跨

数据集简单横向比较并无意义。为此, 本文从“同粒度规模最大”和“被应用次数最多”两个维度遴选典型通用数据集, 并单独列出独立数据集研究。表1汇总了各代表集的核心指标: CWE列出覆盖的通用缺陷枚举数量; 规模列仅统计含漏洞样本数(不含良性/修补样本); “—”表示原文未披露; “\*”表示混合数据集。

现有源代码漏洞检测文献情况如下。

#### (1) 规模

切片级: AutoVAS<sup>[28]</sup>使用 14 194 条 NVD 切片, 为目前最大。

文件级: AMPLE<sup>[29]</sup>含 22 278 个 NVD 文件, 数量居首。

函数级: MVP<sup>[30]</sup>提供 34 378 个 NVD 漏洞函数, 规模最大。

混合集: FUNDED<sup>[31]</sup>构建 150 950 个漏洞函数的 SARD+NVD 混合集, 其中 SARD 函数超过 13 万。

#### (2) 通用性

SARD<sup>[24]</sup>原始数据集被 21 篇文献(28%)用作基准; Devign<sup>[32]</sup>(FFmpeg+Qemu)与 Big-Vul<sup>[33]</sup>分别被 13 篇(17.3%)和 12 篇(16%)文献采用; Reveal<sup>[34]</sup>(Chrome+Debian)亦被 12 篇(16%)文献用于性能比较。

独立数据集按样本形态可分为两类。①同源源代码集: Wang 等<sup>[35]</sup>通过特定 URL 爬取 1999—2019 年 GitHub C/C++ 项目, 获 4 076 对漏洞/补

丁, 经最近链接搜索与过采样扩展为 23 742 条的 PatchDB 并开源; Big-Vul<sup>[33]</sup>仅爬 GitHub 得到 3 754 个漏洞、覆盖 91 种 CWE; Chen 等<sup>[36]</sup>发布含 18 945 个漏洞函数、覆盖 150 种 CWE 的数据集; Zhou 等<sup>[37]</sup>基于 SARD 构造含 Token、AST 等多表征的样本集。②同源二进制集: ROMEO<sup>[38]</sup>将 SARD 的 C/C++ 源码经多编译器/优化等级编译后反汇编为汇编代码, 保留链接时间优化等指令信息, 适用于源代码不可见的黑盒分析。

实证研究方面。Lipp 等<sup>[20]</sup>在 27 个项目(115 万行代码、192 个 CVE)上评估发现, 92% 的漏洞集中在同一函数内, 确认函数粒度是 CVE 数据集的合适抽象层次。Steenhoek 等<sup>[19]</sup>进一步指出, 数据集存在“规模效应”: 仅需 1 134 个漏洞函数即可使模型达到良好性能( $F1 \approx 70\%$ ); 随后规模扩大 10 倍,  $F1$  仅提升约 5%。

综上, 函数粒度被证实为最优样本单元, 当前数据集规模两极分化, 构建大规模、多类型数据集仍是研究重点。与此同时, 随着样本量增加, 模型性能呈“先快后慢”的边际效应增长趋势。

### 1.3 数据集的平衡性

一个平衡的数据集, 其中的漏洞样本和修补样本在数量和规模上通常处于平衡状态。平衡的数据集将减少过拟合和欠拟合、缓解检测偏见问题, 并有助于增强模型的泛化能力。20% 的现有方法提及并解决了数据集的平衡性问题, 其中

表 1 典型漏洞样本数据集的规模情况

| 文献类型  | 代码类型  | 文献                         | 年份   | 数据集来源     | 粒度 | 公开性 | CWE/种 | 规模/个           |
|-------|-------|----------------------------|------|-----------|----|-----|-------|----------------|
| 漏洞检测  | 源代码   | AutoVAS <sup>[28]</sup>    | 2021 | NVD       | 切片 | 公开  | 98    | 14 194         |
|       |       | AMPLE <sup>[29]</sup>      | 2023 | NVD       | 文件 | 公开  | 91    | 22 278         |
|       |       | MVP <sup>[30]</sup>        | 2020 | NVD       | 函数 | —   | —     | 34 378         |
|       |       | Devign <sup>[32]</sup>     | 2019 | NVD       | 函数 | 公开  | —     | 27 318         |
|       |       | Reveal <sup>[34]</sup>     | 2021 | NVD       | 函数 | 公开  | —     | 22 734         |
|       |       | Tao 等 <sup>[39]</sup>      | 2023 | SARD      | 切片 | 公开  | 12    | 48 648         |
|       |       | FUNDED <sup>[31]</sup>     | 2021 | SARD+NVD* | 函数 | 公开  | —     | <b>150 950</b> |
| 独立数据集 | 二进制代码 | BinVulDet <sup>[40]</sup>  | 2023 | SARD      | 文件 | —   | 55    | 51 221         |
|       | 源代码   | PatchDB <sup>[35]</sup>    | 2021 | NVD       | 切片 | 公开  | —     | 4 076 (23 742) |
|       |       | Big-Vul <sup>[33]</sup>    | 2020 | NVD       | 函数 | 公开  | 91    | 3 754          |
|       |       | DiverseVul <sup>[36]</sup> | 2023 | NVD       | 函数 | 公开  | 150   | 18 945         |
|       |       | MVDSC-C <sup>[37]</sup>    | 2022 | SARD      | 函数 | —   | —     | 16 142         |
|       | 二进制代码 | ROMEO <sup>[38]</sup>      | 2023 | SARD      | 函数 | 公开  | 91    | <b>99 298</b>  |

Jeon 等<sup>[28]</sup>、陈小全等<sup>[41]</sup>均采用基于 SMOTE<sup>[42]</sup>的过采样方法 SMOTEENN<sup>[43]</sup>或 GraphSMOTE<sup>[44]</sup>来进行数据集平衡。VulChecker<sup>[45]</sup>采用漏洞样本等比采样的方式来平衡数据集。

具体来说, SMOTE 过采样算法通过在漏洞样本(少数类)和其随机最近的漏洞邻居之间进行插值来构建新的合成样本。SMOTEENN 是对 SMOTE 的改进, 增加了 ENN 欠采样。GraphSMOTE 则先用 GNN 学习节点特征, 执行 SMOTE 过采样后用 GNN 进行分类, 确定合适少数类样本, 在性能上优于 SMOTE、SMOTEENN 以及边界线-SMOTE。但由于该算法需要进行多次迭代和计算, 因此在处理大规模有向图数据时仍然面临效率问题。

从 Steenhoek 等<sup>[19]</sup>对数据集的实证研究可知, 让漏洞检测模型达到同一性能, 所需要的不平衡数据集规模取决于少数类(通常是漏洞样本类而非修补样本类)的规模, 即如果少数类仅占总数的 10%, 那么不平衡数据集规模通常是少数类对应的平衡数据集的 10 倍。因此, 平衡的数据集

在增强泛化能力的基础上, 还可降低检测模型对于数据集规模的需求, 提高机器学习类模型的学习效率。

## 2 代码预处理

代码预处理主要包括代码清理与标准化、反汇编与中间语言表示, 旨在增强检测模型理解代码数据的有效性。在图 2 中, 对于输入数据(待测源代码/二进制代码、源代码/二进制代码漏洞样本), 进行代码预处理将为代码表征、漏洞检测环节提供高信息量的有效数据。

### 2.1 代码清理与标准化

代码清理通常指删除待测代码中的注释、换行、字符串等冗余代码。标准化指将待测代码中的函数名、变量名等进行统一标准化命名, 或将所有符号进行规范化等。现有方法的代码清理与标准化见表 2。

代码清理方面, VulDeePecker<sup>[47]</sup>、AutoVAS<sup>[28]</sup>对非 ASCII 字符、注释进行删除, 部分方法对空行、空格、换行符、制表符进行删除, VulBERTa<sup>[50]</sup>仅删除注释。需要注意的是, 在源

表 2 现有方法的代码清理和标准化

| 检测方法                         | 检测代码  | 代码清理             | 标准化                                                 |
|------------------------------|-------|------------------|-----------------------------------------------------|
| BugGraph <sup>[46]</sup>     | 跨模态   | 空格, 换行符, 制表符, 注释 | 源代码+GCC/LLVM->二进制代码                                 |
| Tao 等 <sup>[39]</sup>        | 跨模态   | 原始代码, 未作清理       | 汇编与源代码之间指令对齐, 寄存器对齐                                 |
| VulDeePecker <sup>[47]</sup> | 源代码   | 非 ascii 字符, 注释   | 变量名->VAR, 函数名->FUN                                  |
| AutoVAS <sup>[28]</sup>      | 源代码   | 同 VulDeePecker   | 变量名->VAR, 函数名->FUN, 序列等长补齐                          |
| CCAligner <sup>[48]</sup>    | 源代码   | 原始代码, 未作清理       | 源代码->TXL, 标识符->Token                                |
| MVP <sup>[30]</sup>          | 源代码   | 空行, 制表符, 注释, 花括号 | 形式参数->PARAM, 变量名->VAR, 字符串->STRING                  |
| V0Finder <sup>[49]</sup>     | 源代码   | 空格, 换行符, 注释      | 字符统一转小写                                             |
| VulBERTa <sup>[50]</sup>     | 源代码   | 注释               | 标识符->Token, 序列等长补齐                                  |
| MOVERY <sup>[51]</sup>       | 源代码   | 空格, 注释           | 字符转成小写, 变量名->DNAME, 变量类型->DTYPE, 被调用函数名->FCALL      |
| PalmTree <sup>[52]</sup>     | 二进制代码 | 原始代码, 未作清理       | 字符串->str, 常量->addr                                  |
| Asteria <sup>[53]</sup>      | 二进制代码 | 常量和字符串           | 未标准化                                                |
| HAN-BSVD <sup>[54]</sup>     | 二进制代码 | 原始代码, 未作清理       | 跳转地址->JA, 整型值->VAL, 特殊寄存器->SR/GR, 内存偏移->OFFSET 等    |
| jTrans <sup>[55]</sup>       | 二进制代码 | 原始代码, 未作清理       | 字符串->str, 常量->const, 内部函数调用->funtion, 跳转对->JUMP_XXX |
| VulHawk <sup>[56]</sup>      | 二进制代码 | 未执行指令            | 操作数->16 类根操作数                                       |
| VulANalyzeR <sup>[57]</sup>  | 二进制代码 | 字符串, 注释, 文件名     | 未标准化                                                |

代码方法中, 由于字符串在源代码中的定位作用, 因此未涉及对字符串的删除, 而二进制代码方法 Asteria<sup>[53]</sup>和 VulANalyzeR<sup>[57]</sup>则均对字符串进行了删除。

代码标准化方面, 55.6%的方法对变量名进行归一化, 即将其映射为 VAR\_0、VAR\_1 等形式。需注意, 该操作通常仅适用于源代码, 因为汇编级二进制代码基本不存在变量名符号信息。此外, 源代码标准化规则还包括: 函数名→FUN、字符串→STRING、形参→PARAM、字母转小写、序列等长补齐等。二进制代码侧则采用: 字符串→str、立即数→addr、跳转目标→JA 等缩写规范。跨模态研究中, Tao 等<sup>[39]</sup>通过指令-寄存器对齐实现汇编与源码的同步表示。图 4 给出 VulDeePecker<sup>[47]</sup> (源码侧) 与 jTrans<sup>[55]</sup> (二进制侧) 的清理-标准化示例: 红色区域为标准化结果, 绿色区域为已清理内容。

代码清理旨在降低检测开销。空格、换行符、制表符等符号在检测中通常属于无效字符; 对于二进制代码, 不同汇编输出的字符串和常量亦存在较大差异。此外, 未被执行的死代码对检测模型而言属于干扰噪声, 应一并剔除。

代码标准化的目的在于缓解词汇表外 (out-of-vocabulary, OOV) 问题。OOV 问题指输入符号在词典或预训练模型中不存在、导致无法识别的情况。不同源代码的变量名、函数名差异显著, 若缺乏标准化, 检测模型需处理的词项数量将大幅增加; 一旦超出处理能力, 就会引发 OOV 并延长处理时间。

## 2.2 反汇编与中间语言表示

反汇编与中间语言表示用于转换代码形态。

反汇编是指借助逆向工具将二进制机器码转换为汇编语言; 中间语言表示则通过逆向工具或中间语言编译器将汇编语言/源代码转换为中间表示 (intermediate representation, IR)。反汇编主要应用于二进制代码方法。表 3 汇总了现有方法的反汇编与中间语言表示。

表 3 现有方法的反汇编与中间语言表示

| 检测方法                                                                       | 反汇编          | 中间语言表示       |
|----------------------------------------------------------------------------|--------------|--------------|
| BinVulDet <sup>[40]</sup> , BugGraph <sup>[46]</sup> , 文献[53-55,59,62-66]等 | IDA pro      | —            |
| VulSeeker-Pro <sup>[67]</sup> , BinSeeker <sup>[68]</sup>                  | IDA pro      | PyVEX VEX-IR |
| ContraFlow <sup>[69]</sup> , VulChecker <sup>[45]</sup>                    | —            | LLVM IR      |
| MDSAE <sup>[59]</sup>                                                      | Capstone     | —            |
| PalmTree <sup>[52]</sup>                                                   | Binary Ninja | —            |
| VulHawk <sup>[56]</sup>                                                    | —            | IDA pro IR   |

BinVulDet<sup>[40]</sup>等 15 种方法使用 IDA Pro<sup>[58]</sup>对二进制机器码进行反汇编, 占全部反汇编方法的 78.9%。IDA Pro 为商业逆向工具, 具备较强的反汇编与反编译能力, 常被学术界和工业界用于二进制代码逆向分析、代码混淆、加解密及漏洞检测等任务。MDSAE<sup>[59]</sup>和 PalmTree<sup>[52]</sup>分别采用 Capstone<sup>[60]</sup>和 Binary Ninja<sup>[61]</sup>进行反汇编。Capstone 与 Binary Ninja 属于轻量级逆向工具, 在准确率、完整率和覆盖率方面均略逊于 IDA Pro。

然而, 即便通用逆向工具 IDA Pro, 也无法处理某些特殊编译的二进制文件, 例如加密文件、被插入花指令或经过代码混淆的文件、部分残缺的文件。此外, IDA Pro 在处理由部分编译器 (如 Clang 设置 PIC 选项) 生成的二进制文件

```

main(int argc,char **argv)
char *userstr;
userstr=argv[1];
test(userstr);
test(char *str)
int MAXSIZE=40;
char buf[MAXSIZE];
strcpy(buf,str);/*string copy*/

main(int argc,char **argv)
char *VAR1;
VAR1=argv[1];
FUN1(VAR1);
FUN1(char *VAR2)
int VAR3=40;
char VAR4[VAR3];
strcpy(VAR5,VAR2);

```

(a) VulDeePecker

```

test    rax, rax
js      0x40047C
mov     rdi, rbx
call    puts
xor     eax, eax
add     rsp, 20h
pop     rbx
retn

test    rax, rax
js      JUMP_20
mov     rdi, rbx
call    puts
xor     eax, eax
add     rsp, <const>
pop     rbx
retn

lea     rdi, str0
call    sub_400570
jmp     0x400476

test    rax, rax
js      JUMP_20
mov     rdi, rbx
call    puts
xor     eax, eax
add     rsp, <const>
pop     rbx
retn

lea     rdi, <str>
call    <function>
jmp     JUMP_14

```

(b) jTrans

图 4 代码清理及标准化示例

时,也无法分析其中的子分支函数。同时,IDA Pro作为递归下降反汇编器,仍面临效率挑战。为确保效率,IDA Pro基于特征算法区分数据与代码,并通过派生变量和函数名尽可能注释反汇编结果,从而最大限度减少原始机器码数量,显著增加符号化信息。尽管如此,IDA Pro仍无法将汇编代码完全还原,可能存在部分信息丢失或错误。

在中间语言表示方面,ContraFlow<sup>[69]</sup>与VulChecker<sup>[45]</sup>利用LLVM将源代码转换为IR,以生成前向和后向切片,在损失部分信息的同时实现跨语言检测;VulHawk<sup>[56]</sup>使用IDA Pro将二进制代码转换为IR以供后续表征;VulSeeker-Pro<sup>[67]</sup>与BinSeeker<sup>[68]</sup>采用PyVEX将二进制代码转换为VEX-IR进行模拟执行,以缓解静态检测的假阳性。

综上,反汇编与中间语言表示作为关键预处理环节,旨在将代码由专用形态转为通用形态,虽然可能带来信息损失、误差及额外开销,但能显著提升检测模型的通用性与适配性,以应对复杂代码类型和多样化检测需求。

### 3 代码表征与漏洞检测

代码表征作为基于代码相似性的漏洞检测关键环节,旨在通过具体表示方式对代码的语法、控制流、数据流等特征进行刻画,使检测模型可据此完成漏洞识别。综合现有研究<sup>[15-23]</sup>,表征方式通常包括:代码片段(切片、序列)、图(CFG、CPG<sup>[70]</sup>等)、树(AST等)、哈希(模糊哈希、局部敏感哈希等)。同时,机器学习类模型在这些表征基础上进行嵌入,如图嵌入、词嵌入、代码嵌入,将具体表示映射至特征空间,供神经网络学习理解。

漏洞检测模型,指在代码表征基础上,结合传统算法或机器学习与代码相似性技术形成的有效检测框架。其中机器学习模型一般包括前期嵌入模型、预训练任务和训练检测模型3部分,其效果将于下文分析。

本文根据文献<sup>[15-23]</sup>研究及近7年相关实际工作,将代码表征方式分为6类,具体见表4。

同时,依据表征方式差异,把现有工作划分为:基于代码片段、基于哈希、基于树、基于图、基于指标、基于混合表征的方法,如图5所示。涵盖源代码(source code, SC)方法、二进

制代码(binary code, BC)方法及跨模态方法(同时处于SC与BC)

表4 代码表征方式分类

| 表征方式 | 说明                  |
|------|---------------------|
| 代码片段 | 代码的直接片段表示           |
| 哈希   | 哈希算法生成的代码唯一映射散列值    |
| 树    | 代码语法特征的树形结构表示       |
| 图    | 代码控制流、数据流等特征的图形结构表示 |
| 指标   | 代码中各类元素的统计学特征       |
| 混合表征 | 混合代码不同类别特征的综合性特征表示  |

#### 3.1 基于代码片段的漏洞检测

基于代码片段的漏洞检测方法,指将代码片段嵌入向量后,利用相似性算法或机器学习模型计算/预测待测向量与漏洞样本向量的相似度,从而识别疑似漏洞片段。

代码片段是代码的直接片段表示,通常仅体现序列化词法特征,包括序列、切片、部件等形式;代码序列指有序代码元素构成的序列;代码切片指对序列进行特定规则提取的片段;代码部件由语义相关但未必连续的代码行组成,可反映行间关联。

本文从表征方式、机器学习技术、末端相似性检测、数据集、性能与特性等方面汇总基于代码片段的漏洞检测方法,具体信息见表5。因篇幅所限,表中仅列典型方法:数据集列中S代表SARD样本,N代表NVD样本;性能为文献自报精确率(Prec)、召回率(Rec)、F1值,且受数据集类型、规模及参数设置影响,带\*号数据仅供参考。

##### (1) 源代码方法

Li等<sup>[47]</sup>首次提出面向深度学习的漏洞样本数据集,并基于此设计VulDecPecker。该方法用代码部件将源代码中的关键行组织成语义关联的前向和后向片段,经代码标准化及Word2Vec<sup>[115]</sup>词嵌入后生成漏洞样本向量;随后采用双向长短期记忆网络(bi-directional long short-term memory, BLSTM)捕获前向和后向信息,最后通过softmax分类器完成漏洞检测。softmax核心函数如式(1)所示,其中 $X_i$ 为第 $i$ 个节点的输出值, $N$ 为输出节点的数量,即类别个数,通过softmax函

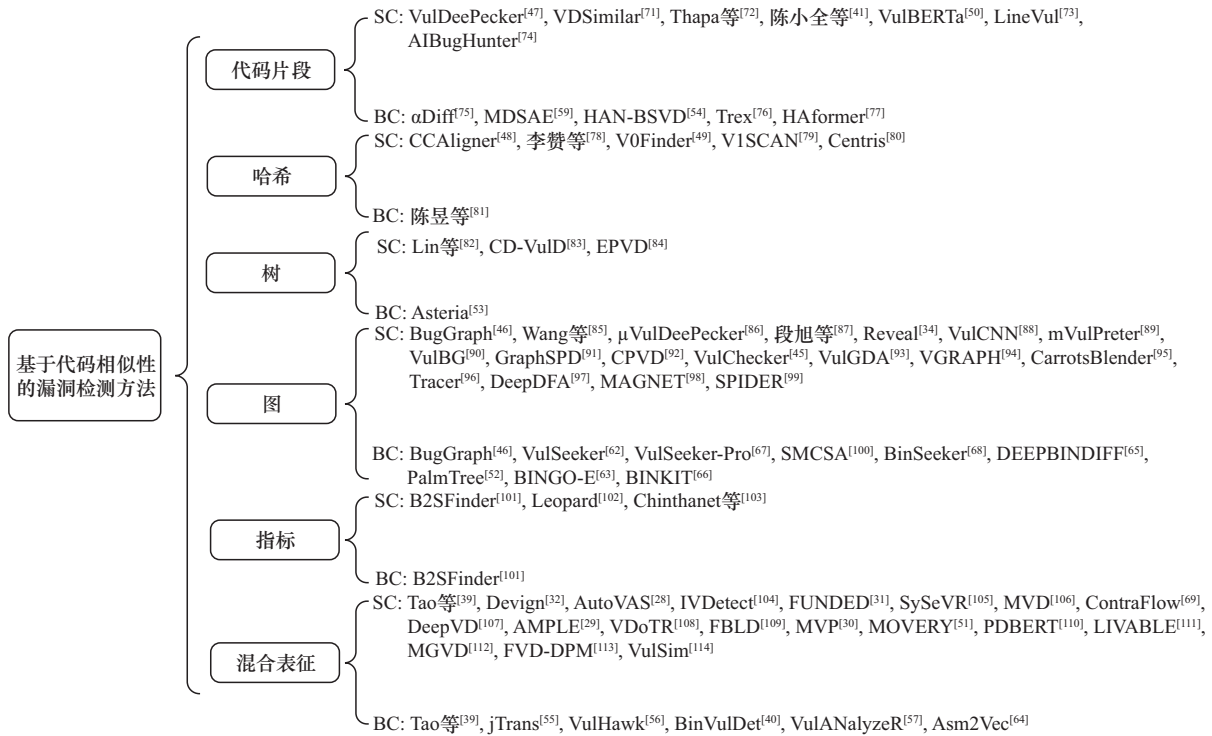


图5 现有工作分类

数将多分类输出转换为(0,1)区间且和为1的概率分布：

$$\text{softmax}(X_i) = \frac{e^{X_i}}{\sum_{n=1}^N e^{X_n}} \quad (1)$$

然而，VulDeePecker的代码部件仅部分反映执行顺序，缺乏更高层抽象与特征提取，难以完整刻画数据依赖和控制依赖，因此在逻辑型漏洞检测上仍面临挑战。

VDSimilar<sup>[71]</sup>在VulDeePecker基础上改进，用孪生网络<sup>[116]</sup>结合欧氏距离取代softmax分类器。孪生网络可将两组输入耦合，直接通过损失函数计算向量间相似度。对于向量 $A$ 和 $B$ ， $A_i$ 和 $B_i$ 分别是 $A$ 和 $B$ 的第 $i$ 个元素，采用欧氏距离及其相似度计算公式如下：

$$\text{Euclidean}(A, B) = \sqrt{\sum (A_i - B_i)^2} \quad (2)$$

$$\text{Sim}(A, B) = \frac{1}{1 + \text{Euclidean}(A, B)} \quad (3)$$

陈小全等<sup>[41]</sup>以全连接神经网络（fully connected neural network, FCNN）为检测模型，末端相似性检测采用通用余弦相似度。对于向量 $A$ 和 $B$ ，计算式为：

$$\text{Cos}(A, B) = \frac{A \cdot B}{|A| \times |B|} \quad (4)$$

Thapa等<sup>[72]</sup>、VulBERTa<sup>[50]</sup>、LineVul<sup>[73]</sup>和AIBugHunter<sup>[74]</sup>采用Transformer<sup>[117]</sup>及其变体BERT<sup>[118]</sup>、RoBERTa<sup>[119]</sup>进行前期嵌入。相较于Word2Vec，Transformer的自注意力机制可充分学习代码上下文信息，支持LineVul与AIBugHunter的行级漏洞检测，在CWE-77（中和不当）、CWE-119（限制不当）、CWE-190（整型溢出）等简单非逻辑型漏洞上表现更优。同时，Thapa等、VulBERTa与LineVul使用BERT的掩码语言模型（masked language model, MLM）作为预训练任务：随机掩盖输入语料中的部分代码元素，并通过上下文预测该元素，从而强化模型对相邻元素关联的理解。

## (2) 二进制代码方法

αDiff<sup>[75]</sup>是首个将深度学习用于跨版本二进制相似性及漏洞搜索的方法。其采用CNN对汇编代码进行嵌入，并利用孪生网络计算CNN嵌入向量的三特征相似度；通过联合学习函数内、函数间、模块间三组特征，缓解二进制在架构、编译器及优化级别上的差异。对于两个待测函数

表5 基于代码片段的典型漏洞检测方法

| 检测方法                          | 年份   | 检测代码 | 表征方式 | 机器学习技术             |     |              | 末端相似性检测  | 数据集                 | 性能     |        |        | 特性          |
|-------------------------------|------|------|------|--------------------|-----|--------------|----------|---------------------|--------|--------|--------|-------------|
|                               |      |      |      | 前期嵌入               | 预训练 | 训练检测         |          |                     | Pre    | Rec    | F1     |             |
| VulDecP-ecker <sup>[47]</sup> | 2018 | SC   | 代码部件 | Word2Vec           | —   | BLSTM        | softmax  | N+S                 | 0.920* | —      | 0.934* | 切片粒度        |
| VDSimilar <sup>[71]</sup>     | 2021 | SC   | 代码序列 | Word2Vec           | —   | BLSTM, 孪生网络  | 欧氏距离     | N: OpenSSL          | 0.918  | —      | 0.914  | —           |
| Thapa 等 <sup>[72]</sup>       | 2022 | SC   | 代码部件 | Word2Vec, BERT     | MLM | GPT-2        | softmax  | N: VulDecP-ecker    | 0.970  | 94.5   | 0.957  | 可扩展         |
| VulBERTa <sup>[50]</sup>      | 2022 | SC   | 代码序列 | RoBERTa            | MLM | MLP, TextCNN | 分类器      | N: VulDecP-ecker    | 0.958* | —      | 0.930* | 鲁棒          |
| LineVul <sup>[73]</sup>       | 2022 | SC   | 代码序列 | CodeBERT           | MLM | Transformer  | 排序       | N: Big-Vul          | 0.967* | 85.7*  | 0.910* | 行粒度, 可解释    |
| AlBugHunter <sup>[74]</sup>   | 2023 | SC   | 代码序列 | Transformer        | —   | CodeBERT     | softmax  | N: Big-Vul          | —      | —      | —      | 行粒度         |
| $\alpha$ Diff <sup>[75]</sup> | 2018 | BC   | 代码序列 | CNN                | —   | 孪生网络         | 三特征距离相似度 | N: CVE-2014-4877 等  | —      | 0.691* | —      | 跨架构-编译器-优化  |
| Trex <sup>[76]</sup>          | 2023 | BC   | 代码序列 | BLSTM, Transformer | MLM | MLP          | 余弦相似度    | N: OpenSSL          | 0.930* | —      | —      | 跨架构-优化, 抗混淆 |
| HAformer <sup>[77]</sup>      | 2024 | BC   | 代码序列 | BERT               | MLM | HAformer     | softmax  | N: CVE-2022-30065 等 | —      | —      | —      | 跨架构-编译器-优化  |

注: \*表示仅供参考。

$I_q$  和  $I_t$ ,  $\alpha$ Diff 使用了函数内距离  $D_1$ 、函数间距离  $D_2$ 、模块间距离  $D_3$  计算相似度, 其中  $\xi$  是在 (0,1) 内预定义的超参数, 用于抑制  $D_2(I_q, I_t)$  的影响:

$$D(I_q, I_t) = D_1(I_q, I_t) + (1 - \xi^{D_2(I_q, I_t)}) + D_3(I_q, I_t) \quad (5)$$

受卷积核感受野限制,  $\alpha$ Diff<sup>[75]</sup> 所用 CNN 更关注局部特征, 难以捕获长距离依赖; 而无论源代码或二进制代码, 函数间的控制与数据依赖常跨越较长距离, 故漏洞检测中 CNN 需与其他网络结合以弥补该缺陷。

HAN-BSVD<sup>[54]</sup> 融合 Text-CNN 与 BGRU 并引入注意力模块, 在 SARD 数据集上取得较好性能。

Trex<sup>[76]</sup> 通过强制执行二进制获取汇编指令序列的动态值, 并实施 MLM 预训练, 以强化模型对二进制定态执行近似语义的理解; 微调后, Trex 在跨架构、优化及混淆函数匹配任务中表现突出。

HAformer<sup>[77]</sup> 通过融合十六进制机器码序列与汇编指令序列, 缓解不同架构下二进制代码在反汇编及标准化时产生的信息损失, 适用于跨架构漏洞检测。

### (3) 基于代码片段的方法小结

一般流程: 代码→代码片段→嵌入向量→机

器学习/特征算法→相似性计算→漏洞检测。

优势及适用场景: 处理简单直接, 契合面向序列的神经网络, 检测效率高, 适合大规模代码检测。

弱点: 片段表示难以捕捉结构、语法、语义相似的漏洞特征, 依赖嵌入模型对序列化词法特征的建模能力; 所分析方法中 50% (6:12) 采用 Transformer 自注意力机制。

### 3.2 基于哈希的漏洞检测

基于哈希的漏洞检测方法, 指利用哈希算法结合漏洞特征进行漏洞检测的方法。哈希, 指哈希算法生成的代码唯一映射散列值, 具有指纹属性; 哈希算法作为一种压缩映射的代码表征方式, 直接将原始代码映射为固定长度哈希值, 以高度压缩形式表征代码的序列化词法特征。基于哈希的典型漏洞检测方法具体信息见表 6, 表格说明与表 5 一致。

#### (1) 源代码方法

CCAligner<sup>[48]</sup> 采用 e 编辑距离检测大间隙克隆, e 编辑距离定义为代码行插入/删除/替换的最小操作次数。在此基础上, CCAligner 使用 MurmurHash 算法将不同代码映射为不同哈希值并存储, 进而定义非对称 Dice 相似系数计算待测代

表 6 基于哈希的典型漏洞检测方法

| 检测方法                      | 年份   | 检测代码 | 表征方式        | 机器学习技术 |            | 末端相似性检测        | 数据集          | 性能     |       |        | 特性               |
|---------------------------|------|------|-------------|--------|------------|----------------|--------------|--------|-------|--------|------------------|
|                           |      |      |             | 前期嵌入   | 训练检测       |                |              | Pre    | Rec   | F1     |                  |
| CCAligner <sup>[48]</sup> | 2018 | SC   | Mur-murHash | —      | —          | 不对称 Dice 相似系数  | JDK 等        | 0.840* | —     | 0.890* | 跨语言, 行粒度, 可扩展    |
| 李赞等 <sup>[78]</sup>       | 2018 | SC   | HashPJW     | —      | —          | 余弦相似度          | N: FFmpeg 等  | —      | —     | —      | 切片粒度             |
| V0Finder <sup>[49]</sup>  | 2021 | SC   | TLSH        | —      | —          | TLSH           | N: Linux 等   | 0.990  | 97.0  | —      | 漏洞溯源             |
| V1SCAN <sup>[79]</sup>    | 2023 | SC   | TLSH        | —      | —          | 行比较算法          | N: ReactOS 等 | 0.960* | 91.0* | —      | 可扩展              |
| 陈昱等 <sup>[81]</sup>       | 2018 | BC   | LSH         | GloVe  | BGRU, 孪生网络 | 余弦相似度, LSH 相似度 | N: openssl 等 | —      | —     | —      | 跨架构-编译器-优化, 文件粒度 |

注: \*表示仅供参考。

码哈希与哈希库之间的相似度, 计算式如下。

$$\text{Sim}(m, n) = \frac{|W_m \cap W_n|}{\min(|W_m|, |W_n|)} \quad (6)$$

其中,  $W_m$  和  $W_n$  是代码块  $m$  和  $n$  的代码窗口集,  $\min$  函数将对较小代码块中包含的窗口数量的相似度得分进行标准化。

李赞等<sup>[78]</sup>引入代码切片技术去除漏洞样本中的无关代码, 通过代码标准化及 HashPJW 算法将漏洞特征代码映射为固定长度哈希值以计算相似度。鉴于 HashPJW 算法本身强度较低, 生成的哈希值更易遭受碰撞攻击, 可能对搜索结果的准确率造成潜在负面影响。

Centris<sup>[80]</sup>对代码进行冗余消除和编码分割以区分开源组件, 再通过局部敏感哈希 (locality sensitive Hash, LSH) 签名匹配, 可缓解因代码分割不准确导致的假阳性问题。

V0Finder<sup>[49]</sup>是检测漏洞零 (vulnerability zero, VZ) 的框架, 通过构建漏洞传播图并结合趋势科技局部敏感哈希 (trend micro locality sensitive Hash, TLSH) 追踪漏洞传播, 发现潜在克隆漏洞。

V1SCAN<sup>[79]</sup>是 Woo 等<sup>[80]</sup>基于 V0Finder<sup>[49]</sup>改进的 1-day 漏洞精确检测方法, 沿用 Centris 和 Ctags<sup>[120]</sup>完成组件识别, 并利用 TLSH 检测漏洞。V1SCAN 将 V0Finder 的假阴性率由 55% 降至 9%, 具备可扩展性。为缓解 TLSH 对微小差异的忽略, Woo 等<sup>[80]</sup>对漏洞代码进行细粒度行级处理, 显著提升检测性能。

## (2) 二进制代码方法

作为一种大规模跨平台同源二进制文件检索方法, 陈昱等<sup>[81]</sup>采用 GloVe 将多架构、多编

译器及多优化等级的机器码漏洞样本进行词嵌入, 通过 BGRU 与孪生网络框架计算余弦相似度以判定二进制文件漏洞同源性, 并利用 LSH 算法实现同源文件快速检索。然而, 该方法检测粒度为文件级, 无法精确定位代码文件内的漏洞位置。

## (3) 基于哈希的方法小结

一般流程: 代码→哈希算法→哈希值→相似性计算→漏洞检测。

优势及适用场景: 得益于哈希算法的高效性, 正向计算开销小, 适用于快速、大批量的漏洞搜索任务。

弱点: 表征方式仅反映代码的直接顺序排列特征; 66.7% (4:6) 的检测方法未采用机器学习技术, 仅依赖匹配算法。检测性能与匹配算法阈值设置强相关: 阈值过低易因无法区分漏洞样本与修补样本的微小差异而产生误报; 阈值过高则使模型泛化能力大幅下降。李赞等<sup>[78]</sup>、V0Finder<sup>[49]</sup>均设置极高阈值以保持高精确率, 此时模型基本不具有泛化性。

## 3.3 基于树的漏洞检测

基于树的漏洞检测方法, 一般指从漏洞样本数据集提取 AST 及其派生树形结构, 并利用机器学习模型学习节点与边特征, 实现漏洞检测。

AST 以层级细分树形结构反映代码词法、语法特征及部分控制流、数据流信息, 但不直接表示所有执行路径与数据依赖。

基于树的典型漏洞检测方法的具体信息见表 7, 表格说明与表 5 一致。

## (1) 源代码方法

Lin 等<sup>[82]</sup>提出一种异构深度学习漏洞检测框

表7 基于树的典型漏洞检测方法

| 检测方法                    | 年份   | 检测代码 | 表征方式 | 机器学习技术    |       |             | 末端相似性检测       | 数据集        | 性能     |        |       | 特性  |
|-------------------------|------|------|------|-----------|-------|-------------|---------------|------------|--------|--------|-------|-----|
|                         |      |      |      | 前期嵌入      | 预训练   | 训练检测        |               |            | Pre    | Rec    | F1    |     |
| Lin 等 <sup>[82]</sup>   | 2019 | SC   | AST  | Word2Vec  | —     | BLSTM       | RF            | S+N        | 0.860* | 0.860* | —     | —   |
| CD-VulD <sup>[83]</sup> | 2020 | SC   | AST  | Word2Vec  | 分层预训练 | BLSTM, MTLF | 马氏距离, LDA 分类器 | S+N        | 0.879  | 0.515  | 0.913 | 跨域  |
| EPVD <sup>[84]</sup>    | 2023 | SC   | AST  | CodeBERT  | —     | CNN, MLP    | MLP, softmax  | N: Big-Vul | 0.665  | 0.669  | 0.667 | 可扩展 |
| Asteria <sup>[53]</sup> | 2021 | BC   | AST  | Tree-LSTM | —     | 孪生网络        | softmax       | N: OpenSSL | —      | —      | —     | 跨架构 |

注：\*表示仅供参考。

架，使用两组BLSTM学习代码AST的词嵌入向量，并将其输入随机森林（random forest, RF）分类器进行分类。RF的计算式见式(7)。

$$S(x) = \sum_{i=1}^n w_i s_i(x) \quad (7)$$

其中， $w_i$ 表示每个决策树的权重， $s_i(x)$ 表示每个决策树的输出值。

CD-VulD<sup>[83]</sup>同样以BLSTM作为特征学习核心，引入度量迁移学习框架<sup>[121]</sup>（metric transfer learning framework, MTLF）学习AST的跨域表示，最终通过马氏距离配合线性判别分析（linear discriminant analysis, LDA）完成漏洞检测。两个向量和之间的马氏距离计算式为：

$$d(Z_i, Z_j) = \sqrt{(Z_i - Z_j)^T M (Z_i - Z_j)} \quad (8)$$

尽管Lin等<sup>[82]</sup>、CD-VulD<sup>[83]</sup>认为BLSTM克服了传统RNN的梯度消失问题，但在处理超长序列时，BLSTM仍面临梯度消失挑战。

EPVD<sup>[84]</sup>提出基于AST构建CFG的方法，设计贪心算法从CFG中选择典型执行路径，使用CodeBERT<sup>[122]</sup>将路径嵌入向量，再利用CNN学习并用MLP分类。与 $\alpha$ Diff<sup>[75]</sup>类似，EPVD<sup>[84]</sup>受CNN卷积核限制，难以检测含长距离执行路径的漏洞。

## （2）二进制代码方法

Asteria<sup>[53]</sup>提出AST编码方法，用Tree-LSTM<sup>[123]</sup>将标准化后的反汇编二进制代码编码为向量，再用孪生网络计算相似度。针对函数内联问题，Asteria<sup>[53]</sup>过滤汇编指令数低于阈值的被调用函数，并以被调用函数集大小校准相似度。

## （3）基于树的方法小结

一般流程：代码→树形结构→嵌入向量→机器学习→相似性计算→漏洞检测。

优势及适用场景：AST对语法特征表征全面，且提取技术成熟，信息损失少；泛化性好且适用于跨领域任务，如CD-VulD<sup>[83]</sup>支持跨域检测，Asteria<sup>[53]</sup>可跨ARM、x86、PowerPC架构检测二进制函数。

弱点：AST多级语法细分导致节点冗余，计算开销大。以Devign<sup>[32]</sup>（27 318个函数样本）为例，Joern提取的AST平均节点数为10 639个，而CFG平均节点数仅1 603个，约为AST的15.1%。

## 3.4 基于图的漏洞检测

基于图的漏洞检测方法，指从漏洞样本数据集提取图形结构，利用机器学习模型学习图的节点特征、边特征及结构关联特征，并借助神经网络实现有效漏洞检测；亦可结合图形结构特征与传统相似性算法完成检测。

图形结构是一种比树形结构更灵活的非线性结构：树形结构必须严格定义父子关系，而图形结构允许任意网状连接（包括环路和双向边）。在代码表征中，图形结构涵盖CFG、数据流图（data flow graph, DFG）、程序依赖图（program dependence graph, PDG）等，可准确刻画控制流与数据流；研究者在CFG、PDG基础上进一步设计出属性控制流图（attribute control flow graph, ACFG）、边程序依赖图（edge program dependence graph, ePDG）等，扩展了图形结构的特征范围与表达能力。

基于图的典型漏洞检测方法的具体信息见表8，表格说明与表5一致。

### （1）跨模态方法

BugGraph<sup>[46]</sup>是一种基于图三元组损失网络（graph triplet-loss network, GTN）的跨模态漏洞检测方法。该方法以汇编代码作为源代码与二进制代码的中间表示：先将二进制代码反汇编为汇

表 8 基于图的典型漏洞检测方法

| 检测方法                        | 年份   | 检测代码   | 表征方式   | 机器学习技术         |                 | 末端相似性检测            | 数据集            | 性能     |        |        | 特性               |
|-----------------------------|------|--------|--------|----------------|-----------------|--------------------|----------------|--------|--------|--------|------------------|
|                             |      |        |        | 前期嵌入           | 训练检测            |                    |                | Pre    | Rec    | F1     |                  |
| BugGraph <sup>[46]</sup>    | 2021 | SC, BC | ACFG   | GNN            | GTN             | 余弦相似度              | N: SNNS 等      | —      | —      | —      | 跨编译器-优化          |
| VGRAPH <sup>[94]</sup>      | 2020 | SC     | CPG    | —              | —               | 三元组匹配算法            | N: Linux 等     | 0.940* | 0.770* | 0.850* | 行粒度              |
| VulCNN <sup>[88]</sup>      | 2022 | SC     | PDG    | Sent2Vec       | CNN             | 分类器                | N+S            | —      | —      | —      | 可扩展, 可解释         |
| mVulPreter <sup>[89]</sup>  | 2022 | SC     | PDG    | Sent2Vec       | GGNN            | 排序                 | N: Big-Vul     | 0.516* | 0.665* | 0.581* | 可扩展, 可解释, 切片粒度   |
| CPVD <sup>[92]</sup>        | 2023 | SC     | CPG    | Word2Vec       | GAT, DANN, FCNN | 分类器                | N+S            | 0.783  | 0.832  | 0.811  | 鲁棒, 跨项目          |
| VulChecker <sup>[45]</sup>  | 2023 | SC     | ePDG   | Structure2-Vec | DNN             | 分类器                | N+S            | —      | —      | —      | 行粒度, 可扩展         |
| MAGNET <sup>[98]</sup>      | 2024 | SC     | CPG    | Word2Vec       | GAT             | 分类器                | N: Big-Vul     | 0.227  | 0.389  | 0.287  | —                |
| BINGO-E <sup>[63]</sup>     | 2018 | BC     | 3D-CFG | —              | —               | Jaccard相似度, SIM相似度 | N: Adobe 等     | —      | —      | —      | 鲁棒, 可扩展, 跨编译器-优化 |
| DEEPBINDIFF <sup>[65]</sup> | 2020 | BC     | CFG    | Word2Vec       | —               | k-Hop 算法           | N: Coreutils 等 | 0.843  | 0.795  | —      | 跨编译器-优化          |

注: \*表示仅供参考。

编代码, 提取 ACFG, 再用 GNN 将其嵌入 GTN 进行训练。ACFG 在 CFG 基础上增加了变量值等属性信息, 可更全面描述程序执行过程。同时, BugGraph<sup>[46]</sup>利用 Origin<sup>[124]</sup>识别待测二进制代码的编译选项, 实现跨编译器、跨优化的检测目标。

## (2) 源代码方法

CFG 类型。CFG 用于表征代码的控制流。Wang 等<sup>[85]</sup>在 CFG 基础上增加数据依赖图 (data dependence graph, DDG) 来表征代码的数据依赖关系。

DDG 类型。DDG 侧重表征代码的数据依赖特征。Tracer<sup>[96]</sup>将源代码 DDG 的静态污点信息与漏洞数据集进行相似性比较, 用于搜索整数溢出、格式化字符串、命令注入等数据流相关漏洞。

PDG 类型。PDG 用于表征代码的控制依赖和数据依赖特征。 $\mu$ VulDeePecker<sup>[86]</sup>是 VulDeePecker<sup>[47]</sup>的改进, 其为每个训练样本生成系统依赖图 (system dependence graph, SDG), SDG 包含一组 PDG。VulCNN<sup>[88]</sup>、mVulPreter<sup>[89]</sup>、VulChecker<sup>[45]</sup>以及 SPIDER<sup>[99]</sup>也采用 PDG 进行代码表征。需要注意的是, VulChecker 将源代码编译成 LLVM IR, 生成更接近底层表示的 ePDG, 用

于解决源代码在数据类型、临时变量和存储位置方面的歧义。

CPG 类型。CPG 作为一种综合图, 同时包含 AST、CFG、PDG 信息, 因此在表征词法、语法、控制流、数据流方面具有显著优势, 但也面临高于 AST 的计算开销。VGRAPH<sup>[94]</sup>、段旭等<sup>[87]</sup>、Reveal<sup>[34]</sup>、mVulPreter<sup>[89]</sup>、GraphSPD<sup>[91]</sup>、CPVD<sup>[92]</sup>、VulGDA<sup>[93]</sup>以及 MAGNET<sup>[98]</sup>均以 CPG 作为主要表征方式。其中 VGRAPH 通过三元组匹配算法使漏洞样本与待测代码进行 CPG 子图匹配, 在泛化性方面存在挑战。

其他图类型。CarrotsBlender<sup>[95]</sup>构建漏洞调用图 (vulnerability call graph, VCG) 来追踪漏洞相关函数的调用关系, 旨在确定漏洞函数在空间维度 (文件粒度) 中的分布。给定一个 VCGG  $(F, V, \delta)$ , 其中  $F$  是一个非空的有限函数集,  $V$  是一个非空的有限的脆弱函数集, 其中  $V \subseteq F$ ,  $\delta \subseteq F \times F$  是函数调用。对于两个漏洞函数  $v_1$  和  $v_2$ , 其中  $f_i \in F$ ,  $f_1 = v_1$ ,  $f_n = v_2$ , 且  $(f_i f_{i+1}) \in \delta$  或者  $(f_{i+1} f_i) \in \delta$ , 计算距离公式如下:

$$d(v_1, v_2) = \arg \min_n |\{f_1, \dots, f_n\}| - 1 \quad (9)$$

## (3) 二进制代码方法

CFG 类型。VulSeeker<sup>[62]</sup>、VulSeeker-Pro<sup>[67]</sup>以及 SMCSA<sup>[100]</sup>均从二进制汇编代码提取 CFG

嵌入向量, 利用余弦相似度 (式(4)) 进行相似漏洞的代码检测。DEEPBINDIFF<sup>[65]</sup>提出一种  $k$ -hop 贪婪匹配算法对基本块级的 CFG 嵌入向量进行漏洞搜索。BINKIT<sup>[66]</sup>通过 CFG 特征值的相对差值计算函数相似度。BINGO-E<sup>[63]</sup>、BinSeeker<sup>[68]</sup>则用 Jaccard 距离衡量待测代码与漏洞样本间的相似度, 并通过仿真消除函数内联影响, 增强模型的鲁棒性。对于集合  $A$  和  $B$ , Jaccard 相似度计算公式如下:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (10)$$

DFG 类型。PalmTree<sup>[52]</sup>在提取 DFG 的基础上设计上下文窗口预测 (context window prediction, CWP)、定义使用预测 (defuse prediction, DUP) 预训练任务, 在真实环境数据集上取得良好检测性能。

#### (4) 基于图的方法小结

一般流程: 代码→图形结构→嵌入向量→机器学习/特征算法→相似性计算→漏洞检测。

优势及适用场景: 图形结构依图类型可分别表征代码的语法、控制流、数据流、依赖或函数调用关系, 覆盖率高, 适用于检测结构、语义复杂漏洞。基于图的方法中, 23.1% (6:26) 具备可扩展性或鲁棒性, 两类占比均高于其他表征方式。

弱点: 图形结构信息丰富且复杂, 时空开销大, 大规模检测面临挑战; 提取工具 (如 Joern) 的系统误差可能引入图信息偏差与遗漏; 图神经网络潜在的过度平滑亦会使部分特征失效或噪声化, 影响检测性能。

### 3.5 基于指标的漏洞检测

指标, 即代码中各类元素的统计学特征, 具体指计算漏洞特征指标得分或漏洞样本匹配指标得分。基于指标的方法通常结合代码相似性技术与统计学指标检测漏洞。

基于典型的漏洞检测方法的具体信息见

表9, 表格说明与表5一致。

#### (1) 跨模态方法

B2SFinder 是 Yuan 等<sup>[101]</sup>提出的二进制到源代码匹配方法, 选用字符串、输出函数、全局字符数组、全局整数数组、全局枚举数组、If/Else、Switch/Case 7 项指标, 并设计基于 S-IDF (TF-IDF 变体) 的匹配算法, 用于检测开源软件中指标显著的克隆漏洞。

#### (2) 源代码方法

Leopard 由 Du 等<sup>[102]</sup>提出, 其使用两组指标: 复杂度指标 (函数、循环) 与漏洞指标 (依赖性、指针、控制结构), 通过计算得分判断漏洞。敏感性分析表明, 复杂度指标对 Leopard 贡献显著; 相比漏洞指标, 复杂度指标在整体相似度上提供粗粒度指向, 并缓解单独使用漏洞指标带来的误报。

Chinthanet 等<sup>[103]</sup>针对 JavaScript 提出基于物料清单的开源组件识别方法以检测克隆漏洞, 物料清单涵盖 Node.js 的包、模块、函数、类、方法及项目等指标, 旨在提升 JavaScript 开源组件识别的可靠性。

#### (3) 基于指标的方法小结

一般流程: 代码→指标→特征算法→相似性计算→漏洞检测。

优势及适用场景: 统计学特征, 具有直接性和全局性, 更适于漏洞检测辅助; 如 Leopard<sup>[102]</sup>的疑似漏洞排序可为其他方法提供有效辅助, 提升检测效率。

弱点: 指标设定主观且有限, 难以检测复杂类型漏洞; 如 B2SFinder<sup>[101]</sup>仅 7 项匹配指标, 无法覆盖复杂语义特征漏洞。

### 3.6 基于混合表征的漏洞检测

基于混合表征的漏洞检测方法, 通常选取代码片段、哈希、树、图等表征中的两种或多种进行综合性表征, 并结合机器学习模型或特征算法

表9 基于典型的漏洞检测方法

| 检测方法                       | 年份   | 检测代码   | 表征方式     | 机器学习技术 |     |      | 末端相似性检测 | 数据集          | 性能     |        |        | 特性   |
|----------------------------|------|--------|----------|--------|-----|------|---------|--------------|--------|--------|--------|------|
|                            |      |        |          | 前期嵌入   | 预训练 | 训练检测 |         |              | Pre    | Rec    | F1     |      |
| B2SFinder <sup>[101]</sup> | 2019 | SC, BC | 代码指标     | —      | —   | —    | 匹配算法    | N: Zlib 等    | 0.932* | 0.885* | 0.908* | 可扩展  |
| Leopard <sup>[102]</sup>   | 2019 | SC     | 复杂度、漏洞指标 | —      | —   | —    | 排序      | N: FFM-peg 等 | —      | —      | —      | 指标敏感 |

注: \*表示仅供参考。

实施漏洞检测。

基于混合表征的典型漏洞检测方法的具体信息见表 10, 表格说明与表 5 一致。

#### (1) 跨模态方法

Tao 等<sup>[39]</sup>以汇编代码为源-二进制中间表示, 设计跨模态细粒度特征对齐算法实现漏洞检测; 其对齐依赖带标注的编译过程, 而编译过程受架构、编译器、优化选项等多因素影响, 提升对其覆盖率仍面临挑战。

#### (2) 源代码方法

树+图类型。Devign<sup>[32]</sup>、FUNDED<sup>[31]</sup>、ContraFlow<sup>[69]</sup>、AMPLE<sup>[29]</sup>、FBLD<sup>[109]</sup>、PDBERT<sup>[110]</sup>采用树-图混合表征。具体地, Devign<sup>[32]</sup>提出节点连通序列 (node-connected sequence, NCS), 按代码顺序连接 AST 相邻节点以保留序列特征; ContraFlow<sup>[69]</sup>在自监督对比学习中利用静态值流 (static value-flow, SVF) 生成值流图 (value-flow

graph, VFG), 并用 Z3 模理论可满足性 (satisfiability modulo theories, SMT) 验证路径可行性; FBLD<sup>[109]</sup>提取漏洞语句的控制/数据依赖构建具有漏洞指向性的特征依赖图 (feature dependence graph, FDG); IVDetect<sup>[104]</sup>在 Token 级代码上提取 AST 与 PDG, 采用 FA-GCN 强化稀疏特征理解; PDBERT<sup>[110]</sup>设计 CDP 与 DDP 预训练任务, 引导模型学习细粒度控制与数据依赖特征。

哈希+树+图类型。MVP<sup>[30]</sup>、MOVERY<sup>[51]</sup>以签名化的代码特征为切入点, 提取多种树/图结构并生成签名, 便于快速匹配和搜索。

代码片段+树+图类型。SySeVR<sup>[105]</sup>、MVD<sup>[106]</sup>、DeepVD<sup>[107]</sup>、VDoTR<sup>[108]</sup>、LIVABLE<sup>[111]</sup>、FVD-DPM<sup>[113]</sup>以及 VulSim<sup>[114]</sup>均使用代码片段、树、图结构作为表征, 以增强模型对多类型特征的学习。MVD<sup>[106]</sup>引入调用图 (call graph, CG) 以捕获更全面的代码语义; DeepVD<sup>[107]</sup>使用 EFG (exception

表 10 基于混合表征的典型漏洞检测方法

| 检测方法                        | 年份   | 检测代码   | 表征方式               | 机器学习技术   |               |                         | 末端相似性检测     | 数据集            | 性能     |        |        | 特性             |
|-----------------------------|------|--------|--------------------|----------|---------------|-------------------------|-------------|----------------|--------|--------|--------|----------------|
|                             |      |        |                    | 前期嵌入     | 预训练           | 训练检测                    |             |                | Pre    | Rec    | F1     |                |
| Tao 等 <sup>[39]</sup>       | 2023 | SC, BC | 代码切片, AST, PDG     | Word2Vec | —             | CNN, BGRU               | softmax     | N+S            | 0.921  | 0.948  | 0.974  | 切片粒度           |
| IVDetect <sup>[104]</sup>   | 2021 | SC     | AST, PDG           | GloVe    | —             | BGRU, Tree-LSTM, FA-GCN | softmax     | N: Big-Vul 等   | 0.230  | 0.720  | 0.350  | 行粒度, 可解释       |
| FUNDED <sup>[31]</sup>      | 2021 | SC     | AST, PCDG          | Word2Vec | —             | GGNN                    | softmax     | N+S            | 0.930  | 0.940  | 0.934  | 跨语言            |
| MVD <sup>[106]</sup>        | 2022 | SC     | 代码切片, PDG, CG      | Doc2Vec  | —             | FS-GNN                  | softmax     | N+S            | 0.615  | 0.694  | 0.652  | 行粒度            |
| MOVERY <sup>[51]</sup>      | 2022 | SC     | 签名, CPG            | —        | —             | —                       | Jaccard 相似度 | N: ReactOS 等   | 0.960  | 0.960  | —      | 行粒度, 可扩展       |
| FBLD <sup>[109]</sup>       | 2023 | SC     | AST, FDG           | Word2Vec | —             | HGT                     | softmax     | N: Devign      | 0.620* | 0.825* | 0.695* | 切片粒度           |
| PDBERT <sup>[110]</sup>     | 2024 | SC     | AST, PDG           | PDBERT   | MLM, CDP, DDP | MLM                     | softmax     | N: Big-Vul     | —      | —      | 0.594  | —              |
| FVD-DPM <sup>[113]</sup>    | 2024 | SC     | 代码切片, CFG, DFG, CG | Word2Vec | —             | GAT, MLP                | softmax     | N+S            | —      | 0.829  | 0.857  | 行粒度            |
| Asm2Vec <sup>[64]</sup>     | 2019 | BC     | 代码序列, CFG          | —        | —             | Asm2Vec                 | 余弦相似度       | N: Busy-Box 等  | 0.940  | —      | —      | 跨编译器-优化, 抗混淆   |
| VulHawk <sup>[56]</sup>     | 2023 | BC     | 代码序列, CFG          | RoBERTa  | MLM, ROP, ABP | GCN, ResNet             | 欧氏距离, 余弦相似度 | N: Coreutils 等 | 0.913  | 0.915  | —      | 跨架构-编译器-优化, 鲁棒 |
| VulANalyzer <sup>[57]</sup> | 2023 | BC     | 代码序列, CFG          | BGRU     | —             | GCN, MLP                | softmax     | N+S            | 0.854  | 0.941  | 0.901  | 行粒度, 可解释       |

注: \*表示仅供参考

flow graph) 表征控制流, 包含异常处理流特征

### (3) 二进制代码方法

代码片段+树+图类型。Asm2Vec<sup>[64]</sup>是一种跨编译器、优化选项且抗混淆的表示学习模型, 从汇编语言 CFG 提取执行路径, 经 Token 化并添加伪逻辑后生成向量; 然而, Asm2Vec 未通过显式克隆子图或符号等价证明来解释结果, 可解释性有限。BinVulDet<sup>[40]</sup>、jTrans<sup>[55]</sup>、VulHawk<sup>[56]</sup>与 VulANalyzer<sup>[57]</sup>联合嵌入汇编指令序列及树/图结构, 以保证特征的充分表征。

### (4) 基于混合表征的方法小结

一般流程: 代码→混合表征方式→嵌入向量→机器学习/特征算法→相似性计算→漏洞检测。

优势及适用场景: 代码表征方式具有可叠加性。基于混合表征的方法泛用性、鲁棒性强, 检测性能较高, 适用于多种类型且结构复杂的漏洞检测。12.5% (3:24) 的方法具有鲁棒性, 比值仅次于基于图的方法; 41.7% (10:4) 的方法检测粒度为切片/行, 比值显著高于任意单一表征方式。此外, 在二进制代码方法中, 80% (4:5) 的方法可跨优化等级, 60% (3:5) 的方法可跨架构、编译器差异。

弱点: 多类特征提取及预处理开销大。例如, FBLD<sup>[109]</sup>在提取 AST、CDG、DDG 的基础上再提取 FDG, 时间复杂度为  $O(n^3)$ , 远高于一般表征方式的预处理时间。此外, 混合表征需处理较多噪声, 并设计复杂的组合式神经网络以融合多类特征。例如, DeepVD<sup>[107]</sup>在提取 AST、PDT、EFG 后, 采用 GRU+Tree-LSTM+FCN 的组

合网络作为中间隐藏层, 最终以 MLP 判别漏洞。

## 3.7 漏洞检测中的机器学习技术

鉴于现有漏洞检测模型在前期嵌入、预训练任务、训练检测环节均涉及机器学习技术, 因此本文将对现有方法所采用的机器学习技术进行综合分析。具体分析见表 11。

### (1) 前期嵌入

词嵌入。机器学习类现有方法中 42.1% 的嵌入采用词嵌入方法 Word2Vec<sup>[115]</sup>, 占比最高。无监督的 Word2Vec 通过 Skip-gram 和连续词袋模型 (continuous bag of words, CBOW) 实现代码到向量的映射。但受限于向量的唯一性, 无法处理多义代码。VulCNN<sup>[88]</sup>采用的 Sent2Vec<sup>[125]</sup>与 MVD<sup>[106]</sup>采用的 Doc2Vec<sup>[126]</sup>均由 Word2Vec 派生。IVDetect<sup>[104]</sup>采用的 GloVe<sup>[127]</sup>考虑全局信息, 可缓解多义代码问题, 但计算开销较高。

图嵌入。mVulPreter<sup>[89]</sup>的 Structure2Vec<sup>[128]</sup>和 VulBG<sup>[90]</sup>的 Node2Vec<sup>[129]</sup>在捕获节点上的代码信息的同时保留边关系特征, 缺点是计算复杂度高, 且对参数敏感。

代码嵌入。Transformer 衍生的 BERT<sup>[118]</sup>、CodeBERT<sup>[122]</sup>、RoBERTa<sup>[119]</sup>被用于前期代码嵌入, 这些模型均含自注意力机制, 可充分学习代码上下文信息。

### (2) 预训练任务

使用预训练的方法中, 75% 采用 MLM<sup>[118]</sup>作为预训练任务。MLM 为 BERT 的默认任务, 用以提升模型对代码相邻元素的理解与抽象。此外, PalmTree<sup>[52]</sup>针对汇编指令设计 CWP 与 DUP

表 11 现有方法的机器学习技术分析

| 应用环节 | 类别                 | 机器学习模型/任务                          | 技术特点              |
|------|--------------------|------------------------------------|-------------------|
| 前期嵌入 | 词嵌入                | Word2Vec, Sent2Vec, Doc2Vec, GloVe | 存在较高的数据依赖         |
|      | 图嵌入                | Structure2Vec, Node2Vec            | 适用于捕获结构关系特征       |
|      | 代码嵌入               | BERT, CodeBERT, RoBERTa            | 利用自注意力机制理解上下文     |
| 预训练  | 预训练任务              | MLM, CWP, DUP, JTP, ROP, ABP       | 强化模型对抽象特征的理解      |
|      | 前馈神经网络             | MLP                                | 适用于分类任务           |
|      | CNN 家族             | CNN, Text-CNN, ResNet              | 侧重局部特征            |
| 训练检测 | 传统 RNN 家族          | BLSTM, Tree-LSTM, GRU, BGRU        | 顺序结构, 存在信息衰减      |
|      | Transformer 及其派生网络 | Transformer, HGT, GPT              | 并行结构, 关注全局特征, 开销大 |
|      | GNN 家族             | FS-GNN, GGNN, GCN, GAT             | 适用于图结构特征的学习和理解    |
|      | 其他网络类型             | 孪生网络、对比学习                          | 适用于相似度计算          |

任务, 学习执行序列与数据依赖。

jTrans<sup>[55]</sup>提出跳转目标预测 (jump target prediction, JTP) 任务, 强化对 jmp/js 后跳转地址的学习; VulHawk<sup>[56]</sup>使用根操作数预测 (root operand prediction, ROP) 与相邻块预测 (adjacent block prediction, ABP) 任务, 学习汇编操作数映射及基本块间数据流关系。

### (3) 训练检测

现有方法使用的机器学习模型主要来自前馈神经网络、CNN 家族、RNN 家族、GNN 家族等。

前馈神经网络。典型代表多层感知机 (multilayer perceptron, MLP), 通过组合多个感知机实现非线性映射, 可处理异质特征并常与 Softmax/Sigmoid 配合完成检测阶段的分类。

CNN 家族。现有方法采用 CNN、Text-CNN 等; 受卷积核限制, 传统 CNN 侧重局部特征, 需增加深度以捕捉长距离依赖。

传统 RNN 家族。包括 BLSTM、Tree-LSTM、GRU、BGRU 等; RNN 为顺序结构, 沿序列方向链式连接节点, 输入序列过长时易发生信息衰减。

Transformer 及其派生网络。Transformer 并行计算, 无信息衰减; 现有方法使用 Transformer、异构图 Transformer<sup>[130]</sup> (heterogeneous graph transformer, HGT)、生成式预训练 Transformer<sup>[131]</sup> (generative pre-trained transformer, GPT) 等。自注意力层可并行关注多组特征, 但随输入规模增大, 时间和空间开销显著增加。GPT 衍生出 ChatGPT、GPT-4 等, 参数量达亿级, 计算成本高昂。

GNN 家族。现有方法采用 FS-GNN、GGNN<sup>[132]</sup>、GCN<sup>[133]</sup>、GAT<sup>[134]</sup> 等; GNN 擅长学习网状图结构的复杂节点特征, 常与图表征结合, 在控制依赖、数据依赖及调用关系理解上具优势。

其他网络类型。孪生网络<sup>[115]</sup>耦合两种神经网络, 可直接通过损失函数计算两组输入的相似度; 对比学习则通过最大化相关样本、最小化无关样本的相似性来学习表示。

## 3.8 漏洞检测的性能评估

现有方法的漏洞检测性能评估指标包括: 准确率、精确率、召回率、F1 值等。75 种方法在既定测试数据集下使用评估指标与同类基线方法

进行比较, 得出该方法的漏洞检测性能情况。由于不同文献的评估指标的测试数据集各异, 因此前文表格中所列出的精确率、召回率、F1 值无法直接用于横向比较。

### (1) 漏洞检测任务类型和通用数据集

本文从漏洞检测任务类型和通用数据集的角度出发, 筛选可横向比较的评估指标。按末端相似性检测技术及任务应用, 将 75 种方法分为两类: 漏洞判别类、漏洞搜索类。

漏洞判别类方法: 以 Softmax/Sigmoid 分类器对单段代码给出“漏洞/良性”标签, 共 44 种 (58.7%, 44:75), 含可兼顾搜索的 VulSim<sup>[114]</sup>与 PalmTree<sup>[52]</sup>模型。

漏洞搜索类方法: 以余弦距离/欧氏距离/哈希算法对单段或多段代码返回数据集中相似漏洞, 共 33 种 (44%, 33:75), 含 VulSim 和 PalmTree 模型。

通用漏洞测试数据集统计如下。

漏洞搜索类无通用数据集: 69.7% (23:33) 使用自建小型孤立数据集, 无法跨文献横向比对。

漏洞判别类存在通用数据集: SARD<sup>[24]</sup>、Devign<sup>[32]</sup>、Big-Vul<sup>[33]</sup>、Reveal<sup>[34]</sup>分别被引用 20、13、13、12 次, 居前 4 位, 可用于直接横向比较。

### (2) 漏洞判别类方法: 精确率和召回率

鉴于数据完整性与指标通用性, 本文选用精确率、召回率对 44 种判别类方法在 SARD、Devign、Big-Vul、Reveal 上的文献结果进行横向比较。同一方法在同一数据集的多次结果取平均。共获得 18、17、16、14 种方法的可用数据; 基于哈希与指标的方法均属搜索类, 不纳入比较。

具体评估指标如图 6 所示, 深灰色条形柱 Rec 表示 HAN-BSVD<sup>[54]</sup>和 VDoTR<sup>[108]</sup>的指标数据仅在部分 SARD 数据集下获得。

### (3) 精确率和召回率评估

分析图 6 发现, 混合表征在 Devign、Reveal 平均精确率最高; 代码片段与树表征分别在 SARD、Big-Vul 最高。事实上, 漏洞特征通常较为复杂, 混合表征能够较为全面地捕获这些特点, 故平均精确率最优。其中 FBLD<sup>[109]</sup>凭漏洞指向性 FDG 在 Devign、Reveal 中取得最高的精

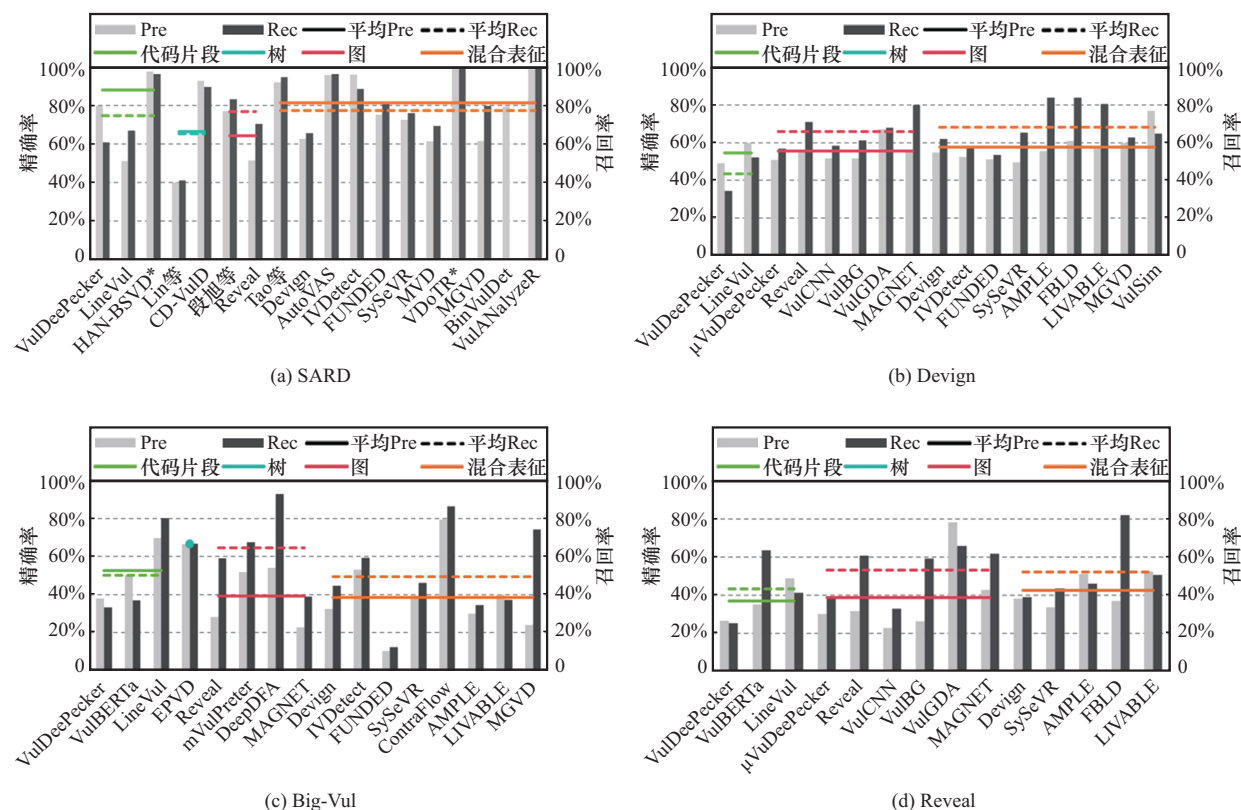


图6 漏洞判别类方法的评估指标

确率。SARD样本简短且人工构造，代码片段契合其简单特征，故精确率相对最佳。Big-Vul仅EPVD<sup>[84]</sup>一种树表征，其基于AST构造控制流，表现良好。

混合表征在SARD、Devign平均召回率最高；树与图表征分别在Big-Vul、Reveal最优。混合表征覆盖多维度特征，故在简单SARD与复杂Devign均能查全较好，其中AMPLE<sup>[29]</sup>、VulANalyzer<sup>[57]</sup>融合图结构与代码序列获最佳。图表征适应Reveal的复杂代码，VulGDA<sup>[93]</sup>以GRU学习CPG特征，召回率佳。代码片段因缺失结构特征，平均召回率在4种数据集下均未领先。

综上，在4种数据集下，基于混合表征的方法总体优于基于代码片段、树、图表征的方法，分别在50%数据集取得最高平均精确率与最高平均召回率。

### 3.9 漏洞检测的开销分析

漏洞检测的开销贯穿各环节。对单个待测样本而言，代码预处理、表征、模型训练及检测（判别/搜索）均产生计算开销。

在本文分析的75篇文献中，分别有25.3%

（19:75）、54.7%（41:75）分别提及了预处理/表征/训练阶段和检测阶段的开销。由于实验平台、环境与测试集差异，单一文献数据无法直接横向比较。

#### （1）锚点数据

本文从锚点数据出发，对文献中可横向对比的开销信息进行分析。具体做法是：按检测任务（判别/搜索）分别筛选各文献与现有方法对比的开销数据，确定“锚点方法”（即被最多文献用于开销比较的方法），再以锚点方法的开销为基准，按比例调整其余参比数据，使不同文献结果具有同一比较基准，从而缓解实验平台、环境、数据集差异带来的不可横向对比问题。

经统计，共确定2个锚点方法：漏洞判别类中，源代码方法Devign<sup>[32]</sup>以5次参比（共涉及10个方法）成为锚点；漏洞搜索类中，二进制方法Gemini<sup>[135]</sup>以4次参比（共涉及5个方法）成为锚点（Gemini系2017年工作，未纳入本文分析范围）。本文取FBLD<sup>[109]</sup>文献中Devign<sup>[32]</sup>开销（训练时间49 min/检测时间3.4 s）与VulSeeker-Pro<sup>[67]</sup>文献中Gemini<sup>[135]</sup>开销（嵌入处理时间

0.15 s) 为基准值 1, 计算其余方法的相对开销; 同一方法存在多组数据时取均值, 结果如图 7 所示, \*标出锚点方法。

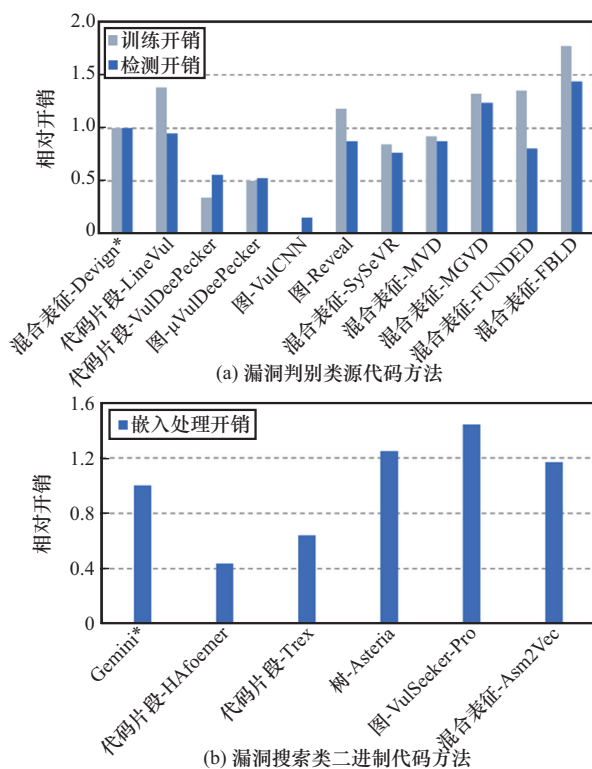


图7 漏洞检测方法的相对开销情况

## (2) 开销分析

漏洞判别类源代码方法方面。由图 7(a)可见, 混合表征方法普遍训练与检测开销较高, 因其需学习混合特征且依赖复杂组合网络, 开销高符合预期。代码片段方法中, LineVul<sup>[73]</sup>的训练开销和检测开销分别高于 81.8% 和 63.6% 的参比方法, 归因于其采用 12 头多层双向自注意力 BERT, 注意力层数与头数显著增加计算开销。值得注意的是, 基于图的方法 VulCNN<sup>[88]</sup>检测开销最低, 因其检测阶段仅用 CNN 对图像化代码分类, 无冗余处理。

漏洞搜索类二进制代码方法方面。由图 7(b)可见, 基于图的 VulSeeker-Pro<sup>[67]</sup>开销最高, 因其额外引入模拟仿真环节; 基于代码片段的 HAformer<sup>[77]</sup>开销最低, 其直接处理汇编与机器码, 无须构图提取结构特征。

综上, 基于混合表征、图的方法普遍计算开销较高, 但需注意到神经网络/相似性算法本身

设计存在多样性和复杂性, 非混合/图的方法在实现层面亦可能出现高开销。

## 3.10 代码表征与漏洞检测总结

第 3.1~3.9 节已完成各类表征漏洞检测方法分析、机器学习技术分析、性能评估及开销分析。现总结代码表征与漏洞检测要点, 按表征类型归纳所用机器学习技术、表征特点及整体特点, 见表 12。

## 4 研究挑战与展望

### 4.1 面临的挑战

尽管近年来基于代码相似性的软件漏洞检测技术取得长足发展, 该领域仍面临以下挑战。

1) 漏洞样本数据集规模不足且滞后。相较于 NVD<sup>[25]</sup>等平台的漏洞总量, 现有数据集规模仍显不足, 且样本时间普遍早于文献公开时间, 多数方法在发布后不再更新数据, 导致检测真实代码的有效性下降。

2) 代码标准化实证研究薄弱。除 OOV 问题外, 虽有研究<sup>[21-22,122]</sup>初步探讨标准化策略对检测效果的影响, 但缺乏系统的定量/定性实证, 尚难形成统一标准。

3) 代码表征需进一步细化、特化。当前表征多与克隆检测重合, 未充分挖掘漏洞特有的结构与语义特征, 难以适配多类型、多语义场景。

4) 多维度表征计算开销大。混合表征通常综合词法、语法、控制流、数据流、调用关系等多维信息, 时空资源消耗显著, 难以兼顾速度与精度。

5) 机器学习技术实证与泛化研究不足。随着技术迭代, 漏洞检测需持续探究更具语义相关性与泛化性的模型。尽管 Transformer 衍生网络、GNN 家族在多个数据集表现良好, 其在真实场景、跨平台下的泛化能力与稳定性仍缺乏大规模验证。

6) 跨语言漏洞检测仍待突破 (源代码方法)。多数方法仅面向单一高级语言 (如 C/C++、Java), 通用的跨语言检测方案仍是难题。

7) 跨架构、编译器、优化等级及抗混淆能力需强化 (二进制代码方法)。虽有 Asm2Vec<sup>[64]</sup>、Trex<sup>[56]</sup>等缓解差异, 但有效性有限; 抗混淆方面, 现有方法难以应对混合式混淆, 问题仍多。

表12 代码表征与漏洞检测总结

| 表征类型   | 主要表征方式         | 主要机器学习技术       |       |                              | 表征特点         | 整体特点                                                     |
|--------|----------------|----------------|-------|------------------------------|--------------|----------------------------------------------------------|
|        |                | 前期嵌入           | 预训练任务 | 训练检测                         |              |                                                          |
| 基于代码片段 | 代码序列           | 词嵌入, 代码嵌入      | MLM   | 传统RNN, Transformer           | 连续代码元素       | (1) 依赖嵌入性能<br>(2) 表征较为简单<br>(3) 开销相对较小                   |
|        | 代码切片           | 词嵌入            | —     | 传统RNN                        | 分段式切片        |                                                          |
|        | 代码部件           | 词嵌入, 代码嵌入      | MLM   | 传统RNN, Transformer           | 行级语义关联       |                                                          |
| 基于哈希   | 哈希值            | —              | —     | —                            | 代码映射         | (1) 适用于快速搜索<br>(2) 泛化性弱                                  |
| 基于树    | AST            | 词嵌入            | —     | 传统RNN, CNN, 前馈神经网络           | 语法           | (1) 检测语法相似漏洞<br>(2) 多层细分结构                               |
| 基于图    | CFG            | 词嵌入, 图嵌入       | —     | GNN, 孪生网络                    | 控制流          | (1) 代码特征覆盖率广<br>(2) 结构信息丰富<br>(3) 计算时空成本高                |
|        | DDG            | —              | —     | —                            | 数据依赖         |                                                          |
|        | PDG            | 词嵌入            | —     | GNN, CNN                     | 控制依赖, 数据依赖   |                                                          |
|        | CPG            | 词嵌入            | —     | GNN, 传统RNN                   | 语法, 控制流, 数据流 |                                                          |
| 基于指标   | 代码指标           | —              | —     | —                            | 统计学特征        | (1) 适用于辅助检测<br>(2) 难以检测语义漏洞                              |
| 基于混合表征 | 代码片段, 哈希, 树, 图 | 词嵌入, 图嵌入, 代码嵌入 | MLM   | 传统RNN, Transformer, GNN, CNN | 混合特征         | (1) 表征具有可叠加性<br>(2) 泛用性、鲁棒性强<br>(3) 开销存在挑战<br>(4) 模型设计复杂 |

8) 可解释性仍不足。黑盒模型难以给出数学自洽的解释, GNNExplainer<sup>[136]</sup>、Grad-CAM++<sup>[137]</sup>等解释器虽能够部分缓解此类问题, 但本质仍为神经网络, 且忽略边联合效应, 解释结果不够忠实且冗长。

9) 缺乏统一全面的评估标准。现有研究多聚焦精确率、准确率、召回率、AUC、F1等传统指标, 忽视可扩展性、可解释性、泛化能力等关键特性, 碎片化评估降低可比性, 制约技术全面评价。

10) 检测准确性问题(假阳性/假阴性)仍未有效缓解。漏洞表征差异大、模型泛化受限, 导致实际应用中误报、漏报普遍, 需从样本质量、特征学习及解释机制3个方面改进。

## 4.2 未来的研究方向

基于代码相似性的软件漏洞检测领域仍存挑战, 但可在未来逐步解决。本文从技术融合趋势与基础设施建设两个视角提出可行研究方向。

(1) 推进跨模态、多语言、集成式神经网络等技术融合的检测模型设计

现有模型多面向单一输入形式(源代码/二进制代码)与单一语言(如C/C++), 通用性不

足。跨模态、跨语言、多维融合的特征学习与集成建模更具表达力, 将成为未来重要研究方向。

跨模态特征统一建模。已有方法<sup>[41,46,101]</sup>较单模态方案更具普适性, 且源码-二进制代码间并无不可逾越壁垒。随着反汇编、反编译、中间语言转换及特征映射技术成熟, 跨模态检测将成为主流。研究建议: 聚焦模态差异, 采用“统一模态形式”或“同特征映射”。前者将源码与二进制转为统一表示(中间语言/汇编等)以消除词法、语法、结构差异; 后者通过映射算法/神经网络实现同类特征对齐。现有方案包括: 汇编统一<sup>[41,46]</sup>、中间语言统一<sup>[138]</sup>、跨模态映射算法<sup>[101]</sup>、基于GNN的结构映射<sup>[9]</sup>等; 可辅以迁移学习或对比学习缓解分布差异, 提升语义对齐。

跨语言漏洞检测。当前方法集中于C/C++。CCAligner<sup>[48]</sup>支持C/Java, FUNDED<sup>[31]</sup>支持C/C++、Java、PHP、Swift。研究建议: 在统一多语言特征基础上, 引入中间语言(如LLVM IR)供模型学习, 拓宽语言覆盖, 增强适应性。

多维且漏洞特化的代码表征。通用数据集评估表明, 混合表征在精确率、召回率上优于单一表征, 验证多维度特征有效性。鉴于漏洞形式化

表征尚缺, 已出现若干面向漏洞的特化方案: Leopard<sup>[102]</sup>设计漏洞关联指标; ContraFlow<sup>[69]</sup>以 VFG+自注意力检测异常值流; CarrotsBlender<sup>[95]</sup>构建 VCG 追踪漏洞调用。研究建议: 通过实证分析各表征的互斥性与互补性, 设计有效耦合的多维融合机制, 并构建面向漏洞特征的形式化表征。

集成式的神经网络检测模型。CPVD<sup>[92]</sup>、VDoTR<sup>[108]</sup>、VulHawk<sup>[56]</sup>均采用集成式神经网络, 在测试中表现良好。Transformer 派生网络与 GNN 家族分别在序列、结构数据上具优势, 将二者及其他先进网络融合可弥补单网络缺陷。此外, ChatGPT 出现后, 大语言模型 (LLM) 已被用于漏洞检测<sup>[139-140]</sup>。Zhang 等<sup>[141]</sup>、Bakhshandeh 等<sup>[142]</sup>发现 LLM 可增强检测能力, 但尚未取得明显领先<sup>[143-144]</sup>, 仍是重要研究方向。研究建议: 在大规模实证基础上, 构建契合漏洞语义的 Transformer+GNN+LLM 集成框架, 探讨 3 类模型在表征、训练、解释上的协同优化, 并对串联/并联/注意力加权等集成方式进行对比评估。

高效简洁的模型可解释化方法。对图解释器, 可通过最大归因化度量筛选边序列因果, 使解释结果高效且忠实, 如 RC-Explainer<sup>[145]</sup>; 对通用解释器, 可通过遮蔽输入预测构建模型无关的 LTX<sup>[146]</sup>, 为 Transformer 和 CNN 提供简洁解释。研究建议: 解析漏洞语义与模型学习机制的关联, 构建高效简洁的解释器, 提升黑盒模型的可用性与可信度。

(2) 建立统一、全面、真实语境下的大规模漏洞数据集与评估体系

漏洞检测评估依赖样本集与有限指标, 而现有数据集滞后、局限, 评估体系不统一, 难以公平对比与持续迭代。构建多类型、多领域有效数据集与统一评估体系将是未来重要研究方向。

构建覆盖多漏洞类型、多语言、多领域、多源形式 (源代码/二进制代码) 的公开数据集。多语言、多类型漏洞集可显著提升模型泛用性; 跨领域、跨模态数据集更具应用潜力。研究建议: 在 NVD、Big-Vul<sup>[33]</sup>、DiverseVul<sup>[36]</sup>等基础上扩展类型、领域、语言、形式, 并推动学界与产业共享, 确保保留真实上下文特征且验证标签

的有效性。

引入自动标注和人机协同标注机制。样本标注自动化、高效化契合规模扩展与质量提升。可利用静态/动态分析工具<sup>[147]</sup>初步筛选和标注, 再经专家审核, 扩大规模并提升质量, 实现持续更新。研究建议: 开发集成自动化工具与专家审核的智能标注平台, 实现人机协同流程自动化、标准化。

建立统一完备的评估指标体系。当前指标相对混乱、不全面, 统一完备的鲁棒性、泛化性、数据敏感性、可扩展性、可解释性、效率等多维指标将提升横向可比性并降低评估复杂度。研究建议: 在传统准确率、F1 分数基础上, 按适用场景设定上述多维指标。

开展标准化 benchmark 竞赛机制。标准化竞赛可吸引研究者参与, 推动领域发展。研究建议: 建立公开透明平台, 定期发布 benchmark 数据集与评估结果, 鼓励模型在真实场景 (开源项目漏洞、CTF 数据) 中实战测试。

## 5 结束语

为把握基于代码相似性的软件漏洞检测领域的研究进展, 本文对近 7 年具有代表性的基于代码相似性的软件漏洞检测方法进行了系统性研究。本文从漏洞样本数据集、代码预处理、代码表征与漏洞检测 3 个方面出发, 对现有方法进行了深入全面的特征总结、性能评估和开销分析, 形成了对基于代码相似性的软件漏洞检测领域面临的挑战分析以及未来的研究方向展望。笔者期待今后能够有更多的研究人员对此领域进行研究, 持续不断地推进此领域向前发展。

## 参考文献:

- [1] LIANG H L, PEI X X, JIA X D, et al. Fuzzing: state of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.
- [2] MANÈS V J M, HAN H, HAN C, et al. The art, science, and engineering of fuzzing: a survey[J]. IEEE Transactions on Software Engineering, 2021, 47(11): 2312-2331.
- [3] BALDONI R, COPPA E, D'ELIA D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys, 2019, 51(3): 1-39.
- [4] CADAR C, SEN K. Symbolic execution for software testing[J]. Communications of the ACM, 2013, 56(2): 82-90.

- [5] NEWSOME J, SONG D X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]//NDSS. 2005, 5: 3-4.
- [6] SCHWARTZ E J, AVGERINOS T, BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]//Proceedings of the 2010 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2010: 317-331.
- [7] LI Z, LU S, MYAGMAR S, et al. CP-Miner: finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on Software Engineering, 2006, 32(3): 176-192.
- [8] WHALE G. Plague: plagiarism detection using program structure[M]. School of Electrical Engineering and Computer Science, University of New South Wales, 1988.
- [9] YU ZQ, ZHENG W X, WANG J Q, et al. CodeCMR: cross-modal retrieval for function-level binary source code matching[J]. Advances in Neural Information Processing Systems, 2020, 33: 3872-3883.
- [10] LOPES C V, MAJ P, MARTINS P, et al. DéjàVu: a map of code duplicates on GitHub[C]//Proceedings of the ACM on Programming Languages, New York: ACM, 2017, 1: 1-28.
- [11] EGHBALI A, PRADEL M. CrystalBLEU: precisely and efficiently measuring the similarity of code[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2022: 1-12.
- [12] 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. Internet Society, 2020.
- [13] ISLAM M R, ZIBRAN M F, NAGPAL A. Security vulnerabilities in categories of clones and non-cloned code: an empirical study[C]//Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). Piscataway: IEEE Press, 2017: 20-29.
- [14] Open source security and risk analysis report [EB]. 2022.
- [15] 李珍, 邹德清, 王泽丽, 等. 面向源代码的软件漏洞静态检测综述[J]. 网络与信息安全学报, 2019, 5(1): 1-14.
- LI Z, ZOU D Q, WANG Z L, et al. Survey on static software vulnerability detection for source code[J]. Chinese Journal of Network and Information Security, 2019, 5(1): 1-14.
- [16] 李韵, 黄辰林, 王中锋, 等. 基于机器学习的软件漏洞挖掘方法综述[J]. 软件学报, 2020, 31(7): 2040-2061.
- LI Y, HUANG C L, WANG Z F, et al. Survey of software vulnerability mining methods based on machine learning[J]. Journal of Software, 2020, 31(7): 2040-2061.
- [17] LIN G J, WEN S, HAN Q L, et al. Software vulnerability detection using deep neural networks: a survey[J]. Proceedings of the IEEE, 2020, 108(10): 1825-1848.
- [18] CHAKRABORTY S, KRISHNA R, DING Y, et al. Deep learning based vulnerability detection: Are we there yet[J]. IEEE Transactions on Software Engineering, 2021.
- [19] STEENHOEK B, RAHMAN M M, JILES R, et al. An empirical study of deep learning models for vulnerability detection[C]//Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2023: 2237-2248.
- [20] LIPP S, BANESCU S, PRETSCHNER A. An empirical study on the effectiveness of static C code analyzers for vulnerability detection[C]//Proceedings of the 31st ACM Sigsoft International Symposium on Software Testing and Analysis. New York: ACM, 2022: 544-555.
- [21] 于颖超, 甘水滔, 邱俊洋, 等. 二进制代码相似度分析及在嵌入式设备固件漏洞搜索中的应用[J]. 软件学报, 2022, 33(11): 4137-4172.
- YU Y C, GAN S T, QIU J Y, et al. Binary code similarity analysis and its applications on embedded device firmware vulnerability search[J]. Journal of Software, 2022, 33(11): 4137-4172.
- [22] NONG Y, SHARMA R, HAMOU-LHADJ A, et al. Open science in software engineering: a study on deep learning-based vulnerability detection[J]. IEEE Transactions on Software Engineering, 2023, 49(4): 1983-2005.
- [23] 邓泉, 叶蔚, 谢睿, 等. 基于深度学习的源代码缺陷检测研究综述[J]. 软件学报, 2023, 34(2): 625-654.
- DENG X, YE W, XIE R, et al. Survey of source code bug detection based on deep learning[J]. Journal of Software, 2023, 34(2): 625-654.
- [24] SARD: Software assurance reference dataset[EB]. 2023.
- [25] NVD [EB]. 2023.
- [26] CVE[EB]. 2023.
- [27] GitHub [EB]. 2023.
- [28] JEON S, KIM H K. AutoVAS: an automated vulnerability analysis system with a deep learning approach[J]. Computers & Security, 2021, 106: 102308.
- [29] WEN X C, CHEN Y P, GAO C Y, et al. Vulnerability detection with graph simplification and enhanced graph representation learning[C]//Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2023: 2275-2286.
- [30] XIAO Y, CHEN B H, YU C D, et al. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures[C]//29th USENIX Security Symposium. 2020: 1165-1182.
- [31] WANG H T, YE G X, TANG Z Y, et al. Combining graph-based learning with automated data collection for code vulnerability detection[J]. IEEE Transactions on Information Forensics and Security, 2020, 16: 1943-1958.
- [32] ZHOU Y Q, LIU S Q, SIOW J, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks[EB]. 2019.
- [33] FAN J H, LI Y, WANG S H, et al. A C/C++ code vulnerability dataset with code changes and CVE summaries[C]//Proceedings of the 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR). Piscataway: IEEE Press, 2020: 508-512.
- [34] CHAKRABORTY S, KRISHNA R, DING Y, et al. Deep learning based vulnerability detection: Are we there yet [J]. IEEE Transac-

- tions on Software Engineering, 2021.
- [35] WANG X D, WANG S, FENG P B, et al. PatchDB: a large-scale security patch dataset[C]//Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Piscataway: IEEE Press, 2021: 149-160.
- [36] CHEN Y Z, DING Z J, ALOWAIN L, et al. DiverseVul: a new vulnerable source code dataset for deep learning based vulnerability detection[C]//Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. New York: ACM, 2023: 654-668.
- [37] ZHOU X, VERMA R M. Vulnerability detection via multimodal learning: datasets and analysis[C]//Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. New York: ACM, 2022: 1225-1227.
- [38] BRUST C A, SONNEKALB T, GRUNER B. ROMEO: a binary vulnerability detection dataset for exploring Juliet through the lens of assembly language[J]. Computers & Security, 2023, 128: 103165.
- [39] TAO W X, SU X H, WAN J Y, et al. Vulnerability detection through cross-modal feature enhancement and fusion[J]. Computers & Security, 2023, 132: 103341.
- [40] WANG Y, JIA P, PENG X, et al. BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention[J]. Computers & Security, 2023, 125: 103023.
- [41] 陈小全, 刘剑, 夏翔宇, 等. 基于比较学习的漏洞检测方法[J]. 计算机研究与发展, 2023, 60(9): 2152-2168.
- CHEN X Q, LIU J, XIA X Y, et al. A vulnerability detection approach based on comparative learning[J]. Journal of Computer Research and Development, 2023, 60(9): 2152-2168.
- [42] CHAWLA N V, BOWYER K W, HALL L O, et al. SMOTE: synthetic minority over-sampling technique[J]. Journal of Artificial Intelligence Research, 2002, 16: 321-357.
- [43] LI H, ZOU P, WANG X, et al. A new combination sampling method for imbalanced data[C]//Proceedings of 2013 Chinese Intelligent Automation Conference. Berlin, Heidelberg: Springer, 2013: 547-554.
- [44] ZHAO T X, ZHANG X, WANG S H. GraphSMOTE: imbalanced node classification on graphs with graph neural networks[C]//Proceedings of the 14th ACM International Conference on Web Search and Data Mining. New York: ACM, 2021: 833-841.
- [45] MIRSKY Y, MACON G, BROWN M, et al. VulChecker: graph-based vulnerability localization in source code[C]//31st USENIX Security Symposium, Security 2022. 2023.
- [46] JI Y D, CUI L, HUANG H H. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network[C]//Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. 2021: 702-715.
- [47] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection[C]//Proceedings 2018 Network and Distributed System Security Symposium. Internet Society, 2018.
- [48] WANG P C, SVAJLENKO J, WU Y Z, et al. CCAAligner: a token based large-gap clone detector[C]//Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2018: 1066-1077.
- [49] WOO S H, LEE D W, PARK S H, et al. V0Finder: discovering the correct origin of publicly reported software vulnerabilities[C]//30th USENIX Security Symposium. 2021: 3041-3058.
- [50] HANIF H, MAFFEIS S. VulBERTa: simplified source code pre-training for vulnerability detection[C]//Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN). Piscataway: IEEE Press, 2022: 1-8.
- [51] WOO S H, HONG H J, CHOI E J, et al. MOVER: A precise approach for modified vulnerable code clone discovery from modified open-source software components[C]//31st USENIX Security Symposium. 2022: 3037-3053.
- [52] LI X, QU Y, YIN H. PalmTree: learning an assembly language model for instruction embedding[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2021: 3236-3251.
- [53] YANG S G, CHENG L, ZENG Y C, et al. Asteria: deep learning-based AST-encoding for cross-platform binary code similarity detection[C]//Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Piscataway: IEEE Press, 2021: 224-236.
- [54] YAN H, LUO S L, PAN L M, et al. HAN-BSVD: a hierarchical attention network for binary software vulnerability detection[J]. Computers & Security, 2021, 108: 102286.
- [55] 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. 2022: 1-13.
- [56] LUO Z H, WANG P F, WANG B S, et al. VulHawk: cross-architecture vulnerability detection with entropy-based binary code search[C]//Proceedings 2023 Network and Distributed System Security Symposium. Internet Society, 2023.
- [57] LI L T, DING S H H, TIAN Y, et al. VulAnalyzer: explainable binary vulnerability detection with multi-task learning and attentional graph convolution[J]. ACM Transactions on Privacy and Security, 2023, 26(3): 1-25.
- [58] IDA[EB]. 2023.
- [59] LE T, NGUYEN T, LE T, et al. Maximal divergence sequential autoencoder for binary software vulnerability detection[C]//International Conference on Learning Representations. 2018.
- [60] Capstone [EB]. 2023.
- [62] GAO J, YANG X, FU Y, et al. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary[C]//Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE Press, 2018: 896-899.
- [63] 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.

- [64] 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 (SP). Piscataway: IEEE Press, 2019: 472-489.
- [65] 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. Internet Society, 2020.
- [66] KIM D, KIME, CHASK, 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.
- [67] GAO J, YANG X, FU Y, et al. VulSeeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM, 2018: 803-808.
- [68] GAO J, JIANG Y, LIU Z, et al. Semantic learning and emulation based cross-platform binary vulnerability seeker[J]. IEEE Transactions on Software Engineering, 2021, 47(11): 2575-2589.
- [69] CHENG X, ZHANG G Q, WANG H Y, et al. Path-sensitive code embedding via contrastive learning for software vulnerability detection[C]//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2022: 519-531.
- [70] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//Proceedings of the 2014 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2014: 590-604.
- [71] SUN H, CUI L, LI L, et al. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches[J]. Computers & Security, 2021, 110: 102417.
- [72] THAPA C, JANG S I, AHMED M E, et al. Transformer-based language models for software vulnerability detection[C]//Proceedings of the 38th Annual Computer Security Applications Conference. New York: ACM, 2022: 481-496.
- [73] FU M, TANTITHAMTHAVORN C. LineVul: a transformer-based line-level vulnerability prediction[C]//Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR). Piscataway: IEEE Press, 2022: 608-620.
- [74] FU M, TANTITHAMTHAVORN C, LE T, et al. AIBugHunter: a Practical tool for predicting, classifying and repairing software vulnerabilities[J]. Empirical Software Engineering, 2023, 29(1): 4.
- [75] LIU B C, HUO W, ZHANG C, et al.  $\alpha$ Diff: cross-version binary code similarity detection with DNN[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. New York: ACM, 2018: 667-678.
- [76] PEI K X, XUAN Z, YANG J F, et al. Learning approximate execution semantics from traces for binary function similarity[J]. IEEE Transactions on Software Engineering, 2023, 49(4): 2776-2790.
- [77] JIANG X Z, WANG S, GONG Y X, et al. HAformer: Semantic fusion of hex machine code and assembly code for cross-architecture binary vulnerability detection[J]. Computers & Security, 2024, 145: 104029.
- [78] 李赞, 边攀, 石文昌, 等. 一种利用补丁的未知漏洞发现方法[J]. 软件学报, 2018, 29(5): 1199-1212.
- LI Z, BIAN P, SHI W C, et al. Approach of leveraging patches to discover unknown vulnerabilities[J]. Journal of Software, 2018, 29(5): 1199-1212.
- [79] WOO S H, CHOI E J, LEE H J, et al. V1SCAN: Discovering 1-day vulnerabilities in reused C/C++ open-source software components using code classification techniques[C]//32nd USENIX Security Symposium. 2023: 6541-6556.
- [80] WOO S, PARK S, KIM S, et al. CENTRIS: a precise and scalable approach for identifying modified open-source software reuse[C]//Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2021: 860-872.
- [81] 陈昱, 刘中金, 赵威威, 等. 一种大规模的跨平台同源二进制文件检索方法[J]. 计算机研究与发展, 2018, 55(7): 1498-1507.
- CHEN Y, LIU Z J, ZHAO W W, et al. A large-scale cross-platform homologous binary retrieval method[J]. Journal of Computer Research and Development, 2018, 55(7): 1498-1507.
- [82] LIN G J, ZHANG J, LUO W, et al. Software vulnerability discovery via learning multi-domain knowledge bases[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2469-2485.
- [83] LIU S G, LIN G J, QU L Z, et al. CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(1): 438-451.
- [84] ZHANG J W, LIU Z X, HU X, et al. Vulnerability detection by learning from syntax-based execution paths of code[J]. IEEE Transactions on Software Engineering, 2023, 49(8): 4196-4212. ]
- [85] WANG X D, SUN K, BATCHELLER A, et al. Detecting “0-day” vulnerability: an empirical study of secret security patch in OSS[C]//Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Piscataway: IEEE Press, 2019: 485-492.
- [86] ZOU D Q, WANG S J, XU S H, et al.  $\mu$ VulDeePecker: a deep learning-based system for multiclass vulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.
- [87] 段旭, 吴敬征, 罗天悦, 等. 基于代码属性图及注意力双向LSTM的漏洞挖掘方法[J]. 软件学报, 2020, 31(11): 3404-3420.
- DUAN X, WU J Z, LUO T Y, et al. Vulnerability mining method based on code property graph and attention BiLSTM[J]. Journal of Software, 2020, 31(11): 3404-3420.
- [88] WU Y M, ZOU D Q, DOU S H, et al. VulCNN: an image-inspired scalable vulnerability detection system[C]//Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2022: 2365-2376.
- [89] ZOU D Q, HU Y T, LI W K, et al. mVulPreter: a multi-granularity

- vulnerability detection system with interpretations[J]. IEEE Transactions on Dependable and Secure Computing, 2022, (99): 1-12.
- [90] YUAN B, LU Y F, FANG Y L, et al. Enhancing deep learning-based vulnerability detection by building behavior graph model[C]// Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2023: 2262-2274.
- [91] WANG S, WANG X D, SUN K, et al. GraphSPD: graph-based security patch detection with enriched code semantics[C]// Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP). Piscataway: IEEE Press, 2023: 2409-2426.
- [92] ZHANG C Y, LIU B, XIN Y, et al. CPVD: cross project vulnerability detection based on graph attention network and domain adaptation[J]. IEEE Transactions on Software Engineering, 2023, 49(8): 4152-4168.
- [93] LI X, XIN Y, ZHU H L, et al. Cross-domain vulnerability detection using graph embedding and domain adaptation[J]. Computers & Security, 2023, 125: 103017.
- [94] BOWMAN B, HUANG H H. VGRAPH: a robust vulnerable code clone detection system using code property triplets[C]// Proceedings of the 2020 IEEE European Symposium on Security and Privacy (EuroS&P). Piscataway: IEEE Press, 2020: 53-69.
- [95] LIU B C, MENG G Z, ZOU W, et al. A large-scale empirical study on vulnerability distribution within projects and the lessons learned[C]// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York: ACM, 2020: 1547-1559.
- [96] KANG W, SON B, HEO K. TRACER: signature-based static analysis for detecting recurring vulnerabilities[C]// Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2022: 1695-1708.
- [97] STEENHOEK B, GAO H Y, LE W. Dataflow analysis-inspired deep learning for efficient vulnerability detection[C]// Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York: ACM, 2024: 1-13.
- [98] WEN X C, GAO C Y, YE J X, et al. Meta-path based attentional graph learning model for vulnerability detection[J]. IEEE Transactions on Software Engineering, 2024, 50(3): 360-375.
- [99] MACHIRY A, REDINI N, CAMELLINI E, et al. SPIDER: enabling fast patch propagation in related software repositories[C]// Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). Piscataway: IEEE Press, 2020: 1562-1579.
- [100] WANG Y S, SHEN J J, LIN J, et al. Staged method of code similarity analysis for firmware vulnerability detection[J]. IEEE Access, 2019, 7: 14171-14185.
- [101] YUAN Z M, FENG M Y, LI F, et al. B2SFinder: detecting open-source software reuse in COTS software[C]// Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE Press, 2019: 1038-1049.
- [102] DU X N, CHEN B H, LI Y K, et al. LEOPARD: identifying vulnerable code for vulnerability assessment through program metrics[C]// Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2019: 60-71.
- [103] CHINTHANET B, PONTA S E, PLATE H, et al. Code-based vulnerability detection in node.js applications: how far are we[C]// Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2020: 1199-1203.
- [104] LI Y, WANG S H, NGUYEN T N. Vulnerability detection with fine-grained interpretations[C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM, 2021: 292-303.
- [105] LI Z, ZOU D Q, XU S H, et al. SySeVR: a framework for using deep learning to detect software vulnerabilities[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2244-2258.
- [106] CAO S C, SUN X B, BO L L, et al. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks[C]// Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2022: 1456-1468.
- [107] WANG W B, NGUYEN T N, WANG S H, et al. DeepVD: toward class-separation features for neural network vulnerability detection[C]// Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Piscataway: IEEE Press, 2023: 2249-2261.
- [108] FAN Y H, WAN C H, FU C, et al. VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs[J]. Computers & Security, 2023, 130: 103247.
- [109] 杨宏宇, 杨海云, 张良, 等. 基于特征依赖图的源代码漏洞检测方法[J]. 通信学报, 2023, 44(1): 103-117.
- YANG H Y, YANG H Y, ZHANG L, et al. Feature dependence graph based source code loophole detection method[J]. Journal on Communications, 2023, 44(1): 103-117.
- [110] LIU Z X, TANG Z J, ZHANG J W, et al. Pre-training by predicting program dependencies for vulnerability analysis tasks[C]// Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York: ACM, 2024: 1-13.
- [111] WEN X C, GAO C Y, LUO F, et al. LIVABLE: exploring long-tailed classification of software vulnerability types[J]. IEEE Transactions on Software Engineering, 2024, 50(6): 1325-1339.
- [112] QIU F C, LIU Z X, HU X, et al. Vulnerability detection via multiple-graph-based code representation[J]. IEEE Transactions on Software Engineering, 2024, 50(8): 2178-2199.
- [113] SHAO M M, DING Y X. {FVD-DPM}: Fine-grained Vulnerability Detection via Conditional Diffusion Probabilistic Models[C]// 33rd USENIX Security Symposium. 2024: 7375-7392.
- [114] SHIMMI S, RAHMAN A, GADDE M, et al. {VulSim}: Leveraging Similarity of {Multi-Dimensional} Neighbor Embeddings for Vulnerability Detection[C]// 33rd USENIX Security Symposium (USENIX Security 24). 2024: 1777-1794.
- [115] MIKOLOV T, CHEN K, CORRADO G, et al. Efficient estima-

- tion of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.
- [116] BROMLEY J, BENTZ J W, BOTTOU L, et al. Signature verification using a " siamese" time delay neural network[J]. Advances in neural information processing systems, 1993, 6.
- [117] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.
- [118] DEVLIN J, CHANG M W, LEE K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.
- [119] LIU Y, OTT M, GOYAL N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.
- [120] Ctags [EB]. 2023
- [121] XU Y H, PAN S J, XIONG H, et al. A unified framework for metric transfer learning[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(6): 1158-1171.
- [122] FENG Z Y, GUO D Y, TANG D Y, et al. Codebert: A pre-trained model for programming and natural languages[J]. arXiv preprint arXiv:2002.08155, 2020.
- [123] WANG W H, LI G, MA B, et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree[C]//Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). Piscataway: IEEE Press, 2020: 261-271.
- [124] ROSENBLUM N, MILLER B P, ZHU X J. Recovering the tool-chain provenance of binary code[C]//Proceedings of the 2011 International Symposium on Software Testing and Analysis. New York: ACM, 2011: 100-110.
- [125] PAGLIARDINI M, GUPTA P, JAGGI M. Unsupervised learning of sentence embeddings using compositional n-gram features[J]. arXiv preprint arXiv:1703.02507, 2017.
- [126] LE Q V, MIKOLOV T. Distributed representations of sentences and documents[C]//International conference on machine learning. PMLR, 2014: 1188-1196.
- [127] PENNINGTON J, SOCHER R, MANNING C. Glove: global vectors for word representation[C]//Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Stroudsburg, PA, USA: ACL, 2014: 1532-1543.
- [128] DAI H J, DAI B, SONG L. Discriminative embeddings of latent variable models for structured data[C]//International conference on machine learning. PMLR, 2016: 2702-2711.
- [129] GROVER A, LESKOVEC J. node2vec: scalable feature learning for networks[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York: ACM, 2016: 855-864.
- [130] HU Z N, DONG Y X, WANG K S, et al. Heterogeneous graph transformer[C]//Proceedings of the Proceedings of The Web Conference 2020. New York: ACM, 2020: 2704-2710.
- [131] RADFORD A, NARASIMHAN K, SALIMANS T, et al. Improving language understanding by generative pre-training[J]. 2018.
- [132] LI Y J, ZEMEL R, BROCKSCHMIDT M, et al. gated graph sequence neural networks[C]//Proceedings of ICLR'16. 2016.
- [133] BERG R V D, KIPF T N, WELLING M. Graph convolutional matrix completion[J]. arXiv e-prints, 2017: arXiv: 1706.02263.
- [134] VELIČKOVIĆ P, CUCURULL G, CASANOVA A, et al. Graph attention networks[J]. arXiv preprint arXiv:1710.10903, 2017.
- [135] 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: ACM, 2017: 363-376.
- [136] YING R, BOURGEOIS D, YOU J X, et al. GNNExplainer: generating explanations for graph neural networks[J]. Advances in Neural Information Processing Systems, 2019, 32: 9240-9251.
- [137] CHATTOPADHAY A, SARKAR A, HOWLADER P, et al. Grad-CAM: generalized gradient-based visual explanations for deep convolutional networks[C]//Proceedings of the 2018 IEEE Winter Conference on Applications of Computer Vision (WACV). Piscataway: IEEE Press, 2018: 839-847.
- [138] GUI Y, WAN Y, ZHANG H Y, et al. Cross-language binary-source code matching with intermediate representations[C]//Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Piscataway: IEEE Press, 2022: 601-612.
- [139] CHEN T Y, LI L, QIAN T T, et al. Identifying vulnerability patches by comprehending code commits with comprehensive change contexts[J]. arXiv preprint arXiv:2310.02530, 2023.
- [140] GAO Z Y, WANG H, ZHOU Y C, et al. How far have we gone in vulnerability detection using large language models[J]. arXiv preprint arXiv:2311.12420, 2023.
- [141] ZHANG C Y, LIU H, ZENG J Y, et al. Prompt-enhanced software vulnerability detection using chatgpt[J]. arXiv preprint arXiv:2308.12697, 2023.
- [142] BAKHSHANDEH A, KERAMATFAR A, NOROUZI A, et al. Using chatgpt as a static application security testing tool[J]. arXiv preprint arXiv:2308.14434, 2023.
- [143] CHESHKOV A, ZADOROZHNY P, LEVICHEV R. Evaluation of ChatGPT model for vulnerability detection[J]. arXiv preprint arXiv:2304.07232, 2023.
- [144] FU M, TANTITHAMTHAVORN C, NGUYEN V, et al. ChatGPT for vulnerability detection, classification, and repair: how far are we[EB]. 2023.
- [145] WANG X, WU Y X, ZHANG A, et al. Reinforced causal explainer for graph neural networks[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2023, 45(2): 2297-2309.
- [146] BARKAN O, ASHER Y, ESHEL A, et al. Learning to explain: a model-agnostic framework for explaining black box models[C]//Proceedings of the 2023 IEEE International Conference on Data Mining (ICDM). Piscataway: IEEE Press, 2023: 944-949.
- [147] 刘航天, 甘水滔, 张超, 等. 物联网设备固件自动化漏洞挖掘技术研究综述[J]. 网络与信息安全学报, 2025, 11(2): 26-49.
- LIU H T, GAN S T, ZHANG C, et al. Survey on automated vulnerability mining techniques for IoT device firmware[J]. Chinese Journal of Network and Information Security, 2025, 11(2): 26-49.

## [作者简介]



**李晴浩**（1998-），男，湖南邵阳人，信息工程大学博士生，主要研究方向为信息安全、代码相似性检测、漏洞检测。



**唐永鹤**（1983-），男，河南确山人，信息工程大学副教授，主要研究方向为计算机安全、恶意代码检测。



**刘春玲**（1981-），女，河南滑县人，信息工程大学讲师，主要研究方向为代码相似性检测、漏洞检测。



**董卫宇**（1976-），男，辽宁沈阳人，信息工程大学教授，主要研究方向为计算机架构、系统虚拟化、计算机安全、漏洞检测。