源代码漏洞静态分析技术

刘嘉勇[1], 韩家璇[1], 黄 诚[1]

[1]四川大学 网络空间安全学院 成都 中国 610207

摘要 漏洞这一名词伴随着计算机软件领域的发展已经走过了数十载。自世界上第一个软件漏洞被公开以来, 软件安全研究者和工程师们就一直在探索漏洞的挖掘与分析方法。源代码漏洞静态分析是一种能够贯穿整个软件开发生命周期的、帮助软件开发人员及早发现漏洞的技术, 在业界有着广泛的使用。然而, 随着软件的体量越来越大, 软件的功能越来越复杂, 如何表示和建模软件源代码是当前面临的一个难题; 此外, 近年来的研究倾向于将源代码漏洞静态分析和机器学习相结合, 试图通过引入机器学习模型提升漏洞挖掘的精度, 但如何选择和构建合适的机器学习模型是该研究方向的一个核心问题。本文将目光聚焦于源代码漏洞静态分析技术(以下简称: 静态分析技术), 通过对该领域相关工作的回顾, 将静态分析技术的研究分为两个方向: 传统静态分析和基于学习的静态分析。传统静态分析主要是利用数据流分析、污点分析等一系列软件分析技术对软件的源代码进行建模分析; 基于学习的静态分析则是将源代码以数值的形式表示并提交给学习模型, 利用学习模型挖掘源代码的深层次表征特征和关联性。本文首先阐述了软件漏洞分析技术的基本概念, 对比了静态分析技术和动态分析技术的优劣; 然后对源代码的表示方法进行了说明。接着, 本文对传统静态分析和基于学习的静态分析的一般步骤进行了总结, 同时对这两个研究方向典型的研究成果进行了系统地梳理, 归纳了它们的技术特点和工作流程, 提出了当前静态分析技术中存在的问题, 并对该方向上未来的研究工作进行了展望。

关键词 源代码漏洞; 静态分析; 数据流分析; 污点分析; 机器学习

中图法分类号 TP312 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.07.08

Vulnerability Detection In Source Code Using Statice Analysis

LIU Jiayong[1], HAN Jiaxuan[1], HUANG Cheng[1]

[1] School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China

Abstract The term vulnerability has gone through several decades with the development of the computer software field. Since the first software vulnerability in the world was made public, software security researchers and engineers have been exploring the methods of vulnerability mining and analysis. The static analysis of source code vulnerability is a technology that can run through the whole software development life cycle and help software developers find software vulnerabilities early. It is widely used in the industry. However, with the increasing volume and complexity of software, how to represent and model the software source code is a difficult problem at present. In addition, in recent years, researchers tend to combine static analysis of source code vulnerabilities with machine learning, trying to improve the accuracy of vulnerability mining by introducing machine learning model. Nonetheless, how to select and build a suitable machine learning model is a core issue in this research direction. This paper focuses on the static analysis technology of source code vulnerability (hereinafter referred to as static analysis technology), and reviews the related work in this field. The research of static analysis technology is divided into two directions: traditional static analysis and learning-based static analysis. Traditional static analysis mainly uses a series of software analysis technologies such as data flow analysis and taint analysis to model and analyze the source code of the software; learning-based static analysis represents the source code in numerical form and submits it to the learning model, then using the learning model to mine the deep representation features and relevance of the source code. This paper first expounds the basic concepts of software vulnerability analysis technology, and compares the advantages and disadvantages of static analysis technology and dynamic analysis technology. Next, the representation method of the source code is explained. After that, this paper summarizes the general steps of traditional static analysis and learning-based static analysis, and systematically combs the typical research results of these two research directions, summarizes their technical characteristics and workflow, puts forward the existing problems in the current static analysis technology, and looks forward to the future research work in these directions.

Key words source code vulnerability; static analysis; dataflow analysis; taint analysis; machine learning

---

通讯作者: 黄诚, 博士, 副教授, Email: codesec@scu.edu.cn.

本课题得到国家自然科学基金资助项目(No. 61902265)、四川省科技厅重点研发资助项目(No. 2020YFG0047, No. 2020YFG0076)资助.
收稿日期: 2021-08-07; 修改日期: 2021-11-01; 定稿日期: 2022-05-11

1 引言

随着互联网技术的发展, 计算机软件系统与用户隐私、资产等重要信息的关系越来越紧密, 大量的用户隐私数据被上传到云端存储。根据第 47 次《中国互联网络发展状况统计报告》[1], 截至到 2020 年 12 月, 我国网民的规模已经达到 9.89 亿, 互联网普及率达 70.4%。网络购物、线上支付、即时通信已然成为当今社会主流的生活方式。据统计, 2020 年疫情期间全国一体化政务服务平台推出的“防疫健康码”使用次数超过 400 亿次, 在线会议和课程等全新的工作和学习模式迅速融入到人们的生活中, 互联网技术为中国抗击疫情提供了强大的支持。然而, 互联网技术的普及也为各类软件系统的安全性提出了巨大的考验。根据 OWASP Top 10 2017 报告[2]显示, 由 node.js 和 Spring Boot 编写的微服务逐渐成为软件开发的新方向; 软件系统中高扩展性的、功能丰富的模块被攻击者们重点关注; XXE(XML External Entities)注入、反序列化攻击等手段逐渐成为主流。在巨大利益等因素的驱使下, 攻击者们不断尝试寻找各类软件系统中存在的漏洞, 试图绕过系统的访问控制, 实现窃取和修改重要数据、控制系统等非法操作。

软件漏洞检测一直是学术界和工业界讨论的核心问题。参考长亭科技发布的《2019 长亭年度漏洞威胁分析与 2020 安全展望》[3], 截至到 2019 年年底, 中国国家信息安全漏洞库(China National Vulnerability Database of Information Security, CNNVD)共收集了 17820 个漏洞, 中国国家信息安全漏洞共享平台(China National Vulnerability Database, CNVD)共收集了 16208 个漏洞, 相当于 2019 年每天约有 48 个漏洞被曝光。同时, 根据著名安全研究机构 SkyBox Security 发布的《2020 Vulnerability and Threat Trends Report Mid-Year Update: Key Findings》报告[4]显示, 2020 年上半年有 9000 多个漏洞被曝光, 移动端漏洞的数量增长迅速。虽然企业为其软件系统制定了一系列安全方案, 也配置和部署了各类安全设备, 但软件漏洞仍然被频频曝出; 其根本原因是开发人员缺乏安全意识, 在软件开发时就为其埋下了不安全因素。此外, 随着计算机软件领域的发展, 软件代码开源化已然成为一种趋势。现如今, 软件的功能越来越多, 系统越来越复杂, 开发人员将开源代码引入到项目中能够极大地提高开发效率; 但在增加开发便利度的同时, 引入开源代码也增加了软件系统存在漏洞的可能。

软件漏洞的检测方法主要有三种: 静态检测(又称: 静态分析)、动态检测(又称: 动态分析)和动静结合的检测(又称: 动静结合的分析或混合分析)。静态分析是指在不运行软件程序的前提下, 对软件代码进行抽象建模, 通过分析程序的属性进而实现漏洞检测; 动态分析是指向软件程序输入特定构造的数据, 观察程序的运行状态, 通过状态判断程序是否存在漏洞; 混合分析则是一种将静态分析和动态分析相结合的混合式漏洞检测方法。本文以面向源代码的静态分析领域典型的研究成果作为切入点, 将该领域的研究分为传统静态分析和基于学习的静态分析两个方向, 分别对这两个方向上的研究进展和研究成果进行归纳总结, 讨论该领域目前存在的困难并对未来的发展方向进行展望。

2 软件漏洞分析技术基本概念

2.1 静态分析技术

静态分析技术是指在不运行程序代码的情况下, 对其进行词法分析、语法分析以及语义分析, 配合数据流分析和污点分析等技术, 对程序代码进行抽象和建模, 分析程序的控制依赖、数据依赖和变量受污染状态等信息, 通过安全规则检查、模式匹配等方式挖掘程序代码中存在的漏洞[5-7]。常见的静态分析工具有: WALA[8]、FindBugs[9]、JSPrime[10]、CodeQL[11]、Fortify[12]、Cppcheck[13]、Cobot[14]等。

依据分析目标的不同, 静态分析可分为: 面向源代码的静态分析和面向二进制代码的静态分析。面向源代码的静态分析以程序的源代码作为输入, 将其转换为某种特定形式的中间表示, 基于该中间表示进行分析。因为分析是基于源代码进行的, 所以能够捕获丰富的代码结构、语义和逻辑等信息。面向二进制代码的静态分析则是以经过反汇编等手段处理后的二进制代码作为输入, 设法恢复程序的信息, 运用模式匹配或补丁对比等方式实现漏洞检测。相比源代码, 二进制代码缺乏与代码相关的高级语义信息, 且以二进制代码形式表示的漏洞模式更为复杂; 然而, 由于存在一部分软件程序是以二进制形式发布的, 并不包含软件程序的源代码, 所以对于非程序开发方的安全人员而言, 研究面向二进制代码的漏洞分析是很重要的。

2.2 动态分析技术

与静态分析不同, 动态分析是指在沙箱等受控环境中执行程序, 向程序输入特定的数据, 监视其运行时的行为, 收集函数的执行结果、程序的异常行为和崩溃情况等信息以判断目标程序是否存在漏

洞[15-16]。常见的动态分析工具有: AFL[17]、Sage[18]和jsfunfuzz[19]等。动态分析技术包括动态符号执行和模糊测试两种。其中, 符号执行的目标是获得让特定代码区域执行的输入, 通常分为静态符号执行和动态符号执行两种。静态符号执行不会真正执行程序, 也不会向程序提供具体的数据输入; 而是通过将程序的输入抽象为符号, 使用约束求解器求解, 进而获得让特定代码区域执行的输入, 如文章[20]中所做的工作; 动态符号执行则是将具体执行和符号执行相结合, 以具体的值作为程序的输入, 执行程序, 收集符号约束, 修改收集的符号约束内容以构造不同的可执行路径, 进而实现对程序所有路径的遍历[21]。相比于动态符号执行, 模糊测试是近年来动态分析领域研究的热门。模糊测试是 Miller 等在 1991 年提出的一个概念[22], 其核心思想是通过向程序输入畸形数据, 观察程序的执行状态(如: 程序的异常行为和崩溃情况), 进而判断程序是否存在漏洞。

2.3 差异性分析

静态分析和动态分析的差异性对比如表 1 所示。静态分析和动态分析面向的目标不同; 静态分析面向的是软件源代码和二进制代码, 而动态分析大多面向软件程序本身。通常, 静态分析可以贯穿整个软件生命周期, 辅助开发人员及早地发现软件代码中存在的问题; 分析过程不需要将程序实际地运行起来, 而是对程序代码进行抽象和建模, 因此分析器能够在低资源需求的前提下实现高代码覆盖率。但是, 由于分析器对程序代码具有极高的依赖性, 导致在程序代码缺失的情况下难以对程序进行分析; 由于分析是基于对程序代码的抽象和建模, 分析器的分析逻辑很大程度上取决于对漏洞已有的先验知识, 所以在面对未定义的程序错误行为时, 静态分析往往存在误报率高的问题; 此外, 静态分析还存在运行时漏洞检测难问题。

对于动态分析而言, 因为分析器需要对程序的运行时信息进行跟踪、收集和分析, 所以对系统环境的要求较高, 需要为不同的程序配置不同的运行环境; 但是由于是基于程序的实际运行情况进行分析的, 因此分析的精确度较高, 而且能够对经过混淆的代码进行检测。不过, 因为需要运行程序观察其执行情况, 而当前环境下很多程序的代码空间很大, 分析器在短时间内难以覆盖整个程序空间, 所以动态分析的漏报率相对较高; 而且以动态符号执行为主的动态分析还面临着路径爆炸[23]等问题。

表 1 静态分析和动态分析差异性对比

Table 1 Comparison of differences between static analysis and dynamic analysis

分析方法	优点	缺点
静态分析	1. 能够在软件生命周期早期发现漏洞 2. 资源消耗少 3. 检测耗时短 4. 代码覆盖率高 5. 漏报率低	1. 在缺少源码(如: 程序引用的库文件源码缺失)的情况下, 需要猜测缺失源码的行为 2. 难以处理未定义的程序错误行为 3. 对漏洞先验知识的依赖性大 4. 难以检测运行时的漏洞 5. 误报率高
动态分析	1. 误报率低 2. 能够检测运行时的漏洞 3. 检测精度高 4. 能够处理经过混淆的代码	1. 代码覆盖率低 2. 存在路径爆炸问题 3. 漏报率高 4. 检测耗时长

无论是静态分析技术还是动态分析技术, 都能够为处在互联网快速发展背景下的软件系统安全提供强有力的保障。近年来, 为了在软件系统上线前就对其可能存在的漏洞进行检测, 将软件系统面临的风险扼杀在摇篮中, 进一步减少软件系统上线后可能受到的安全威胁, 越来越多的企业开始在软件开发过程中使用静态分析对软件漏洞进行挖掘。本节对软件漏洞分析技术的基本概念进行了解释, 让读者对该领域的基本情况有一个初步的了解。在接下来的章节中, 我们会对静态分析领域的相关概念和技术、研究进展以及研究成果进行详细阐述, 并给出我们对该领域未来发展趋势的一些看法。

3 源代码漏洞静态分析技术

3.1 源代码的表示方法

将源代码转换成合适的中间表示是进行源代码漏洞静态分析的第一步。源代码的表示方法主要分为两大类: 树形表示和图形表示。树形表示主要以抽象语法树(Abstract Syntax Tree, AST)为主; 图形表示主要包括: 控制流图(Control Flow Graph, CFG)、数据流图(Data Flow Graph, DFG)、调用图(Call Graph, CG)、程序依赖图(Program Dependence Graph, PDG)、系统

依赖图(System Dependence Graph, SDG)和代码属性图(Code Property Graph, CPG)。如图 1 所示, AST 是由代码解析器对源代码进行词法分析和语法分析后生成的最基本的中间表示, 也是其他中间表示的基础, 能够清楚地反映源代码的结构信息, 在源代码漏洞检测中有着很重要的作用。陈肇炫等人[24]提出的 Astor 模型[24]通过将源代码转换成 AST, 采用深度优先遍历算法获取 AST 的语法表征并训练神经网络模型对其进行学习, 实现了基于源代码结构信息的智能化漏洞检测。Hantao Feng 等[25]将源代码转换成 AST, 然后将 AST 转换为序列, 利用词嵌入技术将序列转换为向量, 保留了源代码的语义特征; 通过训练 BGRU (Bidirectional Gate Recurrent Unite)模型, 从向量中提取特征以实现漏洞检测。文章[26]从源代码中提取 AST, 将其嵌入到向量空间中, 然后使用潜在语义分析技术[27]识别代码的结构模式, 从而实现漏洞外推。

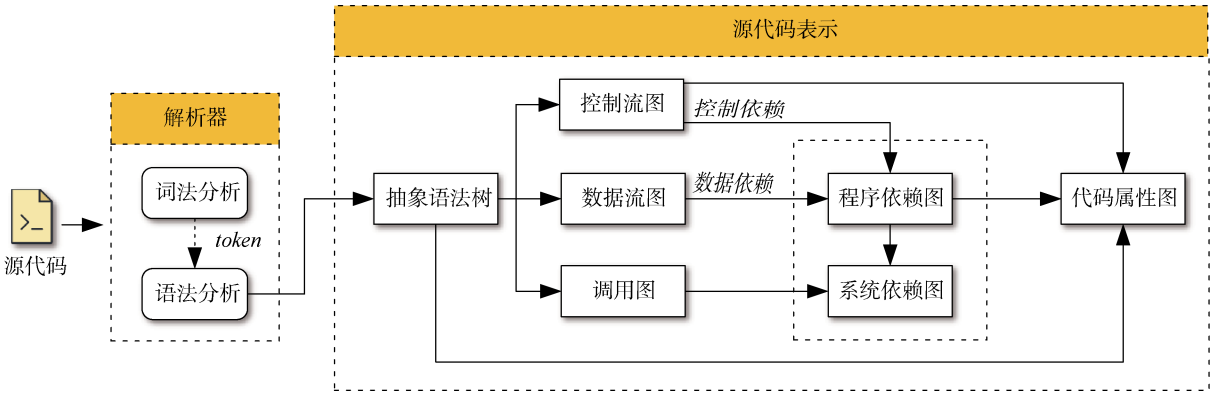

图 1 源代码表示方法的生成流程

Figure 1 Generation process of source code representation method

CFG、DFG 和 CG 可以通过分析 AST 直接获得。CFG 能够描述语句的执行顺序以及语句间的支配关系(即: 控制依赖关系), DFG 能够表示语句和变量间的数据依赖关系, CG 能够表示函数的调用情况。通过从 CFG 中提取控制依赖关系, 从 DFG 中提取数据依赖关系, 将两种关系融合到同一张图中, 进而形成 PDG; 再依据 CG 将不同函数的 PDG 连接起来, 形成过程间程序依赖图, 即 SDG。

CPG 是 Yamaguchi 等人在论文[28]中提出的一种全新的源代码中间表示形式, 是当前源代码漏洞静态分析技术中最新、使用最为广泛的源代码图形表示, 由 AST、CFG 和 PDG 合并而成, 如图 2 所示。

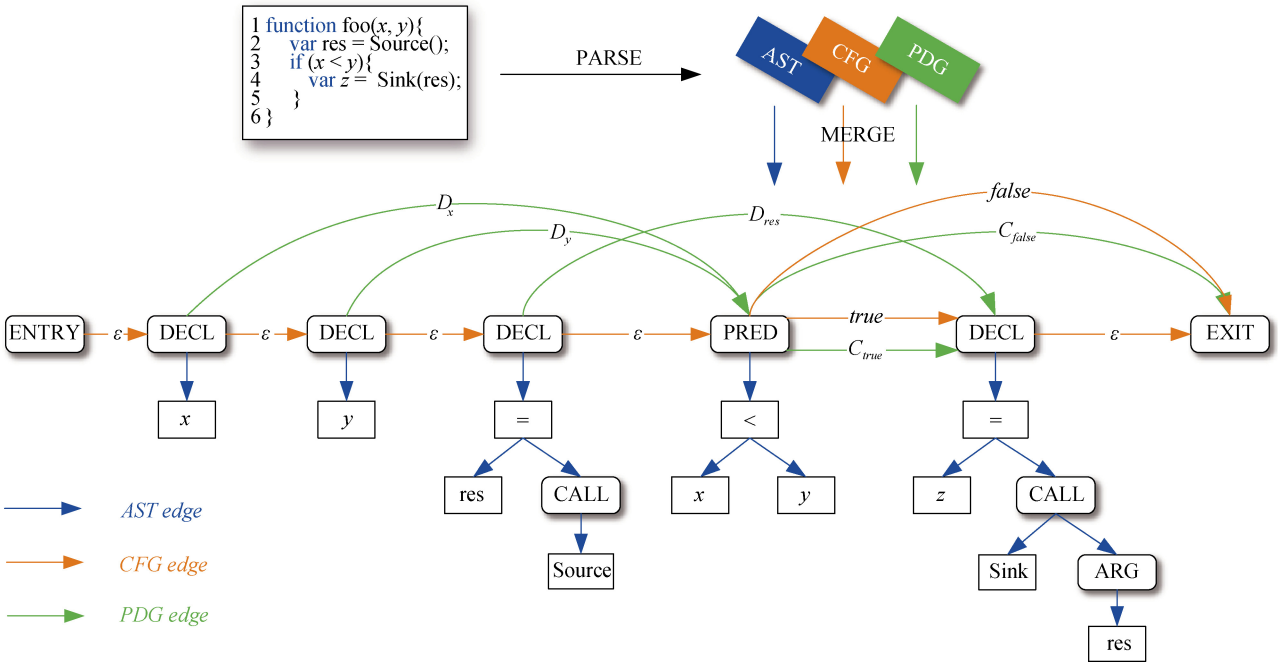

图 2 CPG 示例

Figure 2 CPG example

相比于其他源代码中间表示形式, CPG 能够很好地反映源代码的结构、语句执行顺序、控制依赖和数据依赖等信息; 在文章[28]中, 作者首次提出使用 CPG 表示源代码, 采用图形数据库 Neo4J[29]存储并遍历 CPG 以实现漏洞检测。同时, Yamaguchi 等在文章[30]中还提出了一种自动推断 C 代码中污点型漏洞搜索模式的方法, 该方法通过对 CPG 进行扩展以支持过程间分析; 通过给定 Sink(Sink 指污点的汇聚点), 该方法能够自动识别对应的 Source-Sink(Source 指污点源)系统并对系统中的数据流和 Sanitization (Sanitization 指对污点数据的无害处理, 即净化)进行建模, 生成污点型漏洞的搜索模式; 基于推断的搜索模式对 CPG 进行遍历, 从而检测源代码中存在的漏洞。Peng Wu 等[31]提出基于 CPG 从 AST 和 API 的子树中提取代码特征, 然后使用词袋模型和 TF-IDF 模型将代码特征嵌入到向量空间; 同时从 CVE 上提取漏洞代码及其补丁的代码切片, 计算源代码、漏洞代码、补丁代码之间的相似性以实现漏洞检测。

3.2 源代码的分析方法

源代码的分析方法可以分为两类, 一类是传统静态分析, 另一类是基于学习的静态分析。传统静态分析主要基于数据流分析和污点分析, 通过对源代码进行抽象、建模和分析实现漏洞检测; 基于学习的静态分析则是将机器学习技术运用到静态分析中, 利用其强大的大数据挖掘能力, 学习源代码的运行逻辑、数据流动等信息, 进一步通过学习模型判断源代码中是否存在漏洞。下面将对这两种分析方法的实现原理和典型技术进行详细介绍。

3.2.1 传统静态分析技术

传统静态分析的基本流程如图 3 所示。首先需要对待检测源代码进行抽象和建模, 通过解析器执行词法分析和语法分析, 生成特定的中间表示; 然后基于该中间表示, 采用预设的漏洞分析规则, 结合数据流分析和污点分析, 收集源代码中与漏洞相关的信息; 最后, 对收集的信息进行分析, 给出检测结果。

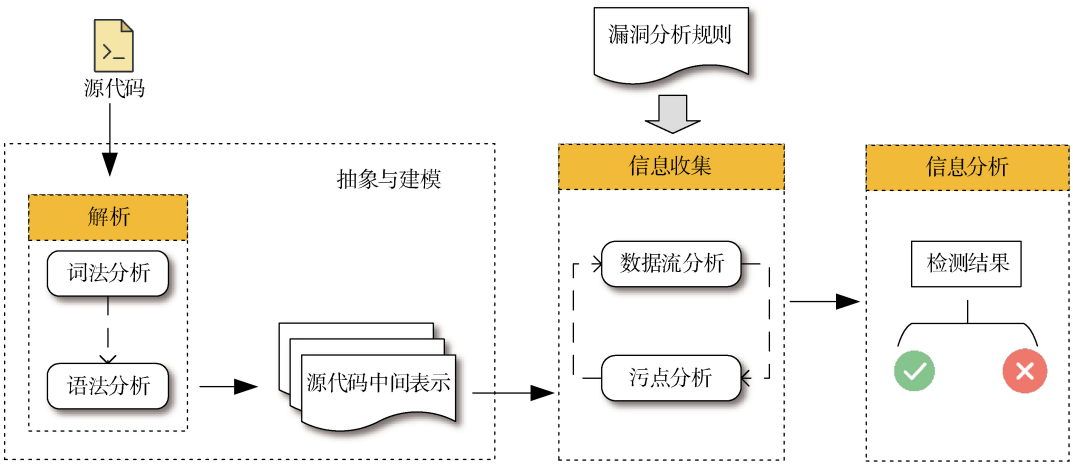

图 3 传统静态分析基本流程

Figure 3 The basic process of traditional static analysis

传统静态分析的核心技术是数据流分析和污点分析。数据流分析是一种基于格(lattice)理论的、用来获取相关数据(如: 变量及其取值)沿着程序执行路径流动的信息的程序分析技术, 常用于源代码的编译优化过程, 能够获得变量在某个程序点上的性质、状态、取值等信息[32-33]。常用的数据流分析方法有:

(1) 到达定义分析(Reaching Definition Analysis): 给定一个程序点 p 和变量定义 d, 判断在 CFG 上是否存在一条从 d 到 p 的路径, 该路径上没有对 d 进行重新定义的指令。如果存在这么一条路径, 则称定义 d 可以到达程序点 p; 否则, 定义 d 不能到达程序点 p。

(2) 存活变量分析(Live Variable Analysis): 给定一个程序点 p 和变量 v, 判断在 CFG 上是否存在一条从 p 开始的路径, 该路径上有使用了 v 的指令。如果存在这么一条路径, 则称变量 v 在程序点 p 处是存活的; 否则, 变量 v 在程序点 p 处不存活。

(3) 可用表达式分析(Available Expression Analysis): 给定一个程序点 p 和表达式 x op y(其中, op 表示 operator, 指操作符), 判断在 CFG 上从 ENTRY 节点到 p 的所有路径是否都执行了 x op y, 且最后一次执行 x op y 后, 没有对 x 和 y 进行重新定义。如果满足上述条件(即: CFG 上从 ENTRY 节点到 p 的所有路径都执行了 x op y, 且最后一次执行 x

op y后没有对x和y进行重新定义), 则称表达式x op y在程序点p处是可用的; 否则, 表达式x op y在程序点p处不可用。

除了上面提到的三种数据流分析方法外, 数据流分析方法还包括: 常量传播分析(Constant Propagation Analysis)[34]、指针分析(Pointer Analysis)[35]等。常量传播分析的目标是判断在给定程序点p处指令i中的变量v的值是否为常数; 指针分析则是一种用于分析变量指向的技术, 如: 对于Java语言, 利用指针分析判断给定变量v所指向的对象o(Object)。

污点分析是一种信息流分析技术, 通过对程序中的敏感数据进行标记, 跟踪标记数据在程序中的传播, 从而检测系统中存在的安全问题[36-37]。根据王蕾等[38]的论文, 污点分析的基本流程如图4所示。污点分析被定义为三元组<Source, Sink, Sanitizer>, 建立在SC: {Tainted, Untainted}集合的格上。如果信息从Tainted类型的变量(图中标记为“污点变量”)传递到Untainted类型的变量(图中标记为“变量”), 则Untainted类型的变量将被标记为Tainted类型; 如果在Tainted类型变量的传播过程中对其进行了净化处理, 则当其传递到Tainted类型或Untainted类型的变量时, 不会改变目标变量的污染状态。在污点分析过程中, 如果Tainted类型的变量能够传递到Sink, 则说明当前程序存在安全问题。

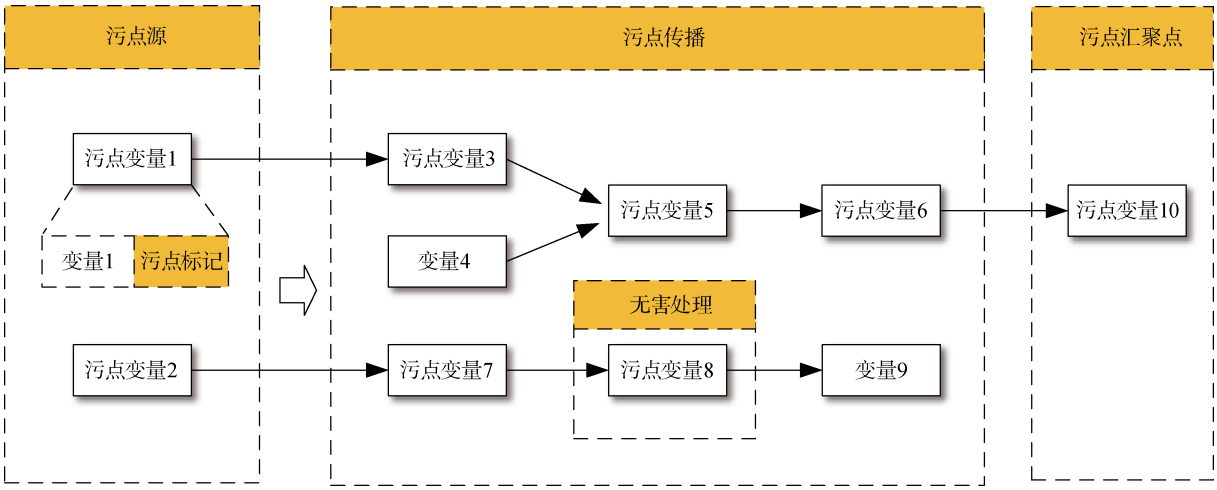

图4 污点分析基本流程

Figure 4 The basic process of taint analysis

传统静态分析技术在软件源代码漏洞检测中有着广泛的应用。Johannes Dahse等[39]对PHP语言进分析, 为每个PHP文件构建对应的AST, 从中提取出用户自定义函数的相关信息, 如: 函数名和参数, 同时构建函数体的AST, 然后将用户自定义函数从前面为PHP文件构建的AST(文中称为主AST)中删除; 接着, 使用名为CFGBuilder的模块将获取的AST转换为CFG; 在构建CFG的过程中, 对每个基本块进行数据流分析和污点分析, 将分析结果存储在一个称为块摘要(Block Summary)的数据结构中, 以便后续执行过程间分析使用。在进行污点分析时, 为保证分析的准确性, 作者对925个PHP内置函数进行建模, 同时还引入了字符串分析(String Analysis); 此外, 通过对包含的文件(即: 通过include等关键字引入的文件)进行建模, 将包含的文件视作为函数, 进一步提高了分析的精确度。作者让模型在5个开源软件上进行了测试, 平均TP(True Positive)达到了72%, 平均FP(False Positive)达到了28%, 平均FN(False Negative)达到了24%。

Xuexiong Yan等[40]提出一种基于后向污点分析的Web应用程序漏洞检测模型。该模型首先从PHP代码中提取AST、CFG和CG, 然后对Sink进行查找, 构建与Sink相关的上下文信息; 接着, 执行污点分析, 在基本块内和基本块间跟踪敏感变量的流动, 最终在测试数据集上成功检测出13个漏洞。Pixy[41]是Nenad Jovanovic等提出的用于检测Web应用程序漏洞的静态分析工具。该工具采用流和上下文敏感的过程间数据流分析方法, 同时基于别名分析和常量传播分析, 对采用PHP语言编写的Web应用程序漏洞进行检测。

Libo Chen等[42]提出一种针对嵌入式设备中提供的Web服务漏洞的污点分析方法。作者发现网络接口上的字符串通常在前端文件和后端二进制文件之间共享以编码用户输入。基于这个发现, 该方法首先在未打包固件的前端文件(HTML、XML和JavaScript文件)中提取关键字, 然后基于前端提取的

关键字, 在后端二进制文件中查找与关键字对应的入口点, 利用路径探索和污点分析技术跟踪输入数据的流动, 实现对嵌入式设备中提供的 Web 服务漏洞的检测。作者使用该方法成功在包括 Tenda W20E 在内的 12 款路由器中找出了 33 个 bug, 其中有 30 个已经被确认。

Yichen Xie 和 Alex Aiken[43]针对 PHP 语言提出了一种结合基本块内、过程内和过程间分析的脚本语言漏洞检测模型。该模型将输入的 PHP 代码解析为 AST, 基于 AST 构建对应的 CFG, 然后使用符号执行技术分析 CFG 的每一个基本块并对基本块内的动态特性进行建模, 生成块摘要; 接着使用可达性分析将每个块摘要合并成相应的函数摘要, 基于函数摘要实现过程间分析。此外, 该模型维护了一个已知正则表达式的数据库(包含正则表达式的效果), 支持对净化的分析, 进一步提高了对 PHP 语言建模的准确性。作者在 6 个流行的开源 PHP 项目代码上对模型进行了测试, 发现了 105 个可疑的漏洞。

V. Benjamin Livshits 和 Monica S. Lam[44]构建了一个 Eclipse 的 Java 应用程序静态分析插件工具。该工具向使用者提供基于 PQL(Program Query Language, 程序查询语言)的分析接口, 采用上下文敏感的指针分析对污点对象的传播进行有效地建模, 实现了对 Java 应用程序中存在漏洞的检测。作者使用该工具在 9 个开源 Java 应用程序中发现了 29 个安全漏洞。

针对 RCE(Remote Code Execution, 远程代码执行)漏洞, Yunhui Zheng 等[45]提出了一种基于路径敏感的静态分析方法。该方法对目标 PHP 代码进行切片, 删除与检测目标无关的语句; 然后对切片中针对字符串和非字符串类型处理的行为进行抽象和建模, 基于路径敏感和上下文敏感的过程间分析对变量进行跟踪, 进而能够跨多个脚本检测 RCE 漏洞。此外, 该方法还能够对动态脚本包含进行处理, 进一步提高了分析的准确性。作者在 10 个 PHP 应用程序上测试了该方法, 成功发现了 21 个真实的 RCE 漏洞, 其中有 8 个是以前从未报告过的。

3.2.2 基于学习的静态分析技术

基于学习的静态分析基本流程如图 5 所示。无论是机器学习模型还是深度学习模型, 都无法直接处理以源代码作为输入的情况, 因此需要对源代码进行预处理。通常, 源代码都是以文本形式给出的, 第一步是对其进行解析或构建程序切片以保留与漏洞检测相关的信息; 接着使用词嵌入等技术将源代码中间表示或切片映射到向量空间; 最后借助机器学习或深度学习模型强大的大数据挖掘能力学习源代码蕴含的各类信息(如: 控制依赖和数据依赖信息), 进而实现漏洞检测。同时, 可以利用传统静态分析方法提取源代码污点变量的传播情况、净化函数的有效性等信息, 丰富模型学习的知识空间, 从而获得性能更好的检测模型。

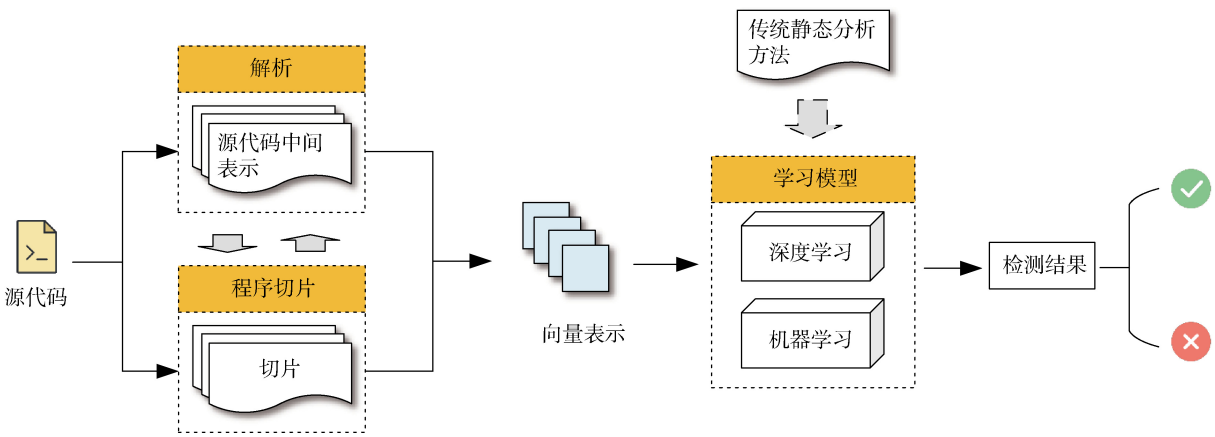

图 5 基于学习的静态分析基本流程

Figure 5 The basic process of learning-based static analysis

VulDeePecker 是 Zhen Li 等[46]提出的一种基于深度学习技术的 C/C++漏洞检测模型。该模型首先提取与 library/API 函数相关的调用并获取与这些调用相关的参数或变量的程序切片; 然后将获得的程序切片组装成一系列在控制依赖或数据依赖上相关的语句集合(文中称为代码 gadget); 接着将代码 gadget 映射为长度相等的向量并利用其训练 BiLSTM(Bi-directional Long Short-Term Memory)模型以实现漏洞检测。作者将 VulDeePecker 应用在 Xen、Seamonkey 和 Libav 这 3 个软件产品上, 检测到了 4 个未出现在 NVD 中的漏洞。

Yuelong Wu 等[47]将 CPG 进行简化, 使其只包含

AST 和 CFG 相关的边, 并结合图神经网络和多层感知机学习代码的图形表示以从中提取特征。Yaqin Zhou 等[48]提出的 Devign 模型以源代码 AST 为主干, 将 CFG、DFG 以及自然代码序列(Natural Code Sequence, NCS)引入其中, 通过过滤、删除等操作对融入了额外信息的 AST 进行优化, 构建了一个称为联合图(Joint Graph, JG)的代码图形表示; 最终使用图神经网络学习源代码综合的语义信息进而实现漏洞检测。经过测试, 该模型相较现有技术, 平均准确率提高了 10.51%, 平均 F1 评分提高了 8.68%。

FTCLNet 模型[49]创新性地将基于傅里叶变换的深度卷积 LSTM(Long Short-Term Memory)神经网络引入到源代码漏洞检测中, 通过代码重写技术对源代码进行规范化, 使其具有统一的代码样式和标识符命名规范, 并基于重写后的代码提取前向切片和后向切片; 接着, 采用预定义的 token 映射表对切片进行编码, 将其映射到向量空间并进行代码嵌入操作。使用傅里叶变换将代码映射到频域, 提取代码的局部特征和全局特征, 使用注意力机制确定代码空间中每个元素的权重。实验表明, 针对缓冲区溢出错误(CWE-119), 模型的 F1 评分达到 90.69%; 针对资源管理错误(CWE-399), 模型的 F1 评分达到 95.69%。

Ibéria Medeiros 等[50]提出了一个以自然语言处理技术为主导的 PHP 源代码漏洞检测工具 DEKANT。该工具首先收集一系列存在漏洞和不存在漏洞的 PHP 代码, 然后对收集的代码进行过程内和过程间分析, 提取从程序入口点开始到 Sink 结束的切片; 通过预定义的中间切片语言(Intermediate Slice Language, ISL)语法规则, 将切片转换为 ISL 表示(即: 将切片转换为 token 表示), 并为每个 token 分配对应的状态(token 和 state(状态)组成的<token, state>二元组, 称为观测序列)。对观测序列进行去重后构建用于训练 HMM(Hidden Markov Model)模型的语料库; 接着, 从语料库中提取知识(HMM 模型的参数)并用概率矩阵表示, 最后训练 HMM 模型实现漏洞检测。作者将模型应用在一组开源的 PHP 软件程序和 WordPress 插件上, 发现了 16 个 0-day 漏洞。

Xin Li 等[51]认为编程语言是人类和计算机之间沟通的桥梁, 与自然语言有着相同的功能, 因此可以将自然语言处理领域的相关技术迁移到对代码的处理中。通过对源代码进行依赖分析、安全切片、标签化(tokenization)和序列化(serialization), 获得一个名为最小中间表示的源代码中间表示结构。接着使用 CBoW(Continuous Bag-of-Word)模型获取源代码的分布式向量表示; 然后使用三个级联的神经网络来学习源代码的高级特征, 最终使用卷积神经网络实现漏洞检测。作者在测试数据集上对模型进行了全方位的评估, 模型最终实现了 1.5%的 FPR(False Positive Rate)、9.6%的 FNR(False Negative Rate)、95.7%的精确率、90.4%的召回率以及 93.0%的 F1 评分。

Huanting Wang 等[52]提出了一种基于 GGNN (Gated Graph Neural Network)的、能够跨编程语言迁移的漏洞检测模型。模型以源代码的函数为输入单位生成对应的 AST。为了获取源代码的语法、数据流和控制流等信息, 文章向 AST 添加了八种类型的边以构成扩展的 AST。接着使用 Word2Vec 将扩展的 AST 节点和 token 映射为向量, 训练神经网络模型实现漏洞检测。作者使用该模型对与包括 CWE-400、CWE-772 在内的前 30 种 CWE 漏洞类型相关的 C 函数进行了检测, 平均准确率达到 92.0%。此外, 作者还提出了一种名为专家混合的模型以解决训练样本短缺的问题。

DeepWuKong[53]是 Xiao Cheng 等提出的一种基于深度图神经网络的源代码漏洞检测模型。作者受到“易受攻击代码和安全代码之间的代码模式可能有很大的不同, 这些不同主要表现在代码的 token、API、控制流和数据流这四个方面”这一观点的启发, 提出一种新的代码切片算法以捕获源代码的高级语义特征。通过对输入的源代码构建过程间控制流图(Inter-procedural Control Flow Graph, ICFG)和值流图(Value Flow Graph), 在别名分析的配合下, 对控制流信息和数据流信息进行整合, 进而构建 PDG。接着采用提出的代码切片算法对 PDG 进行遍历, 提取对应的 XFG(PDG 的子图); 然后基于 XFG 提取结构化信息(XFG 的边)和非结构化信息(使用 Doc2Vec 将代码转换成的向量), 最后训练 GNN 模型实现漏洞检测。作者将 DeepWuKong 运用到 redis 和 lua 两个开源应用程序中, 实现了 5.0%的 FPR、10.0%的 FNR、93.0%的准确率以及 90.0%的 F1 评分。

4 分析与讨论

本文立足于源代码漏洞静态分析技术, 介绍了该领域典型研究的技术原理和方法特点, 并从源代码分析方法、中间表示方法、检测目标、关键工具或技术 4 个方面对上文中涉及的文献进行总结和归纳, 如表 2 所示。同时, 我们还对这些文献中提到的开源数据集或代码进行了总结和归纳, 以支持后续对该领域的进一步研究, 结果如表 3 所示。

表 2　文献关键信息总结与归纳

Table 2　Summary and induction of key information in literatures

源代码分析方法	中间表示方法	文献	检测目标	关键工具或技术
传统静态分析	CPG	[28]	C/C++	1. 基于 Joern[54]对 C/C++代码进行解析, 获取 CPG 2. 使用 Neo4J 图形数据库存储 CPG 3. 使用 Gremlin[55]图语言实现对 CPG 的遍历
	CFG	[41]	PHP	1. 使用 Java 词法分析器 JFlex[56]、Java 解析器 Cup[57]、PHP 解释器(PHP interpreter)的 Flex 和 Bison 文件构成 PHP 代码解析器
		[39]		
		[43]		
	AST CFG CG	[40]	PHP	1. 使用 PHP Parser[58]对 PHP 代码进行解析
	AST CFG	[42]	嵌入式系统	1. 使用标准 XML 处理库解析 XML 文件 2. 使用 Js2Py[59]解析 JavaScript 代码 3. 使用 Ghidra 库[60]和扩展的 KARONTE 的 CPF 模块[61]实现对输入条目的识别 4. 使用 angr[62]实现对二进制文件的污点分析
	程序切片	[44]	Java	1. 基于 joeq Java 编译器[63]和 bddbddb 程序分析工具[64]实现对 Java 应用程序的静态分析
		[45]	PHP	1. 使用 Phc[65]将 PHP 代码转换为 C 代码 2. 使用 LLVM[66]对 C 代码进行解析 3. 依赖于 STP 求解器[67]和 HAMPI 字符串求解器[68]对源代码中的非字符串和字符串行为进行建模
基于学习的静态分析	AST	[26]	C/C++	1. 基于岛语法[69]和 ANTLR[70]构建 C/C++代码解析器, 提取源代码的 AST 2. 使用潜在语义分析技术识别漏洞代码的结构模式 3. 使用 TF-IDF 将 AST 嵌入到向量空间
	CPG	[30]	C	1. 基于 Joern 对 C 代码进行解析 2. 使用 fastcluster[71]库进行聚类 3. 使用完全链接聚类(Complete-linkage Cluster)技术对被调用者和类型进行聚类 4. 使用 BoW(Bag of Word)模型将定义组合映射到向量空间 5. 使用 Gremlin 图语言实现遍历
		[31]		1. 使用 TF-IDF 和 BoW 模型将代码特征嵌入到向量空间 2. 利用向量之间的余弦距离计算函数的相似性 3. 使用 python 的 genism 库进行特征嵌入和相似度计算 4. 使用 Joern 获取 CPG 5. 使用 Gremlin 图语言进行图遍历 6. 使用 Neo4J 图数据库存储 CPG
	代码切片	[46]	C/C++	1. 利用 Checkmarx[72]生成的 PDG 提取切片 2. 使用 Word2Vec 进行将代码 token 转换为向量 3. 使用 BiLSTM 神经网络学习代码特征
		[49]		1. 使用傅里叶变换将代码空间转换到频域 2. 结合 CNN 和 BiLSTM 以提取频域的局部和全局特征 3. 使用 Word2Vec 将代码切片转换为向量 4. 使用 LLVM pass 删除条件编译 5. 利用注意力机制确定代码空间中各个元素的权重
		[50]	PHP	1. 使用 HMM 模型依据代码切片表征漏洞

续表

源代码分析方法	中间表示方法	文献	检测目标	关键工具或技术
基于学习的静态分析	简化的 CPG	[47]	C/C++	1. 使用 Joern 提取 CPG 2. 使用 one-hot 技术对简化的 CPG 中的节点进行编码 3. 使用 Word2Vec 对编码进行嵌入 4. 使用 GNN 和多层感知机学习代码的图形表示并提取特征
	JG	[48]	C	1. 使用 Joern 获取源代码的 AST 和 CFG 2. 使用 Word2Vec 进行嵌入 3. 使用 GGNN 学习代码的联合图表示
	最小中间表示	[51]	C	1. 使用 genism 库实现 CBoW 模型以获取最小中间表示的分布式向量表示 2. 使用 3 个级联的 CNN 学习漏洞的高级特征
	增强的 AST	[52]	C/C++ Java Swift PHP	1. 使用 Word2Vec 将代码图形表示的节点和 token 映射为向量 2. 使用 GGNN 捕获和推理程序的控制依赖、数据依赖以及调用关系 3. 使用 Soot[73]解析 Java 代码 4. 使用 ANTRL 解析 Swift 和 PHP 代码 5. 使用 Joern 解析 C/C++代码 6. 使用 SVM、逻辑回归、KNN、随机森林和 GB(Gradient Boosting)作为预测模型构建数据收集模型 7. 基于 PyCP 库[74]的保形预测(Conformal Prediction)对每个预测模型预测结果的置信度进行量化
	XFG	[53]	C/C++	1. 基于 SVF[75]提取控制流和数据流信息 2. 使用 ANTLR 识别逐位运算符、算术运算符、复合赋值表达式、自增和自减表达式 3. 使用 Doc2Vec 将 XFG 的节点转换为向量表示 4. 采用 GNN 对 XFG 进行分类

表 3　开源数据集或代码

Table 3　Open source dataset or code

文献	开源数据集或代码	链接
Yamaguchi F et al. [30]	代码	https://github.com/fabsx00/querygen
Li Z et al. [46]	数据集	https://github.com/CGCL-codes/VulDeePecker
Wu Y et al. [47]	数据集	https://samate.nist.gov/SRD/testsuite.php
Zhou Y et al. [48]	数据集	https://sites.Google.com/view/devign
Wang H et al. [52]	代码和数据集	https://github.com/HuantWang/FUNDED_NISL
Cheng X et al. [53]	数据集	https://github.com/DeepWukong/DeepWukong

随着计算机软件领域的发展, 软件程序与我们生活的联系越来越密切, 对软件源代码漏洞静态分析技术的需求也会随之变大; 因此, 对源代码漏洞静态分析技术的研究势必会成为今后软件安全研究者们关注的重点。然而, 在对该领域典型的研究进行深入剖析后, 我们认为在进行源代码漏洞静态分析时, 主要存在以下 5 个难点:

(1) **源代码的表示:** 无论是传统静态分析还是基于学习的静态分析, 都需要将源代码转换为某种中间表示。不同的中间表示所包含的源代码信息不同, 如: AST 仅能够表示源代码的结构信息, 无法提供控制流和数据流信息; CFG 能够给出源代码语句之间的控制依赖关系, 但无法提供数据依赖信息。如何选择和构建合理的源代码中间表示方法以尽可能多地包含源代码信息是当前的一个难点; 此外, 对于基于学习的静态分析, 由于学习模型的输入是向量形式, 所以如何有效地将源代码映射到向量空间, 同时尽可能减少源代码原有信息的损失也尤为重要。

(2) **源代码的建模:** 对于传统静态分析而言, 如何对源代码进行有效建模十分重要, 源代码的建模方式直接决定了分析的准确性。通常的做法是对程序的结构和行为进行抽象, 如: 对程序的顺序、选择

和循环结构进行抽象, 亦或是对字符串和非字符串处理例程进行抽象, 以此实现对程序中语句执行逻辑的模拟。然而, 仅通过对源代码的结构、执行逻辑等进行建模很难适应当前的软件开发环境。由于代码对内置函数和外部库的引用越来越频繁, 各种动态代码的引入也越来越常见; 如何在缺乏源代码支持的情况下, 对内置函数和外部引用库的行为进行建模、如何对程序的动态行为进行分析、如何针对大型项目进行高效的过程间分析以保证程序数据流的完整性是当前研究的难点; 同时, 如何自动化地对 Source 和 Sink 进行识别和推理, 对代码中出现的净化例程进行合理地评估也是具有挑战的。

(3) **机器学习方法的选择:** 机器学习技术在数据挖掘、自然语言处理、计算机视觉等领域有非常显著的成果, 如: Huang Chen 等[76]提出的利用机器学习技术自动挖掘关键黑客的模型、Kaiming He 等[77]提出的深度残差神经网络图像识别框架等。将机器学习技术引入到静态分析中是当前的研究趋势, 但并不是所有的机器学习方法都适用于程序分析。如何选择和构建合理的学习模型来理解和学习源代码表示的信息是基于学习的静态分析的关键问题。

(4) **漏洞分析方法的普适性:** 当前的研究大多针对用某一特定类型语言(如: C 和 Java)编写的软件程序展开。虽然已有的检测方法在特定的应用场景下表现不错, 但在对编程语言的多样性和迭代更新快等特性的适应性问题处理上仍然存在不足。因此, 改善漏洞分析方法的普适性, 提高分析模型的适配能力是当前静态分析中的关键问题。

(5) **数据集的匮乏:** 虽然网络上有非常多与软件漏洞相关的信息, 但是不同于其他领域, 能够用于对漏洞分析模型进行训练和评估的数据集很少; 而且正如李韵等在其文章[78]中提到的, 存在部分罕见类型的漏洞, 这类漏洞出现的频率非常低, 但其危害性不容忽视; 然而由于缺乏合适的数据集, 导致现有静态分析面临罕见漏洞挖掘困难的问题。因此, 如何构建统一规范的数据集, 如何在数据稀缺的情况下对漏洞进行有效挖掘是一大难题。

此外, 依据目标场景合理地选择模型在漏报率和误报率上的取向也是关键。无论是静态分析还是动态分析, 我们都希望分析模型能够检测出目标代码中存在的所有漏洞, 而且没有误报; 也就是说分析模型既没有漏报, 也没有误报。然而, 根据 Rice 定理[79]和图灵停机问题[80]可知, 这种情况是不存在的。如图 6 所示, 粉色部分表示 Soundness(可以简单理解为没有漏报), 蓝色部分表示 Completeness(可以简单理解为没有误报), 绿色部分表示 Truth(表示目标程序存在漏洞的真实情况)。对于 Soundness 而言, 包含了目标程序存在漏洞的真实情况, 但是也包含了真实情况之外的结果, 即存在误报。对于 Completeness 而言, 所包含的结果都属于真实情况, 但是没有包含全部的真实情况, 即存在漏报。因此, 在设计检测模型时, 不管是基于静态分析还是动态分析, 都需要根据具体场景确定模型是倾向于 Soundness 还是 Completeness, 然后再优化模型使其靠近 Truth。

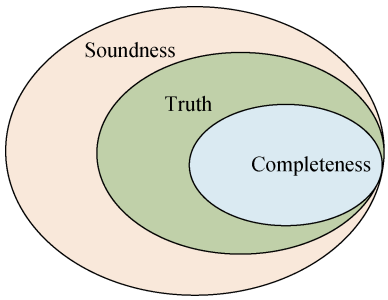

图 6 Soundness 与 Completeness
Figure 6 Soundness and completeness

5 总结

静态分析作为软件分析领域的关键技术, 被广泛运用于对各类软件的健壮性检验和安全检测中。静态分析贯穿整个软件的生命周期, 能够有效地帮助软件开发人员在开发过程中及时发现程序存在的错误和漏洞, 降低软件上线后被攻击的风险。本文介绍了源代码漏洞静态分析技术的核心思想和基本原理, 分析和总结了该领域典型的研究工作, 阐述了我们认为当前静态分析中存在的难点。

近年来, 机器学习技术取得了巨大的突破, 为静态分析注入了新的活力。构建自动化、智能化的静态分析模型势必会成为未来的发展趋势。在这里我们提供了一些研究思路供同方向的研究人员参考。

(1) **使用直观的图形化结构表示源代码**: 使用直观的图形化结构表示源代码不仅便于人类理解源代码, 也能够帮助机器学习模型更好地挖掘源代码中的信息。

(2) **合理的代码嵌入:** 合理地将源代码转换为机器学习模型所需的向量表示, 同时尽可能多地包含源代码的各类信息, 降低代码嵌入时的信息损失, 能够进一步提高机器学习模型对源代码含义的理解。

(3) **将源代码的显式特征和隐式特征相结合**: 使用传统静态分析对源代码的显式特征进行挖掘,

使用机器学习方法对源代码图结构表示中的隐式特征进行挖掘, 将这两种特征相结合, 形成互补, 为源代码漏洞的发现和挖掘提供更加有效地支撑。

(4) **合理、适度地使用深度学习技术**: 从传统机器学习到深度学习, 人工智能领域的发展取得了巨大的进步; 深度学习技术在计算机各个领域上的应用也随之越来越广泛。虽然深度学习技术在解决某些问题上表现出非常不错的性能, 但是其对海量数据和计算机资源的高度需求也是不容忽视的。对于基于学习的静态分析, 由于漏洞数据集稀缺、数据集所包含漏洞样本的覆盖面不够广, 导致训练出来的深度学习模型存在过拟合问题; 而且在面对代码高速迭代更新的大环境时, 模型的性能往往与实验结果相差很大; 因此不能盲目地追求使用深度学习技术。

(5) **从待检测项目中学习**: “对于合理规模的软件项目, 通常包含许多功能相似的代码片段集群, 这些代码片段集群通常由不同的开发人员贡献, 因此它们之间很可能都有相同的 bug, 所以可以通过识别同一代码库中功能相似但不一致的代码片段来检测 bug”, 这是文章[81]中提到的一种 bug 检测思路。文章创新性地提出了一种基于机器学习的 bug 检测技术, 它不需要任何外部代码或样本来进行训练, 仅通过对待检测项目中的函数片段结构进行相似性聚类实现检测。这为我们提供了灵感。通过从待检测项目中学习, 也许能够避免第(4)点中提到的由于数据集稀缺导致模型过拟合的问题, 有助于我们在真实场景下构建高性能的检测模型。

参考文献

[1] The 47th China Statistical Report on Internet Development. CNNIC. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/202102/P020210203334633480104.pdf. 2021.
(第 47 次中国互联网络发展状况统计报告. 中国互联网络信息中心. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/202102/P020210203334633480104.pdf. 2021.)

[2] OWASP Top 10 2017. OWASP. https://owasp.org/www-project-top-ten/2017/. 2017.

[3] 2019 Changting Annual Vulnerability Threat Analysis and 2020 Security Outlook. Changting. https://chaitin-marketing.s3.cn-north-1.amazonaws.com.cn/%E3%80%8A2019%E9%95%BF%E4%BA%AD%E5%B9%B4%E5%BA%A6%E6%BC%8F%E6%B4%9E%E5%A8%81%E8%83%81%E5%88%86%E6%9E%90%E4%B8%8E2020%E5%AE%89%E5%85%A8%E5%B1%95%E6%9C%9B%E3%80%8B.pdf. 2019.
(2019 长亭年度漏洞威胁分析与 2020 安全展望. 长亭. https://chaitin-marketing.s3.cn-north-1.amazonaws.com.cn/%E3%80%8A2019%E9%95%BF%E4%BA%AD%E5%B9%B4%E5%BA%A6%E6%BC%8F%E6%B4%9E%E5%A8%81%E8%83%81%E5%88%86%E6%9E%90%E4%B8%8E2020%E5%AE%89%E5%85%A8%E5%B1%95%E6%9C%9B%E3%80%8B.pdf. 2019.)

[4] 2020 Vulnerability and Threat Trends Report Mid-Year Update: Key Findings. Skybox Security. https://www.skyboxsecurity.com/blog/2020-vulnerability-and-threat-trends-report-mid-year-update-key-findings/. 2020.

[5] Louridas P. Static Code Analysis[J]. *IEEE Software*, 2006, 23(4): 58-61.

[6] Zhioua Z, Short S, Roudier Y. Static Code Analysis for Software Security Verification: Problems and Approaches[C]. *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, 2014: 102-109.

[7] Zou Q C, Zhang T, Wu R P, et al. From Automation to intelligence: Survey of Research on Vulnerability Discovery Techniques[J]. *Journal of Tsinghua University* (*Science and Technology*), 2018, 58(12): 1079-1094.
(邹权臣, 张涛, 吴润浦, 等. 从自动化到智能化: 软件漏洞挖掘技术进展[J]. *清华大学学报(自然科学版)*, 2018, 58(12): 1079-1094.)

[8] T. J. Watson Libraries for Analysis (WALA). Github. https://github.com/wala/WALA. 2021.

[9] Find Bugs in Java Programs. FindBugs. http://findbugs.sourceforge.net/. 2021.

[10] JSPrime. Github. https://github.com/dpnishant/jsprime. 2021.

[11] CodeQL. Github. https://securitylab.github.com/tools/codeql/. 2021.

[12] Fortify Static Code Analyzer. Fortify. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer. 2021.

[13] A tool for static C/C++ code analysis. Cppcheck. http://cppcheck.sourceforge.net. 2021.

[14] Gao Q, Ma S, Shao S H, et al. CoBOT: Static C/C++ Bug Detection in the Presence of Incomplete Code[C]. *The 26th Conference on Program Comprehension*, 2018: 385-3853.

[15] Haugh E, Bishop M. Testing C Programs for Buffer Overflow Vulnerabilities[C]. *NDSS*, 2003: 8.

[16] Ghosh A K, O'Connor T, McGraw G. An Automated Approach for Identifying Potential Vulnerabilities in Software[C]. *Proceedings of 1998 IEEE Symposium on Security and Privacy* (*Cat. No.98CB36186*), 1998: 104-114.

[17] AFL. Github. https://github.com/google/AFL. 2021.

[18] Godefroid P. Random Testing for Security: Blackbox Vs. Whitebox Fuzzing[C]. *The 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007: 1.

[19] jsfunfuzz. Github. https://github.com/MozillaSecurity/funfuzz. 2021.

[20] Li H Z, Kim T, Bat-Erdene M, et al. Software Vulnerability Detection Using Backward Trace Analysis and Symbolic Execution[C]. *2013 International Conference on Availability, Reliability and Security*, 2013: 446-454.

[21] Ye Z B, Yan B. Survey of Symbolic Execution[J]. *Computer Science*, 2018, 45(S1): 28-35.
(叶志斌, 严波. 符号执行研究综述[J]. *计算机科学*, 2018,

45(S1): 28-35.)

[22] Miller B P, Fredriksen L, So B. An Empirical Study of the Reliability of UNIX Utilities[J]. *Communications of the ACM*, 1990, 33(12): 32-44.

[23] Krishnamoorthy S, Hsiao M S, Lingappan L. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs[C]. *2010 19th IEEE Asian Test Symposium*, 2010: 59-64.

[24] Chen Z X, Zou D Q, Li Z, et al. Intelligent Vulnerability Detection System Based on Abstract Syntax Tree[J]. *Journal of Cyber Security*, 2020, 5(4): 1-13.
(陈肇炫, 邹德清, 李珍, 等. 基于抽象语法树的智能化漏洞检测系统[J]. *信息安全学报*, 2020, 5(4): 1-13.)

[25] Feng H T, Fu X T, Sun H Y, et al. Efficient Vulnerability Detection Based on Abstract Syntax Tree and Deep Learning[C]. *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops*, 2020: 722-727.

[26] Yamaguchi F, Lottmann M, Rieck K. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees[C]. *The 28th Annual Computer Security Applications Conference*, 2012: 359-368.

[27] Deerwester S, Dumais S T, Furnas G W, et al. Indexing by Latent Semantic Analysis[J]. *Journal of the American Society for Information Science*, 1990, 41(6): 391-407.

[28] Yamaguchi F, Golde N, Arp D, et al. Modeling and Discovering Vulnerabilities with Code Property Graphs[C]. *2014 IEEE Symposium on Security and Privacy*, 2014: 590-604.

[29] Graph Data Platform. Neo4J. https://neo4j.com/. 2021.

[30] Yamaguchi F, Maier A, Gascon H, et al. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 797-812.

[31] Wu P, Yin L Z, Du X, et al. Graph-Based Vulnerability Detection via Extracting Features from Sliced Code[C]. *2020 IEEE 20th International Conference on Software Quality*, *Reliability and Security Companion*, 2020: 38-45.

[32] Kam J B, Ullman J D. Global Data Flow Analysis and Iterative Algorithms[J]. *Journal of the ACM*, 1976, 23(1): 158-171.

[33] Allen F E, Cocke J. A Program Data Flow Analysis Procedure[J]. *Communications of the ACM*, 1976, 19(3): 137.

[34] Wegman M N, Zadeck F K. Constant Propagation with Conditional Branches[J]. *ACM Transactions on Programming Languages and Systems*, 1991, 13(2): 181-210.

[35] Wilson R P, Lam M S. Efficient Context-Sensitive Pointer Analysis for C Programs[J]. *ACM SIGPLAN Notices*, 1995, 30(6): 1-12.

[36] Tripp O, Pistoia M, Fink S J, et al. TAJ: effective taint analysis of web applications[J]. *ACM SIGPLAN Notices*, 2009, 44(6): 87-97.

[37] Hu Y J, Zhang L L, Zhao K, et al. Android Privacy Leak Detection Method Based on Static Taint Analysis[J]. *Journal of Cyber Security*, 2020, 5(5): 144-151.
(胡英杰, 张琳琳, 赵楷, 等. 基于静态污点分析的 Android 隐私泄露检测方法研究[J]. *信息安全学报*, 2020, 5(5): 144-151.)

[38] Wang L, Li F, Li L, et al. Principle and Practice of Taint Analysis[J]. *Journal of Software*, 2017, 28(4): 860-882.
(王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. *软件学报*, 2017, 28(4): 860-882.)

[39] Dahse J, Holz T. Simulation of Built-in PHP Features for Precise Static Code Analysis[C]. *Proceedings 2014 Network and Distributed System Security Symposium*, 2014, 14: 23-26.

[40] Yan X X, Ma H T, Wang Q X. A Static Backward Taint Data Analysis Method for Detecting Web Application Vulnerabilities[C]. *2017 IEEE 9th International Conference on Communication Software and Networks*, 2017: 1138-1141.

[41] Jovanovic N, Kruegel C, Kirda E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities[C]. *2006 IEEE Symposium on Security and Privacy*, 2006: 6pp.-263.

[42] Chen L, Wang Y, Cai Q, et al. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems[C]. *30th USENIX Security Symposium (USENIX Security 21)*, 2021: 303-319.

[43] Xie Y, Aiken A. Static Detection of Security Vulnerabilities in Scripting Languages[C]. *USENIX Security Symposium*, 2006, 15: 179-192.

[44] Livshits V B, Lam M S. Finding Security Vulnerabilities in Java Applications with Static Analysis[J]. *14th USENIX Security Symposium*, 2005: 271-286.

[45] Zheng Y H, Zhang X Y. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection[C]. *2013 35th International Conference on Software Engineering*, 2013: 652-661.

[46] Li Z, Zou D Q, Xu S H, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[J]. arXiv preprint arXiv:1801.01681, 2018.

[47] Wu Y L, Lu J T, Zhang Y Y, et al. Vulnerability Detection in C/C++ Source Code with Graph Representation Learning[C]. *2021 IEEE 11th Annual Computing and Communication Workshop and Conference*, 2021: 1519-1524.

[48] Zhou Y Q, Liu S Q, Siow J, et al. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks [J]. arXiv preprint arXiv:1909.03496, 2019.

[49] Cao D F, Huang J, Zhang X Y, et al. FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection[C]. *2020 IEEE 19th International Conference on Trust*, *Security and Privacy in Computing and Communications*, 2021: 539-546.

[50] Medeiros I, Neves N, Correia M. DEKANT: A Static Analysis Tool that Learns to Detect Web Application Vulnerabilities[C]. *The 25th International Symposium on Software Testing and Analysis*, 2016: 1-11.

[51] Li X, Wang L, Xin Y, et al. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning[J]. *Applied Sciences*, 2020, 10(5): 1692.

[52] 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*, 2021, 16: 1943-1958.

[53] Cheng X, Wang H Y, Hua J Y, et al. DeepWukong[J]. *ACM Transactions on Software Engineering and Methodology*, 2021, 30(3): 1-33.

[54] The Bug Hunter's Workbench. Joern. http://www.mlsec.org/joern/.

2021.
[55] Gremlin Graph Traversal Machine and Language. Apache TinkerPop. https://tinkerpop.apache.org/gremlin.html. 2021.
[56] JFlex. JFlex. https://jflex.de/. 2021.
[57] LALR Parser Generator for Java. CUP. http://www2.cs.tum.edu/projects/cup/. 2021.
[58] PHP-Parser. Github. https://github.com/nikic/PHP-Parser. 2021.
[59] Js2Py. Github. https://github.com/PiotrDabkowski/Js2Py. 2021.
[60] Ghidra. Github. https://github.com/NationalSecurityAgency/ghidra. 2021.
[61] Redini N, Machiry A, Wang R Y, et al. Karonte: Detecting Insecure Multi-Binary Interactions in Embedded Firmware[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 1544-1561.
[62] angr. angr. https://angr.io/. 2021.
[63] joeq. SOURCEFORGE. http://joeq.sourceforge.net/. 2021.
[64] bddbddb. SOURCEFORGE. http://bddbddb.sourceforge.net/. 2021.
[65] Phc compiler. Github. https://github.com/pbiggar/phc. 2021.
[66] LLVM. LLVM. https://llvm.org/docs/WritingAnLLVMPass.html. 2021.
[67] The Simple Theorem Prover. STP solver. https://stp.github.io/. 2021.
[68] A Solver for String Constraints. HAMPI solver. https://people.csail.mit.edu/akiezun/hampi/. 2021.
[69] Moonen L. Generating Robust Parsers Using Island Grammars[C]. *Proceedings Eighth Working Conference on Reverse Engineering*, 2001: 13-22.
[70] ANother Tool for Language Recognition. ANTLR. https://www.antlr.org/. 2021.
[71] fastcluster. pypi. https://pypi.org/project/fastcluster. 2021.
[72] Checkmarx. Checkmarx. https://www.checkmarx.com/. 2021.
[73] Soot. Github. https://github.com/soot-oss/soot. 2021.
[74] PyCP. Github. https://github.com/dmerejkowsky/pycp. 2021.
[75] SVF. Github. https://github.com/SVF-tools/SVF. 2021.
[76] Huang C, Guo Y Y, Guo W B, et al. HackerRank: Identifying Key Hackers in Underground Forums[J]. *International Journal of Distributed Sensor Networks*, 2021, 17(5): 155014772110151.
[77] He K M, Zhang X Y, Ren S Q, et al. Deep Residual Learning for Image Recognition[C]. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016: 770-778.
[78] 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.
(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.)
[79] Rice's Theorem. Wikipedia. https://en.wikipedia.org/wiki/Rice%27s_theorem. 2021.
[80] Halting Problem. Wikipedia. https://en.wikipedia.org/wiki/ Halting_problem. 2021.
[81] Ahmadi M, Farkhani R M, Williams R, et al. Finding bugs using your own code: detecting functionally-similar yet inconsistent code[C]. *30th USENIX Security Symposium (USENIX Security 21)*, 2021: 2025-2040.

刘嘉勇 于 2008 年在四川大学应用数学专业获得博士学位。现任四川大学教授。研究领域为网络空间安全。研究兴趣包括: 网络信息处理与威胁情报分析、数据挖掘、隐蔽通信构建及分析、虚拟社区及社交机器人自动化分析与检测等。Email: ljy@scu.edu.cn

黄诚 于 2017 年在四川大学信息系统安全专业获得博士学位。现任四川大学副教授。研究领域为网络空间安全。研究兴趣包括: 攻击检测、威胁溯源、数据挖掘、社交网络、机器学习和自然语言处理等。Email: codesec@scu.edu.cn

韩家璇 于 2019 年在贵州大学信息安全专业获得学士学位。现在四川大学网络空间安全专业攻读博士学位。研究领域为网络空间安全。研究兴趣包括: 源码安全、漏洞检测等。Email: jiaxuanhan19@stu.scu.edu.cn