mVulSniffer：一种多类型源代码漏洞检测方法

张学军[1]，张奉鹤[1]，盖继扬[1]，杜晓刚[2]，周文杰[1]，蔡特立[1]，赵博[3]

（1. 兰州交通大学电子与信息工程学院，甘肃 兰州 730070；2. 陕西科技大学电子信息与人工智能学院，陕西 西安 710021；
3. 国家电网甘肃省电力公司，甘肃 兰州 730000）

摘 要：针对现有基于深度学习的源代码漏洞检测方法使用的代码切片不能全面覆盖漏洞类间细微差异特征，且单一深度学习检测模型对跨文件、跨函数代码语句间较长的上下文依赖信息学习能力不足的问题，提出一种多类型源代码漏洞检测方法。首先，基于程序依赖图中的控制依赖和数据依赖信息，抽取包含可区分漏洞类型的细粒度两级代码切片。其次，将两级切片转化为初始表示向量。最后，构建适用于两级代码切片的深度学习漏洞检测融合模型，实现对多类型源代码漏洞的准确检测。在多个合成数据集及 2 个真实数据上的实验结果表明，所提方法的检测效果优于现有的多类型源代码漏洞检测方法。

关键词：多类型漏洞检测；深度学习；注意力机制；数据依赖；控制依赖

中图分类号：TP311

文献标志码：A

DOI: 0.11959/j.issn.1000−436x.2023184

mVulSniffer: a multi-type source code vulnerability sniffer method

ZHANG Xuejun[1], ZHANG Fenghe[1], GAI Jiyang[1], DU Xiaogang[2], ZHOU Wenjie[1], CAI Teli[1], ZHAO Bo[3]

1. School of Electronic and Information Engineering, Lanzhou Jiaotong University, Lanzhou 730070, China
2. School of Electronic and Information and Artificial Intelligence, Shaanxi University of Science and Technology, Xi'an 710021, China
3. State Grid Gansu Electric Power Company, Lanzhou 730000, China

Abstract: Given the problem that the code slice used by existing deep learning-based vulnerability sniffer methods could not comprehensively encompass the subtle characteristics between vulnerability classes, and a single deep learning sniffer model had insufficient ability to learn long context-dependent information between cross-file and cross-function code statements, a multi-type source code vulnerability sniffer method was proposed. Firstly, fine-grained two-level slices containing the types of vulnerabilities were extracted based on the control dependency and data dependency information in program dependency graph. Secondly, the two-level slices were transformed into initial feature vector. Finally, a fusion model of deep learning vulnerability sniffer suitable for two-level slices was constructed to achieve accurate vulnerability detection of multi-type source code. The experimental results on multiple synthetic datasets and two real datasets show that the proposed method outperforms the existing multi-type source code vulnerability sniffer methods.

Keywords: multi-type vulnerabilities sniffer, deep learning, attention mechanism, data dependency, control dependency

0 引言

随着信息技术的高速发展，计算机软件已经渗透到人们生活的各个方面，软件规模及其复杂性不断升高，软件漏洞类型和数量也呈现递增趋势。据美国国家漏洞数据库（NVD, national vulnerability

收稿日期：2023−04−07；**修回日期**：2023−06−28

基金项目：国家自然科学基金资助项目（No.61762058）；甘肃省自然科学基金资助项目（No.21JR7RA282）；甘肃省教育厅产业支撑基金资助项目（No.2022CYZC-38）；国家电网科技基金资助项目（No.W32KJ2722010, No.522722220013）

Foundation Items: The National Natural Science Foundation of China (No.61762058), The Natural Science Foundation of Gansu Province (No.21JR7RA282), The Industrial Support Project of Gansu Provincial Department of Education (No.2022CYZC-38), The State Grid Science and Technology Project (No.W32KJ2722010, No.522722220013)

database）的数据显示：软件漏洞数量已连续 5 年超过 1 万条，除了数量增加，软件漏洞也呈现复杂性和多样性，给计算机系统安全带来了极大的威胁，甚至会造成严重的后果[1]。

软件漏洞是指在软件系统或产品的软件生命周期中，由于操作实体有意或无意的疏忽而产生的设计错误、编码缺陷、运行故障等，它们以不同的形式存在于软件系统的各个层次与环节之中[2-3]。攻击者往往基于这些软件缺陷非法访问目标主机并获取敏感数据。显然，及时对设备、应用及系统软件进行漏洞检测，并修补各类漏洞，对软件系统的安全稳定运行具有重大的意义。

目前，研究者提出了多种漏洞检测方法，其可大致分为动态漏洞检测方法和静态漏洞检测方法[4]。动态漏洞检测方法是在整个程序运行中挖掘程序本身存在的缺陷，如模糊测试[5]和动态符号执行[6]。动态漏洞检测方法虽然在小规模软件测试中取得了一定的成效，但在面对大型复杂的软件系统漏洞时，仍然面临检测效率较低的问题[7]，如符号执行方法虽然能以较少的测试用例覆盖尽可能多的程序路径，但仍存在路径爆炸、约束求解难、内存建模与并行处理复杂等问题[8]。静态漏洞检测方法是在不运行程序的情况下，对程序的二进制代码或源代码的语法、语义、控制流和数据流进行分析，从而检测目标程序是否存在漏洞，主要有基于二进制代码的漏洞检测方法和基于源代码的漏洞检测方法。基于二进制代码的漏洞检测方法先将软件反编译为二进制流，然后通过分析二进制流中是否包含漏洞特征，判断软件是否存在漏洞，但由于二进制流缺失了源代码的语义和语法信息，存在检测误报率较高的问题。

基于源代码的漏洞检测方法能够最大限度地保留源代码中丰富的语义和语法信息，解决了动态漏洞检测无法完全覆盖所有代码的问题，且其不需要代码编译环境，实现效率较高，从而得到了广大研究者的高度重视。基于源代码的漏洞检测方法主要分为基于规则的漏洞检测方法[9]和基于学习的漏洞检测方法[10]。基于规则的漏洞检测方法需要专家手动定义漏洞规则，如开源工具 ITS4（interrogating transactional system for security）、Flawfinder、RATS（rough auditing tool for security）等。基于学习的漏洞检测方法[11-28]利用机器学习、深度学习技术对正常样本和漏洞样本之间的特征差异进行学习来确定漏洞检测边界，但该类方法最初将源代码看作文本序列进行处理[11-13]，忽略了编程语言的结构信息[8]，如控制流和语法结构等，使模型难以学习到程序源代码的重要漏洞特征，影响了模型的检测准确率。Agrawal 等[29]研究发现，在基于学习的软件漏洞检测方法中，数据预处理比模型选择更重要。为了学习到更丰富的代码特征，文献[14-22]在基于学习的漏洞检测方法中使用语义和语法图进行数据预处理，有效提高了模型的漏洞检测准确率。

但是，这些漏洞检测方法的检测粒度为整个程序或函数级，可能会带来较大的噪声和冗余，难以有效捕捉代码缺陷特征。为了实现细粒度的漏洞检测，Li 等[24-25]引入代码切片的概念，最先提出了基于深度学习的源代码漏洞检测方法 VulDeePecker 和软件漏洞检测框架 SySeVR，获得了良好的检测效果，然而它们仅能检测出一段代码（如多行代码）中是否存在漏洞，无法精准指出漏洞的类型[22]。漏洞类型会体现漏洞发生的原理，有助于开发人员和代码审核员快速确定漏洞的准确位置、减少工作量、提高工作效率。鉴于大规模预训练语言模型在程序语言和自然语言[30]中表现出的优异性能以及自然语言和高级编程语言之间的紧密联系，Chandra 等[23]提出了基于 Transformer 大规模预训练模型的多类型软件漏洞检测方法，取得了良好的漏洞检测性能，但其需要大量的算力，限制了其应用。Zou 等[28]提出了一种基于深度学习的多分类漏洞检测方法 μVulDeePecker，它在 VulDeePecker 数据集上进行扩充，增加了控制依赖作为切片依据，并将漏洞类型加入其中，并通过定义 code attention 和构建新的双向长短时记忆（BLSTM, bidirectional long short-term memory）网络漏洞检测模型，在不需要大量算力的情况下，不仅能够检测出一段代码中是否存在漏洞，而且能够准确检测出漏洞的具体类型。然而，μVulDeePecker 数据集中的非漏洞样本均被标记为非缺陷类型，不存在类型信息，可能会影响模型的判断[31]。而且，μVulDeePecker 在切片时进行前后向切片，同时考虑数据依赖和控制依赖，提取可区分漏洞类型的代码切片难度较大；BLSTM 漏洞检测模型面对跨文件、跨函数代码语句间较长的上下文依赖信息时学习能力不足。

综上所述，当前多类型漏洞检测方法依然面临以下挑战：1) 虽然通过代码切片能够进行细粒度的

漏洞类型检测，但是如何进行有效切片以覆盖全面的漏洞特征仍然是一个难题；2) 基于单一深度学习的漏洞检测模型对跨文件、跨函数代码语句间较长的上下文依赖信息的学习能力不足，影响检测效果。针对以上挑战，本文提出了一种多类型源代码漏洞检测方法 mVulSniffer，通过两级代码切片提取更易于区分漏洞类型信息的特征，并设计了适用于两级代码切片的多种深度学习漏洞检测融合模型加强对较长上下文依赖信息的学习能力，有效提高了多类型源代码漏洞的检测能力。本文主要贡献如下。

1) 提出基于两级代码切片的多类型源代码漏洞检测方法。首先，依据 4 种漏洞语法特征对程序依赖图进行切片得到确定是否包含漏洞的初级漏洞代码（PrVC, primary vulnerability code）块；然后，依据 3 种漏洞语法规则对初级漏洞代码块进一步切片，得到包含更加易于区分漏洞类型信息的扩展语法漏洞代码（ExSyVC, extended syntax-based vulnerability code）块，从而获得更全面的漏洞特征。

2) 设计了适用于两级代码切片的双向门控循环单元（BGRU, bidirectional gate recurrent unit）神经网络、卷积神经网络（CNN, convolutional neural network）模型和漏洞检测融合模型来分别提取初级漏洞代码块和扩展语法漏洞代码块的语法特征，在融合层基于注意力机制提取关键漏洞特征，解决已有多类型漏洞检测模型难以对跨文件、跨函数代码语句间较长的上下文依赖信息进行学习的问题，实现了更有效的多类型源代码漏洞检测。

3) 在包含 10 类源代码漏洞的数据集，基于库/应用程序接口（API）函数调用（FC, function call）、数组使用（AU, array usage）、指针使用（PU, pointer usage）和算术表达式（AE, arithmetic expression）4 种易引发漏洞语法特征的 4 个数据集和 2 个真实源代码漏洞数据集 Devign 和 REVEAL 上进行了全面的实验对比和验证。结果表明，mVulSniffer 的漏洞检测效果优于现有方法。

1　相关工作

基于源代码的漏洞检测是一种静态漏洞检测方法[7]，它通过对源代码的语法、语义、控制流和数据流进行分析来检测目标程序代码是否存在漏洞。本文将基于源代码的漏洞检测方法分为基于规则的源代码漏洞检测方法和基于学习的源代码漏洞检测方法。

1.1　基于规则的源代码漏洞检测方法

基于规则的源代码漏洞检测方法发展历史悠久，SteveJohnson 开发了 Lint，通过代码语法规则对 C 语言代码中存在的错误进行挖掘。目前常见的漏洞检测工具主要通过词法分析进行漏洞检测，如 ITS4、Flawfinder、RATS 等。其中，ITS4 通过简单词法分析进行漏洞检测；Flawfinder 和 RATS 对每种漏洞维护内建的特征库，然后通过词法分析算法对其中条目进行匹配，从而挖掘代码中的漏洞，可以有效地挖掘由 API 误用等问题导致的漏洞。

1.2　基于学习的源代码漏洞检测方法

目前，机器学习和深度学习技术在漏洞检测任务中也表现了良好的效果。Yamaguchi 等[11]使用主成分分析（PCA, principal components analysis）和词频和逆文本频率（TF-IDF, term frequency-inverse document frequency）技术得到代码的向量表征，并利用机器学习模型进行漏洞检测。Park 等[12]使用机器学习方法学习变量初始值和其允许取值的范围来检测由变量引起的软件漏洞。

为了更准确地挖掘软件漏洞特征，Russell 等[13]提出了基于深度学习的代码表示，以检测源代码中的软件漏洞。但是，以上 2 种方法将源代码序列当作自然语言序列处理，忽略了编程语言本身的特性，例如控制流和语法结构等。Wang 等[14]和 Li 等[15]基于源代码对应的抽象语法树（AST, abstract syntax tree），分别利用深度置信网络（DBN, deep belief network）和 CNN 实现漏洞检测。Dam 等[16]基于 AST 的树结构信息，利用树状长短时记忆（LSTM, long short-term memory）网络对源代码漏洞特征进行学习，进而实现漏洞检测，但上述工作仅考虑了 AST 的语义、语法信息，忽略了其他代码表示包含的上下文语义特征。Kim 等[17]提出从 AST 和控制流图（CFG, control flow graph）分别获取语义语法和控制流的特征，并采用基于注意力的 LSTM 模型实现漏洞检测。Harer 等[18]对比了基于源代码和基于 CFG 的向量表征，并使用机器学习算法进行漏洞检测，证明了机器学习对函数级漏洞检测的有效性。Duan 等[19]将源代码对应的代码属性图（CPG, code property graph）编码为特征张量并输入神经网

络，实现漏洞检测。为了学习源代码中多维度的数据流、控制流特征，Zhou 等[20]和 Cao 等[21]将图神经网络引入漏洞挖掘任务中，获得良好的漏洞检测性能。Fan 等[22]构建了包含 AST、CFG、数据流图（DFG, data flow graph）和代码序列信息的综合代码图，并设计了圆形门控图神经网络进行漏洞检测，获得了更优的检测效果。

为了实现更细粒度的漏洞检测，Li 等[24]引入代码切片概念，并基于 BLSTM 训练漏洞检测模型来检测源代码是否包含漏洞，但仅引入库/API 函数调用语法规则对源代码进行切片。随后，Li 等[25]又提出了一种源代码漏洞检测框架 SySeVR，该框架将源代码基于 4 种漏洞语法规则进行切片，并基于向量化表征和循环神经网络（RNN, recurrent neural network）训练漏洞检测模型，该框架可用于不同编程语言的源代码漏洞检测。杨宏宇等[26]基于包含数据依赖和控制依赖信息的图结构代码切片，构建了图神经网络模型实现漏洞检测，但漏洞检测模型训练时间较长。胡雨涛等[27]提出了改进的图神经网络解释器（GNNExplainer, generating explanation for graph neural network）对基于深度学习的漏洞检测模型输出结果进行解释。

但上述工作仅检测源代码中是否存在漏洞，没有挖掘漏洞的具体类型。Zou 等[28]最先利用深度学习技术提出一种多类型漏洞检测模型 μVulDeePecker，该方法引入 C/C++漏洞源代码构建包含数据依赖信息的 code gadget 和包含控制依赖且能区分漏洞类型信息的 code attention 代码切片，并通过设计 code attention 抽取方法和构建多层 LSTM 漏洞检测模型实现了多类型的源代码漏洞检测。通过考虑大规模预训练语言模型在程序语言和自然语言任务中优异性能[30]以及自然语言和编程语言之间的紧密联系，Chandra 等[23]提出了基于 Transformer 的大规模预训练模型软件漏洞检测方法，如 BERT（bidirectional encoder representation from transformers）和 CodeBERT，获得比上述方法更优的软件漏洞检测性能，但其需要大量的算力，且检测粒度仍然较粗。

本文基于两级代码切片，设计了一种更细粒度的多类型源代码漏洞检测方法 mVulSniffer。与其他使用单一深度学习模型的软件漏洞检测方法相比，mVulSniffer 通过 BGRU 和 CNN 增加了对两级切片中易于区分漏洞类型的特征与上下文依赖信息的学习能力，在不需要大量算力的情况下有效提高了多类型源代码漏洞的综合检测性能。

2 本文方法

2.1 问题描述

mVulSniffer 方法涉及的主要概念如下。

定义 1 源程序[28]。一个程序 P 是一个有序的程序语句集，$P=\{p_1,p_2,p_3,\cdots,p_n\}$，其中 p_i（$1\leqslant i\leqslant n$）是一条程序语句，也是一个代码令牌的有序集合，表示为 $p_i=\{t_{i1},t_{i2},\cdots,t_{im}\}$，其中代码令牌 t_{ij}（$1\leqslant j\leqslant m$）为变量标识符、函数标识符、常量、关键字或运算符等。

定义 2 数据依赖[28]。给定一个程序 $P=\{p_1,p_2,p_3,\cdots,p_n\}$，存在一条程序语句 $p_i\in P$，一个代码令牌 $t_{ij}\in p_i$ 为数据元素，若 t_{ij} 在 p_u（$p_u\in P$）中被使用，则 p_u 数据依赖于 p_i。

定义 3 控制依赖[28]。存在 2 条程序语句 $p_i,p_j\in P$，且 $i\neq j$，若 p_j 的执行受到 p_i 执行结果的影响，则 p_j 控制依赖于 p_i。

定义 4 初级漏洞代码块。给定一组易引发漏洞的语法规则 $H=\{h_k\}$，$1\leqslant k\leqslant 4$，H 是对引发漏洞语法规则的描述[25]，具体如下：1) h_1 表示基于库/API 函数调用；2) h_2 表示指针使用；3) h_3 表示算术表达式；4) h_4 表示数组使用。这 4 种语法规则引发的漏洞覆盖了 SARD 漏洞库中 93.6%以上的漏洞。同时，给定一个程序语句 $p_i=\{t_{i1},t_{i2},\cdots,t_{im}\}$，若 p_i 至少满足一种易发生漏洞语法规则 h_k，则将与 p_i 具有控制依赖与数据依赖关系的前向切片和后向切片合并构成 PrVC，即 $\mathrm{PrVC}_i=\{p_s,\cdots,p_t\}$，其中，$1\leqslant i\leqslant n$，$1\leqslant s<t\leqslant n$。

初级漏洞代码块 PrVC 是由多行代码组成的有序代码集合，基于满足易引发漏洞的语法规则 H 的语句及源代码对应的程序依赖图进行切片所得，主要包含程序中语句之间的数据依赖关系和控制依赖关系信息，是判断一段程序是否包含漏洞的重要依据。

定义 5 扩展语法漏洞代码块。给定一组描述漏洞语法特征 $R=\{r_k\}$，$1\leqslant k\leqslant 3$，R 是对漏洞语法特征的描述[28]，具体如下：1) r_1 表示代码的语法属性表现为库/API 函数中变量/参数的定义语句；2) r_2 表示代码语法属性表现为条件控制语句；3) r_3 表示代码包含函数库/API 函数表达式。同时，给定一个至少满足一种漏洞语法规则 h_k 的语句

p_i ，将 p_i 所对应的初级漏洞代码块 $\mathrm{PrVC}_i=(p_s,\cdots,p_t)$ 依据漏洞语法特征 R 进行切片构成 ExSyVC_i ，即 $\mathrm{ExSyVC}_i=(p_q,\cdots,p_r)$ ，其中，$s\leqslant q<r\leqslant t$ 。

ExSyVC 为漏洞检测模型提供更加准确的语义、语法信息，主要包含某个特定 PrVC 中可引发漏洞的代码语句，此类代码语句集合的语法信息能够反映不同类型漏洞的细微差异。例如，对库/API 调用函数中数据源、函数执行关键路径以及函数是否正确使用的检查可挖掘漏洞的直接原因。因此，扩展语法漏洞代码块包含的语义、语法信息对确定漏洞的类型具有重要的意义。

2.2　mVulSniffer 整体架构

mVulSniffer 主要由代码解析器、向量转化器和漏洞检测器组成，其架构如图 1 所示。其中，代码解析器主要用于对源代码依据特定的语法规则进行切片，向量转化器主要用于将得到的代码切片转化为神经网络可识别的向量表示，漏洞检测器用于训练深度学习检测模型并挖掘源代码中是否包含漏洞及或检测漏洞类型。

2.3　代码切片过程

2.3.1　PrVC 生成过程

PrVC 生成过程包含生成漏洞候选集、对漏洞候选集中语句进行向前切片、向后切片和切片合并 4 个步骤，具体如算法 1 所示。

算法 1　PrVC 生成算法

输入　程序 P，语法规则 $H=\{h_k\}$，$1\leqslant k\leqslant 4$

输出　PrVC_i

1)生成程序 P 对应的程序依赖图 PDG_P，其顶点集合为 N，单个顶点 $n\in N$

2)在 PDG_P 中进行漏洞语法特征匹配，生成漏洞候选集 $P_s=\{p_i,\cdots,p_j\}$

3) for _ slice$_{p_i}\leftarrow\varnothing$

4) back _ slice$_{p_i}\leftarrow\varnothing$

5) $\mathrm{PrVC}_i\leftarrow\varnothing$

6) for each $p_i\in P_s$ do

7)　遍历 PDG_P，得到 p_i 前驱顶点集合 N_f

8)　for each $n_f\in N_f$ do

9)　if n_f 与 p_i 存在数据依赖或控制依赖关系 then

10)　for_slice$_{p_i}\leftarrow$ for_slice$_{p_i}\cup\{n_f\}$

11)　end if

12)　end for

13)　遍历 PDG_P，得到 p_i 的后继顶点集合 N_b

14)　for each $n_b\in N_b$ do

15)　if n_b 与 p_i 存在数据依赖或控制依赖关系 then

16)　back_slice$_{p_i}\leftarrow$ back_slice$_{p_i}\cup\{n_b\}$

17)　end if

18)　end for

19) end for

20) $\mathrm{PrVC}_i\leftarrow$ for_slice$_{p_i}\cup$back_slice$_{p_i}$

21) return PrVC_i

步骤 1)～步骤 2)生成程序 P 的漏洞候选集 P_s。首先，利用代码分析工具 Joern 生成程序 P 对应的程序依赖图 PDG_P；然后，在 PDG_P 中搜索语法属性满足语法规则 H 的代码语句节点，并将搜索到的代码语句 p_i 的集合称为漏洞候选集 $P_s=\{p_i,\cdots,p_j\}$。具体而言，在遍历 P 对应的程序依赖图 PDG_P 的过程中，若节点代表的程序语句语

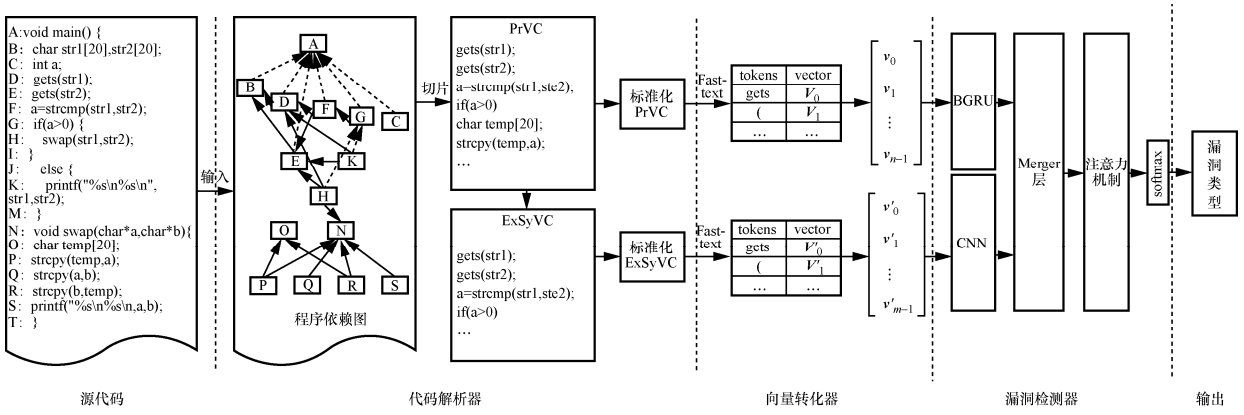


图 1　mVulSniffer 架构

法属性满足漏洞语法规则 H 之一，则将该代码语句并入漏洞候选集 P_s。

步骤 3)～步骤 5)为变量的初始化，将语句 p_i 的前向切片集合 for_slice_{p_i}、后向切片集合 back_slice_{p_i} 和初级漏洞代码块集合 PrVC_i 置空。

步骤 6)～步骤 12)为获得漏洞候选集中语句的前向切片。具体而言，首先，遍历程序依赖图 PDG_P，得到 p_i 前驱顶点集合 N_f；其次，遍历前驱顶点集合 N_f，如果其中的语句 n_f 与 p_i 具有数据依赖或控制依赖关系，则将 n_f 并入 p_i 的前向切片集合 for_slice_{p_i}。

步骤 13)～步骤 19)为获得漏洞候选集中语句的向后切片。首先，遍历程序依赖图 PDG_P，得到 p_i 后继顶点集合 N_b；其次，遍历前驱顶点集合 N_b，如果其中的语句 n_b 与 p_i 具有数据依赖或控制依赖关系，则将 n_b 并入 p_i 的后向切片集合 back_slice_{p_i}。

步骤 20)～步骤 21)为合并前向切片 for_slice_{p_i} 和后向切片 back_slice_{p_i} 形成 PrVC_i，并返回最终的结果。

2.3.2 ExSyVC 生成过程

ExSyVC 的生成过程主要包括分析程序语句语法属性、与漏洞语法规则 R 进行匹配和合并 ExSyVC_i 结果 3 个步骤，具体如算法 2 所示。

算法 2 ExSyVC 生成算法

输入 PrVC_i，语法规则 $R=\{r_k\}$

输出 ExSyVC_i

1) for each $p_i \in \text{PrVC}_i$ do
2) 通过分词及正则表达式解析得出 p_i 的语法属性 c_k
3) for each $r_k \in R$ do
4) if p_i 的语法属性 c_k 与 r_k 匹配 then
5) $\text{ExSyVC}_i \leftarrow \text{ExSyVC}_i \cup \{p_i\}$
6) end if
7) end for
8) end for
9) return ExSyVC_i

首先，输入算法 1 得到的 PrVC_i 以及漏洞语法规则 R；其次，通过分词技术将 PrVC_i 中的每个语句解析为一组有序的 Tokens；接着，使用正则表达式以及上下文语句分析 PrVC_i 中每个代码语句 p_i 的语法属性 c_k，如函数定义、条件语句等，并将 c_k 与 R 中的每种语法规则 r_k 进行匹配，若满足匹配规则，则将对应的代码语句 p_i 纳入扩展语法漏洞代码块集合 ExSyVC_i。具体而言，步骤 1)将 PrVC_i 中每个语句进行遍历；步骤 2)通过词法分析得出 p_i 的语法属性；步骤 3)~步骤 7)通过遍历漏洞语法规则 R 得出该语句 p_i 的语法属性是否满足任一种漏洞语法规则，若满足，则该条语句并入 ExSyVC_i；步骤 9)返回得到的 ExSyVC_i。

2.4 PrVC 和 ExSyVC 的数据清洗及向量化

为了避免源代码中冗余信息及编写代码的主观因素对模型训练产生影响，需要对 PrVC 和 ExSyVC 进行数据清洗。首先，剔除源代码中包含大量与代码执行逻辑和顺序无关的非 ASCII 码字符和代码注释；其次，程序员编码风格各异等因素会导致不同软件源代码中函数名和变量名的定义多样化，可能影响基于源代码的漏洞检测模型的检测准确率，因此利用词法分析技术将切片中的源代码划分为标识符、运算符和关键字等不同标记，对源代码中所有自定义函数名和变量名进行标准化命名，将自定义函数名依次重命名为 FUN1、FUN2 等，变量名依次重命名为 VAR1、VAR2 等；最后，本文利用 FastText 中的 Embedding 词嵌入技术将 PrVC 和 ExSyVC 转化为漏洞检测深度学习可识别的特征向量。

2.5 多类型源代码漏洞检测模型

为了适应两级切片和解决现有多类型漏洞检测模型难以对跨文件、跨函数代码语句间较长的上下文依赖信息进行学习的问题，本文提出一种 BGRU 和 CNN 融合的漏洞检测模型，分为局部模型模块、融合模块和全局模型模块，具体结构如图 2 所示。该融合模型能分别提取初级漏洞代码块 PrVC 和扩展语法漏洞代码块 ExSyVC 的语法特征，并在融合层基于注意力机制提取关键漏洞特征。

由图 2 可知，局部模型模块分为上下两部分。上部分由 BGRU 组成，下部分由 CNN 组成。在 C/C++中，全局变量的定义语句往往距离函数调用较远，所以需要记忆 PrVC 中较长距离的特征，GRU 具有记忆长期依赖信息的特点，相对于 LSTM，其结构简单，参数较少，训练速度更快，同时为了学习 PrVC 中的双向特征，采用双向 GRU 学习 PrVC 代码切片中代码间的长期依赖关系。ExSyVC 作为一种更细粒度的补充训练数据，提取其中的关键漏洞语法特征有助于区分漏洞的特定类型。相对于

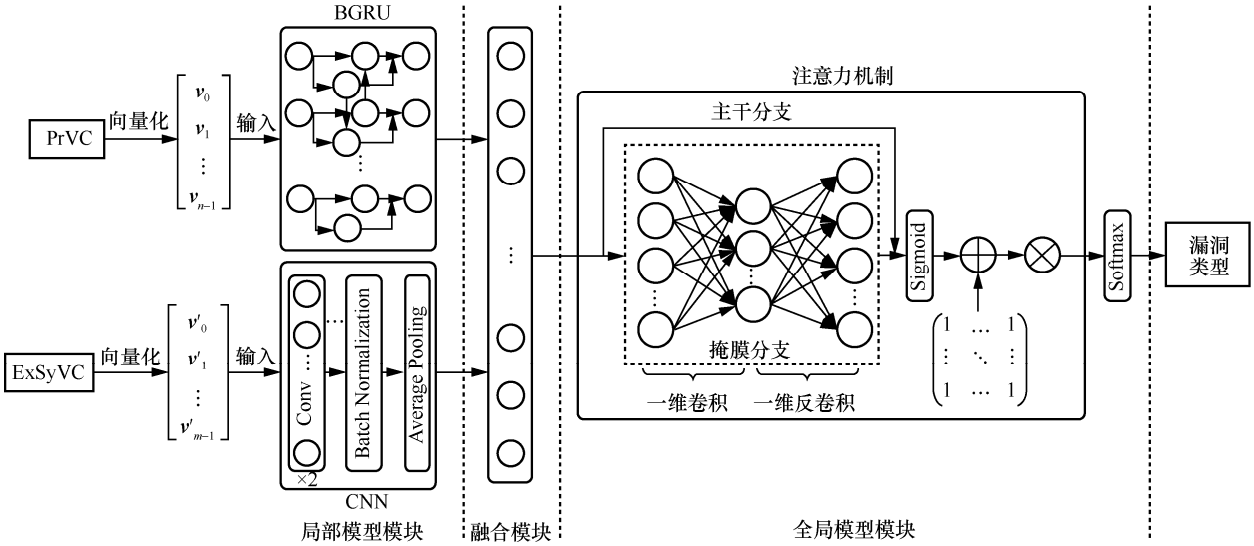


图 2　多类型漏洞检测融合模型

BGRU 模型，CNN 模型提取短文本和细微特征差异的能力更突出，所以本文使用 CNN 对 ExSyVC 进行特征提取。该局部模型由 2 个 Convolution 层、一个 Batch Normalization 层和一个 Average Pooling 组成。融合模块的功能是通过一个 Merger 层将局部模型训练结果中的模型参数进行融合。全局模型模块主要由注意力机制和 Softmax 层组成。Softmax 层为激活层，其功能是输出检测模型的分类结果，如果目标程序有漏洞，那么输出特定漏洞对应的标签，如 1、2 等；否则输出 0，表示不包含漏洞。

3　实验设计与结果分析

3.1　参数配置

实验环境软硬件参数配置如下：操作系统为 Windows10，RAM 大小为 32 GB，GPU 型号为 GeForce RTX 3060。本文所提模型主要基于 TensorFlow 框架实现。

为了验证 mVulSniffer 的综合性能，本文设计并讨论了在多类型源代码漏洞上的检测性能、不同漏洞语法规则引发的漏洞的识别能力和真实软件中漏洞的检测能力，并与最近具有代表性的基于深度学习的源代码漏洞检测方法 Russell[13]、μVulDeePecker[28]、SySeVR [25]、基于 BERT-base[23]和基于 CodeBERT[23]的漏洞挖掘方法进行对比。需要说明的是：本文基于 Zou 等[28]提出的多类型源代码漏洞检测方法 μVulDeePecker，使用 BLSTM 构建源代码漏洞检测模型进行实验；基于 Li 等[25]提出的深度学习漏洞检测框架 SySeVR，使用 BGRU（简称 SySeVR-BGRU）和引入注意力机制的 BGRU（简称 SySeVR-ABGRU）来构建源代码漏洞检测模型进行实验。另外，基于 Chandra 等[23]提出的大规模预训练模型源代码漏洞检测方法，使用 BERT-base（简称 Based BERT-base）和 CodeBERT（简称 Based CodeBERT）来构建源代码漏洞检测模型进行实验。

针对每个实验，本文从实验数据集中选取 80% 的样本作为训练集，20%的样本作为测试集。通过对 PrVC 切片和 ExSyVC 切片中词长进行统计，在将 PrVC 和 ExSyVC 转化为向量时，将其向量维度分别设为 400 和 200。因此，PrVC 和 ExSyVC 在输入神经网络中对应的词向量维度分别设置为 400 和 200，同时添加 Dropout 层防止模型训练过程过拟合，Dropout 值设置为 0.5，优化函数采用 Adamax，损失函数采用 Categorical_Crossentropy，激活函数采用 tanh，学习率设置为 0.001，Batch size 设置为 64，最大 Epoch 设置为 50。其他对比方法及消融实验模型的参数设置与本文方法一致。

3.2　数据集

为了验证模型对多类型源代码漏洞的检测能力，本文从 SARD（software assurance reference dataset）漏洞库中的 C/C++语言的漏洞数据集中选择了 27 605 条数据，以库/API 函数调用作为漏洞语法规则进行切片。其中，包括 21 519 条无漏洞样本数据和 6 086 条包含 10 类特定漏洞的样本数据。每个 CWE-Id 对应漏洞的简单描述以及用于模型训练的样本数量如表 1 所示，本文使用 1～10 对每种包含特定 CWE-Id 漏洞的样本添加标签 Label，便于模型训练。

表 1 10 类漏洞的描述及 CWE 编号

Label	CWE-Id	漏洞类型	数量/个
1	CWE-404	不正确的资源关闭或释放	248
2	CWE-476	空指针解引用	270
3	CWE-119	缓冲区错误	2849
4	CWE-706	消息或数据结构执行不当	167
5	CWE-665	不正确的初始化	289
6	CWE-074	注入	626
7	CWE-704	不正确的类型转换	840
8	CWE-311	敏感数据缺失加密	118
9	CWE-400	不受控制的资源消耗	568
10	CWE-020	输入验证	111

为了验证对不同漏洞语法规则引发的漏洞的识别能力，使用 SySeVR[25]中基于 SARD 库的漏洞样本数据集，引入 API 的 FC、AU、PU 和 AE 这 4 种可引发漏洞的语法规则分别对源代码切片得到 4 个数据集进行实验。此外，为验证 mVulSniffer 对于真实软件中漏洞的检测能力，使用 Zhou 等[20]公开的数据集 Devign 和 Chakraborty 等[32]公开的数据集 REVEAL 进行切片及实验。以上 6 个漏洞数据集样本量如表 2 所示。

表 2 6 个漏洞数据集的样本量

数据集	样本有漏洞/个	样本无漏洞/个	样本总量/个
FC	13 603	50 800	64 403
AE	3 475	18 679	22 154
AU	10 926	31 303	42 229
PU	28 396	263 496	291 892
Devign	11 854	14 124	25 978
REVEAL	2 098	20 050	22 148

3.3 评价指标

实验采用准确率（Acc, accuracy）、加权平均 F1 值（W_F1, weighted-F1）和对单个样本的平均检测时间（DT, detection time）作为验证所提方法对多类型漏洞的检测性能和消融实验的评价指标，并使用各类样本在总体样本中的占比作为加权平均 F1 值的权重。设 N 为类型数，本文设置 N=11，i 为某一样本的标签，X_i 为标签为 i 的样本，当 i=0 时表示该样本无漏洞，当$1 \leqslant i \leqslant N-1$时表示具体漏洞类型。

第 i 类的 Precision 和 Recall 分别为

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i} \tag{1}$$

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i} \tag{2}$$

Acc 和 W_F1 分别为

$$\text{Acc}=\frac{\sum_{i=0}^{N}(\text{TP}_i + \text{TN}_i)}{\sum_{i=0}^{N}(\text{TP}_i + \text{TN}_i + \text{FN}_i + \text{FP}_i)} \tag{3}$$

$$\text{W\_F1} = \frac{1}{\sum_{i=0}^{N} X_i}\sum_{i=0}^{N} X_i \frac{2\text{Precision}_i \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i} \tag{4}$$

实验采用准确率（A, accuracy）、召回率（R, recall）、F1 值（F1-score）作为验证 mVulSniffer 对不同漏洞语法规则引发的漏洞的识别能力和 mVulSniffer 在真实软件漏洞数据集下检测能力的评价指标，计算式分别为

$$A = \frac{\text{TP+TN}}{\text{TP+TN+FN+FP}} \tag{5}$$

$$P = \frac{\text{TP}}{\text{FP+TP}} \tag{6}$$

$$R = \frac{\text{TP}}{\text{TP+FN}} \tag{7}$$

$$\text{F1} = 2\frac{PR}{P+R} \tag{8}$$

其中，TN（true negative）表示无漏洞的样本被检测为无漏洞样本的数量；FN（false negative）表示无漏洞样本被检测为有漏洞样本的数量；TP（true positive）表示有漏洞样本被检测为有漏洞样本的数量；FP（false positive）表示有漏洞样本被检测为无漏洞样本的数量。

3.4 结果与分析

3.4.1 在多类型源代码漏洞上的检测性能

mVulSniffer 及对比方法在 10 类源代码漏洞数据集上 Acc、W_F1 和 DT 的实验结果如表 3 所示。

表 3 不同方法对应的模型性能指标对比

方法	Acc	W_F1	DT/ms
Russell[13]	90.59%	88.36%	**2.08**
μVulDeePecker[28]	95.38%	95.95%	3.57
SySeVR-BGRU[25]	95.94%	95.94%	2.10
SySeVR-ABGRU[25]	96.69%	96.92%	3.62
基于 BERT-base[23]	94.35%	93.60%	5.06
基于 CodeBERT[23]	95.42%	96.33%	5.12
mVulSniffer	**97.41%**	**97.42%**	3.96

从表 3 可以发现，mVulSniffer 的 Acc 和 W_F1 分别达到 97.41%和 97.42%，比 μVulDeePecker 的 Acc 和 W_F1 分别提高了 2.03%和 1.47%，这是因为相比于 μVulDeePecker，mVulSniffer 针对源代码中的上下文依赖信息长的特性，选择 BGRU 和 CNN 学习并保留了源代码中漏洞节点前向和后向切片中重要的依赖信息，并引入注意力机制对可区分是否包含漏洞及漏洞类型的关键特征赋予更大的权重，使训练得到的检测模型具有更优的检测性能。

相比于 Russell 方法、SySeVR-BGRU、SySeVR-ABGRU、基于 BERT-base 的方法和基于 CodeBERT 的方法，mVulSniffer 对应的 Acc 和 W_F1 比对比方法中成绩最好的 SySeVR-ABGRU 分别提高了 0.72%和 0.5%，原因是 mVulsniffer 使用 PrVC 和 ExSyVC 共同作为训练集，并基于 BGRU 和 CNN 训练融合模型，在训练过程中可以学习到更多的语法、语义信息，有助于模型对源代码漏洞进行精确分类。

与基于 BERT-base 的方法相比，基于 CodeBERT 方法的 Acc 达到 95.42%，W_F1 达到 96.33%，优于基于 BERT-base 的方法，其原因是 CodeBERT 预训练模型中包含一定的源代码特征，这些特征有助于对多类型源代码的漏洞检测。

另外，由于 mVulSniffer 方法设计的融合模型由局部和全局模型共同组成，且添加了注意力机制，因此，mVulSniffer 方法的深度学习模型对单个漏洞样本的检测时间为 3.96 ms，高于 Russell、μVulDeePecker 和 SySeVR 这些单模型的方法。使用基于 BERT-base 的方法和基于 CodeBERT 的方法进行编码时需要调用预训练模型，进而增加了单个漏洞样本的检测时间。

3.4.2 *消融实验*

为了证明 ExSyVC 通道和注意力机制对 mVulSniffer 的有效性，本文设计单独使用 PrVC 通道（BGRU）、单独使用 PrVC 通道+注意力机制（BGRU+Att）、使用 PrVC 和 ExSyVC 这 2 个通道（BGRU+CNN）3 组实验与原始 mVulSniffer 在 10 类漏洞数据集下进行实验，使用 Acc、W_F1 和 DT 评价各个方法的综合检测性能，实验结果如表 4 所示。

从表 4 可知，在加入 ExSyVC 训练模型时可以提高多类型源代码漏洞的检测能力。添加注意力机制后，虽然对单个样本的 DT 有所增加，但 Acc 和 W_F1 有所提高。

表 4　不同方法在 10 类源代码漏洞数据集上测试指标

方法	Acc	W_F1	DT/ms
mVulSniffer (BGRU)	83.80%	83.63%	2.08
mVulSniffer (BGRU+Att)	85.57%	86.60%	2.13
mVulSniffer (BGRU+CNN)	95.46%	95.34%	3.88
mVulSniffer (BGRU+CNN+Att)	**97.41%**	**97.42%**	**3.96**

综上所述，mVulSniffer 使用 PrVC 和 ExSyVC 共同作为训练数据为模型提供了丰富的特征，在模型融合后通过添加注意力机制对关键特征分配更高的权重，提高了检测模型对不同类型漏洞的辨识能力。

3.4.3 *在 4 种漏洞语法规则引发的漏洞的检测能力*

mVulSniffer 及对比方法在 API 函数调用、算术表达式、数组使用和指针使用 4 种语法规则对应的 4 个源代码数据集的检测准确率 *A*、召回率 *R* 和 F1 值的实验结果如表 5 所示。从表 5 可以发现，在 4 个数据集上，mVulSniffer 在各项检测指标上均取得了最优的结果；与 μVulDeePecker 相比，mVulSniffer 的检测准确率高 0.48%~1.72%，召回率高 2.31%～9.31%，F1 值高 0.7%~5.19%。尤其在基于 AU 和 PU 的漏洞数据集上，mVulSniffer 表现出了明显的优势。但在基于 AE 漏洞数据集上，mVulSniffer 的模型检测准确率与 SySeVR-ABGRU 相差不大，这可能是因为 mVulSniffer 基于 AE 漏洞语法规则提取的 PrVC 已经包含了足够丰富的可区分源代码漏洞的语义、语法信息，使提取的辅助信息 ExSyVC 没有提供更多重要的特征。从表 5 中也可以看到，相较于其他对比方法，基于 CodeBERT 的方法在 FC、AE 和 AU 数据集中召回率达到最高，分别为 82.15%、86.72%和 85.78%，这说明 CodeBERT 预训练模型中包含的源代码信息对漏洞挖掘任务具有一定的有效性。

总体而言，相比于所有对比方法，本文提出 mVulSniffer 在 4 个数据集上均表现出了最优的综合检测性能。

3.4.4 *在真实源代码漏洞数据集下的检测能力*

为了进一步说明本文方法 mVulSniffer 的优势，在 2 个真实源代码漏洞数据集 Devign 和 REVEAL 上进行了对比实验，检测准确率 *A*、召回率 *R* 和 F1 值的实验结果如表 6 所示。

表 5 在 4 个源代码漏洞数据集上不同方法对应的模型性能指标对比

方法	FC			AE			AU			PU		
	A	*R*	F1	*A*	*R*	F1	*A*	*R*	F1	*A*	*R*	F1
Russell[13]	91.06%	74.52%	83.17%	90.25%	80.74%	85.27%	89.44%	83.45%	81.48%	90.54%	89.77%	88.12%
μVulDeePecker[28]	93.60%	74.77%	83.17%	96.25%	85.32%	87.73%	92.74%	85.45%	85.90%	95.54%	94.59%	94.29%
SySeVR-BGRU[24]	93.20%	81.91%	84.26%	95.62%	80.74%	85.27%	91.28%	85.76%	83.59%	94.40%	91.07%	92.69%
SySeVR-ABGRU[24]	93.39%	77.10%	83.13%	96.61%	85.41%	88.76%	92.56%	79.46%	84.68%	92.95%	89.63%	90.83%
基于 BERT-base[23]	93.04%	79.94%	83.37%	95.14%	85.65%	84.36%	90.72%	85.46%	83.77%	94.90%	90.83%	92.42%
基于 CodeBERT[23]	93.33%	82.15%	83.94%	95.05%	86.72%	86.14%	91.91%	85.78%	83.70%	94.40%	91.43%	92.72%
mVulSniffer	**95.32%**	**84.08%**	**88.36%**	**97.46%**	**92.60%**	**91.96%**	**94.41%**	**91.69%**	**89.46%**	**96.02%**	**96.90%**	**94.99%**

表 6 2 个真实源代码漏洞数据集上不同方法对应的模型性能指标对比

方法	Devign			REVEAL		
	A	*R*	F1	*A*	*R*	F1
Russell[13]	45.39%	48.74%	49.40%	60.90%	10.2%	15.48%
μVulDeePecker[28]	58.39%	24.98%	35.39%	77.91%	34.77%	45.78%
SySeVR-BGRU[24]	58.76%	55.59%	55.31%	75.54%	26.29%	36.58%
SySeVR-ABGRU[24]	56.09%	86.50%	64.25%	76.36%	16.86%	27.67%
基于 BERT-base[23]	56.44%	44.10%	47.99%	75.93%	30.34%	40.57%
基于 CodeBERT[23]	57.01%	49.33%	51.12%	76.33%	32.91%	42.96%
mVulSniffer	**78.80%**	**57.53%**	**71.24%**	**85.76%**	**73.27%**	**80.96%**

从表 6 可知，mVulSniffer 在 Devign 和 REVEAL 数据集上的检测准确率分别达到了 78.80%和 85.76%，均高于其他对比方法。

Russell 方法利用 RNN 构建漏洞检测网络。而 RNN 在处理长距离信息依赖时会出现梯度消失的问题，这在一定程度上影响了其漏洞检测性能。

与仅使用 BLSTM 模型的 μVulDeePecker 相比，mVulSniffer 采用基于 BGRU 和 CNN 的融合模型，有助于提高在真实漏洞数据中的漏洞检测性能；与 SySeVR、基于 BERT-base 和基于 CodeBERT 的漏洞检测方法相比，mVulSniffer 使用 PrVC 和 ExSyVC 共同作为训练数据为模型提供了丰富的漏洞特征，使 mVulSniffer 在真实源代码漏洞数据集上具有更好的性能表现。

此外，相比于基于 BERT-base 的漏洞检测方法，基于 CodeBERT 的漏洞检测方法在生成预训练模型的同时使用程序语言和自然语言进行预训练任务，由于 CodeBERT 预训练模型中包含一定的代码结构信息，基于 CodeBERT 的漏洞检测方法在整体性能上优于基于 BERT-base 的方法。但是，由于 CodeBERT 预训练数据集中缺乏 C 语言代码，故基于 CodeBERT 方法在基于 C 语言的真实数据集上的漏洞检测性能低于本文提出的 mVulSniffer 方法。

4 结束语

针对目前多类型源代码漏洞检测方法难以获得准确区分漏洞类型的代码切片，以及单一漏洞检测模型难以对跨文件、跨函数代码语句间较长的上下文依赖信息进行充分学习造成对多类型源代码漏洞检测不准确的问题，本文提出了一种基于两级代码切片的 BGRU 和 CNN 融合多类型软件漏洞检测方法 mVulSniffer，在代码切片阶段基于 C/C++代码对应的程序依赖图提取更易于区分漏洞类型信息的两级代码切片，基于 BGRU 和 CNN 及注意力机制设计，提高对较长的上下文依赖信息的学习，从而提高多类型源代码漏洞的检测能力。为了验证 mVulSniffer 的有效性，设计了三组对比实验进行纵向比较，并于其他 6 种经典方法进行横向对比。在包含 10 类漏洞的数据集上，mVulSniffer 的检测准确率达到 97.41%，加权平均 F1 值达到 97.42%；在 API 函数调用、算术表达式、数组使用和指针使用这 4 种数据集上的检测准确率分别达到 95.32%、97.46%、94.41%和 96.02%，在 Devign 和 REVEAL 数据集上

的检测准确率分别达到 78.80%和 85.76%；消融实验证明了 ExSyVC 通道和注意力机制对多类型源代码漏洞检测的有效性。

然而，mVulSniffer 仍存在一定的局限，具体如下：1) 使用的数据样本均来源于 C/C++语言的源代码，暂未考虑对跨语言源代码的漏洞检测；2) 无法定位漏洞的具体位置，不便于漏洞的高效修补。未来的研究将对以上 2 个问题展开研究，设计一种跨语言、多类型的源代码漏洞定位方法。

参考文献：

[1] 刘剑, 苏璞睿, 杨珉, 等. 软件与网络安全研究综述[J]. 软件学报, 2018, 29(1): 42-68.
LIU J, SU P R, YANG M, et al. Software and cyber security-a survey[J]. Journal of Software, 2018, 29(1): 42-68.

[2] 吴世忠. 信息安全漏洞分析回顾与展望[J]. 清华大学学报(自然科学版), 2009, 49(S2): 2065-2072.
WU S. Review and outlook of information security vulnerability analysis[J]. Journal of Tsinghua University (Science and Technology), 2009, 49(S2): 2065-2072.

[3] 吴世忠, 郭涛, 董国伟, 等. 软件漏洞分析技术进展[J]. 清华大学学报(自然科学版), 2012, 52(10): 1309-1319.
WU S Z, GUO T, DONG G W, et al. Software vulnerability analyses: a road map[J]. Journal of Tsinghua University (Science and Technology), 2012, 52(10): 1309-1319.

[4] BROOKS T N. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems[C]//Proceed ings of the Science and Information Conference. Berlin: Springer, 2018: 1083-1102.

[5] BÖHME M, PHAM V T, ROYCHOUDHURY A. Coverage-based greybox fuzzing as Markov chain[J]. IEEE Transactions on Software Engineering, 2017, 45(5): 489-506.

[6] STEPHENS N, GROSEN J, SALLS C, et al. Driller: augmenting fuzzing through selective symbolic execution[C]//Proceedings of the Network and Distributed System Security Symposium. Piscataway: IEEE Press, 2016: 16(2016): 1-16.

[7] 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.
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, 2018, 58(12): 1079-1094.

[8] 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.

[9] 王雅文, 姚欣洪, 宫云战, 等. 一种基于代码静态分析的缓冲区溢出检测算法[J]. 计算机研究与发展, 2012, 49(4): 839-845.
WANG Y W, YAO X H, GONG Y Z, et al. A buffer overflow detection algorithm based on static analysis of code[J]. Journal of Computer Research and Development, 2012, 49(4): 839-845.

[10] 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.

[11] YAMAGUCHI F, LINDNER F, RIECK K. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning[C]//Proceedings of the 5th USENIX Conference on Offensive Technologies. Berkeley: USENIX Association, 2011: 118-127.

[12] PARK J, SHIN J, CHOI B. Detection of vulnerabilities by incorrect use of variable using machine learning[J]. Electronics, 2023, 12(5): 1197-1212.

[13] RUSSELL R, KIM L, HAMILTON L, et al. Automated vulnerability detection in source code using deep representation learning[C]//Proceedings of the 17th IEEE International Conference on Machine Learning and Applications. Piscataway: IEEE Press, 2018: 757-762.

[14] WANG S, LIU T Y, TAN L. Automatically learning semantic features for defect prediction[C]//Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering. Piscataway: IEEE Press, 2016: 297-308.

[15] LI J, HE P J, ZHU J M, et al. Software defect prediction via convolutional neural network[C]//Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security. Piscataway: IEEE Press, 2017: 318-328.

[16] DAM H K, PHAM T, NG S W, et al. A deep tree-based model for software defect prediction[J]. arXiv Preprint, arXiv: 1802.00921, 2018.

[17] KIM J, HUBCZENKO D, MONTAGUE P. Towards attention based vulnerability discovery using source code representation[C]//Proceedings of the International Conference on Artificial Neural Networks. Berlin: Springer, 2019: 731-746.

[18] HARER J A, KIM L Y, RUSSELL R L, et al. Automated software vulnerability detection with machine learning[J]. arXiv Preprint, arXiv: 1803.04497, 2018.

[19] DUAN X, WU J, JI S, et al. VulSniper: focus your attention to shoot fine-grained vulnerabilities[C]//Proceedings of the 28th International Joint Conference on Artificial Intelligence. Menlo Park: AAAI Press, 2019: 4665-4671.

[20] ZHOU Y, LIU S, SIOW J, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in Neural Information Processing Systems, 2019, 32: 10197-10207.

[21] CAO S, SUN X, BO L, et al. BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection[J]. Information and Software Technology, 2021, 136: 106576-106587.

[22] 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: 103247-103259.

[23] CHANDRA T, SEUNG I J, MUHAMMAD E A, et al. Transformer-based language models for software vulnerability detection[C]//Proceedings of the 38th Annual Computer Security Applications Conference. New York: ACM Press, 2022: 481-496.

[24] 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.

[25] 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.

[26] 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.

[27] 胡雨涛, 王溯远, 吴月明, 等. 基于图神经网络的切片级漏洞检测及解释方法[J]. 软件学报, 2023, 34(6): 2543-2561.
HU Y T, WANG S Y, WU Y M, et al. Slice-level vulnerability detection and interpretation method based on graph neural network[J]. Journal of Software, 2023, 34(6): 2543-2561.

[28] ZOU D Q, WANG S J, XU S H, et al. μVulDeePecker: a deep learning-based system for multiclass vulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.

[29] AGRAWAL A, MENZIES T. Is “better data” better than “better data miners”? on the benefits of tuning smote for defect prediction[C]// Proceedings of the 40th International Joint Conference on Software Engineering. New York: ACM Press, 2018: 1050-1061.

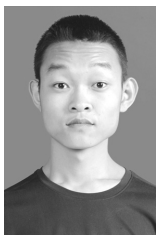
[30] FENG Z Y, GUO D Y, TANG D Y, et al. CodeBERT: a pre-trained model for programming and natural languages[C]//Proceedings of Findings of the Association for Computational Linguistics. Stroudsburg: ACL Press, 2020: 1536-1547.

[31] 邓枭, 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.

[32] CHAKRABORTY S, KRISHNA R, DING Y, et al. Deep learning based vulnerability detection: are we there yet?[J]. IEEE Transactions on Software Engineering, 2022, 48(9): 3280-3296.

[作者简介]

张学军（1977− ），男，宁夏中卫人，博士，兰州交通大学教授，主要研究方向为网络安全、数据隐私与机器学习等。



张奉鹤（1999− ），男，山西大同人，兰州交通大学硕士生，主要研究方向为漏洞挖掘。

盖继扬（1995− ），男，甘肃平凉人，兰州交通大学硕士生，主要研究方向为入侵检测与漏洞挖掘。

杜晓刚（1985− ），男，陕西宝鸡人，博士，陕西科技大学副教授，主要研究方向为机器学习与计算机视觉等。

周文杰（1979− ），女，湖南长沙人，博士，兰州交通大学副教授，主要研究方向为大数据与复杂网络等。

蔡特立（1997− ），男，陕西汉中人，兰州交通大学硕士生，主要研究方向为漏洞挖掘。

赵博（1981− ），男，甘肃平凉人，国家电网甘肃省电力公司高级工程师，主要研究方向为电网安全与软件安全。