面向源代码的软件漏洞静态检测综述

李珍[1,2,3,4]，邹德清[1,2,3,4,5]，王泽丽[1,2,3,4]，金海[1,2,3,4]

（1. 华中科技大学计算机科学与技术学院，湖北 武汉 430074；
2. 华中科技大学服务计算技术与系统教育部重点实验室，湖北 武汉 430074；
3. 华中科技大学集群与网格计算湖北省重点实验室，湖北 武汉 430074；
4. 华中科技大学湖北省大数据安全工程研究中心，湖北 武汉 430074；
5. 深圳华中科技大学研究院，广东 深圳 518057）

摘 要：软件静态漏洞检测依据分析对象主要分为二进制漏洞检测和源代码漏洞检测。由于源代码含有更为丰富的语义信息而备受代码审查人员的青睐。针对现有的源代码漏洞检测研究工作，从基于代码相似性的漏洞检测、基于符号执行的漏洞检测、基于规则的漏洞检测以及基于机器学习的漏洞检测 4 个方面进行了总结，并以基于源代码相似性的漏洞检测系统和面向源代码的软件漏洞智能检测系统两个具体方案为例详细介绍了漏洞检测过程。

关键词：软件漏洞；源代码漏洞检测；代码相似性；深度学习

中图分类号：TP393

文献标识码：A

doi: 10.11959/j.issn.2096−109x.2019001

Survey on static software vulnerability detection for source code

LI Zhen[1,2,3,4], ZOU Deqing[1,2,3,4,5], WANG Zeli[1,2,3,4], JIN Hai[1,2,3,4]

1. School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
2. Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China
3. Clusters and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan 430074, China
4. Big Data Security Engineering Research Center, Huazhong University of Science and Technology, Wuhan 430074, China
5. Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China

Abstract: Static software vulnerability detection is mainly divided into two types according to different analysis objects: vulnerability detection for binary code and vulnerability detection for source code. Because the source code

收稿日期：2018−11−18；**修回日期：**2018−12−26

通信作者：邹德清，deqingzou@hust.edu.cn

基金项目：科技部“网络空间安全”重点专项基金资助项目（No.2017YFB0802205）；国家自然科学基金资助项目（No.61672249）；深圳市基础研究（学科布局）基金资助项目（No.JCYJ20170413114215614）

Foundation Items: The Ministry of Science and Technology’s “Network Space Security” Key Special Project (No.2017YFB0802205), The National Natural Science Foundation of China (No.61672249), The Shenzhen Fundamental Research Program (No.JCYJ20170413114215614)

论文引用格式：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.

contains more semantic information, it is more favored by code auditors. The existing vulnerability detection research works for source code are summarized from four aspects: code similarity-based vulnerability detection, symbolic execution-based vulnerability detection, rule-based vulnerability detection, and machine learning-based vulnerability detection. The vulnerability detection system based on source code similarity and the intelligent software vulnerability detection system for source code are taken as two examples to introduce the process of vulnerability detection in detail.
Key words: software vulnerability, vulnerability detection for source code, code similarity, deep learning

1 引言

计算机科学技术的发展在为人类带来便利的同时，也给恶意分子提供了犯罪的工具。尤其是在网络空间安全领域，黑客攻击、数字资产盗窃、用户隐私信息泄露等各种网络安全事件屡屡出现，严重危害了网络的进一步发展和用户对其的信任度。软件系统作为网络空间的核心组件，其本身存在的漏洞是导致这些攻击的根本原因。软件漏洞是指软件在其生命周期（即开发、部署、执行整个过程）中存在的缺陷，而这些缺陷可能会被不法分子利用，绕过系统的访问控制，非法窃取较高的权限从而任意操纵系统，如触发特权命令、访问敏感信息、冒充身份、监听系统运行等。随着现有软件系统愈加复杂庞大，漏洞出现频率不断提升，急需针对漏洞检测领域开展系统的研究工作，以便高效及时地发现软件系统的漏洞，实时修补，提高网络空间的安全等级。

依据分析对象，软件漏洞静态检测可以分为二进制漏洞检测和源代码漏洞检测两类。二进制漏洞检测方法通过直接分析二进制代码检测漏洞，漏洞检测的准确度较高，实用性广泛；但缺乏上层的代码结构信息和类型信息，分析难度较大，因此基于二进制代码的漏洞检测研究工作相对较少。而源代码相对于编译后的二进制代码拥有更丰富的语义信息，因此更利于快捷地找出漏洞，得到了漏洞检测研究人员的广泛关注。

源代码漏洞检测针对软件设计开发阶段，通过提取源代码模型和漏洞规则，基于静态程序分析技术检测源代码中的漏洞，具有代码覆盖率高、漏报低的优点，但对已知漏洞的依赖性较大，误报较高。源代码漏洞检测方法主要包括基于中间表示的漏洞检测和基于逻辑推理的漏洞检测。基于中间表示的漏洞检测方法首先将源代码转换为有利于漏洞检测的中间表示，然后对中间表示进行分析，检查是否匹配预定义的某个漏洞规则，从而判断源程序中是否含有对应漏洞规则相关的漏洞。基于逻辑推理的漏洞检测方法将源代码进行形式化描述，然后利用数学推理、证明等方法验证形式化描述的一些性质，从而判断程序是否含有某种类型的漏洞。基于逻辑推理的漏洞检测方法由于以数学推理为基础，因此分析严格，结果可靠。但对于较大规模的程序，将代码进行形式化表示本身是一件非常困难的事情。基于中间表示的漏洞检测方法没有上述局限性，适用于分析较大规模程序，因此得到了更为广泛的应用。

本文针对基于中间表示的源代码漏洞检测方法开展研究。依据对中间表示的分析技术，漏洞检测方法可以分为 4 类：基于代码相似性的漏洞检测、基于符号执行的漏洞检测、基于规则的漏洞检测以及基于机器学习的漏洞检测。其中，第一类方法主要针对由于代码复制（code clone）导致的相同漏洞进行检测；后三类方法基于漏洞模式，针对各种原因导致的漏洞进行检测。

2 基于代码相似性的漏洞检测

基于代码相似性进行漏洞检测的核心思想是相似的代码很可能含有相同的漏洞，主要包括 3 个

属性：代码表征、代码段级别和比较方法。具体来讲，通过代码表征和代码段级别**抽象地描述代码段**，然后利用比较方法说明如何依据两个代码段表征判断它们的相似性。

1) 代码表征

代码表征的方式有 5 种，分别是**文本、度量、标记、树和图表**。基于文本的表征由于缺少对代码语法和语义信息的表达，很少用于检测或预测漏洞。

基于度量的表征通过从代码段采集不同的度量，用相应的度量值衡量代码相似度，主要用于漏洞检测领域。Chowdhury 等[1]基于复杂性、内聚和耦合 3 个度量自动预测漏洞。Neuhaus 等[2]发现漏洞构件具有相似的导入和函数调用的集合，基于此预测一个新的构件是否含有漏洞。基于度量的表征方法包含程序的语法信息，容易扩展到多类编程语言中，但无法表示比较单元的全部特征，限定条件较多，检测效果的好坏需要进一步验证。

基于标记（token）的表征不需要语法分析，仅通过词法分析将源代码转换为一个标记序列，通过比较这些标记数组的行来发现相似代码。ReDeBug[3]能够迅速发现操作系统规模代码库中未打补丁的漏洞代码。它使用特征散列法编码位向量中的 *n* 个标记，使 ReDeBug 以 cache 高效方式执行相似性检测，并借助 diff 补丁代码段发现漏洞代码。Li 等[4]将每个文件按行标记，即每行作为一个标记，滑动窗口大小为 *n*，每 *n* 个标记作为比较的基本单位。从 diff 补丁代码段中提取得到漏洞代码，**当漏洞代码的 *n* 个标记集合为目标代码的 *n* 个标记集合的子集时，则发现代码复制。**Scandariato 等[5]对应用软件中的构件源码进行文本挖掘，每个构件用源码中一系列术语及出现频率标识，基于这些特征来预测含有漏洞的构件。Yamaguchi 等[6]将每个函数表示为主要 API 使用模式的组合。通过抽取每个函数的类型名和函数名，嵌入向量空间，使用机器学习方法得到 API 使用模式，基于此辅助发现源码中的漏洞。CP-Miner[7]采用频繁子序列挖掘技术，以程序中的基本块为单位识别基本的复制粘贴代码段，之后发现复制粘贴代码段中的缺陷。**基于标记的表征方法在词法级别对代码复制漏洞有良好的检测效果，但未考虑到语法语义信息。**

在基于树的表征中，采用树来表示源代码中变量、常量、函数调用以及其他标记的语法结构。Yamaguchi 等[8]从源码中提取所有函数的抽象语法树，嵌入向量空间中，使用机器学习方法分析函数的结构模式，利用函数的结构模式组合来发现漏洞。SecureSync[9]采用扩展的抽象语法树（xAST）来表征漏洞代码段，用于复制源码的再现漏洞检测。**基于树的表征方法虽然检测效果较好，但复杂度高，难以用在大规模软件系统中**。

在基于图的表征中，图的节点表示表达式或语句，边表示控制流、控制依赖或数据依赖。通过分析源代码的语法结构以及函数调用关系、控制依赖关系、数据流等，构建程序依赖图（PDG）。匹配图中的节点，由这些节点组成的连通图称为相似子图，由此判断代码的相似性。这种语义表征已用于复制检测、缺陷检测和漏洞检测。在漏洞检测方面，Yamaguchi 等[10]采用代码属性图表征函数源码来发现漏洞，代码属性图结合了抽象语法树、控制流图和程序依赖图。CBCD[11]用子图同构匹配来确定缺陷代码的 PDG 是否是软件系统 PDG 的子图，并提供了 4 种 PDG 查询的优化方法。SecureSync[9]采用基于图的 API（xGRUM）来表征漏洞代码段，用于检测针对共享库的再现漏洞。**基于图的表征方法综合考虑程序的语法和语义特征，有很好的检测能力，但建立图结构代价非常高，寻找图相似的匹配算法复杂度高，时空效率低，较难运用于大型软件检测。**

2) 代码段级别

代码段是程序比较的单位，这意味着代码需要以特定的粒度级别进行抽象。**现有的工作包括5个级别的代码段：不带上下文的补丁级、切片级、带上下文的补丁级、函数片段级和文件/构件代码段级。**在不带上下文的补丁级别，通过提取前缀为“-”的连续行（表示删去的行），从 diff 文件中获取代码段。这种粒度已用于错误检测。在切片级别，基于程序依赖图对程序进行切片。由于切片通常保留了程序依赖图的结构，因此代码相似性通过子图间的同构来表示。切片级粒度已用于代码复制检测[12]和漏洞检测[13]。在带上下文的补丁级别，通过提取前缀为“-”的行和没有前缀的行，从 diff 补丁文件中获取片段。这种粒度已用于缺陷检测[7]和漏洞检测[3,4]。在函数片段级别，函数作为独立的单位，已用于漏洞检测[6,8]和代码复制检测[1]。在文件/构件代码段级别，每个文件/构件都被视为一个单元，这种粗粒度级别主要用于漏洞预测[1-2]。

3) 比较方法

代码相似性方法中的比较主要包括两种方法：向量比较和近似/精确匹配。向量比较方法首先将漏洞的表征和目标程序的表征转换为向量，然后通过比较这些向量来检测漏洞。近似/精确匹配方法是通过包含关系[3,4,7]、子串匹配[14]、完整子图同构匹配或近似 γ-同构匹配，在目标程序的代码表征中查找漏洞表征。

基于代码相似性的漏洞检测方法只需要单个漏洞代码实例就可以检测目标程序中的相同漏洞，但它局限于检测类型 I 和类型 II 的代码复制[15]（即相同或几乎相同的代码复制）和部分类型 III 的代码复制[15]（如语句的删除、插入和移动）引发的漏洞。即使使用人工定义的特征来增强基于代码相似性的漏洞检测能力，也很难检测那些不是由代码复制引发的漏洞，因此当用来检测不是由代码复制引发的漏洞时，会导致很高的漏报率。

3 基于符号执行的漏洞检测

基于符号执行的漏洞检测方法使用中间语言，结合符号执行和约束求解来检测漏洞。通过使用符号执行技术，将程序中变量的值表示为符号值和常量组成的计算表达式，而一些程序漏洞可以表现为某些相关变量的取值不满足相应的约束。**通过判断表示变量取值的表达式是否满足相应的约束，来检测程序是否存在相应的漏洞。**约束求解过程一方面判断路径条件是否可满足，根据判断条件对分析的路径进行取舍，另一方面检查程序存在漏洞的条件是否可以满足。符号执行的过程常常需要利用一定的漏洞分析规则，分析规则描述在什么情况下需要引入符号，以及在什么情况下程序可能存在漏洞等信息。通过对漏洞分析初步结果进行进一步的确认处理，得到最终的漏洞分析结果。

Liang 等[16]提出了一个基于 LLVM 中间表征的静态分析工具，结合符号执行和 Z3 SMT 解释器来发现缺陷，能够检测 3 种类型的程序缺陷，即除零错误、指针溢出和死代码。Cassez 等[17]提出了一个静态分析工具，能够分析 LLVM 中间表征，并检查是否存在可以到达 LLVM 中间表征中某个指定错误块的一次运行。Thome 等[18]提出一个搜索驱动的约束求解技术，采用基于蚁群优化元启发式算法的混合约束求解过程，作为任何现有字符串求解器提供的复杂字符串操作的补充。沈维军等[19]基于动静态相结合的程序分析与符号执行技术，通过数值变量符号式提取、静态攻击流程分析以及高精度动态攻击验证来检测和分析软件中可能存在的数值稳定性相关安全漏洞。

基于符号执行的漏洞检测方法能够生成触发漏洞的具体输入，能够利用符号执行工具生成的

输入验证并分析漏洞。**然而由于约束求解器无法求解所有形式的约束，符号执行的分析精度也会受到影响；同时由于开销大，往往无法扩展到大规模程序。**

4　基于规则的漏洞检测

基于规则的漏洞检测方法由专家针对各类漏洞人工分析生成漏洞规则，在词法语法解析基础上，对源代码建模[20]，进行数据流分析、污点分析等。数据流分析是一种用来获取相关数据沿着程序执行路径流动的信息分析技术，分析对象是程序执行路径上的数据流动或可能的取值。数据流分析可以获得程序变量在某个程序点上的性质、状态或取值等关键信息，而一些程序漏洞的特征恰好可以表现为特定程序变量在特定程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可直接应用于检测程序漏洞。**污点分析[21]是一种跟踪并分析污点信息在程序中流动的技术**。在漏洞分析中，使用污点分析技术将所感兴趣的数据标记为污点数据，然后通过跟踪和污点数据相关信息的流向，可以分析这些信息是否会影响某些关键的程序操作，进而挖掘程序漏洞。

常见的软件漏洞检测工具包括开源工具 Flawfinder、RATS、ITS4[22-24]，商业产品 Checkmarx、Fortify、Coverity[25-27]等。开源工具一般采用简单的解析器和漏洞规则，因此误报漏报高；商业产品 Checkmarx 基于源代码进行数据流分析，不需要编译，解析能力明显优于开源工具；商业产品 Fortify 和 Coverity 基于中间语言进行数据流分析，需要对源代码进行编译，检测效果一般优于直接分析源代码的方法，但专家生成的漏洞规则仍然不够完善。

基于规则的漏洞检测方法中漏洞规则的生成依赖人类专家，因此主观性强。漏洞检测能够精确定位到漏洞行，但由于人工定义的漏洞规则很难考虑全各种区分有漏洞和无漏洞的情况，规则的不完善导致漏洞检测具有较高的误报和漏报[28]。

5　基于机器学习的漏洞检测

机器学习技术可以分为 3 种主要方法。**1) 监督学习：**学习系统基于一组标记的训练样本学习所需的模型，其中每个样本由输入数据（通常是向量）和所需的相应输出值（标签）组成。**2) 无监督学习：**在没有标记训练数据的情况下，学习系统的目标是识别给定数据集中的模式和结构。**3) 强化学习：**通过与动态环境的互动来接受奖励和惩罚，训练学习系统达到某个目标。应用于漏洞检测的机器学习技术目前主要涉及前两种，下面按照是否需要人类专家定义特征分为基于传统机器学习方法和基于深度学习方法两类，并分别对其进行介绍。

5.1　基于传统机器学习的方法

传统的机器学习方法通过人工定义特征属性，然后采用机器学习方法，如支持向量机、k 近邻等进行分类。**基于传统机器学习的漏洞检测方法包括两类：针对特定漏洞类型的方法和漏洞类型无关的方法。**

针对特定漏洞类型的方法前提是借助专家知识（如漏洞原理）将漏洞分为不同类型，而某种类型的漏洞，通过机器学习技术学习漏洞模式。Yamaguchi 等[29]针对 C 程序缺少检查漏洞，提出了一个在源代码中自动识别缺少检查的方法 Chucky，静态地对源代码加污点，并识别与安全关键对象关联的异常条件或缺少的条件；针对 C 程序污点类型漏洞，提出了自动推断搜索模式的方法，给定一个安全敏感的 sink，如内存函数，该方法自动识别相应的 source-sink 系统以及构建系统中数据流和净化的模式[30]。此外，还有针对格式化字符串漏洞[31]、信息泄露漏洞等的漏洞检测[32]等。针对特定漏洞类型的方法中，每种方法

仅限于检测一种类型漏洞，而且要求专家定义帮助识别特定类型漏洞的特征。

漏洞类型无关的方法针对各种类型的漏洞，采用机器学习技术，如支持向量机、k 近邻等学习漏洞模式依靠专家手工定义特征来刻画漏洞。Grieco 等[33]采用系统调用作为特征来刻画漏洞，以整个程序为粒度来检测漏洞。Neuhaus 等[34]采用导入和函数调用作为特征来刻画漏洞，以构件为粒度来预测漏洞，采用依赖的名字来预测哪个包中含有漏洞。Shin 等[35]采用复杂度、代码变化和开发人员活动作为特征来刻画漏洞，以文件为粒度来预测漏洞。Moshtari 等[36]针对跨项目的漏洞预测，对复杂性、耦合度以及新提出的耦合度量指标集进行了评价和比较。Scandariato[37]等将每个构件表征为一系列的源代码中的词以及出现的频率，通过对构件源代码的文本挖掘来预测一个构件是否含有漏洞。

基于传统机器学习的漏洞检测方法依赖于专家手工定义特征属性，采用机器学习模型自动对漏洞代码和无漏洞代码进行分类。但由于输入机器学习模型的代码粒度通常较粗，无法确定漏洞行的确切位置。

5.2 基于深度学习的方法

基于深度学习的方法不需要专家手工定义特征，可以自动生成漏洞模式，在漏洞检测方面的相关研究目前刚刚起步。Lin 等[38]针对跨项目情况，采用深度学习模型在函数级别检测洞。Xu 等[39]采用神经网络方法在函数级别进行基于代码相似性的二进制漏洞检测。Rajpal 等[40]针对模糊测试发现漏洞的过程，采用神经网络从过去模糊探测的输入文件中学习模式来指导未来的模糊探测。Russell 等[41]针对 C/C++开源软件代码，开发了一个大规模函数级漏洞检测系统，针对词法解析后的源代码学习深度特征表征，并利用 3 个不同的静态分析检测工具的结果构建了一个开源软件数据集。Harer 等[42]针对 C/C++程序，采用机器学习方法进行数据驱动的漏洞检测，基于一个静态分析器的结果构建了开源函数数据集，比较了应用到源代码和编译后代码的效果。**然而目前基于深度学习的漏洞检测方法存在很多不足，主要表现在以下 4 个方面：**1) **无法精确定位各种类型漏洞**，目前工作的漏洞检测粒度基本都在函数级，粒度太粗导致无法定位漏洞的具体位置；2) **缺少涵盖各种类型漏洞的大规模标注数据集**；3) **现有工作面向漏洞检测采用的深度学习模型有限**，哪种深度学习模型更适合检测漏洞，或者更适合检测哪种类型的漏洞尚不清楚；4) **现有工作只能检测是否含有漏洞，无法提供更全面的漏洞信息。**

在软件缺陷预测方面，Yang 等[43]采用深度学习技术来预测代码更改中的缺陷。Wang 等[44]采用深度信念网络从源代码中自动学习程序的语义表征，在文件级预测软件缺陷。Phan 等[45]利用表征程序执行流程的控制流图来自动学习缺陷特征，进行缺陷预测。Li 等[46]提出了基于卷积神经网络进行缺陷预测的框架 DP-CNN，基于程序的抽象语法树，利用深度学习来生成有效的特征。Dam 等[47]提出能够自动学习源码特征，采用树结构的长短期记忆网络，来直接匹配抽象语法树表征的源代码，用于缺陷预测。然而这些软件缺陷预测方法无法用于漏洞检测，因为文献[43]提出的缺陷预测方法适用于检测代码更改中的缺陷，无法针对整个目标程序，文献[44-47]的方法在文件级或程序级表征程序进行缺陷预测，若应用于漏洞检测，则粒度太粗导致无法定位漏洞。

在软件缺陷检测方面，通常结合针对缺陷报告的信息检索技术来检测和定位源代码中的缺陷。Huo 等[48]利用词法和程序结构信息，学习来自自然语言和编程语言源代码中的统一特征，提出了一个卷积神经网络 NP-CNN，依据缺陷报告自动定位潜在的缺陷代码。Xiao 等[49]采用卷积神

经网络和级联森林提取语义和结构特征，根据缺陷报告来定位缺陷文件。然而缺陷检测方法不能用于检测漏洞，主要原因是：一方面，缺陷不一定是漏洞；另一方面，上述缺陷检测通常依赖缺陷报告，而在漏洞检测中是无法提供漏洞报告的。

此外，深度学习技术也开始应用到程序分析中与漏洞检测相关性较小的其他研究领域，如异常检测[50]、软件语言建模[51]、代码复制检测[52]、API 学习[53]、二进制函数边界识别[54]、恶意 URL、文件路径检测和注册表键检测[55]、修复程序错误[56]、软件的可追溯性[57]、预测程序的属性[58]、代码作者归属[59]等。

基于深度学习的漏洞检测方法不需要专家手工定义特征，可以自动生成漏洞模式，有望改变软件源代码漏洞检测方法，使面向各种类型漏洞的漏洞模式从依赖专家手工定义向自动生成转变，并且显著提高漏洞检测的有效性。然而目前该方法的相关研究刚刚起步，在漏洞定位、数据集构建、深度学习模型解释等方面有待深入研究。

6　实例 1：基于源代码相似性的漏洞检测

6.1　问题阐述

代码复制的广泛存在使一个软件漏洞可能存在于多个应用程序中，修补主机的某个漏洞并不意味着能够完全排除该漏洞对主机的潜在威胁。因此，当有针对某个漏洞的补丁公布时，应及时检查主机中其他软件是否也存在该漏洞，即在给定漏洞和源代码的前提下，能够自动判断源代码中是否含有该漏洞，如果有，给出具体位置，以便于及时修补。上述问题主要面临两个挑战：一是尚不存在能够用来评测基于代码相似性进行漏洞检测研究的数据集；二是不存在某个代码相似性算法适用于所有漏洞。

6.2　解决方案

针对上述问题与挑战，在构建用于评价的数据集基础上，基于漏洞代码特征，实现基于源代码相似性的漏洞检测系统 VulPecker[60]，降低误报与漏报。系统整体结构图 1 所示，包括两个阶段：学习阶段和检测阶段。学习阶段用来选择对给定漏洞有效的代码相似性算法，选择的算法反过来指导漏洞签名的生成以及检测阶段的复制漏洞检测。

给定一个漏洞及其补丁，该漏洞可以通过描述漏洞补丁的 diff 文件来刻画。漏洞补丁 diff 文件由一个或多个 diff 块组成，对于每个 diff 块，定义以下两个特征集合：基本特征和修补特征，如表 1 所示。基本特征对应表 1 中的类型 1，包括漏洞的唯一标识符 CVE ID、描述漏洞类型的通用弱点枚举标识符 CWE ID、厂商、受影响的产品和漏洞严重度。修补特征对应表 1 中的类型 2~类型 6，描述了从修补前代码段到修补后代码段的代码变化。

表 1　　漏洞 diff 块特征

类型	描述	类型	描述
1	基本特征	3-9	删除函数声明
1-1	CVE ID	3-10	修改运算符
1-2	CWE ID	4	表达式特征
1-3	产品发行商	4-1	修改赋值表达式
1-4	影响的产品	4-2	修改 if 条件
1-5	漏洞严重度	4-3	修改 for 条件
2	非本质性特征	4-4	修改 while 条件
2-1	空格、格式或注释的修改	4-5	修改 do while 条件
3	元素特征	4-6	修改 switch 条件
3-1	修改变量名	5	语句特征
3-2	修改常量	5-1	增加行
3-3	修改变量类型	5-2	删除行
3-4	修改函数名	5-3	移动行
3-5	增加函数参数	6	函数特征
3-6	删除函数参数	6-1	增加整个函数
3-7	修改函数参数	6-2	删除整个函数
3-8	增加函数声明	6-3	函数外的修改

为了构建数据集，首先选择具有一系列发布版本且由 C/C++语言开发的开源软件产品，从美

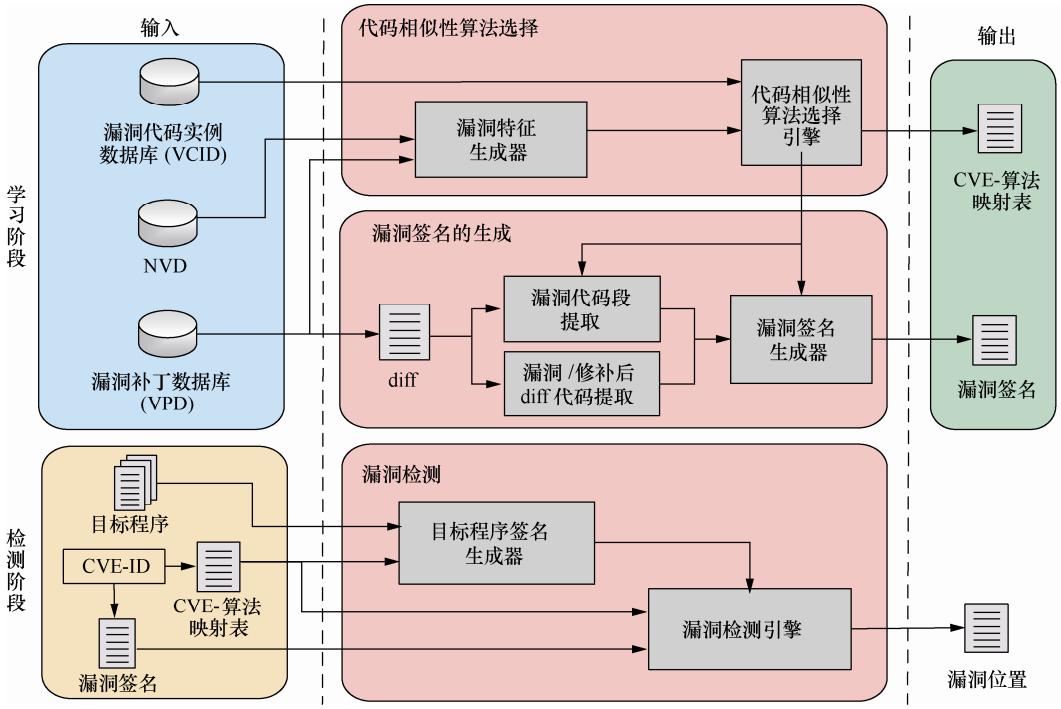


图 1　VulPecker 的结构

国国家漏洞库（NVD）中筛选出 19 个软件产品。基于筛选后的产品，构建了一个包含 1 761 个漏洞（含 3 454 个漏洞补丁 diff 块）的漏洞补丁数据库（VPD）和一个包含 455 个漏洞代码复制实例的漏洞代码实例漏洞库（VCID）。diff 块数量和漏洞代码复制实例数量的差距表明，许多 diff 块没有得到对应的漏洞代码复制实例。

基于上述构建的数据集，直接从 NVD 漏洞库中提取类型 1 的特征；通过对 diff 块进行文本分析和简单的语法分析，分别提取类型 2 和类型 6 的特征；接着针对漏洞代码和修补后代码，采用 Joern[10]工具在函数级分别生成各自的抽象语法树（AST），通过 gumtree 算法[61]对两棵 AST 中节点进行匹配，根据匹配节点集合以及不匹配节点信息，生成从漏洞代码到修补后代码的编辑操作序列，最后根据编辑操作对应的节点信息提取出 diff 块具有类型 3~类型 5 的特征。

代码相似度算法选择引擎是 VulPecker 的核心部分，用于确定哪个代码相似性算法对哪个漏洞有效。通过向算法引擎中输入候选代码相似性算法、漏洞 diff 块特征向量、准确率阈值和 VCID 数据库，引擎会自动输出 CVE-算法映射表。整个过程主要包括以下 3 个步骤，如图 2 所示。首先，选择能够区分漏洞代码和修补后代码的代码相似性算法；然后，识别具有最合适代码段级别的代码相似性算法；最后，选择对于 VCID 具有最低漏报率的代码相似性算法。

基于上述步骤选择合适的代码相似性算法后，需要生成漏洞签名。漏洞签名的生成包括两个步骤。首先，通过提取前缀为“-”的行和没有前缀的行来获取漏洞 diff 代码，通过提取前缀为“+”的行和没有前缀的行来获取修补后 diff 代码。根据每个 diff 块以及为该 diff 块选择的代码相似性算法所使用的代码段级别，从漏洞软件的源代码中提取出漏洞代码段。然后，对于每个 diff 块，对上一步获得的漏洞/修补后 diff 代码和漏洞代码段进行预处理并表示。因为 diff 块中给出的代码语句可能不完整，因此对于基于树或图的代码相

似性算法，需要从漏洞代码段中提取缺失的部分。根据为 diff 块选择的代码相似性算法所使用的代码表征，来表示预处理后的漏洞/修补后 diff 代码和漏洞代码段，作为漏洞签名用于漏洞检测。

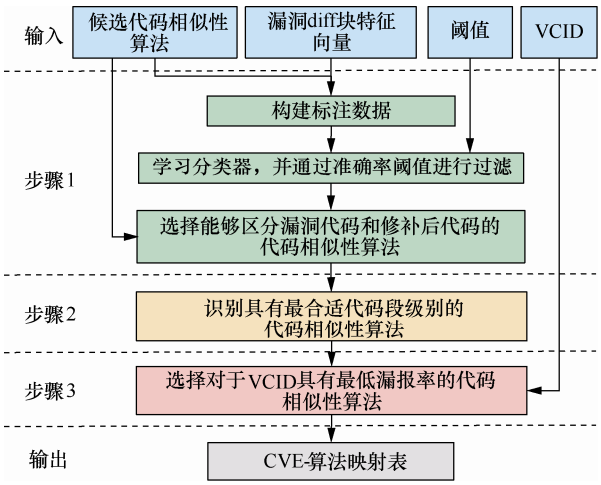


图 2　代码相似性算法的选择过程

最后，基于已有的 CVE-算法映射表进行漏洞检测。CVE-算法映射表能够提供关于指定 diff 块的代码相似性算法及其代码片段级别、代码表征和比较方法的信息。给定 diff 块和目标程序，在对目标程序经过有关空格、格式和注释等预处理后，通过为 diff 块选择的代码相似性算法使用的代码表征生成目标程序签名。漏洞检测引擎采用为 diff 块选择的代码相似性算法使用的比较方法从目标程序签名中搜索漏洞签名。如果找到漏洞签名，则报告目标程序中漏洞的位置。

6.3　效果评测

针对 3 个开源软件产品（即 Firefox、FFmpeg 和 Qemu）在 2013 到 2015 年公布的 246 个漏洞，使用上述漏洞的漏洞签名和 CVE-算法映射表判断目标产品中是否含有上述漏洞中的一个或多个。VulPecker 检测出了 40 个在 NVD 漏洞库中没有公布的漏洞。在这些漏洞中，有 18 个未知漏洞，已报告厂商。对于剩余的 22 个漏洞进行了手动检查和确认，在相关软件的后续发布版本中默默地进行了修补，从漏洞发布到修补的首个版本发布的平均时间为 7.3 个月。

7　实例 2：面向源代码的软件漏洞智能检测

7.1　问题阐述

现有的漏洞静态分析方法存在两个问题。第一，依赖人类专家定义漏洞特征。由于漏洞特征复杂，即使对专家而言也是一个冗长乏味、主观性强、易出错的工作。不同专家定义的漏洞特征可能不同，漏洞特征的质量决定了漏洞检测系统的有效性。第二，现有的漏洞检测方法漏报较高。一个具有高误报的漏洞检测系统是不可用的，而具有高漏报的漏洞检测系统是无用的。理想的漏洞检测系统是同时满足低误报和低漏报的，但通常二者很难同时满足，更好的处理方法是强调低漏报，只要误报在可接受的范围内。

借鉴计算机视觉领域中的目标检测过程，将深度学习用于漏洞检测领域，主要存在以下三方面挑战：一是目标检测能够很自然地利用图像中的纹理、边缘和颜色等信息定义候选区域，漏洞检测则没有明显的细粒度代码结构来描述漏洞的候选区域；二是目标检测拥有海量的人工标注类别的图像数据集，但目前没有标注好的涵盖各种类型漏洞的大规模数据集，且人工标注漏洞的难度远比标注图像大得多；三是目标检测采用适合图像处理的卷积神经网络（CNN）模型来学习特征，然而程序源代码与图像不同，更关注语句上下文信息，且漏洞源代码数据具有自身的特点。

7.2　解决方案

为了解决上述问题，本文开展了基于深度学习的漏洞检测研究。该方法具有很大潜力，因为深度学习不需要人类专家定义特征，但同时也具有挑战，因为深度学习不是为漏洞检测这种应用而产生的。本文主要探讨了将深度学习用于漏洞检测的指导原则，包括将深度学习用于漏洞检测

的程序表征、代码粒度以及神经网络的选择。具体来说，采用代码段（code gadget）来表征程序，其中代码段是语义相关的多行代码（可以不连续），通过编码为向量作为深度学习模型的输入。针对缓冲区漏洞和资源管理异常漏洞，构建了含 61 638 个代码段的训练集，以代码段为粒度检测漏洞，提出了基于深度学习的漏洞检测系统 VulDeePecker[62]，漏洞检测过程如图 3 所示。

基于双向长短期记忆网络模型（BLSTM）自动学习生成漏洞模式，在不需要人类专家定义特征的前提下，自动检测目标程序是否含有漏洞，并给出漏洞代码的位置。该模型包括两个阶段：学习阶段和检测阶段。学习阶段针对训练程序，包括以下 4 个步骤。

步骤 1 生成代码段。首先提取库/API 函数调用，然后针对库/API 函数调用的每个参数提取一个或多个程序切片，最后将针对同一个库/API 函数调用的多个切片组合成为一个代码段。

步骤 2 为代码段加标签。根据已知的漏洞信息及漏洞位置，给生成的代码段加漏洞标签“1”或无漏洞标签“0”。

步骤 3 将代码段转换为向量。首先将代码段转换为符号表征来容纳更多的语义信息，然后将符号表征转换为向量，作为 BLSTM 神经网络的输入。

步骤 4 训练 BLSTM 神经网络。将转换为向量的代码段及其标签输入标准的 BLSTM 神经网络进行训练。

检测阶段针对目标程序，包括步骤 5~步骤 7，其中步骤 5 与步骤 1 类似，步骤 6 与步骤 3 类似。在步骤 7 中，对代码段进行分类。采用学习阶段训练好的 BLSTM 神经网络，对目标程序的代码段进行分类，若分类为 1，则为有漏洞，否则为无漏洞。

7.3 效果评测

本文针对能否同时处理多类漏洞、人类经验能否改进有效性、与其他静态检测方法的有效性比较这 3 个方面对 VulDeePecker 的有效性进行了评价。实验结果表明，VulDeePecker 可以应用到多类漏洞，其有效性与安全相关库/API 函数的个数有关；人类经验可用于选择和安全有关的库/API 函数，能够改进 VulDeePecker 的有效性；VulDeePecker 比人工定义规则的静态分析工具（开源工具和商业工具）更有效，比基于代码相似性的漏洞检测方法具有更低的漏报，其有效性受数据量的影响。此外，VulDeePecker 在 Xen、Seamonkey 和 Libav 这 3 个开源软件产品中检测到 4 个在 NVD 漏洞库中未公布的漏洞，这些漏洞在相应软件的后续版本中默默地进行了修补。而这些漏洞几乎未能被其他漏洞检测系统检测到。更准确地说，一个漏洞检测系统检测出了 4 个漏洞中的 1 个，漏掉了 3 个，而其他漏洞检测系

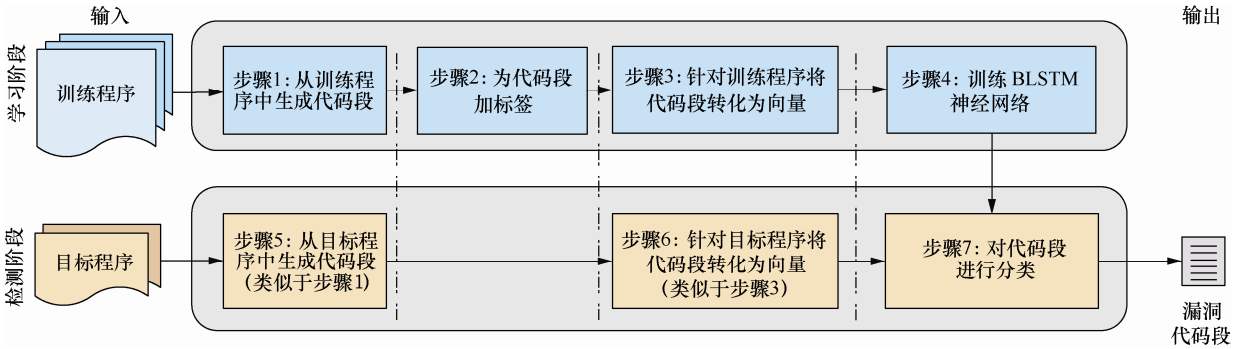


图 3 VulDeePecker 的漏洞检测过程

统漏掉了全部。

8　局限性及未来展望

本节对第 6 节和第 7 节实例解决方案的局限性进行阐述，并展望了未来的漏洞检测研究工作。

基于源代码相似性的漏洞检测系统 VulPecker 的局限性主要表现在以下方面：首先，目前的实验集中于 C/C++ 开源软件，虽然 VulPecker 本身对语言没有限制，但需要针对其他语言程序（如 Java 或者 python）的效果进行实验研究，也有待于扩展到中间语言等级别；其次，VPD 和 VCID 数据库的构建尚不完善，如在创建 VPD 的过程中使用了启发式方法。尽管通过取样并进行人工分析验证了启发式能产生相对正确的结果，但有待通过大规模的实验进行进一步的验证。最后，在未来的工作中，需要在性能方面进行改进，使其能够针对大规模软件进行检测，提高可扩展性。

面向源代码的软件漏洞智能检测系统 VulDeePecker 虽然检测效果比传统的漏洞检测方法更好，但仍存在一些局限性。第一，目前只能处理 C/C++程序，未来工作希望能够适用于处理更多其他的编程语言。第二，目前只能处理与库/ API 函数调用相关的漏洞，如何针对其他类型漏洞提取代码段需要进一步研究。第三，虽然代码段可同时基于数据依赖和控制依赖分析，但是目前借助商业工具提取的代码段只涵盖了数据依赖。提高数据依赖分析的利用以及采用控制依赖提高漏洞检测能力是未来的一项重要工作。第四，在标记代码段、转化为符号表征等阶段使用了启发式方法，未来需要对启发式给漏洞检测结果的有效性影响进行评估。第五，采用的深度学习模型局限于 BLSTM 神经网络，对其他可用于漏洞检测的神经网络的有效性需要进一步研究。最后，目前用于实验的数据集仅包含缓冲区漏洞和资源管理异常漏洞，需要利用更多漏洞类型、更大规模的数据集对方法的有效性进行评测。

总体来说，未来将陆续开展以下三方面的研究工作：1) 研究面向训练程序的代码段标注，基于公开漏洞数据库中的大量数据，对训练程序中的代码段进行自动标注，并实现已标注代码段的数据量扩充，有望构建涵盖各种类型漏洞的大规模标注数据集；2) 研究面向训练程序的漏洞模式智能化学习，采用面向漏洞代码数据特性的深度学习模型实现各类型漏洞模式的自动生成，有望给出哪种深度学习模型更适合检测哪种类型的漏洞；3)研究面向目标程序的漏洞检测与模型解释，基于代码段进行多层级漏洞检测，有望提供除是否含有漏洞外更全面的漏洞信息，并且基于漏洞检测结果进行深度学习模型的解释，进一步改进模型的有效性。

9　结束语

基于源代码的软件漏洞静态检测是保障网络空间安全技术的重要研究领域。通过对给定源代码进行分析，检测软件系统中存在的安全缺陷，从而维护整个系统的稳定运行。本文从实现源代码漏洞检测的方法角度出发，以采用的技术类型为分类依据，总结了现有的源代码漏洞检测研究工作，并重点阐述了基于源代码相似性的漏洞检测系统以及基于深度学习的软件漏洞智能检测系统两个方案。在此基础上，分析了源代码漏洞检测研究存在的问题，并对未来的研究工作进行了展望。

参考文献：

[1] CHOWDHURY I, ZULKERNINE M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities[J]. Journal of Systems Architecture, 2011, 57(13):244-313.

[2] NEUHAUS S, ZIMMERMANN T, HOLLER C, et al. Predicting

vulnerable software components[C]//The 14th ACM Conference on Computer and Communications Security (CCS). 2007: 529-540.

[3] JANG J, AGRAWAL A, BRUMLEY D. ReDeBug: finding unpatched code clones in entire OS distributions[C]//2012 IEEE Symposium on Security and Privacy (S&P). 2012: 48-62.

[4] LI H, KWON H, KWON J, et al. A scalable approach for vulnerability discovery based on security patches[C]//International Conference on Applications and Techniques in Information Security (ATIC). 2014: 109-122.

[5] SCANDARIATO R, WALDEN J, HOVSEPYAN A, et al. Predicting vulnerable software components via text mining[J]. IEEE Transactions on Software Engineering, 2014, 40(10): 993-1006.

[6] YAMAGUCHI F, LINDNER F, RIECK K. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning [C]//The 5th USENIX Workshop on Offensive Technologies (WOOT). 2011: 118-127.

[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] YAMAGUCHI F, LOTTMANN M, AND RIECK K. Generalized vulnerability extrapolation using abstract syntax trees[C]//The 28th Annual Computer Security Applications Conference (ACSAC). 2012: 359-368.

[9] PHAM N H, NGUYEN T T, NGUYEN H A, et al. Detection of recurring software vulnerabilities[C]//The IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France. 2010: 447-456.

[10] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//The IEEE Symposium on Security and Privacy (S&P). 2014: 590-604.

[11] LI J, ERNST M D. CBCD: cloned buggy code detector[C]//The 34th International Conference on Software Engineering (ICSE). 2012: 310-320.

[12] KOMONDOOR R, HORWITZ S. Using slicing to identify duplication in source code[C]//The International Static Analysis Symposium. 2001: 40-56.

[13] 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.

[14] KOSCHKE R, FALKE R, FRENZEL P. Clone detection using abstract syntax suffix trees[C]//The 13th Working Conference on Reverse Engineering (WCRE). 2006: 253-262.

[15] RATTAN D, BHATIA R, SINGH M. Software clone detection: a systematic review[J]. Information and Software Technology, 2013, 55(7): 1165-1199.

[16] LIANG H, WANG L, WU D, et al. MLSA: a static bugs analysis tool based on LLVM IR[C]//The 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). 2016: 407-412.

[17] CASSEZ F, SLOANE A M, ROBERTS M, et al. Skink: static analysis of programs in LLVM intermediate representation (competition contribution)[C]//The 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2017: 380-384.

[18] THOME J, SHAR L K, BIANCULLI D. et al. Search-driven string constraint solving for vulnerability detection[C]//The 39th International Conference on Software Engineering (ICSE). 2017: 198-208.

[19] 沈维军, 汤恩义, 陈振宇, 等. 数值稳定性相关漏洞隐患的自动化检测方法[J]. 软件学报, 2018, 29(5):1230-1243.
SHEN W J, TANG E Y, CHEN Z Y, et al. Method for automated detection of suspicious vulnerability related to numerical stability[J]. Journal of Software, 2018, 29(5):1230-1243.

[20] 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 method of buffer overflow detection based on static code analysis[J]. Journal of Computer Research and Development, 2012,49(4):839-845.

[21] 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.
WANG L, LI F, LI L, et al. Principle and practice of taint analysis[J]. Journal of Software, 2017, 28(4): 860-882.

[22] RAHMA M, QUSAY H M. Evaluation of static analysis tools for finding vulnerabilities in Java and C/C++ source code[J]. CoRR abs/1805.09040, 2018.

[23] HOSSAIN S, MOHAMMAD Z. Mitigating program security vulnerabilities: approaches and challenges[J]. ACM Computer Survey, 2012, 44(3): 1-46.

[24] VIEGA J, BLOCH J T, KOHNO Y. et al. ITS4: a static vulnerability scanner for C and C++ code[C]//The 16th Annual Computer Security Applications Conference, New Orleans, Louisiana. 2000: 257-267.

[25] ANIQUA Z B, TAMARA D. IDE plugins for detecting input-validation vulnerabilitie[C]//IEEE Symposium on Security and Privacy Workshops. 2017: 143-146.

[26] JAMES W, MAUREEN D. SAVI: static-analysis vulnerability indicator[J]. IEEE Security & Privacy, 2012,10(3): 32-39.

[27] BILL B. How to find and fix software vulnerabilities with Coverity static analysis[C]//IEEE Cybersecurity Development (SecDev). 2016:153.

[28] YAMAGUCHI F. Pattern-based vulnerability discovery[D]. Dissertation: University of Gottingen, 2015.

[29] YAMAGUCHI F, WRESSNEGGER C, GASCON H. et al. Chucky: exposing missing checks in source code for vulnerability discovery[C]//The 2013 ACM SIGSAC Conference on Computer and Communications Security. 2013: 499-510.

[30] YAMAGUCHI F, MAIER A, GASCON H. et al. Automatic inference of search patterns for taint-style vulnerabilities [C]//The 2015

IEEE Symposium on Security and Privacy (S&P). 2015: 797-812.

[31] SHANKAR U, TALWAR K, FOSTER J S, et al. Detecting format string vulnerabilities with type qualifiers[C]//The 10th USENIX Security Symposium. 2001: 201-220.

[32] BACKES M, KOPF B, RYBALCHENKO A. Automatic discovery and quantification of information leaks[C]//The 30th IEEE Symposium on Security and Privacy (S&P). 2009: 141-153.

[33] GRIECO G, GRINBLAT G L, UZAL L, et al. Toward large-scale vulnerability discovery using machine learning[C]//The 6th ACM Conference on Data and Application Security and Privacy. 2009: 85-96.

[34] NEUHAUS S, ZIMMERMANN T, HOLLER C, et al. Predicting vulnerable software components[C]//The 14th ACM Conference on Computer and Communications Security (CCS). 2007: 529-540.

[35] SHIN Y, MENEELY A, WILLIAMS L, et al. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities[J]. IEEE Transactions on Software Engineering, 2011, 37(6): 772-787.

[36] MOSHTARI S, SAMI A. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction[C]//The 31st Annual ACM Symposium on Applied Computing. 2016: 1415-1421.

[37] SCANDARIATO R, WALDEN J, HOVSEPYAN A, et al. Predicting vulnerable software components via text mining[J]. IEEE Transactions on Software Engineering, 2014, 401(10): 993-1006.

[38] LIN G, ZHANG J, LUO W, et al. POSTER: vulnerability discovery with function representation learning from unlabeled projects [C]//The 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). 2017: 2539-2541.

[39] XU X, LIU C, FENG Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection[C]//The 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). 2017: 363-376.

[40] RAJPAL M, BLUM W, SINGH R. Not all bytes are equal: neural byte sieve for fuzzing[J]. CoRR, vol. abs/1711.04596, 2017.

[41] RUSSELl R L, KIM L Y, HAMILTON L H, et al. Automated vulnerability detection in source code using deep representation learning[J]. CoRR, vol. abs/1807.04320, 2018.

[42] HARER J A, KIM L Y, RUSSELL R L, et al. Automated software vulnerability detection with machine learning[J]. CoRR, vol. abs/1803.04497, 2018.

[43] YANG X, LO D, XIA X, et al. Deep learning for just-in-time defect prediction[C]//The 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS). 2015: 17-26.

[44] WANG S, LIU T, TAN L. Automatically learning semantic features for defect prediction[C]//The 38th International Conference on Software Engineering (ICSE). 2016: 297-308.

[45] PHAN A V, NGUYEN M L, BUI L T. Convolutional neural networks over control flow graphs for software defect prediction [C]//29th IEEE International Conference on Tools with Artificial Intelligence (ICTAI). 2017: 45-52.

[46] LI J, HE P, ZHU J. et al. Software defect prediction via convolutional neural network[C]//The 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). 2017: 318-328.

[47] DAM H K, PHAM T, NG S W, et al. A deep tree-based model for software defect prediction[J]. CoRR abs/1802.00921, 2018.

[48] HUO X, LI M, ZHOU Z H. Learning unified features from natural and programming languages for locating buggy source code[C]// The 25th International Joint Conference on Artificial Intelligence (IJCAI). 2016: 1606-1612.

[49] XIAO Y, KEUNG J, MI Q, et al. Bug localization with semantic and structural features using convolutional neural network and cascade forest[C]//The 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE). 2018: 101-111.

[50] DU M, LI F, ZHENG G, et al. DeepLog: anomaly detection and diagnosis from system logs through deep learning[C]//The 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). 2017: 1285-1298.

[51] WHITE M, VENDOME C, LINARES-VÁSQUEZ M, et al. Toward deep learning software repositories[C]//The 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR). 2015: 334-345.

[52] WHITE M, TUFANO M, VENDOME C, et al. Deep learning code fragments for code clone detection[C]//The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2016: 87-98.

[53] GU X, ZHANG H, ZHANG D, et al. Deep API learning[C]//The 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). 2016: 631-642.

[54] SHIN E C R, SONG D, MOAZZEZI R. Recognizing functions in binaries with neural networks[C]//The 24th USENIX Security Symposium. 2015: 611-626.

[55] SAXE J, BERLIN K. eXpose: a character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys[J]. CoRR, vol. abs/1702.08568, 2017.

[56] RAHUL G, SOHAM P, ADITYA K, et al. DeepFix: fixing common C language errors by deep learning[C]//The 31st AAAI Conference on Artificial Intelligence (AAAI). 2017: 1345-1351.

[57] GUO J, CHENG J H, HUANG J C. Semantically enhanced software traceability using deep learning techniques[C]//The 39th IEEE/ACM International Conference on Software Engineering (ICSE). 2017: 3-14.

[58] ALON U, ZILBERSTEIN U, LEVY O, et al. A general path-based representation for predicting program properties[C]//The 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2018: 404-419.

[59] ALSULAMI B, DAUBER E, RICHARD E. Source code authorship attribution using long short-term memory based networks[C]//

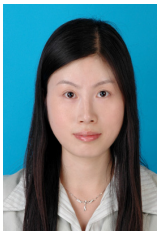
The 22nd European Symposium on Research in Computer Security (ESORICS). 2017: 65-82.

[60] LI Z, ZOU D Q, XU S H, et al. VulPecker: an automated vulnerability detection system based on code similarity analysis[C]//The 32nd Annual Computer Security Applications Conference (ACSAC). 2016: 201-213.

[61] FALLERI J R, MORANDAT F, BLANC X, et al. Fine-grained and accurate source code differencing[C]//The 29th ACM/IEEE International Conference on Automated Software Engineering (ASE). 2014: 313-324.

[62] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection[C]//The 25th Annual Network and Distributed System Security Symposium (NDSS). 2018.

[作者简介]



李珍（1981− ），女，河北保定人，华中科技大学博士生，主要研究方向为软件安全、漏洞检测。

邹德清（1975− ），男，湖南湘潭人，华中科技大学教授、博士生导师，主要研究方向为云计算安全、网络攻防与漏洞检测、软件定义安全与主动防御、大数据安全与人工智能安全、容错计算。

王泽丽（1995− ），女，湖北襄阳人，华中科技大学博士生，主要研究方向为区块链系统安全、智能合约安全。

金海（1966− ），男，上海人，华中科技大学教授、博士生导师，主要研究方向为计算机系统结构、虚拟化技术、集群计算、网格计算、并行与分布式计算、对等计算、普适计算、语义网、存储与安全。