基于抽象语法树的智能化漏洞检测系统

陈肇炫[1,2], 邹德清[1,3,4], 李 珍[1,3], 金 海[1,2]

[1]大数据技术与系统国家工程研究中心 服务计算技术与系统教育部重点实验室 集群与网格计算湖北省重点实验室 大数据安全湖北省工程研究中心, 武汉 中国 430074

[2]华中科技大学 计算机科学与技术学院, 武汉 中国 430074

[3]华中科技大学 网络空间安全学院, 武汉 中国 430074

[4]深圳华中科技大学研究院, 深圳 中国 518000

摘要 源代码漏洞的自动检测是一个重要的研究课题。目前现有的解决方案大多是基于线性模型, 依赖于源代码的文本信息而忽略了语法结构信息, 从而造成了源代码语法和语义信息的丢失, 同时也遗漏了许多漏洞特征。提出了一种基于结构表征的智能化漏洞检测系统 Astor, 致力于使用源代码的结构信息进行智能化漏洞检测, 所考虑的结构信息是抽象语法树(Abstract Syntax Tree, AST)。首先, 构建了一个从源代码转化而来且包含源码语法结构信息的数据集, 提出使用深度优先遍历的机制获取 AST 的语法表征。最后, 使用神经网络模型学习 AST 的语法表征。为了评估 Astor 的性能, 对多个基于结构化数据和基于线性数据的漏洞检测系统进行比较, 实验结果表明 Astor 能有效提升漏洞检测能力, 降低漏报率和误报率。此外, 还进一步总结出结构化模型更适用于长度大, 信息量丰富的数据。

关键词 漏洞检测; 结构表征; 抽象语法树; 神经网络
中图法分类号 TP393 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2020.07.01

Intelligent vulnerability detection system based on abstract syntax tree

CHEN Zhaoxuan[1,2], ZOU Deqing[1,3,4], LI Zhen[1,3], JIN Hai[1,2]

[1]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Clusters and Grid Computing Lab, Big Data Security Engineering Research Center, Wuhan 430074, China

[2]School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

[3]School of Cyber Science and engineering, Huazhong University of Science and Technology, Wuhan 430074, China

[4]Institute of Huazhong University of Science and Technology, Shenzhen 518000, China

Abstract Automatic detection of source code vulnerability is an important research topic. However, most existing solutions are based on linear models. They rely on the text information of source code but ignore the grammatical structure information. This will cause the loss of source code syntax and semantic information, but also miss many vulnerability features. In this paper, an Abstract Syntax Tree (AST) based source code structured representation learning system is proposed to study the structured information of source code and detect the vulnerabilities, called Astor. First, we present a data set that is transformed from the source code and contains information about the syntax structure of the source code. In addition, we propose using a depth first information extraction scheme to obtain the syntax and semantic representation of AST. In Astor, the neural network based detection system is used to learn the representation of AST. In order to evaluate the Astor, we compare vulnerability detection systems based on structured data and linear data. The results show that Astor can achieve much fewer false negative and false positive than other approaches. In addition, this paper further concludes that the structured model is more suitable for data with rich semantic information.

Key words vulnerability detection; structured representation; abstract syntax tree; neural network

1 引言

软件漏洞是一个不容低估的问题, 它与社会、金融、政治密切相关。虽然已经产生了很多漏洞检测工具, 并及时检测出了很多漏洞, 但是软件漏洞的现状仍然不容乐观。2017 年, 美国国家漏洞数据库(National

通讯作者: 邹德清, 教授, Email: deqingzou@hust.edu.cn。

本课题得到国家自然科学基金项目(No.U1936211), 深圳市基础研究(学科布局)(No.JCYJ20170413114215614), 广东省省级科技计划项目(No.2017B010124001), 广东省重点领域研发计划项目(No.2019B010139001)的资助。
收稿日期: 2019-12-06; 修改日期: 2020-04-20; 定稿日期: 2020-06-19

Vulnerability Database, NVD)公布了超过 13400 个漏洞, 是 2016 年的两倍, 而且这个数字还在上升。

当前针对源代码的漏洞检测工具可以简单地分为三类。第一类是基于规则的方法, 这种方法难以满足检测未知漏洞的要求; 第二种是基于代码相似度的方法, 它只能检测代码复用中的漏洞; 第三类是基于机器学习的方法, 根据自动化程度进一步分为两类。(i)基于软件度量的漏洞预测方法: 这些方法需要定义数据的属性, 然后使用机器学习模型根据属性值来识别是否存在漏洞。这种方法存在人工参与度高、误报率高的问题。(ii)基于神经网络的漏洞检测方法: 不需要人工专家定义属性, 通过自动化学习对源代码进行高层次的特征刻画。目前, 对于细粒度样本而言, 基于神经网络的漏洞检测方法的效果是最好的, 可以高效地检测出样本是否包含漏洞, 且极大降低了人力资源的消耗[1]。

将神经网络应用于漏洞检测的大部分工作基本遵循以下三个步骤: (i)对数据集进行一定的预处理操作; (ii)使用词向量模型将数据转换为向量; (iii)利用神经网络模型学习向量表征, 构造漏洞检测器。通过这三个步骤即可实现漏洞检测的功能, 但当前的许多方法在使用这套流程时存在以下问题。

问题 1: 目前漏洞检测领域所公开的数据集都是线性文本这一级别的, 当前的公共数据集都在源代码级(如 NVD 数据集, 其中包含大量的软件源代码), 而没有能够反映代码结构信息的数据集。而由于缺少完整的语法关系, 细粒度的源代码数据(如切片)很难进一步解析得到准确的语法结构信息。因此, 现有的漏洞检测解决方案大多基于源代码进行信息提取和表征。

问题 2: 目前已有的模型大多只适用于学习简单的代码文本信息, 基于文本依赖和单词相似性进行代码表征。这种表征方法不能充分学习代码所承载的语法和语义信息, 会导致较高的漏报和误报率。

问题 3: 虽然目前也有一些结构化模型被提出来学习代码表征, 但由于缺乏能够跨函数的结构化数据集, 这些模型只能在函数级对数据进行漏洞检测。而在真实的应用场景中仅依靠一个封闭的函数很难检测出漏洞的存在。

当前工作中的缺陷使得漏洞检测领域需要产生一些新的方法来自动学习源代码的完整表征并有效地进行漏洞检测。同时, 可以保证低误报率和漏报率。基于此, 本文构建了一个包含源代码结构化信息的漏洞数据集, 并提出了一个基于源代码结构化表征的漏洞检测系统, 以克服现有工作中所存在的无法完全表征代码信息的不足。主要思想是: 利用静态分析工具将源代码解析成抽象语法树(Abstract Syntax Tree, AST)形式, 然后在 AST 的基础上提取结构化信息, 并将其转化为源代码的完整表征。最后, 使用神经模型学习这种结构化表征, 构建漏洞检测器。

本文的贡献包括以下三方面的内容:

1) 构建了一个包含源代码结构信息的漏洞数据集。首先将源代码解析为 AST, 之后基于 AST 构造了一个包含结构化信息的漏洞数据集, 研究人员可以利用该数据集探索不同的结构表征方法, 并将其应用于漏洞检测。所构建的数据集是跨函数的, 且粒度为切片级。通过所构建的数据集, 可以解决前文所提的问题 1 和问题 3。

2) 为了解决问题 2, 构建了一个基于源代码结构化表征的漏洞检测系统, 称为 Astor, 并将检测粒度控制在切片级。本文提出使用 AST 完整表征源代码的结构信息, 在此基础上, 设计了一个基于抽象语法树的结构化表征学习系统, 该系统能够准确地学习源代码的内部结构信息, 并利用得到的有效表征构建漏洞检测器。

3) 总结出适用于结构化表征模型的数据特点。结构表示和线性表示是不同的程序表示方法。事实上, 它们都可以保留程序的语义和语法信息, 但程度不同。在漏洞检测领域, 对于不同类型的数据, 使用结构化表示或线性表示可以得到更好的检测结果, 并没有明确的界限, 也没有相关的工作提出解决方案。在搜集了线性数据和结构化数据后, 根据数据的特性进行大量的实验, 率先明确了使用结构化表示和使用线性表示的界限, 即数据集的平均长度。在真实的漏洞检测场景中, 它可以作为一种技术标准来指导什么样的表示方法可以得到特定数据类型的准确结果。

论文其余部分组织如下: 第二节介绍了相关工作; 第三部分讨论了 Astor 的设计; 第四部分描述了本文针对 Astor 所设计的实验与结果分析; 第五部分是本文的总结。

2 相关工作

相关工作可以分为以下两方面分别总结:

1) 漏洞检测相关工作: 这类方法可以进一步分为三类。第一类是基于规则的方法, 由人类专家或一些自动化算法生成漏洞规则[2], 然后基于模式匹配算法进行漏洞检测[3]。这些方法通常需要对源代码采用词法分析、语法分析、函数调用流分析等技术, 提

取有效信息并生成漏洞模式[4-7]。这类方法具有较高的人员参与性和较弱的可扩展性, 难以满足检测未知漏洞的要求; 第二类是基于代码相似性的方法。这个方法通常有三个步骤[8], 第一步是把一个程序分成一些代码片段[9-10]; 第二步是以抽象的方式表示每个代码片段[11-12], 包括树[10,13]和图[9]; 第三步是通过在第二步中得到的抽象表示来计算代码片段之间的相似性[13]。这类方法方法具有单实例漏洞代码就足以检测目标程序中相同漏洞的优点。但它只能检测代码复用中的漏洞。然而, 本文所提出的漏洞检测系统是通用的, 可以通过扩展数据集来满足各种类型的漏洞; 第三类是基于机器学习的漏洞检测方法。按自动化程度还可以进一步分为两类: (i)基于软件度量的方法。它也需要人类专家的参与, 主要是数据属性的定义, 然后根据属性值采用机器学习的方法识别是否存在漏洞。数据的属性包括有代码复杂度[15]、词频[16]、分支数量[17]、变量定义数量等。通常是利用随机森林、支持向量机等算法, 基于数据属性对模型进行训练, 生成漏洞分类器[17-18]。然而, 这样的方法也存在着大量人员参与的问题。且计算复杂度高, 误报率和漏报率也是居高不下。(ii)基于神经网络的漏洞检测方法。这一类不需要人工专家定义属性, 而是通过自动化学习来构建漏洞检测器[19]。利用现有的模型, 如 LSTM[1]、GRU[20]等神经网络来表示和学习代码漏洞的特征, 对数据进行高层次的表征。然而, 将深度学习应用于漏洞检测的大部分工作是基于线性模型的, 缺乏对程序结构信息的学习, 不能完全表征源代码[21-23]。然而, 本文所提出的系统可以学习结构化的信息, 构建更准确的漏洞检测模型。

2) 基于结构化模型的神经网络相关工作: 随着神经网络技术的不断变化和发展, 研究人员开始考虑使用一些结构化模型来学习自然语言或源代码的结构信息, 希望能够完整地学习源数据的表征信息[24-26]。目前,基于树结构[27]的神经网络和图结构[28]最适用于结构化数据。例如, Tai 等人[29]提出使用 Tree-LSTM 模型预测两个句子的语义联系; Mou 等人[30]则将自然语言任务的方法转移到程序语言领域, 提出了一种称为 TBCNN 的树结构卷积神经网络, 其目的是通过学习程序的抽象语法树表征来捕获代码的结构信息。另一方面, 随着图神经网络的不断兴起和完善, 一些研究人员认为图神经网络还可用于代码处理, Allamanis 等人 [28]提出使用图来表示代码的语法结构和语义信息, 使用基于深度学习的方法学习图表征, 实验结构证明它对 VarNaming 任务(预测变量的名字)是有效的。此外, 还有很多关于实体分类的工作也是使用图神经网络来完成的[31-32]。与以前的线性模型相比, 结构化模型可以学习源代码特有的结构信息, 并且在学习表征的方法上具有更多的优势[22-23]。在漏洞检测领域, Lin 等人[21]提出一种函数表征学习方法来获取 AST 的高层表征和函数结构信息。Zhou 等人[33]设计了一个叫做 Devign 的系统, 它是一种基于图层次分类的通用图神经网络模型, 能够有效地从丰富的图节点中学习到代码的特征。虽然以上两项工作都将结构模型应用于漏洞检测领域, 但都是基于函数级的漏洞检测, 即不能很好地识别跨功能漏洞数据, 检测的粒度也很粗。然而, 本文所提出的系统可以识别跨函数的漏洞数据。另一方面, 基于图的神经网络模型需要使用程序依赖图来学习漏洞特征, 而程序依赖图无法表示代码行内部的语法结构, 因此不适合用于代码的结构化表征。

3 Astor 系统设计

本文的目标是设计一个漏洞检测系统, 致力于源代码的结构化信息表征学习并在低误报率和漏报率的基础上实现漏洞智能检测。本章将介绍 Astor 的设计与实现。首先讨论了结构化数据集的概念, 之后给出了 Astor 的框架概述并详细阐述了它的组成部分。

3.1 结构化数据集的生成

为获取代码的完整表征信息(包括语法结构和语义信息), 首先需要将代码切片转换为 AST。为了更好的解释, 此处给出代码切片的定义:

定义 1. 代码切片(Code Gadget, *CG*)[1], 代码切片由许多程序语句组成, 这些语句彼此之间具有数据依赖或控制依赖方面的语义关联。本文后续部分使用符号 *CG* 表示代码切片。

具体来说, 生成 *CG* 的过程分以下两个步骤:

1) 构建候选漏洞元素。首先为源代码中的每个函数生成对应的抽象语法树, 遍历所有抽象语法树, 搜集能匹配到同一种漏洞语法特征的代码元素, 将所述代码元素称为候选漏洞元素, 确定每一种漏洞语法特征对应的所有候选漏洞元素。漏洞语法特征共分为 4 类: 库/API 函数调用、数组使用、指针使用或表达式定义[20]。图 1 展示了四类语法特征的具体例子。图 1(a)针对的是“memset”元素, 可以看出它是一个被调用的元素, 通过检索匹配发现“memset”又是属于 API 函数调用中的一个, 则将“memset”这个元素定义为属于库/API 函数调用这一类语法特

征。图 1(b)当中, 对于图中的“source”元素, 可以看出它是一个标识符, 并且它出现在前面分支的标识符声明语句中, 且该标识符声明语句包含有“[”的字符, 则将“source”这个元素定义为数组使用这一类。图 1(c)当中, 对于图中的“data”元素, 可以看出它是一个标识符, 并且它出现在前面分支的标识符声明语句中, 且该标识符声明语句包含有“*”的字符, 则将“data”这个元素定义为指针使用这一类。图 1(d)当中, 对于图中的“data=dataBuffer-8”元素, 可以看出它一个表达式语句, 并且该语句包含有“=”字符, 且在右边有 1 个或多个标识符, 则将“data=dataBuffer-8”这个元素定义为表达式定义这一类。

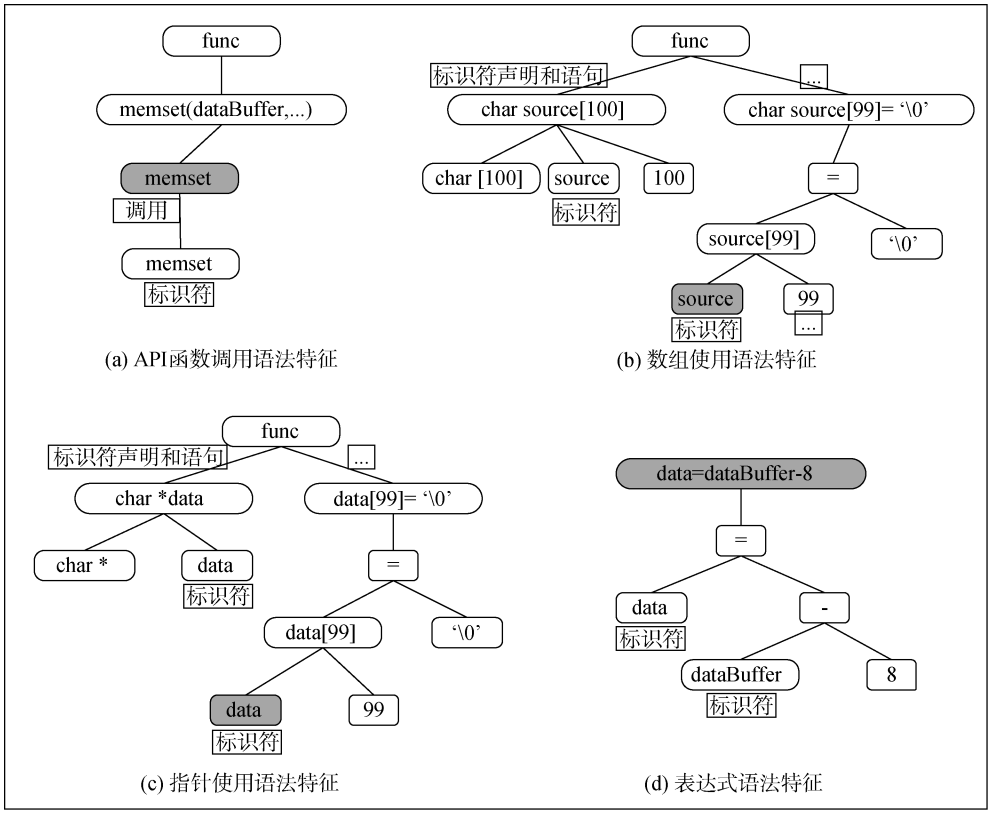


图 1　四类语法特征实例

Figure 1　An example of four types of syntax features

2) 构建候选漏洞代码段。首先基于静态解析工具 Joern[1]为源代码中的每个函数生成程序依赖图(Program Dependence Graph, PDG)[34]。简单来说, PDG 是以图形化展现程序间的控制依赖关系和数据依赖关系, PDG 中每个节点对应源代码中的语句(如声明、赋值、表达式、控制逻辑等); 针对每个候选漏洞元素, 在对应的 PDG 中, 根据每个候选漏洞元素所在位置的前后两个方向进行前向切片和后向切片, 并将前向切片和后向切片结合得到每个候选漏洞元素对应的程序切片; 根据每个候选漏洞元素对应的程序切片在 PDG 中所包含的节点, 将节点所对应的代码语句按照其在源代码中的顺序保存, 得到每个候选元素对应的候选代码段。这里所构建的候选漏洞代码段就是上述提到的代码切片 *CG*。

CG 是对大量的 C 语言执行上述 2 个步骤后所得到的样本数据, 样本粒度会小于函数级。之所以针对 C 语言是因为它是一种类型不安全的语言。由于开发人员有意或无意的行为, 以及 C 语言固有的一些特性, 使得所开发的程序中存在某些瑕疵或安全缺陷, 而这些瑕疵或安全缺陷就有可能造成被攻击者所利用的漏洞[35]。因此基于 C 语言所编写的代码会比其他语言更有可能出现漏洞。

从 *CG* 的定义中可以获取以下信息: 首先, *CG* 可以展现代码的语义信息, 但它无法反映语法信息, 也就是代码的结构信息。其次, *CG* 由具有语义依赖相关的程序语句组成, 因此它可能会不关注于对结构信息很重要但与其他信息没有语义关系的语句。为了得到完整、准确的结构化数据集, 必须解决上述问题。

如图 2 所示, 结构化数据集的生成包括两个步骤: 补充语法信息和解析语法信息。而补充语法信息还可以进一步分为以下两个子步骤:

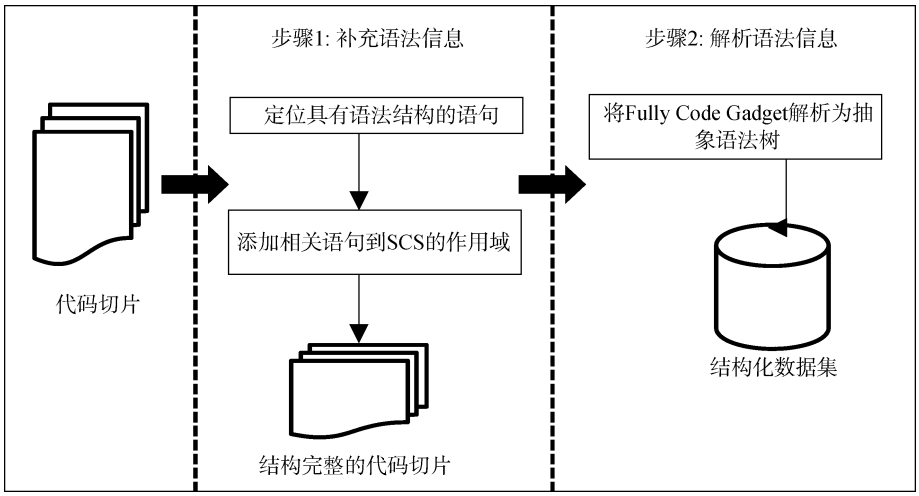


图 2　结构化数据集生成过程

Figure 2　Structured data set generation process

1) 定位具有语法结构的语句。作为一个简单的例子, 图 3(a)显示的是漏洞号为 *CWE-121* 的 *CG*。通用缺陷枚举(Common Weakness Enumeration, CWE)[36]是漏洞类型标识, 该标识是由美国国土安全部国家计算机安全部门资助的软件安全战略性项目所提出的, 随后逐渐成为主流的漏洞类型编号方式, 每个 CWE 号对应一种公认的漏洞类型, 如 *CWE-121* 表示的是栈溢出类型的漏洞。第一行是 *CG* 的源文件文件名, 剩下的每一行都是分为两个部分,一部分是代码语句, 另一部分是该语句所对应的行号, 如“329 char *数据;”意味着该语句“char * data;”是源文件的第 329 行。*CG* 中的每个语句都与其他语句有依赖关

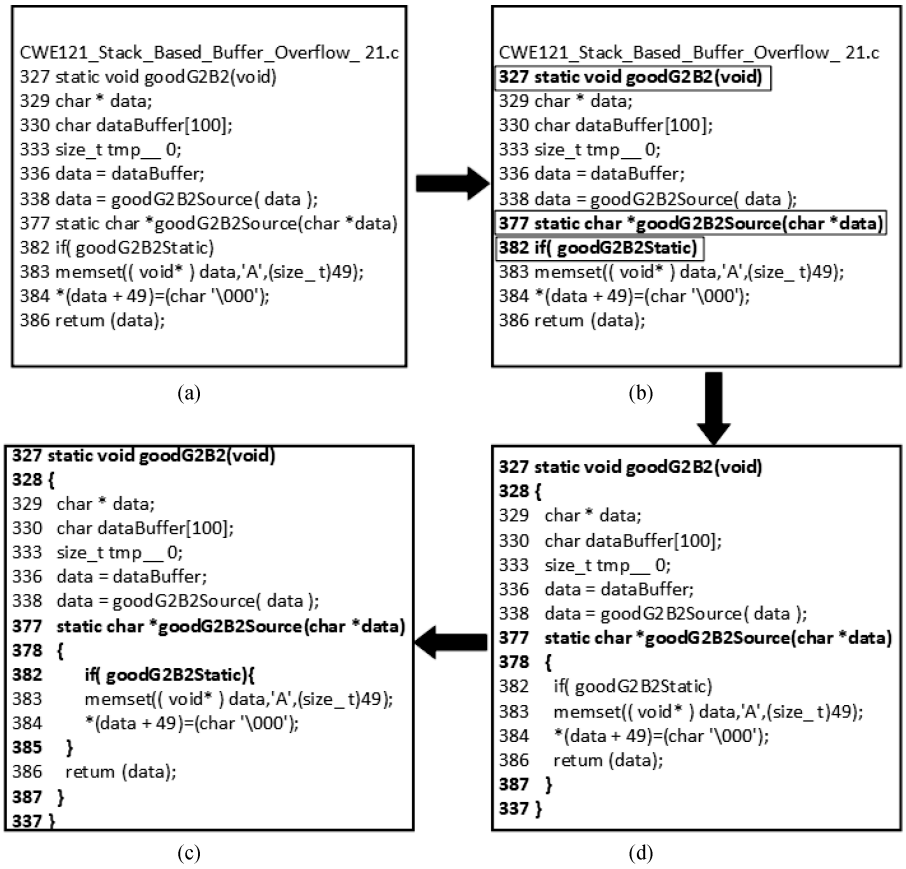


图 3　补充语法信息的一个简单例子

Figure 3　A simple example of supplementing syntax information

系, 如对于“char * data”和第 336 和 338 行有数据依赖关系, 与第 384 行有控制依赖关系。但是, 从例子中我们也可以看出, *CG* 不能显示代码的语法结构关系。例如, 第 382 行是一个 *if* 分支语句, 随后的一些语句应该包含在“{}”符号中, 以显示 *if* 的作用域, 从而反映出代码要表示的结构关系。否则, 每个语句只是一个独立的个体, 我们无法得到完整的结构依赖关系。同样的, 第 327 行和第 377 行分别是代码切片中的两条函数声明语句, 但并未给出“{}”符号将各自的函数作用域区分开来, 无法显示结构关系。因此, 提出了语法结构完整的代码切片的概念, 它是 *CG* 的改进版本。下面给出语法结构完整的代码切片的定义。

定义 2. 语法结构完整的代码切片(Fully Structured Code Gadget, *FG*), 语法结构完整的代码切片由许多程序语句组成, 语句彼此之间具有数据依赖或控制依赖方面的语义关联, 并且能显示出完整而准确的语法结构关系。从定义可以看出, 语法结构完整的代码切片所包含的程序语句内容与代码切片 *CG* 一致, 但增加了语句之间存在的语法结构关系。后续部分使用符号 *FG* 表示语法结构完整的代码切片。

为了将 *CG* 转换成 *FG*, 首先需要找到具有语法结构的语句。实现阶段共考虑了两种语句, 一种是函数声明语句, 另一种是分支控制语句。函数声明语句是包含函数定义的语句, 例如图 3(a)中的第 327 和 377 行。首先统计现有的数据集中函数声明语句涉及的所有可能的代码元素。将其构造为规则集 1, 规则集 1 包含 6 个代码元素规则, 包括: *static*、*void*、*int funcname()*、*char *funcname()*、*char funcname()*、*short funcname()*。分支控制语句则是包含分支代码元素的语句。总共有 5 条规则, 称其为规则集 2, 包括 *if*、*while*、*for*、*else* 和 *switch*。基于规则集 1 和规则集 2 在 *CG* 上执行字符串匹配算法, 可以得到所有包含语法结构的语句。图 3(b)显示了所有具有语法结构的语句(用黑矩形框标记)。为了后续描述方便, 此处给出定义 3。

定义 3. 语法结构语句, 包含语法结构的语句由许多程序语句组成, 这些语句属于 *CG*, 并且与规则集 1 和规则集 2 匹配。使用集合 *SCS* 来表示这些语句集合。

2) 添加相关语句到 *SCS* 的作用域。在定位到属于 *SCS* 的语句后, 需要恢复它的语法结构, 即识别出它的结构体作用域内应该包含的语句, 并添加“{}”符号表示该结构关系。为次, 提出了一套作用域匹配算法用于恢复语法结构, 称为算法 1, 算法细节如下所示。

算法1. 作用域匹配算法

输入: *CG*; *SCS*

输出: *FG*

```
1: For i in SCS:
2:    get symbol "{" and line number a;
3:    get symbol "}" match to "{" and line number b;
4:    IF i∈ 规则集1:
5:       select statements between (a,b) to "{" and "}";
7:    ELSE:
8:      IF not other branch:
9:       select statements between (a,b) to "{" and "}";
10:   ELSE:
11:       goto 2;
12: return FG;
```

该算法针对 *CG* 所对应的 *SCS* 中的每一条语句 *i*(第 1 行), 首先获取该语句所拥有的作用域开端符号“{”, 其次通过字符识别和统计的方法获取作用域开端所对应的结束符“}”, 并将开端和结束的行号记录(第 2-3 行)。在取得行号后, 根据上述两种语法结构语句, 可以分两种情况进行考虑。如果 *i* 属于函数声明语句, 则判断 *CG* 中有哪些语句的行号是在上述两个行号的范围内, 将这些语句纳入 *i* 的作用域, 并用符号“{”和“}”包围这些语句(第 4~5 行)。如果 *i* 属于分支控制语句, 则首先需要识别出在这个分支控制语句下是否还有嵌套的分支语句, 若没有, 则做跟函数声明语句一样的操作(第 7~9 行)。若有包含嵌套分支, 则需要先将内层分支嵌套的语法结构补充完整(第 10~12)行。所有的 *i* 都执行执行结束, 集合 *SCS* 就匹配到了对应的作用域代码, 也就能够将 *CG* 转化为 *FG*, 输出 *FG* 即可(第 12 行)。图 3(a)到图 3(c)展示了为 *CG* 补充函数声明语句作用域的过程, 可以看出为 327 行和 277 行两个函数声明语句都补充了对应的语法结构作用域。图 3(d)则展示了为分支控制语句补充语法结构作用域后的结果, 也是最终得到的 *FG*。可以看出, 执行语法信息补充这个操作后, *FG* 可以完全展示代码的语法结构和语义信息。

为了学习和表征代码的结构化信息, 需要进一步将 *FG* 表征为具有结构化语法信息的形式。已有的研究工作中用于表征结构化信息的载体有很多, 例如控制流图(Control Flow Graph, CFG)或程序依赖图 PDG, 但 CFG 中的节点代表一个基本块, 它是以图的形式表示一个过程内所有基本块执行的可能流向。因此虽然 CFG 可以展示结构关系, 但是粒度较粗, 无法展现基本块的具体代码元素以及基本块内不同代码元素

存在的语义关系。PDG 也同样存在这样的问题, 无法展示最细粒度的依赖关系。本文需要综合考虑语义信息和语法结构信息, AST 可以充分展示代码的语法结构关系, 其次AST的节点可以展示具体的代码元素信息, 不同代码元素之间的依赖关系也可以通过节点的层次关系获取。因此选择使用 AST 作为结构化信息的载体构造结构化数据集。首先, 使用静态解析工具 Joern[1]解析完整的 *FG* 生成抽象语法树, 解析完成后会将代码元素转换成节点, 并将代码之间的结构关系转换成边, 这些节点和边保存在 Joern 自带的数据库中。将具有结构依赖的节点按照层次结构连接起来, 就可以构建出一个完整的抽象语法树。由于每个 *CG* 都能生成对应的*FG*, 并且每个*FG*可以进一步解析成一个抽象语法树。因此基于上述方法可以构建出一个与 *CG* 数量相等的结构化数据集。由于 *CG* 本身是细粒度数据集, 并且具有跨函数的特性, 因此所构建的结构化数据集也是可以跨函数的细粒度结构化数据集。目前共包括 20 万个样本。

3.2 Astor 的总体架构

Astor 有两个阶段: 学习阶段和检测阶段。学习阶段的输入是大量的训练数据集, 其中一些是有漏洞的, 另一些是无漏洞的。此处所说的“漏洞”是指一个样本包含一个或多个已知的漏洞。学习阶段的输出是漏洞模式, 这些模式被编码到神经网络中。

如图 4 所示, 学习阶段共有两个子步骤: 表征抽象语法树和训练神经网络模型。

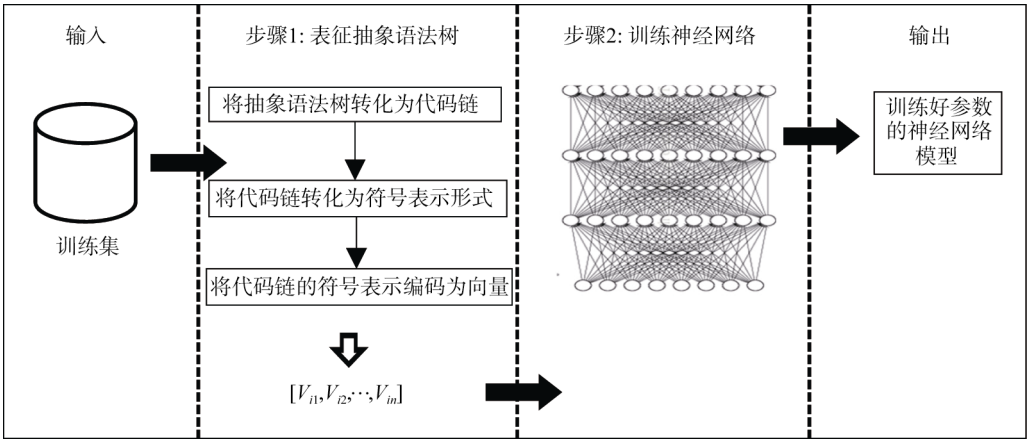


图 4 Astor 的学习阶段

Figure 4 The training phase of Astor

1) 表征抽象语法树。该步骤还分为 3 个子步骤, 下面将详细说明: (i)将抽象语法树转化为代码链。本文构建了一个跨函数的细粒度结构化数据集, 该数据集基于 AST 表征代码的结构信息, 每个样本实际上是一个抽象语法树。树是由节点和边组成的, 每个节点存储着一些代码信息, 边代表了节点之间的结构关系。为了进一步获得这种结构信息, 本文提出使用深度优先遍历方法将树转换成线性链。具体来说, 它以深度优先的方式遍历树中的每个节点, 逐个记录每个节点上的代码, 最后将代码连接成一个线性链, 称之为代码链(Code Chain, *CC*)。在此步骤之后, 进一步将每个完整的 *FG* 所对应的 AST 转换为一个代码链。(ii)将代码链转化为符号表示形式。这一步的目的是启发式捕获训练集中的一些语义信息。首先, 删除非 ASCII 字符和注释, 因为它们与漏洞无关。其次, 以一对一的方式将用户定义的变量映射到符号名(例如, *VAR1*、*VAR2*), 同时, 当多个变量出现在不同的代码链中时, 可能会映射到相同的符号名。第三, 以一对一的方式将用户定义的函数映射到符号名称(如 *FUN1*、*FUN2*), 同时注意, 当多个函数出现在不同的代码链中时, 可能会映射到相同的符号名称。(iii)将代码链的符号表示编码为向量。每个代码链需要通过其符号表示形式编码到一个向量中。为此, 我们首先利用词法分析将符号表示中的代码链划分为一系列标记, 包括标识符、关键字、操作符和符号。例如, 代码链中的“strcpy(VAR5;VAR2)”语句可以由 7 个 token 序列表示: “*strcpy*”, “*(*”, “*VAR5*”, “*;*”, “*VAR2*”, “*)*”, “*;*”。神经网络模型只能接受向量格式的样本数据, 因此进一步引入了 Word2Vec[37]工具将 token 标记转换为向量, Word2Vec 可以根据给定的语料库, 通过优化后的训练模型快速有效地将一个词语表达成向量形式, 之所以选择它是因为它在文本挖掘中被广泛使用。

2) 训练神经网络模型。训练神经网络模型是为了能够学习到包含结构化信息的代码链样本所具有的数据特征, 形成判断样本有无漏洞的能力。Astor

是基于有监督的神经网络模型而构建的, 由于代码链样本对应于前文所提到的*CG*样本, 因此二者具有相同的标签, 即带有漏洞的代码链具有“1”的标签, 反之具有“0”的标签。在将大量代码链样本利用 Word2Vec 工具转化为向量后, 就可以作为神经网络的输入进行训练。

本文使用 Keras 框架辅助构建神经网络模型, 这是一种高度模块化的框架, 可以根据需要对网络层进行线性堆叠, 相关参数可以在模型构建过程中进行设置。在神经网络领域, 不同网络结构的模型对数据的表征学习能力是不同的, 如卷积神经网络(Convolutional Neural Networks, CNN)和循环神经网络(Recurrent Neural Network, RNN)所针对的数据类型就会存在差异, 即使是 RNN 这一类的模型, 不同的线性堆叠方式也会对学习能力产生影响。因此, 在 Astor 的实现阶段设计了不同网络结构的模型对结构化数据进行学习, 旨在找出最适合于结构化数据的模型, 模型设计将在第四章实验部分介绍。

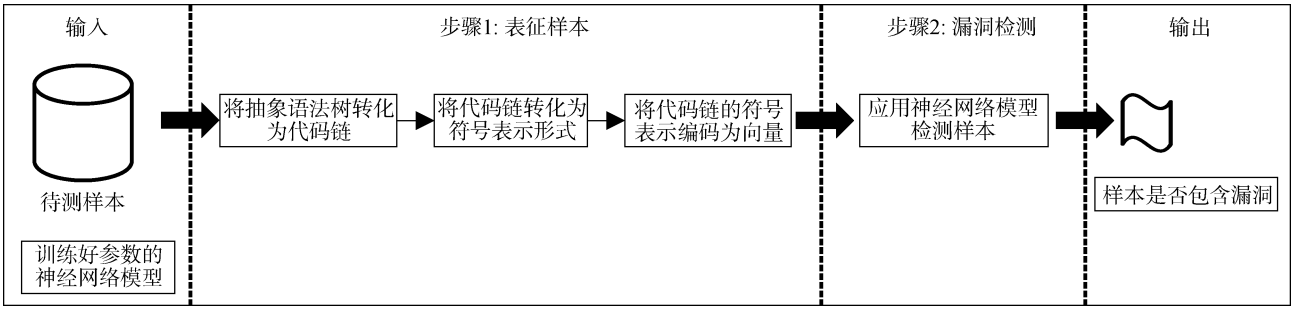


图 5 Astor 的检测阶段

Figure 5 The detection phase of Astor

检测阶段: 给定一个或多个目标样本, 将其转换为代码链 *CC*, 之后进行符号映射, 映射后的数据再被进一步编码成向量, 并被用作已经训练好的漏洞检测模型的输入。模型根据向量值输出检测结果。如图 5 所示, 检测阶段也可以分为两个子步骤:

1) 表征抽象语法树。它有三个子步骤。这与学习阶段的步骤是相似的, 这里就不再赘述。

2) 漏洞检测。该步骤利用已经训练完成的神经网络模型对所输入的 *CC* 对应的向量进行分类。当一个向量被分类为“1”意味着相应的 *CC* 是有漏洞的, 分类结果为“0”则代表这个样本无漏洞。Astor 系统是基于深度学习模型学习大量 C/C++源码数据存在的漏洞特征而生成的漏洞检测系统, 因此可以用于检测 C/C++类型的源码数据是否存在漏洞, 但系统本身是通用且泛化的, 若引进更多的源码解析工具总结其他语言(如 JAVA)的语法特征以及生成细粒度样本, 即可实现对其他语言的漏洞检测。

4 实验设计与结果分析

本文所设计的实验用于回答以下三个研究问题:

RQ1: 哪种神经网络模型更适合学习结构化数据?

RQ2: 什么类型的数据更适合结构表征?

RQ3: 结构表征比线性表征更好吗?

4.1 评估漏洞检测系统的度量标准

实验阶段使用以下 5 个指标来评估漏洞检测系统 Astor 的性能。其中 *TP* 为正确检测到漏洞的样本数量, *FP* 为误报的样本数量, *FN* 为未检测到真实漏洞的样本数量, *TN* 为未检测到漏洞的样本数量。

(1) 误报率: 误报样本占所有无漏洞样本数量的比例。计算公式如下:

$$FPR = \frac{FP}{FP + TN}$$

(2) 漏报率: 漏报样本数量占所有漏洞样本数量的比例。计算公式如下:

$$FNR = \frac{FN}{FN + TP}$$

(3) 召回率: 检测出的真实漏洞样本占全部漏洞样本的比例。计算公式如下:

$$R = \frac{TP}{FN + TP}$$

(4) 精度: 检测正确的漏洞样本数目与所有被分类成漏洞的样本数目之比。计算公式如下:

$$P = \frac{TP}{TP + FP}$$

(5) F1-指标: 同时考虑了精度和召回率的 F1 标准的算术平均数, F1 标准是精确率(*P*)和召回率(*R*)的调和平均值, 反映了模型的整体识别能力。计算公

式如下:

$$F1=\frac{2\times P\times R}{P+R}$$

4.2 结构化数据集的生成

本文构建的结构化数据集来源于 SySeVR[20]中所使用的 SARD(Software Assurance Reference Dataset, SARD)数据集, 该数据集是美国国家标准与技术研究院(National Institute of Standards and Technology, NIST)的参考数据集, 公开了超过 100000 个测试用例, 这些测试用例来源于人工编写的漏洞代码。SARD 数据集将这些程序标记为“good”(即, 无漏洞的), “bad”(即, 有漏洞的)或“mixed”(即, 有漏洞程序和其修补版本都可用)。SARD 数据集按照漏洞类型对数据样本划分种类, 有多种漏洞类型, 以 CWE 号作为类型标识。将 SARD 数据集构建为结构化数据的步骤如下:

1) 将源代码解析为有标签的代码切片。基于 4 类语法特征(库/API 函数调用、数组使用、指针使用和表达式定义)可以得到 4 种类型的代码切片 *CG*。根据切片是否包含漏洞代码为其添加“1”(有漏洞)或“0”(无漏洞)标签。该部分的实现是参考 VulDeepecker[19]的方法, 基于 Joern 工具对源码做静态解析而得到的。

2) 将代码切片补全语法信息后进一步转化为抽象语法树。使用 Python 语言实现本文在系统设计部分所提出的语法信息补全机制, 该机制首先自动识别代码切片中的函数声明语句以及分支控制语句, 之后再溯源到代码切片所在的源文件中, 进一步将属于分支结构和函数结构的代码分配到各自的作用域中得到语法结构和信息完整的 *FC* 切片。之后再基于 Joern 工具将其解析为抽象语法树, Joern 并不能直接得到完整的抽象语法树, 而是分为节点和边两个文件, 节点文件保存的是抽象语法树上每个节点所包含的信息, 包括: 代码, 节点类型, 节点 ID 等, 而边文件则是以父节点和子节点的形式保存节点与节点之间的层次结构关系。因此需要在解析完成后遍历解析过程中得到的两个文件, 从根节点开始基于节点 ID 将解析结果还原为抽象语法树的完整形式。

3) 将抽象语法树表征为代码链并映射成向量格式。本文使用 Python 语言构建基于深度优先遍历算法的结构化表征模型, 将抽象语法树表征为代码链 *CC* 的形式。具体来说, 它是以深度优先的方式从根节点开始遍历抽象语法树中的每个节点, 逐个记录每个节点上的代码信息, 最终得到可以表征其文本和结构信息的线性链 *CC*。之后再构建 Word2Vec 模型将 *CC* 映射为神经网络可以接受的向量样本, 本文所使用的 Word2Vec 模型是从开源的第三方 Python 工具包 Gensim 中导入的。将 *CC* 表征为向量样本包含以下几个步骤: 1)语料库预处理: 对 *CC* 执行分词处理, 得到每个 *CC* 的 token 序列, 并将所有的 token 构建为语料库, 本文使用正则化匹配的方式为 *CC* 分词; 2)将预处理得到的语料库输入 Gensim 内建的 Word2Vec 对象训练词向量模型, 本文设置了 5 轮迭代训练得到最终的模型; 3)将每个 *CC* 的 token 序列输入词向量模型, 映射为向量格式的样本。

本文共生成 420627 个 *CC*。每种类型的 *CC* 数量分别是: 64403 个 FC-kind(与 API 相关), 42229 个 AU-kind(与数组使用相关), 291814 个 PU-kind(与指针相关), 22154 个 AE-kind(与表达式相关)。基于这些 *CC*, 构建了代码链数据库(Code Chain Database, *CCD*)。

4.3 实验与结果分析

为了训练深度神经网络, 从 *CCD* 中为每一种类型的数据选取 10000 个样本(4 类共 40000 条样本)作为实验数据, 用 80%的数据进行训练, 剩下的 20%进行测试。从以下三个方面评估 Astor 的性能:

1) 针对 RQ1 的实验: 在本次实验中, 为了确定哪种模型对结构化数据最友好, 共设计了 5 种神经网络模型分别进行实验, 包括: 卷积神经网络(Convolutional Neural Networks, CNN)、长短期记忆网络(Long Short-Term Memory, LSTM)、双向长短期记忆网络(Bi-Direction Long Short-Term Memory, BLSTM)、门控循环神经网络(Gated Recurrent Unit, GRU)和双向门控循环神经网络(Bi-Direction Gated Recurrent Unit, BGRU)。表 1 给出了 5 种模型得到最优结果时所设置的参数值。其中, Layers 表示对应神经网络的层数; Dropout 是指在训练过程中, 为避免出现过拟合, 按照一定的概率将部分神经网络单元从网络中丢弃, 如 0.2 代表丢弃 20%的神经网络单元; Batchsize 表示一次训练所选取的样本数, 不同的取值对模型的优化程度和训练速度有不同的影响; Units 指的是神经网络中隐藏层的神经元个数, 神经元个数不同对模型训练也有一定影响; VectorDim 和 MaxLen 针对的是样本数据的数据格式而设置的参数, VectorDim 表示样本的向量维度, MaxLen 表示向量的长度。参数的取值是由将样本转为向量的 Word2Vec 模型决定的。根据 Word2Vec 模型的训练结果, 这两个参数值为固定值 40 和 500, 即实验的样本数据都是 40 维, 长度为 500 的向量。

表 1 五种神经网络参数设置

Table 1 The parameter settings of five neural network

模型	Layers	Dropout	VectorDim	BatchSize	MaxLen	Units
CNN	2	0.2	40	16	500	256
GRU	2	0.2	40	32	500	128
LSTM	1	0.4	40	32	500	128
BGRU	1	0.4	40	16	500	128
BLSTM	1	0.4	40	16	500	128

每个模型的输入包含来自 *CCD* 的四种类型的数据。有 32000 个样本作为训练集, 8000 个样本作为测试集。实现阶段基于 Keras 框架构建上述几种神经网络模型, 这是一个用 Python 编写的高级神经网络 API, 它能够以 TensorFlow, CNTK, 或者 Theano 作为后端运行, 此外, Keras 可以支持 GPU 和 CPU。本文在实现上述几种神经网络时都是采用 TensorFlow 作为后端实现的, 并且使用 GPU 训练神经网络。实验环境的具体配置参数和软件版本信息如表 2 所示。

表 2 实验环境详细配置参数

Table 2 Detailed configuration parameters of experimental environment

硬件配置	CPU 版本	Intel Xeon E5-1620 3.50GHz
	GPU 版本	NVIDIA GeForce GTX 1080
	内存大小	32GB
	磁盘大小	2TB
软件配置	Keras2.1.2、Tensorflow1.3.0、Python3.5、Gensim3.2.0	
操作系统	Linux 3.10.0-514.6.2.el7.x86_64	

表 3 给出了针对 RQ1 的实验结果。可以观察到针对 *CCD* 的数据, 使用 BGRU 模型更有效, 漏洞率和误报率都低于其他模型。这是因为 GRU 模型内部单元具有特殊的门结构, 这种结构使得模型可以在训练过程中很好的保留数据的重要特征, 并且可以保证神经网络在长周期的学习传播过程当中不会丢失信息。而 BGRU 是双向的 GRU 模型, 在保持了 GRU 模型优势的基础上, 可以通过双向传播学习到数据样本内部的前后依赖关系。

表 3 五种神经网络对比结果(单位(%))

Table 3 Experimental results of five neural networks were compared(unit (%))

模型	FPR	FNR	A	P	R	F1
CNN	6.7	11.3	91.3	88.9	88.7	88.8
GRU	5.6	7.3	93.6	90.8	92.7	91.7
LSTM	6.0	18.5	89.1	88.8	81.5	84.9
BGRU	4.5	6.8	95.5	92.4	93.2	92.8
BLSTM	7.5	8.4	92.0	87.8	91.6	89.6

2) 针对 RQ2 的实验: 为了回答 RQ2, 本文进一步使用 BGRU 模型建立了 Astor, 并对四类数据分别进行了实验。对于每种类型, 我们随机选择了 8000 个样本作为训练集, 2000 个样本作为测试集。

表 4 给出了实验结果。可以发现 Astor 对指针类型的数据(PU-kind)效果是最好的。初步分析是因为指针类型的数据较为复杂, 所涉及的数据依赖和控制依赖较多, 所以该类型样本的代码长度较大, 可以蕴含足够多的语法和语义信息供模型学习。为验证这个观点, 本文进一步做了补充实验, 统计了四种类型数据的样本平均长度。统计结果如表 5 所示。可以看出, 指针类型数据(PU-kind)的平均长度为 28.3, 远高于其他三类。这证明该类型数据样本更适合本文所提出的 Astor 系统是因为其数据较为复杂, 代码的长度最大, 所包含的信息量是最多, 因此可以通过其抽象语法树学到足够多的结构信息。而表 4 另外三类数据检测结果的 F1 值与表 5 中对应的平均长度相结合, 可以进一步证明这个结论的准确性。

表 4 四种类型数据对比结果(单位(%))

Table 4 Experimental results of four types of data were compared (unit (%))

模型	FPR	FNR	A	P	R	F1
Astor/FC	12.1	8.8	89.0	80.8	91.2	85.7
Astor/AU	20.5	8.0	83.9	71.5	92.0	80.5
Astor/AE	2.0	32.2	86.0	93.1	67.8	78.2
Astor/PU	5.7	5.3	95.0	90.2	94.7	92.4

表 5 四种类型数据的平均长度(单位(行数))

Table 5 Four types data's average number of rows (unit (the number of rows))

	FC-kind	AU-kind	PU-kind	AE-kind
平均长度	17.9	15.2	28.3	13.7

3) 针对 RQ3 的实验: 在这个实验中, 将 Astor 与线性模型(Linear Model, LM)进行比较, 在线性模型中, 文本代码直接输入神经网络进行特征学习并应用于漏洞检测。LM 采用 SySeVR[12]所提出的 SeVCs 数据集, Astor 采用本文所构建的 *CCD* 数据集。为了充分展示模型的性能, 分别测试了之前实验中提到的指针类型数据(PU-kind)和 4 种类型混合(ALL-kind)的数据。

表 6 给出了实验结果, 可以发现: 针对指针类型数据(PU-kind)数据, Astor 比线性模型具有更好的检测效果, F1 值可以提高 2%。此外, 尤其需要注意的是, 在漏报率(FNR)方面, Astor 可以比线性模型降低

将近 9%。这对于漏洞检测来说是至关重要的, 理想的漏洞检测检测方法是同时满足低误报和低漏报, 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, 因此漏报率是决定一个检测系统性能的关键因素。从这方面来看, 本文所提出的 Astor 是一个切实可行, 且效果要优于当前线性模型的系统。而对于通用类型的数据, 即混合的随机数据, Astor 与线性模型的效果不相上下, 能达到 92%以上, 这说明 Astor 可以作为一个通用型的系统使用, 能够满足检测多种漏洞, 多种类型数据的要求。

表 6 Astor 和线性模型对比结果(单位(%))
Table 6 Experimental results of Astor versus linear model (unit (%))

模型	FPR	FNR	A	P	R	F1
Astor/PU	5.7	5.3	95.0	90.2	94.7	92.4
LM/PU	1.0	14.2	94.8	95.2	85.8	90.1
Astor/ALL	4.5	6.8	95.5	92.4	93.2	92.8
LM/ALL	1.0	10.2	96.1	96.2	89.8	92.7

实验表明, 在通用数据集上, Astor 可以达到与线性模型相同的性能, 但在复杂数据集上, Astor 的性能更好。因此, 本文提出的方法和系统是有效的, 并且在实际应用中会比线性模型更实用, 因为在实际应用中, 代码通常较为复杂, 蕴含多种语法结构和语义信息。

5 总结与展望

本文提出了智能漏洞检测系统 Astor, 致力于学习源代码的结构化信息并应用于智能化漏洞检测。基于所构建的结构化数据集, 进行了大量的实验。结果表明该系统是有效的, 且针对现实场景中较为复杂、且语法丰富的数据, 效果会比线性模型更好。此外, 所构建的数据集以及构建方法可以为漏洞检测领域探索新方法和新模型提供思路。

具体来说, Astor 是基于神经网络训练而成的二分类系统, 这是由于训练神经网络时使用的是二分类数据(即“0”和“1”标签), 未来的研究中若有足够的样本用于构建多分类的数据集, 就可以基于本文所提出的方法训练出适用于多分类的漏洞检测系统, 能够识别出样本存在哪一类漏洞。此外, Astor 的检测粒度为切片级, 切片级的样本所包含的代码行较少, 因此对于识别漏洞位置有非常大的帮助, 后续可以针对切片级的样本开展漏洞定位研究, 识别具体漏洞代码位置。

致 谢 本课题得到国家自然科学基金项目(No.U1936211), 深圳市基础研究(学科布局)(No.JCYJ20170413114215614), 广东省省级科技计划项目(No.2017B010124001), 广东省重点领域研发计划项目(No.2019B010139001)的资助, 在此表示感谢。

参考文献

[1] Li Z, Zou D Q, Xu S H, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[C]. *2018 Network and Distributed System Security Symposium*, 2018: 258-263.

[2] F.Yamaguchi, A.Maier, H.Gascon. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities[C]. *IEEE Symposium on Security and Privacy(SP'15)*, 2015:54-62.

[3] Wang L, Chen G, Jin M Z. Detection of Code Vulnerabilities via Constraint-Based Analysis and Model Checking[J]. *Journal of Computer Research and Development*, 2011, 48(9): 1659-1666. (王雷, 陈归, 金茂忠. 基于约束分析与模型检测的代码安全漏洞检测方法研究[J]. 计算机研究与发展, 2011, 48(9): 1659-1666.)

[4] Hoang T, Oentaryo R J, Le T D B, et al. Network-Clustered Multi-Modal Bug Localization[J]. *IEEE Transactions on Software Engineering*, 2019, 45(10): 1002-1023.

[5] Li Z J, Zhang J X, Liao X K. Survey of Software Vulnerability Detection Techniques[J]. *Chinese Journal of Computers*. 2015, 388(04): 19-34. (李舟军, 张俊贤, 廖湘科. 软件安全漏洞检测技术[J]. *计算机学报*, 2015, 388(04): 19-34.

[6] W.Wong, R.Gao, Y.Li. A Survey on Software Fault Localization[J]. *IEEE Trans. Software Engineering*. 2016, 42(02): 707-740.

[7] F.Yamaguchi, N.Golde, D.Arp. Modeling and Discovering Vulnerabilities with Code Property Graphs[C]. *IEEE Symposium on Security and Privacy(SP'14)*, 2014:259-263.

[8] 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 Conference on Computer Security Applications*, 2016: 201-213.

[9] J.Li, M.Ernst. CBCD: cloned buggy code detector[C]. *The 34th International Conference on Software Engineering (ICSE'12)*, 2012: 310-320.

[10] 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 - ASE '10*, 2010: 447-456.

[11] 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 - ASE '10*, 2010: 369-375.

[12] J.Jang, A.Agrawal, D.Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions[C]. *IEEE Symposium on Security and Privacy(SP'12)*, 2012: 48-62.

[13] S.Ki , S.Woo, H.Lee. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery[C]. *IEEE Symposium on Security and Privacy(SP'17)*, 2017: 2375-2387.

[14] L.Jiang. DECKARD : Scalable and accurate tree-based detection of code clones[C]. *29th International Conference on Software Engineering (ICSE'07)*, 2016: 96-105.

[15] Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: Scaling Code Clone Detection to Big-code[C]. *The 38th International Conference on Software Engineering*, 2016: 236-245.

[16] 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.

[17] Meneely A, Williams L. Strengthening the Empirical Analysis of the Relationship between Linus' Law and Software Security[C]. *The 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010: 258-269.

[18] Shin Y, Williams L. An Initial Study on the Use of Execution Complexity Metrics as Indicators of Software Vulnerabilities[C]. *The 7th international workshop on Software engineering for secure systems*, 2011: 1-7.

[19] N.Nagappan, T.Zimmermann, L.Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista[C]. *Third International Conference on Software Testing, Verification and Validation(ICST'10)*, 2010:268-276.

[20] X.Yang, D.Lo, X.Xia. Deep Learning for Just-in-Time Defect Prediction[C]. *IEEE International Conference on Software Quality, Reliability and Security (QRS'15)*, 2015:625-635.

[21] Li Z, Zou D Q, Xu S H, et al. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities[EB/OL]. 2018: arXiv:1807.06756[cs.LG]. https://arxiv.org/abs/1807.06756.

[22] L.Guanjun, J.Zhang, W.Lu, et al. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017: 2539-2541.

[23] P.Michael P, S.Koushik. Deep learning to find bugs. Technical Report TUD-CS-2017-0295.

[24] Xu X J, Liu C, Feng Q, et al. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 363-376.

[25] B.Alsulami, E.Dauber, and R.Harang, "Source Code Authorship Attribution Using Long Short-Term Memory Based Networks," in Proc. European Symposium on Research in Computer Security (ESORICS'17), pp.65-82,2017.

[26] Alsulami B, Dauber E, Harang R, et al. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks[J]. *Computer Security - ESORICS 2017*, 2017: 65-82.

[27] Kipf T N, Welling M. Semi-Supervised Classification with Graph Convolutional Networks[EB/OL]. 2016: arXiv:1609.02907[cs.LG]. https://arxiv.org/abs/1609.02907.

[28] Wei H H, Li M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code[C]. *The Twenty-Sixth International Joint Conference on Artificial Intelligence*, 2017: 3034-3040.

[29] Dam H K, Pham T, Ng S W, et al. A Deep Tree-based Model for Software Defect Prediction[EB/OL]. 2018: arXiv:1802.00921[cs.SE]. https://arxiv.org/abs/1802.00921.

[30] A.MMiltiadis, B.Marct, K.Mahmoud. Learning to Represent Programs with Graphs[C]. 6th International Conference on Learning Representations(ICLR'18), 2018:865-872.

[31] K.Tai, R.Soche, C.Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks[C]. *The 53rd Annual Meeting of the Association for Computational Linguistics(ACL'15)*, 2015: 123-135.

[32] L.Mou, G.Li, Z.Jin. Convolutional Neural Network over Tree Structures for Programming Language Processing[C]. *Thirtieth Aaai Conference on Artificial Intelligence(AAAI'16)*, 2016: 368-372.

[33] L.Yujia, T.Daniel, B.Marc, et al. Gated Graph Sequence Neural Networks[C]. *4th International Conference on Learning Representations(ICLR'16)*, 2016: 65-75.

[34] Schlichtkrull M, Kipf T N, Bloem P, et al. Modeling Relational Data with Graph Convolutional Networks[M]. The Semantic Web. Cham: Springer International Publishing, 2018: 593-607.

[35] Z.Yaqin, L.Shangqing, S.Jingkai, et al. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks[C]. *Advances in Neural Information Processing Systems 32(NIPS'19)*, 2019:56-68.

[36] PDG, https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E4%BE%9D%E8%B5%96%E5%9B%BE/22341209?fr=aladdin.

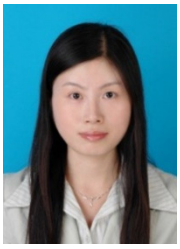
[37] Chen X Q, Xue R. Cause, Exploitation and Mitigation of Program Vulnerability C and C++ Language as an Example[J]. *Journal of Cyber Security*, 2017, 2(4): 41-56. (陈小全, 薛锐. 程序漏洞:原因、利用与缓解——以 C 和 C++语言为例[J]. *信息安全学报*, 2017, 2(4): 41-56.)

[38] CWE, https://cwe.mitre.org/.

[39] Word2Vec, http://radimrehurek.com/gensim/models/word2vec.html, Nov, 2019.

陈肇炫 (1995-), 男, 福建泉州人, 硕士研究生, 就读于华中科技大学, 主要研究方向为: 软件安全、漏洞检测。Email: zhaoxaunchen@hust.edu.cn

邹德清 (1975-), 男, 湖南湘潭人, 于 2004 年在华中科技大学获得博士学位, 华中科技大学教授, 博士生导师, 主要研究方向: 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。Email: deqingzou@hust.edu.cn



李珍 (1981-), 女, 河北保定人, 博士研究生, 就读于华中科技大学, 主要研究方向为: 软件安全、漏洞检测。Email: lizhen_hust@hust.edu.cn

金海 (1966-), 男, 上海人, 于 1994 年在华中科技大学获得博士学位, 华中科技大学教授, 博士生导师, 主要研究方向: 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。Email: hjin@hust.edu.cn