基于中间语言的 PHP 注入漏洞检测方法研究

张国栋[1], 刘子龙[1], 姚天宇[1], 靳　卓[1], 孙东红[2], 秦佳伟[3]

[1]沈阳航空航天大学 计算机学院 沈阳 中国 110136
[2]清华大学 网络科学与网络空间研究院 北京 中国 100084
[3]国家计算机网络应急技术处理协调中心 北京 中国 100029

摘要　Web 应用数量快速增长并广泛用于各领域, 所存在的漏洞数量也随之增长。注入漏洞是 Web 应用漏洞中最具广泛性和破坏性的, 漏洞检测工具所提取的信息中会缺失部分与漏洞相关的语义信息, 且包含大量与漏洞信息无关的噪声数据, 导致误报和漏报。针对此问题, 提出了一种命名为 Alpherg 的中间语言表示, 具有保留源代码信息、提取源代码中仅与漏洞相关的语义信息和表示源代码控制流信息等特点。利用其进行漏洞特征提取时, 表示结果丢弃了与漏洞无关的噪声数据, 保留了源代码中的上下文信息, 形式上可脱离原有的编程语言, 具有可读性。利用 Alpherg 进行漏洞特征提取, 提出了基于 Bi-LSTM 和注意力机制的 PHP 注入漏洞检测模型, 利用 Bi-LSTM 得到 Alpherg 长序列表示中的上下文关系; 进一步, 通过加入注意力机制计算每个时间步的注意力分布, 更好地利用 Alpherg 中与漏洞相关的信息, 提高了模型的漏洞检测能力。将 Alpherg 与其他特征提取方法处理结果进行了比较, 结果表明 Alpherg 能精确地提取与漏洞存在直接关系的信息, 避免引入过多噪声, 并保留了漏洞的语义信息。在 SARD 数据集上验证了所提出的漏洞检测模型, 漏洞检测结果表明该模型漏洞检测准确率为 98%, 高于作为对比的三个静态检测工具和基于 PHP token 的深度学习漏洞检测模型, 证明了此方法的可行性和有效性。

关键词　注入漏洞检测; 深度学习; 漏洞语义特征; 代码切片
中图法分类号　TP 393.08　DOI 号 10.19363/J.cnki.cn10-1380/tn.2024.11.08

Research on PHP Injection Vulnerability Detection Method Based on Intermediate Language

ZHANG Guodong[1], LIU Zilong[1], YAO Tianyu[1], JIN Zhuo[1], SUN Donghong[2], QIN Jiawei[3]

[1] School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China
[2]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
[3] National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China

Abstract　With the rapid growth of Web applications and use in various fields, the number of vulnerabilities in Web applications has increased. Injection vulnerabilities are the most widespread and destructive in Web application vulnerabilities. However, the information extracted by vulnerability detection tools will miss semantic information related to vulnerability and contain lots of noise data unrelated to vulnerability, which leads to false positives and false negatives. To solve this problem, an intermediate language representation named Alpherg is proposed, which can retain the code information, extract the semantic information only related to the vulnerability, and represent the control flow information in the source code. Using Alpherg to extract vulnerability features, the results discard the noise data unrelated to vulnerability, retain the context information in the source code, and the form can be separated from the original programming language. Using Alpherg, a PHP injection vulnerability detection model based on Bi-LSTM and attention mechanism is proposed. The model uses Bi-LSTM to obtain the context relationship in Alpherg's long sequence representation. Furthermore, attention mechanism is added to the model to utilize the information related to vulnerabilities in the Alpherg representation by calculating the attention distribution at each time step and improving the vulnerability detection ability. Compared Alpherg with other methods, the results show that it can accurately extract information related to vulnerability directly, avoid noise and retain the semantic information of vulnerability. The proposed model is verified on the SARD dataset. The results show that the vulnerability detection accuracy of the proposed model is 98%, which is higher than the three static detection tools and the PHP Token-based deep learning vulnerability detection model, which proves the feasibility and effectiveness of this method.

Key words　injection vulnerability detection; deep learning; semantic features of vulnerabilities; code slicing

通讯作者：秦佳伟, 工学博士, 工程师, 硕士生导师, Email: qinjiawei@cert.org.cn。
本课题得到国家重点研发计划项目(No. 2022YFB3103901)和中关村实验室项目(No. ZGC-02-20220211)资助。
收稿日期: 2023-02-06; 修改日期: 2023-05-31; 定稿日期: 2024-10-12

1 引言

自分布式计算模式(如信息物理系统、物联网等)流行以来, Web 应用数量持续快速增长, 漏洞数量也随之增加, 其中注入攻击是所有 Web 漏洞中最具破坏性的威胁之一。注入攻击可将脚本嵌入到用户输入, 导致 Web 应用程序后台关系数据库管理系统执行恶意 SQL 语句, 或在用户浏览器内执行恶意脚本, 造成安全隐患。据 2020 年国家信息安全漏洞共享平台(China National Vulnerability Database, CNVD)公布的漏洞分布数据[1], Web 应用漏洞占比高达 29.5%, 其中注入攻击占 50%以上, 因此检测和防止注入漏洞对于提高Web应用程序的可靠性和可信度至关重要。在各种 Web 应用程序开发语言中, PHP 因其编写灵活、运行速度快等特点, 被广泛用于 Web 应用开发。据 W3Techs 统计[2], 截至 2022 年底, PHP 在 Web 服务器的编程语言中占比达 77.5%, 因此针对 PHP 语言的 Web 漏洞检测仍是当前研究热点。

为了提高漏洞检测自动化程度及对未知漏洞的发现能力, 将机器学习用于漏洞检测成为技术发展趋势。2017 年, Ma 等[3]提出使用抽象语法树(Abstract Syntax Tree, AST)表示 Java 代码, 并用机器学习模型对 Java 程序进行检测, 但文献[4]研究表明直接使用 AST 进行漏洞提取易导致误报等问题。2018 年, Li 等[5]针对 C 语言的程序漏洞提出使用名为 Code Gadgets 的程序特征表示方法表示 C 语言程序, 并结合深度学习模型实现了漏洞检测系统。但由于 Web 应用程序中没有主函数程序入口, 上述思路无法直接用于 Web 应用漏洞检测。2021 年, Rabheru 等[6]提出通过分析控制流图并利用图卷积网络来学习 PHP 源代码的语义, 该方法在训练过程中未对源代码进行处理, 直接将源代码的语义和上下文信息嵌入分类模型, 由此引入的大量噪声数据影响了检测效果。

依据分析方法, PHP 注入漏洞检测分为两种: 一是动态检测法[7-8], 基本思想是模拟黑客设计各种攻击向量对 Web 应用程序发起攻击, 使用动态分析技术模拟攻击流程, 并根据攻击结果判断是否存在漏洞。如 Liu M 等人[7]认为恶意的 SQL 语句中具有独特的语义信息, 因此可基于语义知识生成语义相同的测试用例, 用于检测 SQLi 漏洞, 并基于此思想设计了 DeepSQLi。DeepSQLi 利用神经语言模型对给定的测试用例或正常用户输入进行变异, 产生更多样化的测试用例用于检测。但该方法类似黑盒测试, 一次无法进行完全的、无遗漏的输入测试, 因用例语义缺失无法覆盖所有执行路径而导致检测失败[9]。另一种是静态检测法[10], 其本质是通过词法和语法分析静态分析变量间的依赖关系, 以检测污点数据能否从污点源传播到污点汇聚点, 检测过程既不运行目标程序, 也无需修改源代码。因具体采用的技术差异, 静态检测又分为两类: 一类是基于规则的静态检测[11-12], Son 等[11]结合污染分析和控制流图(Control Flow Graph, CFG)实现了对丢失授权和拒绝服务攻击等特定语义漏洞的发现, 但其文中给出的检测结果表明该方法的误报率较高, 检测结果无法直接使用; Wasef 等[12]提出了一种基于遗传算法和静态分析的 PHP Web 应用程序检测方法, 通过消除 CFG 中的不可行路径降低了误报率。另一类是基于学习的静态检测[13-18], 此类方法在对源代码完成静态分析预处理后, 使用分析结果训练用于漏洞检测的网络模型, 无需人工设计规则库, 算法适用性强, 如 Medeiros 等[18]采用数据挖掘技术替代了人工审核, 降低了检测误报率; Zheng 等[19]通过对传统机器学习和深度学习方法漏洞检测效果的比较, 证实了深度学习方法可得到更优的漏洞检测结果。

采用深度学习进行漏洞检测, 需对源代码进行预处理, 找到漏洞所在位置并提取出漏洞特征, 对现有特征提取方法分析发现, 存在不同程度的漏洞信息缺失。如 Fidalgo 等[20]利用操作码进行特征提取时, 因操作码无法表示高级语言中所包含的完整语义信息, 导致提取到的漏洞语义信息缺失。Li 等[21]在操作码表示的基础上人为添加了部分缺失的漏洞信息, 一定程度上改善了检测效果, 但受操作码自身局限, 无法彻底避免漏洞特征提取过程中的部分信息缺失。基于以上分析可知, 要实现基于 PHP 的 Web 应用漏洞检测需解决以下 3 个问题: 1)如何对 Web 应用程序源代码进行分析, 确定漏洞可能存在的位置? 2)选择何种漏洞特征表示方法, 以及如何在保证漏洞信息不缺失的情况下实现对程序特征的提取? 3)如何实现漏洞检测, 并获得较高的准确率和较低的误报率?

为了解决上述问题, 本文针对基于 PHP 的 Web 应用 SQL 注入漏洞(SQL injection, SQLi)、跨站脚本攻击(Cross Site Scripting, XSS)和命令注入漏洞(Command Injection, CI)的检测方法展开研究, 提出了一种基于中间语言(命名为 Alpherg)的 PHP 漏洞检测方法, 该方法对源代码进行代码切片、Alpherg 中间语言表示后, 采用基于 Bi-LSTM 和注意力机制的

注入漏洞检测模型完成漏洞检测。综上, 本文的主要贡献如下:

1) 为获取 Web 应用漏洞的具体位置, 提出了一种基于污点分析的程序切片方法。该方法在跟踪污染源执行路径过程中, 通过判断污染源是否经过敏感函数, 确定漏洞可能存在的位置并得到其代码切片, 用于后续漏洞分析。

2) 设计了一种用于漏洞语义信息表示的中间语言 Alpherg。Alpherg 可保留源代码中的代码信息、提取源代码中仅与漏洞相关的语义信息并可表示源代码的控制流信息。利用 Alpherg 进行漏洞特征提取时, 表示结果丢弃了与漏洞无关的噪声数据, 保留了源代码中的上下文信息, 且形式上可脱离原有的编程语言, 具有良好的可读性。

3) 提出了一种基于 Bi-LSTM 和注意力机制的 PHP 注入漏洞检测模型。该模型在使用 Alpherg 提取漏洞特征的基础上, 利用 Bi-LSTM 处理长期依赖关系的能力, 得到 Alpherg 长序列表示中的上下文关系; 在此基础上, 通过在模型中加入注意力机制, 计算每个时间步的注意力分布, 利用 Alpherg 表示中与漏洞相关的信息提高模型的漏洞检测能力。

2 中间语言设计

针对程序切片、漏洞特征提取等常用源代码预处理方法对不同类型漏洞表示一致性差、特征提取不全、检测效果不稳定等问题, 基于漏洞三元组原则设计了中间语言 Alpherg, 通过将漏洞语义统一抽象为漏洞三元组, 消除了源代码中不同命名方式和语法差异带来的干扰, 得到更好的源代码预处理结果, 为后续漏洞检测的准确性和可靠性提供重要保障。

2.1 漏洞语义特征

Web 应用程序包含诸多功能, 各功能的实现都需要用到各种变量和方法, 但一个 Web 漏洞的存在仅与敏感函数的上下文相关, 与用于实现应用功能的其他代码无关。因此本文在对某一漏洞进行分析时, 仅分析与敏感函数相关的变量和方法, 其他代码则视为噪声数据而丢弃。

以 SQLi 为例, 代码 1 是 Web 应用中用于查询用户名的代码, 第 4~6 行读取用户 ID 并直接用于数据库查询, 得到与此 ID 对应的用户名。由于代码中没有对用户输入进行验证, 直接将其作为参数用于数据库查询, 这种情况下, 若攻击者输入恶意代码, 如“-1 union select user, password from users”就会导致 SQLi 的发生。

代码 1 SQLi 示例代码

```
1.  <?php
2.  if(isset($_POST[‘Submit’])){
3.      //Get input
4.      $id = $_POST[ 'id' ];
5.      $query=“SELECT username FROM users
         WHERE user_id = $id;”;
6.  $result=mysql_query($GLOBALS["___mysql_
sto n"], $query);
7.      //Get results
8.      while($row=mysql_fetch_assoc($result)){
9.          //Display values
10.         $username=$result["username"];
11.         echo "<pre>ID: {$id}<br />First name:
{$first}<br />Surname: {$last}</pre>";
12.     }
13. }
14. ?>
```

综上可知 SQLi 与数据库的操作有关, 因此在分析 SQLi 时, 只需关注 Web 应用程序源代码中数据库操作函数的上下文, 即与其相关的变量和方法即可。可见代码 1 中与 SQLi 漏洞相关的语句只有第 4~6 行, 其他语句与漏洞无关, 可被视为噪声数据丢弃。

代码 2 是用于检查用户 ID 合法性的代码, 包含一个 XSS 漏洞。在第 3~4 行, 对用户合法性进行判断, 并在第 8 行显示 ID 的用户权限类型。该行代码执行后会将$_GET[“oauth_signature”]的值直接显示在浏览器中, 若攻击者使用 GET 方式注入恶意攻击脚本, 在受害者访问此页面时执行攻击脚本, 盗取用户信息。以上分析说明 XSS 与输出函数有关, 因攻击脚本需要输出到浏览器中才会被执行, 所以代码 2 中只有第 8 行是影响该漏洞的代码, 其他语句与漏洞无关, 可被视为噪声数据丢弃。

代码 2 XSS 示例代码

```
1.  <?php
2.  //检查账号是否是合法 ID
3.  if($_GET["openid"]{
4.      if(!is_valid_openid($_GET["openid"],$_
GET["timestamp"])){
5.          showerr("API 账号有误");
6.          //输出错误信息
7.          echo "#invalid openid\n";
8.  echo "sig: "$_GET["oauth_signature"]."\n";
9.          exit;
10. }}?>
```

代码 3 是一段 CI 示例代码, 在 Web 应用中用于管理员访问操作系统并执行系统命令。在第 2 行中取得管理员要执行的系统命令, 在第 4 行执行命令, 并在第 6~7 行显示执行结果。由于管理员在应用命令执行函数时, 没有对提交的数据内容进行严格过滤就在函数中执行而造成 CI 漏洞。说明 CI 与命令执行函数有关, 在此段代码中只有第 2 行和第 4 行与此漏洞相关, 其他代码均为噪声数据。

代码 3 CI 示例代码

```
1.  <?php
2.  $cmd=$_GET["cmd"];
3.  echo "<pre>";
4.  exec($cmd, $output);
5.  echo "</pre>";
6.  while(list($key, $value)=each($output))
7.      echo $value. "<br>";
8.  ?>
```

综上, 检测 SQLi、XSS 与 CI 时只需分析与其相关的部分代码, 其他代码对于漏洞检测而言是无关数据, 由此提出一种基于污点分析的程序切片方法。此方法在跟踪污染源执行路径过程中, 通过判断污染源是否经过敏感函数, 确定漏洞可能存在的位置并得到其代码切片。代码 4 即为代码 1 程序切片示例结果。其中, 第 1 行获取了用户 ID, 第 2 行将用户 ID 拼接到 SQL 查询语句中, 第 3 行执行数据库查询并取得查询结果。

代码 4 SQLi 的切片示例代码

```
1.  $id = $_POST[ 'id' ];
2.      $query= “SELECT username FROM user
    WHERE user_id = $id;”;
3.  $result=mysql_query($GLOBALS["___mysql_ston"],
$query);
```

以上结果说明, 程序切片代码保留了漏洞相关代码、语义信息及代码的执行顺序, 剔除了漏洞无关代码。但由于程序切片代码只是代码的堆叠, 没有描述语句间的执行关系, 且没有对用户自定义变量与方法进行归一化处理, 仍无法直接用于漏洞检测。本文针对这一问题设计了名为 Alpherg 的中间语言, 该语言消除了源代码中不同命名方式和语法差异带来的干扰, 保留了源代码中语句执行关系, 可得到更好的源代码预处理结果。

2.2 Alpherg 的设计原则

程序切片是在跟踪污染源执行路径过程中, 从源代码中通过判断污染源是否经过敏感函数得到的代码堆叠。由于程序切片对用户自定义变量和方法未做归一化处理, 用户开发习惯的差异在漏洞检测中会导致检测结果不一致, 无法直接用于漏洞检测。

针对上述情况, 可采用中间语言对程序切片进行语义提取和抽象表示, 避免直接使用简单的代码堆叠, 以提高漏洞检测率。现今针对 PHP 源代码漏洞检测所采用的中间语言有操作码、机器码和 PHP token 等, 但这些中间语言均无法完整表示漏洞信息, 对不同漏洞类型需补足不同的缺失语义信息, 且不具可读性。如在将源代码转义为机器码时, 机器码仅能表示基本机器指令, 不能完整表示高级语言中的语义信息, 造成漏洞语义信息缺失。

为避免上述问题, 本文设计的中间语言 Alpherg 应具备以下特点:

1) 将源代码转义为中间语言后, 应能用特定的语法形式正确表示源代码中的漏洞语义信息。即在转义过程与转义结果中不改变源代码内部的漏洞语义信息, 但需针对源代码中的语法形式进行转义与抽象表示。

2) 可对 PHP 源代码中的漏洞信息进行统一、完整的抽象表示, 在转义过程中无需对漏洞语义信息进行补足。

3) 可简化源代码的代码信息, 抽象表示结果具有可读性。

2.3 Alpherg 的关键字设计

Alpherg 的整体设计基于漏洞三元组原则, 对于任意漏洞的发生过程均可抽象为<污染源, 清洁函数, 敏感函数>的三元组表示, 其中污染源包括用户输入、外部文件等所有不可信数据; 清洁函数是对污染源进行检测过滤的所有操作, 可改变污染源的自身属性; 敏感函数是存在安全隐患的函数, 污染源被敏感函数执行后会引发如 SQL 注入漏洞等安全问题。

将与三元组相关的代码元素抽象后, 设计得到中间语言的关键字, 分别用于表示污染源、清洁函数、敏感函数等漏洞信息。漏洞检测时, 将从源代码中抽象出的关键字表示结果按特定语法规则组合, 得到保留了源代码中的代码信息、漏洞语义信息和代码的控制流信息的 Alpherg 最终表示结果。当有新的漏洞出现时, 只需根据三元组原则在漏洞信息库中添加新漏洞的三元组标识, 即可使用 Alpherg 表示新型漏洞。

表 1 展示了所设计的全部 Alpherg 关键字, 第二列到第四列分别给出了 Alpherg 关键字的所属类型、作用描述及对应示例。需强调的是, 在 Alpherg 中清

洁函数不仅包括检测过滤污染源的所有操作, 还包括源代码中诸如分支语句一类的流程控制语句, 流程控制信息的加入可保证控制语句能够按照程序的正确逻辑执行, 避免在执行路径分析中出现不符合程序执行逻辑的结果, 保证了 Alpherg 中间表示的正确性。

设计过程中, Alpherg 将污染源统一表示为 tainted_var; 将敏感函数按漏洞类型表示为 sink_sql、sink_ci 和 sink_xss; 将清洁函数分为清洁操作(包括字符串操作、检测过滤等)和流程控制语句两部分表示。其中字符串操作分为字符串拼接和字符串替换; 检测过滤包括清洁函数、白名单和赋值语句等所有可能将污染源无害化的操作; 流程控制语句是为了保留源代码中的执行逻辑, 包括分支语句和循环语句等。统一的抽象表示可以消除源代码中不同命名方式和语法差异带来的干扰, 利于后续的漏洞检测分析。

表 1 Alpherg 的关键字列表

Table 1 The list of Alpherg's token

序号	Token	描述	举例
1	sink_sql	SQL 注入	mysql_query
2	sink_ci	命令注入	system
3	sink_xss	跨站脚本	echo
4	sanit	清洁函数	aaddslashes
5	char0	字符串长度为 0 或易被攻击	“”
6	char5	字符串长度<5	“asd”
7	char6	字符串长度>5	“123456”
8	conc	串联运算符	.
9	addstr	添加字符串操作	sprintf
10	m	拼接在字符串中间	“ls’ ” . var . “ ’ ”
11	f	拼接在字符串后部	“cat” . tainted .
12	replace_str	字符串替换操作	preg_replace
13	whitelist	白名单	preg_match
14	forma_v	格式验证	filter_var
15	cond	条件语句条件部分	if()
16	do	条件语句成立部分	if(){..do..}
17	else	条件语句不成立部分	else{..do..}
18	end	条件语句结束	
19	comp	比较符	>
20	assign	常量赋值	=
21	tc_num	类型转换为数字	(int)$tainted
22	tc_str	类型转换为字符串	(string)$tainted
23	ty_num	检测是否为数字	bind_param
24	ty_str	检测是否为字符串	bind_param
25	tainted_var	被污染的变量	

此外, 所设计的部分关键字作为对其他关键字的补充说明, 在中间表示结果中不能单独出现, 需与对应的关键字配合使用。如针对字符串的连接操作“‘ find / size’ ”. $tainted . “ ’ ”, 需根据污染源的位置及前后字符串的形式做补充说明, 最终中间语言表示结果为 conc_m_char0, 其中 conc 表示连接操作, m 代表污染源在字符串连接操作中处于中间位置, char0 表示将污染源闭合的字符串长度为 0 或其闭合字符串易于被攻击, 这里关键字 m 和 char0 不能单独出现, 只能作为关键字 conc 的补充说明。

2.4 Alpherg 的语法设计

根据 Alpherg 的设计原则, 其语法结构应能统一、完整地表示 PHP 源代码中的漏洞语义信息, 并按照漏洞三元组设计原则, 将源代码中的漏洞信息抽象成由漏洞类型、对污染源进行的操作和污染源组成的三元组; 为了后续准确描述漏洞在源代码文件中的详细信息, 在语法结构中还需保留文件名、代码起始行等基本信息; 基于语法结构能用特定形式将漏洞信息、基本信息等组合得到具有可读性的中间语言表示。

根据此语法规则, 可将由源代码抽象出的行转义结果组合为以污染源为主语, 清洁函数为谓词, 敏感函数为宾语的 Alpherg 中间语言表示。谓词部分描述了对污染源的操作和执行关系, 包括字符串操作、检测过滤和流程控制语句等。宾语部分则用于说明可检测的漏洞类型, 包括 SQL 注入、跨站脚本和命令注入。

图 1 给出了 Alpherg 语法规则的形式化描述, Alpherg 以污染源为主语, 清洁函数为谓词, 敏感函数为宾语按第 1 行所示规则表示出源代码中的漏洞语义信息, 并用注释信息补充说明漏洞的基本信息。图 1 中第 2 行所示的敏感函数用于说明源代码中可能存在的漏洞类型, 目前 Alpherg 中仅考虑了 SQL 注入漏洞、跨站脚本攻击和命令注入漏洞三种类型, 后续若需增加可表示漏洞类型时, 按相同语法规则进行扩展即可。Alpherg 中清洁函数由清洁操作和流程控制语句两部分组成(见第 3 行), 在整个 Alpherg 表示结果中以谓词形式出现。第 4~7 行给出了清洁操作的语法结构及具体内容; 流程控制语句的语法规则由第 8 行给出。Alpherg 语法规则中注释信息包括文件名和代码起始行信息, 用于后续分析中定位漏洞, 在第 9~11 行规定了其语法构成及具体内容。

图 2 给出了一个 Alpherg 转义过程示例, 左侧为切片代码, 右侧为对应的行转义结果。第 1 行被识别为污染源并转义为 tainted_var; 第 2 行是分支语句的条件部分, 按照 Alpherg 的语法规则, 该行首先被转义为 cond, 但对于分支语句还需要根据条件内容对该转义进行补充说明, 示例中的条件内容为使用

filter_var 函数验证输入格式, 则第 2 行转义结果被补充为 cond format_v; 第 3 行是条件成立时的赋值语句, 本质是为污染变量取别名, 按照 Alpherg 语法规则统一被识别为 tainted_var, 不需要转义; 第 4~5 行是条件不成立时的赋值语句, 按规则首先被转义为 else, 但对于分支语句还需进行补充说明, 由于此语句用常量将污染变量重新赋值, 则第 4~5 行转义结果被补充为 else assign; 第 6 行是字符串拼接操作, 因此被转义为 addstr, 此操作中闭合污染变量 $tainted 的字符串长度为 1, 易被攻击, 该行转义结果被补充为 addstr_char0; 第 7 行是命令注入漏洞的敏感函数被转义为 sink_ci。

```
Alpherg语法设计 (BNF):
1.      IR ::= SINK PREDICATE+ “tainted_var” ANNOTATION
2.        SINK::= “sink_sql”|“sink_xss”|“sink_ci”
3.        PREDICATE::= {TERM|COND}
4.          TERM::= CONC|ADD_STR|“saint”|“replace_str”|
                    “comp”|“assign”|“whitelist”|“tc_num”|
                    “tc_str”|“ty_num”|“ty_str”|“format_v”
5.            CONC::= “conc”[“_f”|“_m”]“_” CHAR
6.            ADD_STR::= “addstr_” CHAR
7.            CHAR::= “char0”|“char_5”|“char_6”
8.          COND::= “cond” TERM+ [“do” TERM+]
                    [“else” TERM+] “end”
9.        ANNOTATION::= “//” SOURCE_ROW “and” SINK_ROW
10.         SOURCE_ROW::= filename row
11.         SINK_ROW::= filename row
```

图 1 Alpherg 的语法规则

Figure 1 Syntax rules of Alpherg

上述转义过程完成后, 得到的行转义结果会根据 Alpherg 的语法规则按执行顺序组合成代码切片的 Alpherg 表示"sink_ci addstr_char0 cond format_v else assign tainted_var"。若将污染源和敏感函数所在行及文件名等信息作为注释添加到上述中间表示的末尾, 则可得到完整的中间表示结果"sink_ci addstr_char0 cond format_v else assign tainted_var // tainted.php 5 and tainted.php 15"。

2.5 源代码的转义流程

从 Web 应用的源代码转义为疑似漏洞的 Alpherg 表示, 需先从源代码得到代码切片, 再从代码切片得到行转义结果, 最后将行转义结果组合成 Alpherg 中间语言表示。在此过程中, 主要执行的操作包括程序切片、行转义和 Alpherg 表示组合。

1) 程序切片

程序切片是通过跟踪污染源及其依赖关系, 遍历不同的可执行域, 并对源代码进行过程间上下文分析后得到的代码段, 算法 1 详细描述了此过程。因此程序切片中包含了从污染源开始到敏感点结束的所有与污染源相关的代码, 可将源代码中的漏洞信息完整保留, 但其中包含了不是 ASCⅡ的字符和注释, 需进一步清理。

2) 程序切片行转义及 Alpherg 表示组合

为了消除源代码中不同命名方式和语法差异带来的干扰, 保留源代码中的语句执行关系, 需对程序切片进行行转义处理, 并按照图 2 所示的语法规则将得到的行转义结果组合为以污染源为主语, 清洁函数为谓词, 敏感函数为宾语的 Alpherg 中间语言表示。

算法 2 给出了程序切片行转义算法描述, 其过程包括以下两个步骤:

步骤 1: 判断待分析语句中的变量是否存在于污染变量列表, 确定该语句是否需要行转义。若需要, 则根据语义将其转义为由 Alpherg 关键字表示的行转义结果, 在转义过程中如果出现依赖于污染源的新变量, 则将其更新到污染变量列表中; 否则跳过该语句按源代码流程分析下一条语句。这里引入的污染变量列表记录了所有被污染变量的信息, 用于跟踪污染源的传播路径。

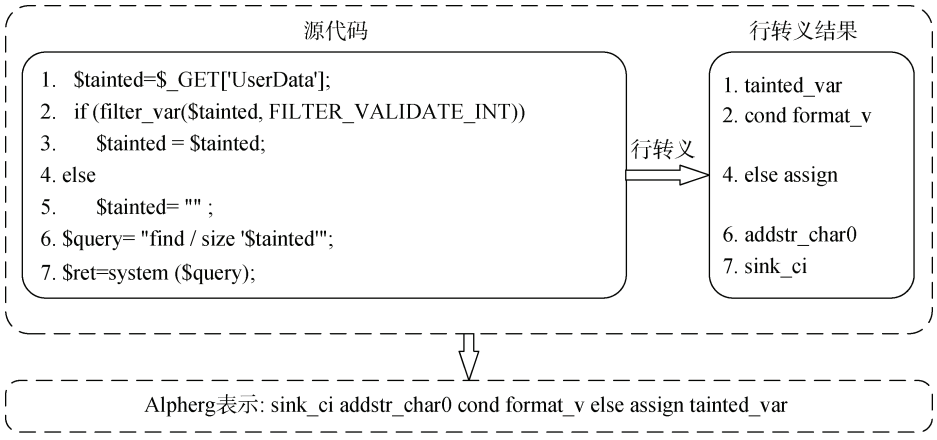

图 2 源代码转义示例

Figure 2 Example of source code escape

算法 1 程序切片算法

输入 源代码 *stmts*, 敏感函数集合 *sink*, 污染源集合 *source*

输出 疑似存在漏洞的代码切片

```
初始化 SLICES = ϕ
FOR 遍历 stmts 中的代码行
    IF 代码行调用的函数属于 sink THEN
        初始化 slice = ϕ
        FOR 从函数所在代码行向前遍历
            IF 代码行中变量定义属于 source THEN
                记录当前代码行位置
            END IF
        END FOR
        FOR 从当前代码行向后遍历, 直到敏感函数所在代码行
            IF 代码行操作或依赖污染源 THEN
                将代码行存入 slice
            END IF
        END FOR
    END IF
    存入 SLICES
END FOR
返回所有的代码切片 SLICES
```

步骤 2: 将行转义结果按照图 2 所示的语法规则进行组合, 得到 Alpherg 表示。若行转义结果对应的程序切片是类定义或自定义函数, 则将得到的 Alpherg 表示添加到已转义列表(详见算法 2 需注意的问题 3)中; 否则将 Alpherg 表示以字符串形式保存, 经向量化处理后添加到训练数据集, 用于后续漏洞检测模型的训练。

算法 2 需注意的问题:

1) 在分析分支语句时, 需将条件部分与执行部分视为一个整体进行分析, 将两部分的转义结果按照语法规则组合得到分支语句的 Alpherg 表示。

算法 2 程序切片行转义算法

输入 切片代码 *stmts*, 污染变量列表 *list*, 已转义列表 *c_IR* 和类与变量的映射表 *name_var*

输出 切片转义后的 Alpherg 表示

```
将源代码转换为 AST 并初始化 IR, stmts = AST(stmts); IR = ϕ
FOR 遍历 AST 中的代码语句 v
    IF 此语句是赋值语句 THEN
        提取语句左值变量名 var 和右值变量名 expr
        IF 此语句是调用类的语句 THEN
            调用类分析算法, 查询此语句的 Alpherg 表示
        END IF
        IF 右值变量属于污染变量列表 list THEN
            将左值变量 var 存入 list
        END IF
        IF var 或 expr 中的变量名属于 list THEN
            根据右值对此赋值语句转义
        END IF
    END IF
    IF 此语句是分支语句 THEN
        获取分支语句的 Alpherg 表示
    END IF
    IF 此语句是循环语句 THEN
        IF 循环条件中的变量依赖于污染源 THEN
            更新 list
        END IF
        获取循环语句的 Alpherg 表示
    END IF
    IF 此语句是方法调用 THEN
        提取方法名和其参数名 func_args
        IF func_args 属于 list THEN
            查询该方法的 Alpherg 表示
        END IF
    END IF
END FOR
对所有语句转义后的结果按照语法重新排列得到切片的 Alpherg 表示, 返回 Alpherg 表示
```

处理过程中, 条件部分首先被统一转义为关键字 cond, 在此基础上判断条件部分中的变量是否依赖污染源, 若依赖则对条件部分进行进一步分析, 为 cond 添加补充转义说明, 并将该变量更新至污染变量列表; 否则仅保留关键字 cond 用于表示分支语句的条件部分。而执行部分按正常转义处理即可。

2) 对循环语句转义时, 需将其视为顺序语句进

行处理。但因循环语句中的变量值会不断变化, 若将其直接视为顺序语句进行转义, 会导致变量信息缺失, 因此需先将该变量的取值存入污染变量列表, 再将循环语句视为顺序语句进行转义。

算法 3 类定义的转义算法

输入 切片代码 *stmts*, 已转义列表 *c_IR*

输出 更新后的 *c_IR*

1. 初始化疑似被污染属性 *attr*, 提取类名 *c_name*
2. FOR 遍历类定义语句
3. IF 此语句是类属性定义并有初始值 THEN
4. 提取此属性名 *name* 和初始值
5. IF 初始值是疑似污染源 THEN
6. 对此语句转义并得到结果 *IR*, 存入已转义列表, *c_IR*[*c_name*][*name*] = *IR*
7. 将此属性存入 *attr*
8. END IF
9. END IF
10. IF 此语句是方法定义 THEN
11. 提取方法名 *method* 与变量名 *arg_list*
12. 合并 *arg_list* 和 *attr* 作为污染变量列表 *list*
13. 对定义部分进行转义, *IR*=IR(*v*, l*ist*, *c_IR*, *name_var*)
14. IF $IR \neq \phi$ THEN
15. 将转义结果存入已转义列表, *c_IR*[*c_name*][*method*] = *IR*
16. END IF
17. END IF
18. END FOR
19. 返回已转义列表 *c_IR*

3) 对类定义和自定义函数转义时, 应提前将其转义结果存入已转义列表, 用于后续过程间转义。但由于类定义和自定义函数中可能不存在污染源, 不能转义, 导致信息丢失。为解决此问题, 算法 3 将类成员和函数参数均视为污染源, 并以此作为分析的起点进行转义。需要说明的是, 尽管算法 3 只给出了类定义的处理过程, 但对于独立的自定义函数可将其视为同名类定义, 按该算法进行处理。

为实现过程间分析, 设计了已转义列表。如算法 2 和算法 3 中所述, 在切片转义自定义函数时, 会先对函数定义分析并转义, 转义过程中将函数参数视为被污染变量, 并将此函数转义后的结果保存到已转义列表中, 如果转义结果为空则不记入已转义列表, 该自定义函数与漏洞无关。当程序切片转义分析到自定义函数调用时, 则按算法 2 中第 5~7 行和第 24~29 行给出的方法查询已转义列表, 用查询结果替换自定义函数调用的转义结果, 实现了过程间漏洞信息的传递。

图 3 给出了从源代码到 Alpherg 表示的完整转义过程, 包括程序切片、行转义和 Alpherg 表示组合三个主要处理环节。在转义过程中, 先执行算法 1, 跟踪示例中的污染源$_POST["id"]及其依赖关系, 遍历污染源的可执行域, 并对源代码进行过程间上下文分析, 直至到达敏感函数 mysql_query(),

代码 5 源代码过程分析示例

```
1.  function sanitize($var_in){
2.                          return htmlspecial-
chars($var_in,ENT_QUOTES);
3.  }
4.  function gets($tainted){
5.     return $tainted;
6.  }
7.  $tainted = $_GET["t"];
8.  $tainted = gets($tainted);
9.  $dataflow = sanitize($tainted);
10. $context = ("Hello" . $dataflow);
11. echo($context);
```

记录污染源到敏感函数间的全部有效源代码(源代码中的第 5 和第 10 行在区间内, 但为无效代码), 得到程序切片; 再执行算法 2, 对程序切片进行行转义, 并依据图 1 所示的语法规则将行转义结果组合得到最终的 Alpherg 中间语言表示。

3 基于 Alpherg 的漏洞检测模型

本文提出了一种基于 Bi-LSTM 和注意力机制的 PHP 注入漏洞检测模型(模型结构及训练使用过程见图 4)。该模型利用 Bi-LSTM 处理长期依赖关系的能力, 可有效处理由 Alpherg 中间语言表示的漏洞特征序列中的长期依赖关系, 得到 Alpherg 长序列表示中的上下文关系; 进一步在该模型中加入注意力机制, 通过计算每个时间步的注意力分布, 及计算结果的加权平均, 将得到的加权向量作为模型的最终表示, 经分类器分类得到漏洞检测结果。由 Bi-LSTM 和注意力机制构成的注入漏洞检测模型可更好地利用 Alpherg 表示中保留的源代码上下文关系及漏洞相关信息, 提高了模型的漏洞检测能力。

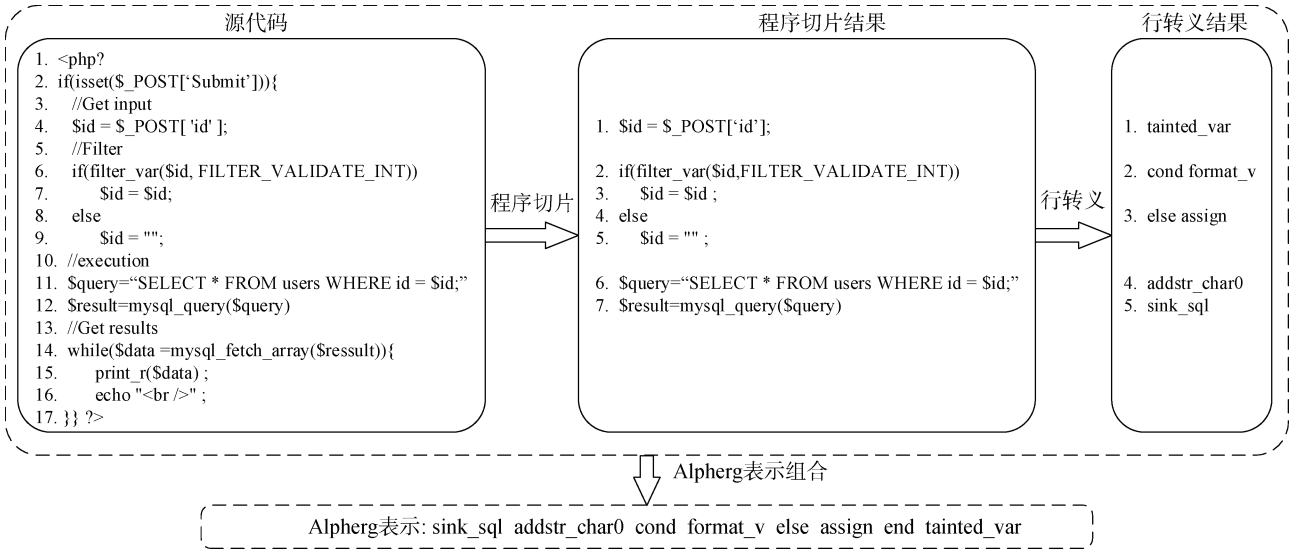

图 3　源代码转义过程

Figure 3　The procedure of source code escape

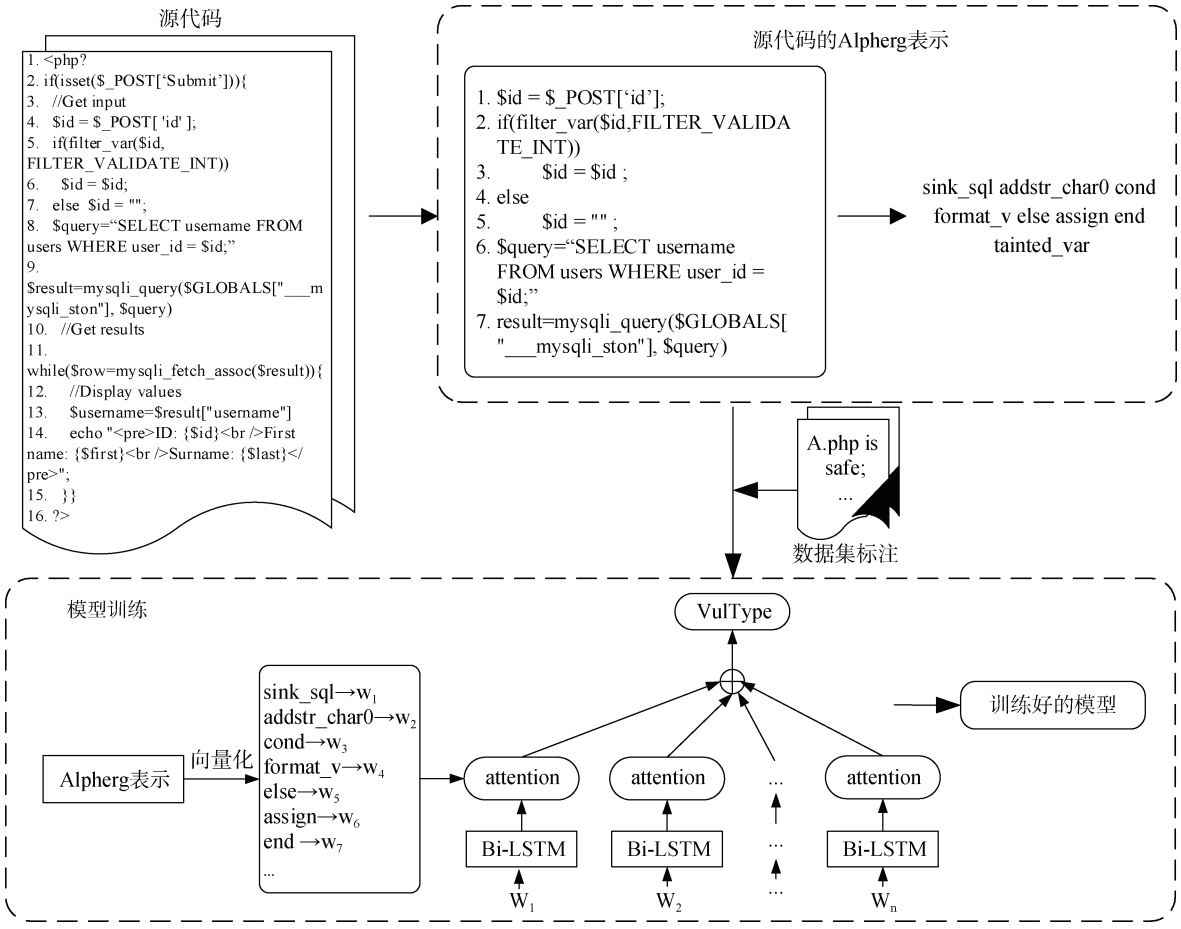

图 4　模型训练过程

Figure 4　The procedure of model training

1) 源代码转义

算法 4 给出了源代码转义环节中对源代码进行程序切片和转义的详细描述。为实现过程间分析, 首先需对源代码中的类定义和自定义函数按算法 2、3 所述进行转义处理(写入或不写入已转义列表)。当程序切片转义分析出类定义或自定义函数调用时, 按算法 2 给出的方法查询已转义列表, 并用查询结果替换类定义或自定义函数对应的转义结果, 实现过

程间漏洞信息的传递。然后按算法 1 所述, 跟踪污染源及其依赖关系, 遍历不同的可执行域, 得到包含从污染源开始到敏感函数结束的所有源代码, 得到可保留完整漏洞信息的程序切片, 经字符处理去除所包含的非 ASCⅡ码字符和注释信息, 得到程序切片的行转义序列。最后按图 1 所示的语法规则将行转义序列组合得到最终的 Alpherg 中间语言表示。

算法 4 源代码切片及转义算法

输入 源代码 *stmts*

输出 源代码转义结果 *IR_ALL*

1. 定义转义结果 *IR_ALL*, 已转义列表 *c_IR*
2. FOR 遍历源代码中所有的类定义
3. 按照算法 3 对类定义进行转义并存入 *c_IR*
4. END FOR
5. FOR 遍历源代码中所有的自定义函数
6. 对函数定义进行转义并将结果存入 *c_IR*
7. END FOR
8. 按照算法 2 对源代码进行切片 *SLICES*
9. FOR 遍历 *SLICES*
10. 去除切片内的注释和不是ASCⅡ的字符
11. 按照算法 3 将切片转义为 Alpherg
12. 提取切片的开始行和结束行, 即污染源和敏感函数所在行
13. 提取所在文件名
14. 按照 Alpherg 的语法规则将行信息和文件名添加到 *IR*
15. 将 *IR* 存入 *IR_ALL*
16. END FOR
17. 返回 *IR_ALL*

算法 4 中需注意以下三点:

a) 对类定义进行转义时, 给出了不同类的转义顺序规则。若类间存在依赖关系, 则按依赖关系的逆序进行类转义, 并将类定义的转义结果存入已转义列表。

b) 程序切片的转义结果不直接包含漏洞的基本信息。这些信息需在转义结束后, 通过将切片内的污染源和敏感函数等信息按图 1 所示的语法规则, 添加到 Alpherg 表示中而得到。

c) 算法 4 会对源代码中所有疑似漏洞进行转义得到对应的 Alpherg 表示。因此最终的 Alpherg 表示中会因记录了疑似漏洞信息, 存在假负类漏洞的可能, 但本文未对此问题进行深入分析。

2) 数据集标注

本文所用训练数据来源于 SARD[22](Software Assurance Reference Dataset)。该数据集是一组用于软件安全检测的漏洞数据, 其中包含了真实的软件应用程序源代码及已知的漏洞代码, 在 PHP Web 应用漏洞检测研究中被广泛使用[23]。

本文从 SARD 中提取了全部命令注入漏洞、跨站脚本漏洞和 SQL 注入漏洞的源代码数据(在 SARD 中分别对应标签 CWE-78、CWE-79 和 CWE-89), 并对数据集中的错误进行了修正。如将 PHP 测试用例中存在的未定义常量“checked_data”替换为正确变量“$tainted”; 将错误拼写的变量名“tained”修正为“tainted”; 将数组中的错误使用的赋值符号“=>”修正为“=>”; 并修正了大量引号使用错误。将这些修正后的源代码数据转义后, 在对应的 Alpherg 表示中添加对应标签得到本文所用的训练数据集(该数据集包括的各类漏洞数量如表 2 所示)。

表 2 数据集中各类型漏洞的数量

Table 2 The number of different vulnerabilities in the dataset

漏洞名称	存在漏洞/个	安全/个	汇总/个
SQLi	912	2640	3552
XSS	4352	5728	10080
CI	624	1755	2379
汇总	5888	10123	16011

3) 数据预处理及模型训练

按网络模型输入要求, 使用 word2vector 将标注后以字符串形式存储的 Alpherg 中间表示向量化为词向量 binData, 并对 binData 进行统一长度调整, 若 binData 长度小于规定的阈值 w(本文中 w=200), 则在 binData 后进行补 0 操作; 若 binData 长度大于阈值 w, 则按 w 的取值进行截断操作。最后将处理后的词向量 binData 存入训练数据集, 用于对模型训练得到漏洞检测模型, 输入模型完成模型的训练。

4 实验和结果分析

本文将通过实验回答以下 3 个问题。

问题 1: Alpherg 与其他表示方式相比是否能更好地表示漏洞。

问题 2: 所提出的漏洞检测模型是否能有效识别多种 PHP 注入漏洞。

问题 3: 与现有工具/模型相比, 本文所提出的漏洞检测模型是否具有更好的检测效果。

4.1 实验环境和评价指标

本文使用表 2 中描述的数据集, 在 CPU 处理器为英特尔 Xeon Silver 4210R, 主频为 2.40Ghz, 内存为 512GB, GPU 为 NVIDIA TITAN RTX 3090, 显存

为 24GB 的设备上完成实验。

本文所提出的漏洞检测模型解决了一个多分类问题, 用于检测多种漏洞。在性能评价时, 先分别计算每个漏洞的对应指标值, 然后计算全部类别对应指标的平均值, 作为最终性能评价依据。以第 i 类漏洞的检测为例, 具体评价指标确定如下:

真正类 TP_i: 样本是 i 类漏洞, 模型预测结果也是 i 类漏洞。

假负类 FN_i: 样本是 i 类漏洞, 模型预测结果不是 i 类漏洞。

假正类 FP_i: 样本不是 i 类漏洞, 模型预测结果是 i 类漏洞。

真负类 TN_i: 样本不是 i 类漏洞, 模型预测结果也不是 i 类漏洞。

误报率(*FPR*): 不是漏洞的样本被预测成漏洞的占比, 如公式(1)所示。

$$FPR=\sum_{i}^{n}\frac{\frac{FP_i}{FP_i+TN_i}}{n} \tag{1}$$

召回率(*TPR*): 真实存在漏洞的样本被判定为存在漏洞的占比, 如公式(2)所示。

$$TPR=\sum_{i}^{n}\frac{\frac{TP_i}{TP_i+FN_i}}{n} \tag{2}$$

精确度(*P*): 真实存在漏洞的样本被预测为漏洞的比例, 如公式(3)所示。

$$P=\sum_{i}^{n}\frac{\frac{TP_i}{TP_i+FP_i}}{n} \tag{3}$$

F-Measure (*F*1): 精确度和召回率的调和平均, 如公式(4)所示。

$$F1=\sum_{i}^{n}\frac{\frac{2P_iTPR_i}{FP_i+TN_i}}{n} \tag{4}$$

精度(*Acc*): 判为漏洞的样本数量占样本总数量的比例, 如公式(5)所示。

$$Acc=\sum_{i}^{n}\frac{\frac{TP_i+TN_i}{TP_i+TN_i+FP_i+FN_i}}{n} \tag{5}$$

4.2 实验分析

1) 回答问题 1: Alpherg 与其他表示方式相比是否能更好地表示漏洞。

为了说明使用 Alpherg 提取特征的有效性, 本文将 Alpherg 表示与不同的中间表示方法进行比较, 证明 Alpherg 表示可更好地表示漏洞。本文选取的中间表示包括 PHP 源代码文本, AST 表示[24], PHP 操作码[21]和 PHP token[25]。PHP 源代码文本是将源代码中无意义的标记符号去除后得到的结果; AST 是将源代码信息用抽象语法树表示, 并去除源代码中的特殊标记符后得到的; 操作码是 PHP 语言里 zend 引擎执行的中间代码, 可用作漏洞特征的中间表示; PHP token 是利用 PHP 内置函数 token_get_all, 将源代码按 PHP 标记进行分割后得到的 PHP token 序列, 用作源代码的漏洞特征表示。代码 6 给出了 PHP 的 SQLi 漏洞源代码样例。本文采用上述方法对代码 6 分别进行了表示, 通过实验表明所提出的 Alpherg 中间语言表示与其他方法相比能更好地表示源代码中的漏洞信息。

代码 6 中第 12~18 行为与漏洞无关信息, 且变量未做归一化表示, 直接用于漏洞检测时效果不佳。同样, 在使用 AST 进行漏洞检测时与直接使用源代码面临相同的问题。

代码 6 SQLi 漏洞原始代码样例

```
1.  <?php
2.  $tainted = $_GET["UserData"];
3.  if(filter_var($tainted, FILTER_VALIDATE_INT))
4.  $tainted = $tainted;
5.  else
6.  $tainted = "";
7.  $query=sprintf(“SELECT username FROM users
    WHERE user_id = %s”,$tainted);
8.  // Connection to the database
9.  $conn = mysql_connect('localhost', 'mysql_user',
    mysql_password');
10. //execution
11. $result=mysql_query($query);
12. while($row=mysql_fetch_assoc($result)){
13.       //Display values
14.       $username=$result["username"];
15.       echo "<pre>ID: {$id}<br />First name: {$fir
              st}<br />Surname: {$last}</pre>";
16. }
17. mysql_close($conn);
18. ?>
```

代码 7 样例的 Alpherg 表示

```
1.  tainted_var
2.  cond   format_v
3.  else
4.        assign end
5.  addstr_char0
6.  sink_sql
```

代码 7 给出了样例代码的 Alpherg 行转义结果, 其中的序号代表源代码样例中的代码行号。将行转义结果按照语法规则组合得到样例的 Alpherg 表示“ sink_sql addstr_char0 cond else assign end tainted_var”。该结果中, 漏洞信息被抽象为由漏洞类型、对污染源进行的操作和污染源组成的三元组, 丢弃了与漏洞无关的噪声数据, 保留了源代码中漏洞的上下文信息, 在形式上脱离了原有编程语言, 且具有可读性。

代码 8 是样例转义为 PHP 操作码的结果(限于篇幅仅对样例中前 11 行做了处理), 其中的序号同样代表源代码样例中的代码行号。将源代码转义为操作码可归一化表示自定义变量和方法, 简化代码表示。但 PHP 操作码表示函数时使用相同的标识符, 无法根据操作码判断函数具体类型。且无法表示变量内的具体数据, 因此无法跟踪污染源的执行路径。因此在利用操作码进行漏洞特征提取时, 需人工添加缺失的部分漏洞信息, 因操作码可读性差, 无法彻底避免漏洞特征提取过程中的信息缺失。

代码 8 样例的 PHP 操作码表示

```
1. FETCH_R
   FETCH_DIM_R
   ASSIGN
2. INIT_FCALL
       SEND_VAR
       SEND_VAL
       DO_ICALL
       JMPZ
3. ASSIGN
       JMP
4. ASSIGN
5. INIT_FCALL
       SEND_VAL
       SEND_VAR
       DO_ICALL
       ASSIGN
6. INIT_FCALL_BY_NAME
       SEND_VAL_EX
       SEND_VAL_EX
       SEND_VAL_EX
       DO_FCALL
7. INIT_FCALL_BY_NAME
       SEND_VAR_EX
       DO_FCALL
       ASSIGN
```

代码 9 是利用 PHP token 对样例特征提取的结果(限于篇幅只转义了前 11 行代码, 并在结果中去除了表示空格的标识 T_WHITESPACE、标签的标识 T_OPEN_TAG 和注释部分), 其中的序号为源代码样例中的代码行号。在转义过程中, PHP token 将函数名统一转义为 T_STRING, 转义结果无法定位疑似漏洞代码段。Fang 等人[25]对 PHP token 进行了重新设计, 保留了源代码中的函数名, 设计了函数名列表, 用来标记相似函数。虽然避免了函数名统一转义引发的问题, 但转义结果中函数参数会被舍弃, 对于部分根据参数不同而具有不同功能的函数依然会造成漏洞信息缺失, 降低漏洞检测准确率。

代码 9 样例的 PHP token 表示

```
1. T_OPEN_TAG
2. T_VARIABLE
     T_OR_EQUAL
     T_VARIABLE
     T_CONSTANT_ENCAPSED_STRING
3. T_IF
     T_STRING
     T_VARIABLE
     T_STRING
4. T_VARIABLE
5. T_ELSE
6. T_VARIBLE
     T_OR_EQUAL
     T_ CONSTANT_ENCAPSED_STRING
7. T_VARIABLE
      T_OR_EQUAL
      T_STRING
      T_CONSTANT_ENCAPSED_STRING
      T_VARIABLE
8. T_VARIABLE
      T_OR_EQUAL
      T_STRING
      T_STRING
      T_STRING
      T_STRING
9. T_VARIABLE
      T_OR_EQUAL
      T_STRING
      T_VARIABLE
```

综上, 本文设计的 Alpherg 中间语言可正确表示源代码中漏洞的原始语义, 清晰地描述了漏洞语义特征, 去除了无用的噪声数据, 仅保留了必要的漏洞信息, 与其他方法相比, 可更好地描述漏洞。

为进一步说明 Alpherg 相对于其他表示方式的优势, 本文比较了从源代码转义为各种表示方式所需时间。该实验使用多组 PHP 源代码文件, 比较了转义为 AST、Opcode、PHP token 及 Alpherg 表示所

需时间(如表 3 所示)。实验结果显示, 将源代码转义为 Alpherg 表示所需时间仅次于转义为 AST 所需时间, 表明 Alpherg 中间语言可以在相对较短的时间内获得更好的漏洞信息抽象表示。

表 3　中间语言转义时间比较结果

Table 3　Results of intermediate language escape time comparison

文件数	Alpherg(s)	AST(s)	Opcode(s)	PHP token(s)
1	0.5227	0.5139	0.6682	0.5318
2	1.0525	1.0283	1.2871	1.0543
4	2.0999	2.0558	2.5528	2.1036
8	4.2322	4.1454	5.2129	4.2381

2) 回答问题 2: 所提出的漏洞检测模型是否能有效识别多种 PHP 注入漏洞。

表 4 给出了按照一定比例将漏洞数据拆分为训练集、验证集和测试集的不同方案。虽然方案中各类漏洞数量分布不均, 且不同漏洞表示的实现语句存在明显不同, 得到的 Alpherg 表示存在明显的差别, 但 Alpherg 中间语言仍能将漏洞的上下文语义信息准确表示出来, 不会在模型识别结果中出现假负类问题。根据图 5 可知, 在训练过程中随着训练迭代次数的增加, 训练集和验证集的损失值都在逐步下降, 准确率在逐步上升, 最终达到平稳态。本文对各个分类结果进行了整理, 得到了如表 5 所示的混淆矩阵, 表明基于 Alpherg 的漏洞检测模型可正确地对漏洞进行分类, 不会将某一类漏洞错误地识别为其他类型漏洞, 仅会将其误判为安全类型。综上所述, 基于 Alpherg 中间语言表示的漏洞检测模型可实现对多种 Web 漏洞的检测。

表 4　用于训练和验证的数据集分布

Table 4　Distribution of data sets used for training and validation

数据集	SQL 注入		跨站脚本攻击		命令注入	
	漏洞/个	安全/个	漏洞/个	安全/个	漏洞/个	安全/个
训练集	638	1900	3046	4009	436	1228
验证集	91	211	435	574	62	175
测试集	183	529	871	1145	126	352
汇总	912	2640	4352	5728	624	1755

3) 回答问题 3: 与现有工具/模型相比, 本文所提出的漏洞检测模型是否具有更好的检测效果。

为了验证所提出的漏洞检测模型的有效性, 本文分别将其与以下几种漏洞检测工具/模型进行了比较。(1) RIPS[26], 是使用静态分析技术, 用 PHP 编写的源代码分析工具, 可自动挖掘 PHP 源代码中潜在的安全漏洞; (2) Web Application Protection(WAP)[27], 是用于检测 PHP Web 应用程序漏洞的源代码静态分析和数据挖掘工具; (3) progpilot[28], 是一款利用控制流程图和抽象语法树的 PHP 漏洞检测工具; (4) TAP[25], 是基于 PHP token 和深度学习技术的 PHP 漏洞检测模型。以上工具/模型均可用本文处理得到的数据集, 实现 PHP 源代码漏洞检测。

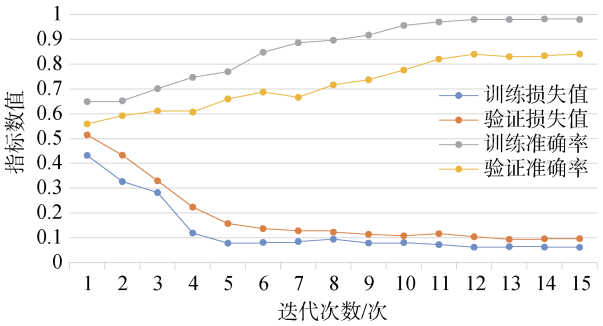

图 5　模型不同指标的变化曲线

Figure 5　Change curves of different indicators of the model

表 5　分类模型的混淆矩阵

Table 5　Confusion matrix of classification model

	CWE-78	CWE-79	CWE-89	SAFE
CWE-78	122	0	0	4
CWE-79	0	845	0	26
CWE-89	0	0	177	6
SAFE	6	14	3	2003

表 6 列出了基于 Alpherg 的漏洞检测模型及上述工具/模型在 PHP 源代码漏洞检测实验中的评价指标, 结果显示, 本文所提出的漏洞检测模型与结果较好的 TAP 模型相比, 各项指标均有提升, 准确率达 98%。主要原因为 TAP 是基于 PHP token 设计的, 需人为补充漏洞信息, 存在漏洞语义信息缺失的情况, 导致检测准确率下降。而本文设计的 Alpherg 不但可更好地表示漏洞语义信息, 还可去除与漏洞信息无关的噪声数据。

表 6　测试结果

Table 6　Test results

检测工具	*Acc*(%)	*P*(%)	*TPR*(%)	*FPR*(%)	*F*1(%)
本文模型	98.15	98.02	96.94	1.79	97.54
TAP	93.85	92.50	83.70	2.46	87.88
WAP	65.70	28.43	19.02	17.36	22.79
RIPS	69.23	33.33	15.62	11.33	21.27
progpilot	41.30	22.27	48.39	61.27	30.50

为了说明本文所提模型的实用性, 选取了 GitHub 开源项目 sqli-lab[29]和 DVWA[30], 人工审查了

项目源代码并标注了 83 个注入漏洞, 用于测试不同漏洞检测工具的准确性和实用性。表 7 所示实验结果表明本文模型在检测项目漏洞时, 精确度为 83.72%, 召回率为 86.75%。相比于 RIPS、WAP、Progpilot 和 TAP, 本文模型在此项目上的漏洞检测表现更优, 在漏洞检测中具有更好的准确率和实用性。

表 7 开源项目检测结果

Table 7 Test results of open source project

检测工具	*TP*	*FP*	*FN*	*P*(%)	*TPR*(%)
本文模型	72	14	11	83.72	86.75
TAP	66	16	17	80.49	79.52
WAP	36	69	47	34.29	34.29
RIPS	54	69	29	43.91	65.06
progpilot	46	66	37	41.07	55.42

5 结语

本文针对漏洞检测工具所提取的漏洞信息中会缺失部分与漏洞相关的语义信息, 且包含大量与漏洞信息无关的噪声数据, 导致漏洞检测存在误报和漏报的问题, 提出了一种名为 Alpherg 的中间语言表示, 此语言可保留 PHP 源代码中的代码信息、提取源代码中仅与漏洞相关的语义信息, 并能表示源代码的控制流信息, 可精确提取与漏洞存在直接关系的信息, 避免引入过多噪声, 并保留了漏洞的语义信息。利用 Alpherg 进行漏洞特征提取, 提出了一种基于 Bi-LSTM 和注意力机制的 PHP 注入漏洞检测模型, 与现有的检测工具/模型相比, 可有效检测 PHP 源代码中的注入漏洞。

在未来的工作中, 一方面可以继续完善 Alpherg 的设计使其适用于多编程语言的中间表示; 另一方面可在 Alpherg 中间表示的基础上结合其他神经网络模型, 设计面向多语言的源代码审计或漏洞检测系统, 帮助开发人员更快定位并修复漏洞, 提高 Web 应用的安全性。

参考文献

[1] Vulnerability Distribution Statistics. National Information Security Vulnerability Sharing Platform. https://www.cnvd.org.cn/flaw/ statistic. 2020.
(漏洞分布统计数据. 国家信息安全漏洞共享平台. https://www.cnvd.org.cn/flaw/statistic. 2020.)

[2] Usage Statistics of Server-side Programming Languages for Websites. W3Techs. https://w3techs.com/technologies/overview/ programming_language. October 2022.

[3] Ma S Q, Thung F, Lo D, et al. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017: 229-246.

[4] Giuffrida C, Bardin S, Blanc G. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript[C]. *DIMVA 2018: Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018: 303-325.

[5] Li Z, Zou D Q, Xu S H, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[EB/OL]. 2018: 1801.01681. https://arxiv.org/abs/1801.01681v1.

[6] Rabheru R, Hanif H, Maffeis S. DeepTective: Detection of PHP Vulnerabilities Using Hybrid Graph Neural Networks[C]. *The 36th Annual ACM Symposium on Applied Computing*, 2021: 1687-1690.

[7] Liu M Y, Li K, Chen T. DeepSQLi: Deep Semantic Learning for Testing SQL Injection[C]. *The 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020: 286-297.

[8] Aliero M S, Ghani I, Qureshi K N, et al. An Algorithm for Detecting SQL Injection Vulnerability Using Black-Box Testing[J]. *Journal of Ambient Intelligence and Humanized Computing*, 2020, 11(1): 249-266.

[9] Shuai B, Li H F, Zhang L, et al. Software Vulnerability Detection Based on Code Coverage and Test Cost[C]. *2015 11th International Conference on Computational Intelligence and Security*, 2015: 317-321.

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

[11] Son S, Shmatikov V. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications[C]. *The ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011: 1-13.

[12] Wasef A, Fitri Z, Khalid H. Web Security: Detection of Cross Site Scripting in PHP Web Application Using Genetic Algorithm[J]. *International Journal of Advanced Computer Science and Applications*, 2017, 8(5): 64-75.

[13] Nguyen T T, Maleehuan P, Aoki T, et al. Reducing False Positives of Static Analysis for SEI CERT C Coding Standard[C]. *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice*, 2019: 41-48.

[14] Catal C, Akbulut A, Ekenoglu E, et al. Development of a Software Vulnerability Prediction Web Service Based on Artificial Neural Networks[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017: 59-67.

[15] Guo N, Li X Y, Yin H, et al. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020: 199-218.

[16] 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.

[17] Jahanshahi R, Doupé A, Egele M. You shall Not Pass: Mitigating SQL Injection Attacks on Legacy Web Applications[C]. *The 15th ACM Asia Conference on Computer and Communications Security*, 2020: 445-457.

[18] Medeiros I, Neves N, Correia M. Detecting and Removing Web

Application Vulnerabilities with Static Analysis and Data Mining[J]. *IEEE Transactions on Reliability*, 2016, 65(1): 54-69.

[19] Zheng W, Gao J L, Wu X X, et al. An Empirical Study of High-Impact Factors for Machine Learning-Based Vulnerability Detection[C]. *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing*, 2020: 26-34.

[20] Fidalgo A, Medeiros I, Antunes P, et al. Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants[C]. *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2020: 465-476.

[21] Li C H, Wang Y D, Miao C W, et al. Cross-Site Scripting Guardian: A Static XSS Detector Based on Data Stream Input-Output Association Mining[J]. *Applied Sciences*, 2020, 10(14): 4740.

[22] B. Stivalet. PHP Vulnerability Test Suite[DS]. https://samate.nist.gov/SARD/view.php?tsID=103.

[23] Kronjee J, Hommersom A, Vranken H. Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning[C]. *The 13th International Conference on Availability, Reliability and Security*, 2018: 1-10.

[24] Kurniawan A, Abbas B S, Trisetyarso A, et al. Static Taint Analysis Traversal with Object Oriented Component for Web File Injection Vulnerability Pattern Detection[J]. *Procedia Computer Science*, 2018, 135: 596-605.

[25] Fang Y, Han S J, Huang C, et al. TAP: A Static Analysis Model for PHP Vulnerabilities Based on Token and Deep Learning Technology[J]. *PLoS ONE*, 2019, 14(11): e0225196.

[26] 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: 23-26.

[27] Medeiros I, Neves N F, Correia M. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives[C]. *The 23rd international conference on World wide web*, 2014: 63-74.

[28] designsecurity. Progpilot. https://github.com/designsecurity/progpilot.

[29] Audi-1. Sqli-labs. https://github.com/Audi-1/sqli-labs.

[30] Digininja. DVWA. https://github.com/digininja/DVWA.

张国栋 于 2003 年在华南理工大学机械设计及理论专业获工学硕士学位。现任沈阳航空航天大学计算机学院教授。研究领域为模式识别与智能系统、网络与信息安全。研究兴趣包括: 漏洞检测、医学影像理解与分析。Email: zhanggd@sau.edu.cn

刘子龙 于 2020 年在沈阳航空航天大学软件工程专业获工学学士学位。2023 年在沈阳航空航天大学计算机科学与技术专业获工学硕士学位。研究领域为网络与信息安全。研究兴趣包括: Web 安全、人工智能。Email: 2278117063@qq.com

姚天宇 于 2022 年在沈阳航空航天大学计算机科学与技术专业获学士学位。现在沈阳航空航天大学电子信息专业攻读硕士学位。研究领域为网络与信息安全。研究兴趣包括: Web 安全、人工智能。Email: 1401856303@qq.com

靳卓 于 2021 年在沈阳航空航天大学物联网工程专业获工学学士学位。2023 年在沈阳航空航天大学电子信息专业获工学硕士学位。研究领域为网络与信息安全。研究兴趣包括: 渗透技术、代码审计。Email: 1239581167@qq.com

孙东红 于 2010 年在北京理工大学计算机科学与技术专业获工学博士学位。现在清华大学网络科学与网络空间研究院任副研究员。研究领域为网络安全。研究兴趣包括: 大数据分析、智能计算。Email: sundonghong@tsinghua.edu.cn

秦佳伟 于 2021 年在北京邮电大学计算机科学与技术专业获博士学位。现在国家计算机应急技术处理协调中心任工程师。研究领域为网络与信息安全。研究兴趣包括: Web 安全、移动端及物联网安全分析。本文通讯作者。Email: qinjiawei@cert.org.cn