基于 Bi-LSTM 模型的恶意 JavaScript 代码检测方法

纪育青[1] 方艳红[1,2*] 谭顺华[1] 王学渊[1,2]

[1](西南科技大学信息工程学院 四川 绵阳 621010)

[2](西南科技大学特殊环境机器人技术四川省重点实验室 四川 绵阳 621010)

摘 要 传统的静态检测恶意 JavaScript 代码方法十分依赖于已有的恶意代码特征,无法有效提取混淆恶意代码特征,导致检测混淆恶意 JavaScript 代码的精确率低。针对该问题提出基于双向长短期记忆网络(Bidirectional Long Short-term Memory,Bi-LSTM)的恶意代码检测模型。通过抽象语法树将 JavaScript 代码转化为句法单元序列,通过 Doc2Vec 算法将句法单元序列用分布式向量表示,将句向量矩阵送入 Bi-LSTM 模型进行检测。实验结果表明,该方法对于混淆恶意 JavaScript 代码具有良好的检测效果且检测效率高,准确率为 97.03%,召回率为 97.10%。

关键词 恶意 JavaScript 代码检测 Bi-LSTM 深度学习 Doc2Vec

中图分类号 TP399 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2024.09.049

MALICIOUS JAVASCRIPT CODE DETECTION METHOD BASED ON BI-LSTM MODEL

Ji Yuqing[1] Fang Yanhong[1,2*] Tan Shunhua[1] Wang Xueyuan[1,2]

[1](*School of Information Engineering, Southwest University of Science and Technology, Mianyang 621010, Sichuan, China*)

[2](*Key Laboratory of Robot Technology Used for Special Environment of Sichuan Province, Southwest University of Science and Technology, Mianyang 621010, Sichuan, China*)

Abstract The traditional static detection methods of malicious JavaScript code rely heavily on existing malicious code features, which can't effectively extract the obfuscated malicious code feature, resulting in low accuracy of detecting obfuscated malicious JavaScript code. To solve this problem, a malicious code detection model based on bidirectional long short-term memory (Bi-LSTM) is proposed. This method transformed JavaScript code into syntactic unit sequence through abstract syntax tree, and used the Doc2Vec algorithm to represent the syntactic unit sequence with distributed vectors. The sentence vector matrix was sent to the Bi-LSTM model for detection. The experimental results show that this method has good detection effect and high detection efficiency for obfuscated malicious JavaScript code, with the accuracy rate of 97.03 % and the recall rate of 97.10 %.

Keywords Malicious JavaScript code detection Bi-LSTM Deep learning Doc2Vec

0 引 言

随着互联网的发展与普及,人们越来越感受到互联网带来的便利,其中就包括大量的 Web 服务以及各种服务应用。现在 Web 前端技术飞速发展,许多 Web 服务以及各类服务应用已经越来越多采用 JavaScript 语言进行开发,大大提高了人机的交互能力,极大地方便人们的生活。然而由于 JavaScript 语言的动态性,攻击者很容易利用网站漏洞,在网站中嵌入恶意代码以窃取用户信息,对用户的信息安全造成了极大的困扰。同时攻击者还会利用各种混淆技术如随机混淆、压缩混淆等技术[1]对恶意的 JavaScript 代码进行混淆以防止传统的依赖特征提取的静态检测技术的检测,这导

收稿日期:2021-03-03。国家自然科学基金项目(61771411)。**纪育青**,硕士生,主研领域:计算机网络,信息安全。**方艳红**,副教授。**谭顺华**,副教授。**王学渊**,副教授。

致检测的难度大大增加。

近几年来,混淆恶意 JavaScript 代码检测得到了广泛的研究。徐青等[2]通过分析恶意 JavaScript 代码,将恶意代码特征类型分为基于统计信息、基于 URL 重定向、基于攻击过程、基于代码混淆等四类特征类型,并通过分类算法构建检测模型。马洪亮等[3]先判断是否为混淆代码,若为混淆代码则通过 AST 进行代码反混淆,之后再进行恶意代码的检测。该方法不仅十分耗时,且只能针对固定几种混淆方法进行反混淆,不适合进行大规模的检测。Fass 等[4]提出了 JaST 模型,通过 AST 提取相应词法单元,并结合 N-gram 进行频率分析。该模型对恶意代码有较高的识别率,但是对于一些恶意代码片段只占很小比例的 JavaScript 样本的检测效果并不理想。Liang 等[5]分别使用基于树的卷积神经网络和基于图的神经网络从 JavaScript 代码的抽象语法树和控制流图提取 JavaScript 的语法特征和语义特征,但是该方法在合并特征时会产生较大误差。邱瑶瑶等[6]将 JavaScript 代码转换成抽象语法树,结合 Word2Vec 算法[7]生成词向量,并将词向量送入深度学习模型进行检测。该方法对于混淆的恶意 JavaScript 文件有较高的准确率和召回率,但该方法存在对大型 JavaScript 文件解析时损失大量信息的问题,只能检测较小的 JavaScript 文件。

一般来说,静态检测方法是基于代码文本特征匹配的检测方法,该方法占用的系统资源少,检测效率高,但是特别依赖复杂的特征工程和分类算法,如文献[8-10]。动态检测方法是依据代码执行结果来判别是否为恶意代码的检测方法,该方法的检测准确度高,但需要消耗更多资源且对于检测环境有更高要求,如文献[11-13]。现阶段,无法有效地在混淆的恶意 JavaScript 代码文本上提取恶意特征,因此静态检测方法在检测混淆恶意代码上表现不佳,但是混淆前后的恶意代码的攻击逻辑即语义信息并没有发生根本性的改变。

故为了从 JavaScript 代码获得语义信息,本文通过 AST 将 JavaScript 代码转换为句法单元序列,该序列保留丰富的语义信息且容易通过 Doc2Vec 算法进行句向量表征。然后在此基础上结合 Bi-LSTM 网络[14]构建恶意 JavaScript 代码检测模型。

1 检测方法

本文提出的检测方法的整体流程图如图 1 所示。首先将 JavaScript 样本通过词法分析和语法分析映射成为抽象语法树,然后遍历抽象语法树节点记录节点信息和代码模块调用顺序,接着根据代码模块调用顺序对节点信息进行分词得到句法单元序列,最后通过 Doc2Vec 算法将句法单元序列转化为句向量表示并输入到 Bi-LSTM 模型中进行分析,检测是否为恶意 JavaScript 代码。

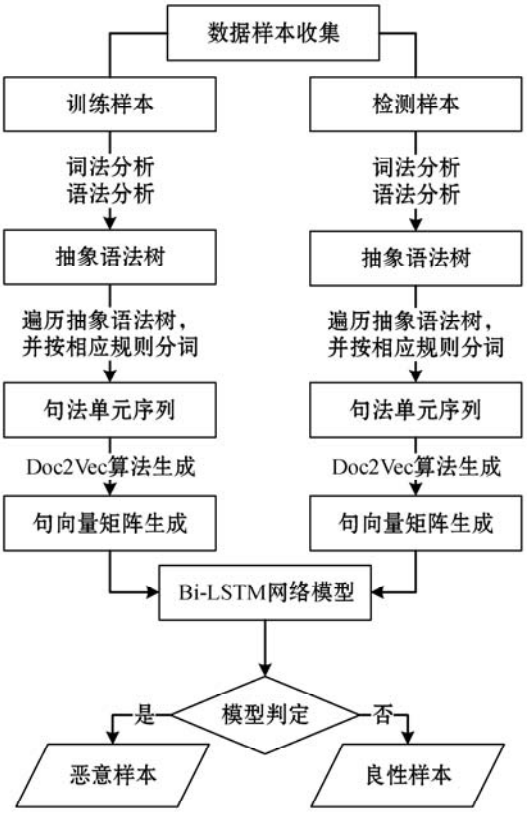

图 1 整体流程

1.1 句法单元序列生成

抽象语法树是源代码抽象语法结构的树状表示[15],树上的每一个节点都表示源代码中的一种结构。AST 通常被用来检查、分析转换代码,比如说浏览器执行 JavaScript 代码前需要将 JavaScript 代码转化为 AST 进行编译工作。通过解析 AST 可以得到 JavaScript 代码的语义信息,这对于本文的 JavaScript 文件的预处理工作是十分重要的。本文首先采用 Esprima. js 对 JavaScript 代码解析成为 AST,然后遍历 AST 节点得到词法单元序列和代码模块的调用顺序,最后根据代码模块的调用顺序对词法单元序列进行划分得到句法单元序列。图 2 为一段示例代码,图 3 为示例代码解析成的抽象语法树,表 1 为示例代码得到的句法单元序列。

```
var flag = true
while(flag){
  alert(1)
}
```

图 2 示例代码

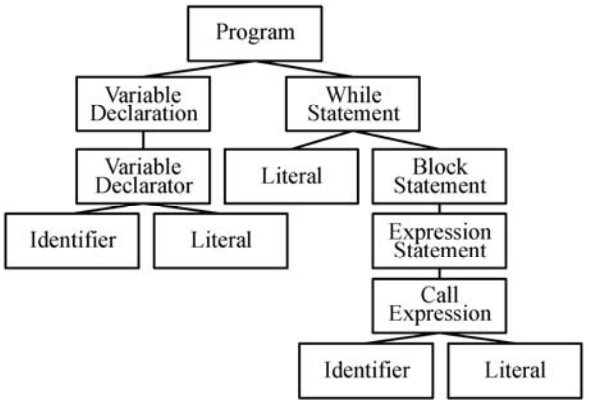

图 3　抽象语法树示例

表 1　句法单元序列示例

序号	句法单元序列
1	Identifier、Literal、Variable Declarator、Variable Declaration
2	Literal、Identifier、Literal、Call Expression、Expression Statement、Block Statement、While Statement
3	Variable Declaration、While Statement、Program

1.2　基于 Doc2Vec 算法的句向量训练

本文采用 Doc2Vec 算法对生成的句法单元序列进行向量化表征。Doc2Vec 算法是由 Le 等[16]在 2014 年提出来的一种非监督式算法,是在 Word2Vec 算法的基础上改进的,它能够将可变长度文本转化为固定长度的向量特征表示。该方法与 Word2Vec 算法相比,在输入层多增加了一个段落向量,解决了 Word2Vec 算法忽略词的上下文信息的问题,减少了在向量化过程中所造成的信息损耗。Doc2Vec 算法提出了两种模型来生成段落向量,分别是分布式的记忆模型(PV-DM)和分布式的词袋模型(PV-DBOW),其中,PV-DM 模型通过利用段落向量和段落中词的上下文信息去预测当前词的概率,PV-DBOW 通过当前词去预测段落中的任意词的概率。因为分析代码模块功能的时候常常需要结合上下文进行分析,故本文选用 PV-DM 模型对生成的句法单元序列进行向量表征。

1.3　基于 Bi-LSTM 的检测模型

利用 Doc2Vec 算法得到的句向量矩阵属于时间序列,为了对句向量矩阵进行更为可靠的分类,需要选择能够较好地处理时间序列数据和充分挖掘句向量矩阵信息的深度学习模型。故本文采取基于长短期记忆网络[17](Long Short-Term Memory,LSTM)的网络模型对代码生成的句向量矩阵进行训练。LSTM 网络结构如图 4 所示。

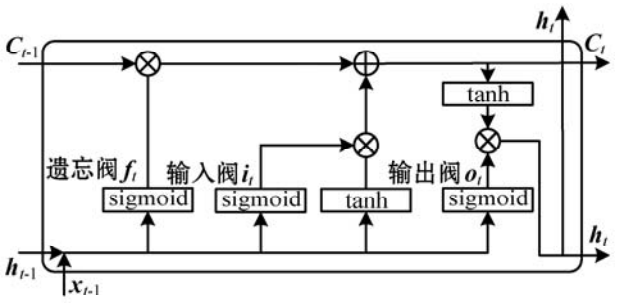

图 4　LSTM 网络结构图

循环卷积神经模型(Recurrent Neural Networks,RNN)能够处理时间序列数据,但是 RNN 由于本身结构的缺陷,存在长距离依赖问题,且容易发生梯度爆炸或者梯度消失[18]等问题。LSTM 是 RNN 的一种改进网络,通过遗忘阀 $\boldsymbol{f}_t$、输入阀 $\boldsymbol{i}_t$ 和输出阀 $\boldsymbol{o}_t$ 控制单元状态 $\boldsymbol{C}_t$,决定哪些数据需要保留,哪些数据需要被遗忘,进而解决 RNN 的长距离依赖问题。遗忘阀、输入阀和输出阀的计算式如下:

$$\boldsymbol{f}_t=\sigma(\boldsymbol{W}_f\odot[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_f) \tag{1}$$

$$\boldsymbol{i}_t=\sigma(\boldsymbol{W}_i\odot[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_i) \tag{2}$$

$$\boldsymbol{o}_t=\sigma(\boldsymbol{W}_o\odot[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_o) \tag{3}$$

式中:$\boldsymbol{W}$ 为权向量,$\boldsymbol{b}$ 为偏置向量,σ 是 sigmoid 函数。LSTM 的输出结果 $\boldsymbol{h}_t$ 由输出门 $\boldsymbol{o}_t$ 和单元状态 $\boldsymbol{C}_t$ 决定。计算式如下:

$$\tilde{\boldsymbol{C}}_t=\tanh(\boldsymbol{W}_c\odot[\boldsymbol{h}_{t-1},\boldsymbol{x}_t]+\boldsymbol{b}_c) \tag{4}$$

$$\boldsymbol{C}_t=\boldsymbol{f}_t\odot\boldsymbol{C}_{t-1}+\boldsymbol{i}_t\odot\tilde{\boldsymbol{C}}_t \tag{5}$$

$$\boldsymbol{h}_t=\boldsymbol{o}_t\odot\tanh(\boldsymbol{C}_t) \tag{6}$$

虽然 LSTM 解决了长期记忆问题,并且能够有效地利用文本的依赖关系,但是对于代码样本来说,当前代码模块可能收到之前代码模块的影响,而可能受到之后代码模块的影响,因此分析 JavaScript 代码产生的句法单元序列,需要充分考虑到句法单元序列的上下文信息,故本课题采用双向的 LSTM[18](即 Bi-LSTM)模型对输入的句法单元序列进行处理。Bi-LSTM 在 LSTM 的基础上添加了后向的 LSTM,使得 Bi-LSTM 可以充分挖掘时间序列的上下文信息。此外,由于句法单元序列代表了 JavaScript 源代码的语义信息和部分代码模块的调用顺序等信息,因此句法单元序列的上下文信息是检测恶意 JavaScript 代码的关键。

假设存在一个 JavaScript 代码的句向量表征矩阵 $\boldsymbol{S}$,它由 n 个句法单元组成,其中 $\boldsymbol{W}_i(1\leqslant i\leqslant n)$ 表示每一个句法单元的向量表示,如式(7)所示。这些句法单元序列将被送入 Bi-LSTM 模型进行训练。

$$S=(\boldsymbol{W}_1,\boldsymbol{W}_2,\cdots,\boldsymbol{W}_n) \tag{7}$$

此时 Bi-LSTM 的输出结果 h_t 由式(8)－式(10)决定,其中,$\overrightarrow{\boldsymbol{h}}_t$ 为前向的 LSTM 的输出结果,$\overleftarrow{\boldsymbol{h}}_t$ 为后向的

LSTM 的输出结果。

$$h_t = [\overrightarrow{\boldsymbol{h}}_t, \overleftarrow{\boldsymbol{h}}_t] \tag{8}$$

$$\overrightarrow{\boldsymbol{h}}_t = \overrightarrow{\boldsymbol{LSTM}}(\boldsymbol{W}_t, \overrightarrow{\boldsymbol{h}}_{t-1}) \tag{9}$$

$$\overleftarrow{\boldsymbol{h}}_t = \overleftarrow{\boldsymbol{LSTM}}(\boldsymbol{W}_t, \overleftarrow{\boldsymbol{h}}_{t-1}) \tag{10}$$

最终本文提出的深度学习网络检测模型如图 5 所示。本文先将 JavaScript 代码样本转化为句法单元序列,然后使用 Doc2Vec 算法将句法单元序列转化为句向量矩阵,接着将该句向量矩阵输入到 Bi-LSTM 网络中进行训练,最后用 softmax 函数对 Bi-LSTM 网络的输出进行分类,得到预测标签。

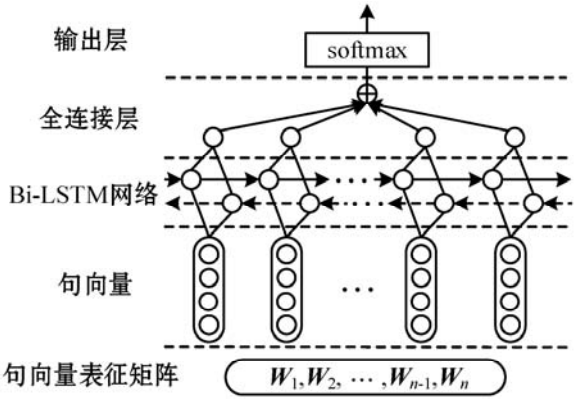

图 5 检测网络结构

2 实验与分析

2.1 实验数据及环境

本文的实验数据分为良性 JavaScript 样本和恶意 JavaScript 样本,其中良性样本是通过 Alexa 公布排名靠前的网站进行爬取得到的,恶意样本是 Github 公开的数据。在实验中,最终获得良性样本 7 056 份,恶意样本 5 453 份。将数据集的 90% 作为训练集,10% 作为测试集。

本文搭建的恶意 JavaScript 代码检测模型的实验环境如表 2 所示。其中检测模型是在 Ubuntu 上使用 tensorflow2.1 搭建并训练的,JavaScript 代码的数据预处理工作是在 Node.js 环境下使用 Esprima.js、Estraverse.js、Escodegen.js 完成的,句法单元序列的向量表征是利用 Genism 库完成的。

表 2 实验环境配置

名称	信息
操作系统	Ubuntu 18.04.5 LTS
系统配置	CPU: Intel Xenon CPU X5672,内存:12 GB
Python 库	Genism、TensorFlow2.1、Scikit-learn
Node.js	Node.js 12.6.3

2.2 实验评价指标

本文标记 T_P 为恶意代码被预测为恶意代码的个数,T_N 为良性代码被预测为良性代码的个数,F_N 为恶意代码被预测为良性代码的个数,F_P 为良性代码被预测为恶意代码的个数。本文实验使用精确率(Precision),召回率(Recall),准确率(Accuracy)和 F1 值等指标作为检验模型是否有效的标准,具体含义如下。

$$P_{\text{recision}} = \frac{T_P}{T_P + F_P} \tag{11}$$

$$R_{\text{ecall}} = \frac{T_P}{T_P + F_N} \tag{12}$$

$$A_{\text{ccuracy}} = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \tag{13}$$

$$F_1 = \frac{2 \times P_{\text{recision}} \times R_{\text{ecall}}}{P_{\text{recision}} + R_{\text{ecall}}} \tag{14}$$

2.3 实验设计与结果分析

2.3.1 向量表征方式对比实验

在同样分类模型训练的情况下,实验对比了向量表征方式、输入长度对检测模型的效果。其中分类模型选用朴素贝叶斯算法。表 3 结果为对代码样本转化为 AST 之后进行分词操作之后的统计结果,表 4 为采用朴素贝叶斯算法对句向量矩阵和词向量矩阵分类的结果。

表 3 文本序列类型

文本序列类型	最小序列长度	最长序列长度	平均序列长度
词法序列文本	4	702 543	6 432
句法序列文本	2	4 532	575

表 4 向量表征方式实验对比结果

向量表征方式	序列长度	准确率/%	精确率/%	召回率/%	F1 值/%
Word2Vec	512	72.32	77.23	59.47	67.19
Doc2Vec	512	82.43	81.33	83.32	81.87
Word2Vec	1 024	83.23	84.34	83.52	83.93
Doc2Vec	1 024	84.63	85.12	81.45	83.24

从表 4 可看出,当输入序列长度一致时,采用句向量表征的方式在检测准确率和精确率上是优于词向量表征方式的。结合表 3 可知,句向量表征的方式相比于词向量表征的方式,可以更好地对 JavaScript 样本进行向量表征,减少在表征过程中造成的信息损失消耗,有利于提高模型检测准确率,提高检测效率。

2.3.2 Bi-LSTM 检测模型实验

本文在模型的训练过程中,将输入的句法单元序列的长度设置为 512,每个句向量的维度设置为 100,损失函数使用交叉熵损失函数,并使用 Adam 优化算

法对模型进行优化。本文提出来的方法的实验结果如图 6 和图 7 所示,随着迭代次数的增加,交叉熵损失函数逐渐下降,最终收敛到一定值且误差较低,准确率逐渐上升,最终收敛于一定值且准确率较高,这表明该模型具备良好的训练效果。

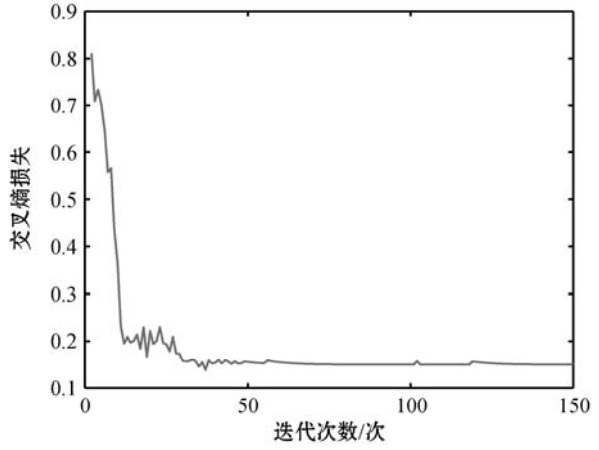

图 6　交叉熵损失函数变化曲线

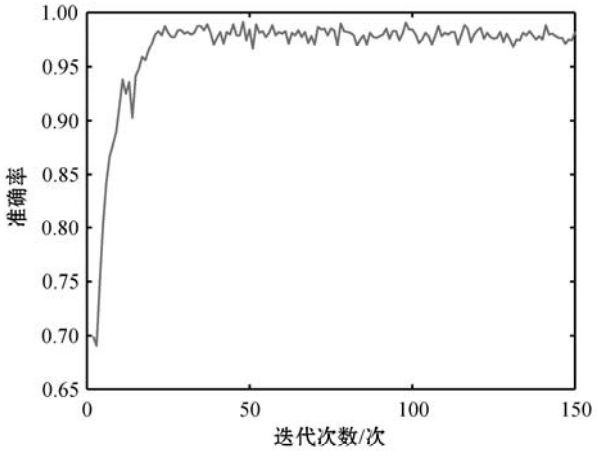

图 7　准确率变化曲线

在数据输入相同的特征情况下,本文将 Bi-LSTM 与 Naïve Bayes、SVM、LSTM 等分类算法进行比较,比较结果如表 5 所示。

表 5　不同分类算法检测结果(%)

分类算法	准确率	精确率	召回率	F1 值
Naïve Bayes	82.43	81.33	83.32	81.87
SVM	92.40	92.81	92.73	92.76
LSTM	95.46	95.74	95.13	95.43
Bi-LSTM	97.03	96.13	97.10	96.61

由表 5 可知,在各类分类算法中,Naïve Bayes 的各项评估指标最差,Bi-LSTM 的各项评估指标最好,且取得了最好的检测效果,准确率为 97.03%,召回率为 97.10%,表明 Bi-LSTM 分类算法对于恶意代码的检测效果是要优于其他算法的。

文献[19]获得的良性样本同样为 Alexa 公布排名靠前网站爬取的数据,恶意样本为从 Vxheaven 公布的恶意网站爬取,大部分与本文获取的恶意样本攻击类型相同。文献[4]提出的 JaST 模型已在 Github 上开源,故本文对文献[4]和文献[19]中提出的检测模型做了对比。结果如表 6 所示。本文算法相对于文献[4]和文献[19]提出的检测模型在准确率和召回率上是更优的,且精确率与 JaST 模型相差无几,由此可以证明本文算法的有效性。

表 6　不同模型有效性对比结果(%)

分类算法	准确率	精确率	召回率	F1 值
JaST[4]	92.16	96.71	92.53	94.57
SdA-LR[19]	94.82	94.90	94.80	94.80
本文算法	97.03	96.13	97.10	96.61

2.3.3　模型运行效能分析

为了评估本文提出的模型的运行效能,随机地选择了 500 个样本并记录了每个阶段的时间消耗,具体情况如表 7 所示。

表 7　样本在各阶段的时间消耗

阶段	最小值/ms	最大值/ms	平均值/ms	总计/s
解析生成 AST	6.382	531.154	78.592	73.010
生成句法单元序列	0.613	197.078	37.494	18.746
生成句向量矩阵	0.001	0.355	0.041	0.020
模型验证	0.036	0.057	0.043	0.021
总计	7.032	728.644	116.170	91.798

由表 7 中的数据可知,500 个样本的时间消耗为 91.798 s,平均时间消耗 116.170 ms。考虑到 JavaScript 样本平均大小为 74.84 KB,因此认为该开销是合理的。另外,本文对比了文献[4]提出来的 JaST 模型,在同一平台同一数据集下进行实验,结果如表 8 所示。通过表 8 可知,JaST 模型的平均检测时间近乎是本文提出来的模型的 7.5 倍,故本文提出的检测模型的检测效率是优于 JaST 的,且检测时间消耗较小。

表 8　模型检测时间对比

模型	平均检测时间/ms
本文模型	116.170
JaST	879.304

3　结　语

为了应对日益增多的基于 JavaScript 的网络攻击,特别是被混淆过的恶意 JavaScript 代码,本文提出了一种基于 Bi-LSTM 的恶意 JavaScript 代码检测方法。本文首先通过 AST 将 JavaScript 代码转化为句法单元序

列,然后利用 Doc2Vec 算法将句法单元序列转化为句向量表示并输入到 Bi-LSTM 网络中进行训练和检测。本文通过进行对比实验验证了本文提出的检测方法拥有良好的表现,准确率达到 97.03%,召回率达到 97.10%,并且检测的时间消耗短。由于本文收集到的恶意 JavaScript 样本数据来自 2016 年—2019 年的互联网,导致本文提出来的模型无法检测到新型攻击,计划在后续研究中不断加入新型恶意 JavaScript 示例,提高模型的性能。另外,本文所使用的数据预处理方式虽然较大程度避免了信息损失消耗,但还是造成了一定的损失,如何尽量减少在向量表征过程中造成的信息损耗将是我们进一步的研究方向。

参 考 文 献

[1] Xu W, Zhang F, Zhu S. The power of obfuscation techniques in malicious JavaScript code: A measurement study [C]//7th International Conference on Malicious and Unwanted Software,2012:9 - 16.

[2] 徐青,朱焱,唐寿洪. 分析多类特征和欺诈技术检测 JavaScript 恶意代码[J]. 计算机应用与软件,2015,32(7):293 - 296.

[3] 马洪亮. 基于 JavaScript 的恶意网页异常检测方法研究[D]. 北京:北京交通大学,2018.

[4] Fass A, Krawczyk R P, Backes M, et al. JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript[C]// International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,2018:303 - 325.

[5] Liang H L, Yang Y X, Sun L, et al. JSAC: A novel framework to detect malicious JavaScript via CNNs over AST and CFG[C]//International Joint Conference on Neural Networks,2019:1 - 8.

[6] 邱瑶瑶,方勇,黄诚,等. 基于语义分析的恶意 JavaScript 代码检测方法[J]. 四川大学学报(自然科学版),2019,56(2):273 - 278.

[7] Mikolov T, Sutskever I, Chen K, et al. Distributed representations of words and phrases and their compositionality [J]. Advances in Neural Information Processing Systems, 2013,26:3111 - 3119.

[8] Canal D, Cova M, Vigna G, et al. Prophiler: A fast filter for the large-scale detection of malicious web pages[C]// 20th International Conference on World Wide Web, 2011: 197 - 206.

[9] Curtsinger C, Livshits B, Zorn B G, et al. ZOZZLE: Fast and precise in-browser JavaScript malware detection[C]// 20th USENIX Security Symposium,2011:33 - 48.

[10] Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques[C]//4th International Conference on Malicious and Unwanted Software,2009 47 - 54.

[11] Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript code [C]//19th International Conference on World wide Web, 2010:281 - 290.

[12] Kim K, Kim I L, Kim C H, et al. J-force: Forced execution on JavaScript[C]//26th International Conference on World Wide Web,2017:897 - 906.

[13] Mao J, Bian J D, Bai G D, et al. Detecting malicious behaviors in JavaScript applications[J]. IEEE Access,2018, 6:12284 - 12294.

[14] Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures[J]. Neural Networks,2005,18(5 - 6):602 - 610.

[15] Yue C, Wang H N. Characterizing insecure JavaScript practices on the web[C]//18th International Conference on World Wide Web,2009:961 - 970.

[16] Le Q, Mikolov T. Distributed representations of sentences and documents[C]//31st International Conference on Machine Learning,2014:1188 - 1196.

[17] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural Computation,1997,9(8):1735 - 1780.

[18] Kolen J, Kremer S. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies[M]//A Field Guide to Dynamical Recurrent Neural Networks. New York: Wiley-IEEE Press,2001:237 - 243.

[19] Wang Y, Cai W D, Wei P C. A deep learning approach for detecting malicious JavaScript code[J]. Security & Communication Networks,2016,9(11):1520 - 1534.

(上接第 356 页)

[12] 贾宝柱,韩森,顾一鸣,等. 基于降维优化的主动式锚泊定位系统张力分配算法[J]. 电机与控制学报,2020,24(7):156 - 164.

[13] Sangaiah A K, Fakhry A, Abdel-Basset M, et al. Arabic text clustering using improved clustering algorithms with dimensionality reduction[J]. Cluster Computing,2019,22(2): 1 - 15.

[14] 李青彦,彭进业. 一种基于空间金字塔特征的图像分类降维算法[J]. 微型电脑应用,2020,36(2):17 - 19.

[15] 丁小艳. 基于 PSO 优化的盲源分离式文本特征降维分类方法[J]. 山东农业大学学报(自然科学版),2019,50(5):881 - 884.

[16] Xu X Z, Liang T M, Zhu J, et al. Review of classical dimensionality reduction and sample selection methods for large-scale data processing[J]. Neurocomputing, 2019, 328 (7):5 - 15.

[17] Elyanow R, Dumitrascu B, Engelhardt B E, et al. NetNMF-SC: Leveraging gene-gene interactions for imputation and dimensionality reduction in single-cell expression analysis[J]. Genome Research,2020,30(2):119 - 129.