计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(8):266−273 [doi: 10.15888/j.cnki.csa.008036]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

BiLSTM 在 JavaScript 恶意代码检测中的应用①

雷天翔[1,2], 万　良[1,2], 于　淼[1], 褚　堃[1,2]

[1](贵州大学 计算机科学与技术学院, 贵阳 550025)
[2](贵州大学 计算机软件与理论研究所, 贵阳 550025)
通讯作者: 万　良, E-mail: wanliangtr@163.com

摘　要: 传统的机器学习方法在检测 JavaScript 恶意代码时, 存在提取特征过程复杂、计算量大、代码被恶意混淆导致难以检测的问题, 不利于当前 JavaScript 恶意代码检测准确性和实时性的要求. 基于此, 提出一种基于双向长短时神经网络 (BiLSTM) 的 JavaScript 恶意代码检测方法. 首先, 将得到的样本数据经过代码反混淆, 数据分词, 代码向量化后得到适应于神经网络输入的标准化数据. 其次, 利用 BiLSTM 算法对向量化数据进行训练, 学习 JavaScript 恶意代码的抽象特征. 最后, 利用学习到的特征对代码进行分类. 将本文方法与深度学习方法和主流机器学习方法进行比较, 结果表明该方法具有较高的准确率和较低的误报率.

关键词: 恶意代码检测; 双向长短时神经网络; JavaScript 脚本; 词向量

引用格式: 雷天翔,万良,于淼,褚堃.BiLSTM 在 JavaScript 恶意代码检测中的应用.计算机系统应用,2021,30(8):266–273. http://www.c-s-a.org.cn/1003-3254/8036.html

Application of BiLSTM in JavaScript Malicious Code Detection

LEI Tian-Xiang[1,2], WAN Liang[1,2], YU Miao[1], CHU Kun[1,2]

[1](College of Computer Science and Technology, Guizhou University, Guiyang 550025, China)
[2](Institute of Computer Software and Theory, Guizhou University, Guiyang 550025, China)

Abstract: The JavaScript malicious code detection by existing machine learning methods is complex, with large amount of calculation and difficult detection caused by maliciously confused codes. Existing approaches, therefore, fail to realize accurate and real-time detection. For this reason, a method based on Bidirectional Long Short-Term Memory (BiLSTM)-based method for JavaScript malicious code detection is proposed. Firstly, standardized data adapting to be input into the neural network is obtained by code de-obfuscation, data segmentation, and code vectorization. Secondly, the BiLSTM algorithm is used to train the vectorized data and learn the abstract features of JavaScript malicious code. Finally, the abstract features are used to assort codes. The proposed method is compared with deep learning and existing mainstream machine learning approaches, and the results show that this method exhibits a higher accuracy rate and a lower false alarm rate.

Key words: malicious code detection; Bidirectional Long-Short Term Memory (BiLSTM) network; JavaScript's scripts; word vector

1　引言

近年来, 随着 Web 应用程序以浏览器/服务器 (B/S) 架构占据主流市场, Web 服务已经被广泛的应用, 浏览器和网页已然成为传播恶意代码的重要途径. 攻击者使用网站代码漏洞, 第三方应用程序漏洞, 浏览器漏洞和操作系统漏洞对网站执行跨站点脚本攻击, 注入

① 基金项目: 国家自然科学基金 (62062020)
Foundation item: National Natural Science Foundation of China (62062020)
收稿时间: 2020-11-28; 修改时间: 2020-12-21; 采用时间: 2021-01-07; csa 在线出版时间: 2021-07-31

Web 木马, 篡改网页, 网络钓鱼和窃取个人信息, 造成用户个人信息泄露和财产损失. 根据《2019 中国网络安全报告》, 瑞星“云安全”在 2019 年全球共截获恶意网址 (URL) 总量 1.45 亿个, 其中挂马类网站 1.2 亿个, 钓鱼类网站 2454 万个[1]. 中国的恶意 URL 总数为 471.63 万, 位列全球第五. 恶意内容包含在恶意网页中, 这些恶意网页很容易使访问者不知不觉地受到网络攻击, 例如病毒传播, 特洛伊木马植入, 信息泄漏等. 他们的恶意代码主要是脚本语言, 例如 JavaScript. 为了避免检测, 这些恶意脚本还以不同的编码方法进行了混淆[2]. 经过混淆后的 JavaScript 恶意代码更是成为提高检测性能的一大难点.

当前, 静态检测方法为大多数研究人员在对 JavaScript 恶意代码检测过程中使用的方法. 此方法通过对 JavaScript 源码的语法、过程、结构等进行分析从而提取特征达到对恶意代码的检测的目的. 随后使用机器学习[3–5] 的检测方法被提出. Likarish 等[6] 从每个代码文件中选择 65 个统计特征作为输入, 包括可读序列的数量, 每个 JavaScript 关键字的频率, 脚本的长度, 每行平均字符数和 Unicode 符号. 数量等. 评估代码的可读性, 然后利用机器学习来检测模糊的恶意 JavaScript 代码, 取得了很好的分类效果. Wang[7] 采用机器学习技术抽取和分析恶意脚本特征, 使用 SVM 分类模型进行分类, 结果表明具有较高的检测率低误报率. Li 等[8] 使用自编码器将高维数据转换为低维数据, 紧接着采用深度自编码网络自动学习 JavaScript 恶意代码的特征, 从而对其进行检测和分类. 通过研究发现现有的检测技术存在着手工提出特征, 工作量大主观性太强和对混淆的恶意代码检测难度大等不足, 针对上述不足之处, 一些研究人员开始使用深度学习的方法来检测 JavaScript 恶意代码, 深度学习神经网络可以自动学习特征, 因此避免了手动提取特征的复杂性和主观性. Cui[9] 将深度学习中的卷积神经网络 (Convolutional Neural Networks, CNN) 运用到 JavaScript 恶意代码检测. Wu 等[10] 使用卷积神经网络、LSTM、CNN-LSTM 模型进行漏洞检测. 上述实验结果显示, 相比于传统的方法, 上述方法的优势尽显无疑. Fang 等[11] 使用 LSTM 来对 JavaScript 恶意代码. LSTM 能有效地检测 JavaScript 恶意代码, 但对于混淆的 JavaScript 恶意代码的检测还不够准确. Choi 等[12]和 Visaggio 等[13] 使用 3 个衡量标准来对混淆进行检测, 衡量指标分别是字符串的最大长度、无序状态的熵、在忽略大小写后, 非数字字母在字符串中所占的比例. Lu 等[14] 从代码的语义方面对代码进行反混淆. 马洪亮等[15] 将静态分析和动态分析相结合起来, 以此来达到恶意代码反混淆的目的.

基于上诉文献, 为了获取恶意代码深层本质特征, 提出了一种基于双向长短时记忆网络的 JavaScript 检测方法. LSTM 是带有记忆细胞单元的循环神经网络, 常用于处理长序列数据, 能解决长序列数据依赖的问题. 但是单向的 LSTM 只能解决上文对下文的依赖而无法解决下文对上文的依赖, 使得对 JavaScript 代码特征提取不够充分. 在此基础上本文选择双向长短时记忆网络来学习 JavaScript 代码, 获得更加全面的代码特征, 提高检测的准确度. 实验结果分析, 该方法具有较优的分类效果, 强鲁棒性和较强的泛化能力.

2 研究方法

2.1 系统概括

本文检测方法的整体框架如图 1 所示, 首先通过爬虫工具从 Alexa 和 PhishTank 网站收集大量数据, 随后经过数据清洗、数据分词和数据向量化后得到能输入到神经网络中的标准化数据集, 最后使用双向长短时神经网络对标准数据集进行训练和分类, 得到最终的结果.

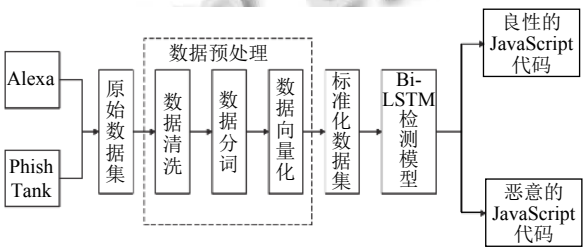

图 1 JavaScript 检测整体框架

2.2 长短时神经网络 (LSTM)

在 1991 年, 研究人员提出了一种改进的 RNN 网络, 也就是长短时神经网络 (Long Short term Memory Networks, LSTM). 这一网络的出现成功的解决了 RNN 的长期依赖问题和梯度消失问题.

LSTM 网络结构如图 2 所示, 其有 4 个重要组成部分为遗忘门、输入门、输出门、用于更新细胞状态的部分[16]; 遗忘门决定保留和遗忘上一时刻状态c_{t-1}的哪些信息; 输入门负责决定在 t 时刻的输入x_t有多少信

息被保留到c_t; 输出门选择c_t有多少输送到了 LSTM 在 t 时刻的输出值h_t; 数学表达式如下:

更新遗忘门输出f_t:

$$f_t = \sigma_1(w_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$

式 (1) 中, 遗忘门的权重矩阵为w_f, h_{t-1}和x_t拼接成的一个新向量$[h_{t-1}, x_t]$, 偏置为b_f, $\sigma_1(x)$为激活函数使用 Sigmoid.

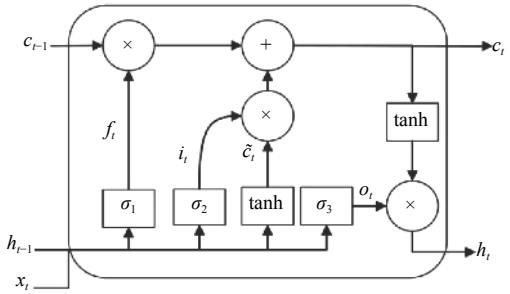

图 2 长短时记忆网络结构

更新输入门的输出i_t, 同时计算候选状态$\tilde{c}_t$, 计算公式如下:

$$i_t = \sigma_2(w_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$

$$\tilde{c}_t = \tanh(w_c \cdot [h_{t-1}, x_t] + b_c) \tag{3}$$

其中, w_i为输入门的权重, b_i为偏置向量, 激活函数$\sigma_2(x)$同样使用 Sigmoid 函数, w_c为权重矩阵, b_c为偏置向量, 激活函数为$\tanh(x)$.

更新当前时刻细胞状态c_t:

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{4}$$

更新输出门输出h_t:

$$o_t = \sigma_3(w_o \cdot [h_{t-1}, x_t] + b_0) \tag{5}$$

$$h_t = o_t * \tanh(c_t) \tag{6}$$

式 (5) 中, 输出门的权重矩阵为w_o, 激活函数使用$\tanh(x)$, b_o是输出门的偏置.

2.3 双向长短时记忆网络 (BiLSTM)

LSTM 解决了 RNN 的梯度消失问题, 但单向的 LSTM 只能从前向后传递依赖, 无法充分利用上下文信息, 在面对预测问题时, 需要上下文信息来共同决定当前预测结果, 使得预测结果更加准确. 因此, 双向 LSTM 被提出来解决此问题, 其结构如图 3 所示.

2.4 一种基于 BiLSTM 的检测模型

2.4.1 模型构建

为了充分利用上下文的依赖关系, 本文提出一种基于 BiLSTM 的检测模型. 如图 4 所示.

Input layer: 输入层. 数据以向量的形式进行输入, 同时还要设置相关参数 (batchsize).

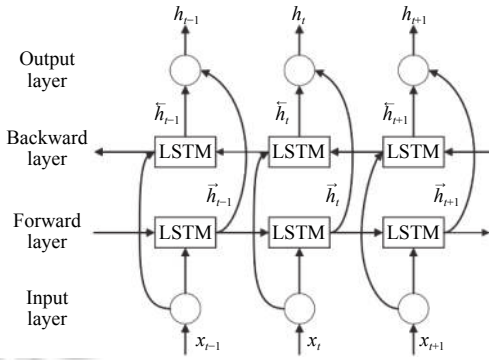

图 3 BiLSTM 网络结构

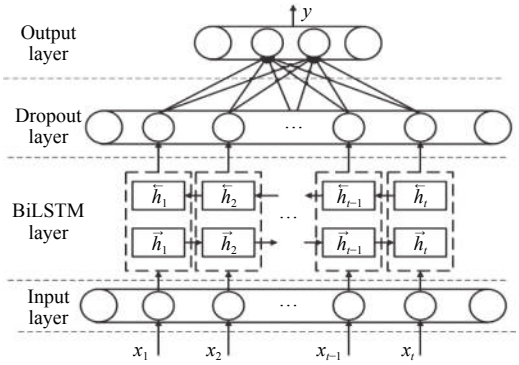

图 4 基于 BiLSTM 的检测模型

BiLSTM layer: 双向长短时记忆网络层. BiLSTM 具有能解决数据长期依赖并且能保留上下文语义信息的优点, 让其自动学习 JavaScript 恶意代码的特征. 其中包括了两部分, 从前向后传递的 LSTM 层和从后向前传递的 LSTM 层, 在某一时刻, 将两个方向不同的层的结果结合起来, 得到最终的输出结果传递给下一层.

Dropout layer: 为了使检测模型具有较强的泛化能力, 防止过拟合问题的产生.

Output layer: 输出层. 输出分类器分类后的结果.

本文的 JavaScript 恶意代码检测模型步骤如下, 具体分为两步, 第 1 步为提取代码的抽象特征, 第 2 步对其进行分类检测. 输入: $X_i=\{x_1, x_2, \cdots, x_{49}, x_{50}\}$

步骤 1:

(1) 将 X_i 作为输入, 经过输入层得到输出向量 $I_i=\{i_1, i_2, \cdots, i_{49}, i_{50}\}$.

(2) I_i 作为 BiLSTM 层的输入向量, 经过这一层

的前向 LSTM 得到前向的隐向量 $h_{L_i}=\{h_{L_1},h_{L_2},\cdots,h_{L_{63}},h_{L_{64}}\}$. 经过后向的 LSTM 得到后向的隐向量 $h_{R_i}=\{h_{R_1},h_{R_2},\cdots,h_{R_{63}},h_{R_{64}}\}$.

(3) 最后将 (2) 中的两个向量拼接成为一个新的向量$\{[h_{L_1},h_{R_1}],[h_{L_2},h_{R_2}],\cdots,[h_{L_{63}},h_{R_{63}}],[h_{L_{64}},h_{R_{64}}]\}$, 即$h_i=\{h_0,h_1,\cdots,h_{63},h_{64}\}$.

步骤 2:

(1) 为了防止过拟合问题的产生, 在 dropout 层, 随机的使若干个神经元失活, 这一层的输入为 h_i, 这一层的输出为 $d=\{d_1,d_2,\cdots,d_{63},d_{64}\}$.

(2) 将 dropout 层的输出向量 d 输入到 Output 层的 Softmax 函数中进行分类, 得到最终的二维输出 y.

2.4.2 算法设计

为了检测出恶意代码, 设计了算法 1.

算法 1. BiLSTM 训练算法

1) 构建 BiLSTM 神经网络并对网络的权重和偏置进行初始化操作;
2) 构建 Softmax 并初始化其参数;
3) for i in epoch, 进行迭代循环训练, epoch 为训练迭代次数;
4) 将预处理后的训练集 X_i 作为输入层的输入, 得到输出 I_i;
5) 将 I_i 作为 BiLSTM 的输入, 经过前向 LSTM 和后向 LSTM 分别得到前向隐向量 h_L 和后向隐向量 h_R, 将 h_L 和 h_R 拼接得到抽象特征 h;
6) 经过 dropout 层避免模型出现过拟合;
7) 将抽象特征向量输入到 Softmax 分类其中得到分类结果;
8) 根据最后的输出结果和真实结果之间的差距, 通过反向传播算法依次调整各个参数;
9) 更新 Softmax 的参数;
10) 对 BiLSTM 神经网络的权重和偏置进行更新;
11) end for
12) 将 $T_i=\{t_1,t_2,\cdots,t_{49},t_{50}\}$作为测试集输入到训练好的模型中, 得到测试结果.

3 实验

本文实验的硬件条件为: 处理器 Intel(R) Core(TM) i7-9750H CPU@2.60 GHz, 内存 8 GB, 图像处理器 NIVDIA GeForce GTX 1650, 显存 4 GB. 实验环境为 Python3.6.2、Tensorflow-gpu1.15.0.

3.1 实验数据收集

本文的数据集中, 良性的数据来自于 Alexa 排名靠前的网站所抓取的数据, 恶意的数据来源于知名网站 PhishTank 的数据库, 经过数据预处理后, 获得良性的代码数据 84 208 条和恶意代码数据 26 216 条. 为良性的数据打上标记为 1, 恶意的数据打上标记为 0. 实验中, 从样本中以 7:3 的比例随机选取训练集和测试集数据. 表 1 展示了数据集分布情况.

3.2 数据预处理

为了提高输入数据的质量以此来提高整个模型性能, 在此基础上, 本文对收集的所有数据进行了数据预处理.

(1) 代码反混淆: 利用解码技术对于恶意的 JavaScript 混淆代码进行了反混淆处理. 本文采用动态分析的技术进行反混淆. 首先生成混淆代码的抽象语法树 AST, 遍历 AST 的所有节点, 若该节点为变量, 数组, 方法等, 则将该节点进行保留, 其余的节点则删除, 对保留的节点使用变量值读取器[17] 读取终值. 经过上述反混淆过程,隐藏在变量终值中的初始 JavaScript 恶意代码会被还原出来. 如图 5 所示为混淆代码还原为原始代码.

表 1 数据集分布

标签	类别	训练集	测试集	总计
0	恶意	18 349	7867	26 216
1	良性	59 946	24 262	84 208

```
eval(function(p,a,c,k,e,d,){e=function(c){return c}}
if(!''.replace(/^/,String)){while(c--)
{d[c]=k[c]||c}k=[function(e){return[e]}];
e=function(){return'\\w+'};c=1;while(c--)
{if(k[c]){p=p.replace(newRegExp('\\b'+e(c)+'\\
b','g),k[c])}}return p}('178(9){0 6=9;0 4=9;0 7;0
11="5(\'7=6+4;\');";0 10="6=8(6-1);5(11);";0
13='4=8(4);5(10);":015="18(9<2){7=1;}20{4=42;5(13);}
;5(15);197}0 14=3;012=8(14);16(12);'',10,21,
'var||||t2|eval|t1|k|f|n|str3|str4|y|str2|x|str1
print|function|if|return|lese'.split('|'),0,{}))
```

(a) 混淆代码

```
function f(n) var t1=n;var t2=n;var k
;var s4="eval('k=t1+t2');"; var s3="t1=f(t1-1);
eval(s4);";var s2="t2=f(t2);eval(str3);"
var s1="if(n<2){k=1;} else{t2=t2-2;eval(s2);}";
eval(s1);return k;} var x=3;var y=f(x) print(y);
```

(b) 原始代码

图 5 混淆代码经过反混淆还原为初始代码

(2) 分词: 对数据进行代码反混淆的工作后, 由于神经网络的输入为向量,因此先进行分词处理, 使用分词工具 NLTK 对代码数据进行分词, 去除停用词, 为后续训练词向量做准备. 图 6 展示了分词结果.

(3) 向量化: 首先根据词频-逆文件频率 (TF-IDF) 算法建立词汇库, 选择那些对识别恶意代码具有关键作用的词. 根据词汇库将代码数据转化为数值型数据, 使用 Gensim 工具包中的 Word2Vec 模型对分词的数据训练, 将分词的数据转化为向量, 表 2 展示了一个向

量化的样例. 由于神经网络输入长度固定, 而向量化后的代码长度不固定. 因此选择合适的向量维度对于模型精度极其重要, 于是在训练词向量的过程中根据最后的向量维度, 如果数据的长度超过向量维度,则对其进行截断, 如果数据的长度没有超过向量维度, 后面的数据用–1 进行填充. 使得所有输入数据都保持在合适的向量维度.

```
<script>function f(n){
  var t1=n;var t2=n;var k;
 var s4="eval('k=t1+t2');";
 var s3="t1=f(t1-1);eval(s4);";var
  s2="t2=f(t2);eval(str3);";
 var s1="if(n<2){k=1;} else{t2=t2-
2;eval(s2);}"; eval(s1);return k;}
 var x=3;var y=f(x) print(y);</script>
```

(a) 原始样例

```
'<script>','function','f(n)','{' 'var','t1=','n','var'
't2=','n''var','k' 'var','s4=','eval','(','k=','t1+t2'
')','var','s3=','t1=','f(t1-1)', 'eval','(','s4',')'
,'var','s2=', 't2=','f(t2)','eval','(','str3'
')','var','s1=','if','(','n<2',')' '{','k=','0','}',
'else','{','t2=' 't2-2','eval','(','s2',')',
'}' 'eval','(','s1',')' 'return','k' '}','var',
'x=','0','var' 'y=','f(x)' 'print',
'(','y',')','</script>'
```

(b) 分词后的结果

图 6 分词后的结果

表 2 向量化结果

操作	样例
分词样例	‘<script>’, ‘var’, ‘s4=’, ‘eval’, ‘(’, ‘k=’, ‘t1’, ‘+’, ‘t2’ ‘)’, ‘var’, ‘s3=’, ‘</script>’
向量化	[4,20,7,64,34,3,0,31,0,38,20,12,5]

3.3 实验结果及其分析

表 3 展示了 BiLSTM 的实验结果. 此结果为代码数据经过 BiLSTM 神经网络的训练和分类后得到的.

表 3 BiLSTM 实验结果混淆矩阵

实际	预测为恶意	预测为良性	总数
恶意	*TP*=25 849	*FN*=367	26 216
良性	*FP*=1179	*TN*=83 029	84 208
总数	27 028	83 396	110 424

3.3.1 评价指标

为了验证本文方法的性能, 本文方法的评判标准为准确率 (*Accuracy*)、误报率 (False Positive Rate, *FPR*) 和召回率 (Recall, *RE*), 评价分类模型的 3 个重要指标如下:

准确率: 被正确检测出的数据占所有样本的比例, 即:

$$Accuracy = \frac{TP+TN}{TP+TN+FN+FP} \times 100\% \quad (7)$$

召回率: 正确分类为恶意的代码占所有恶意代码的比例, 即:

$$RE = \frac{TP}{TP+FN} \times 100\% \quad (8)$$

误报率: 良性的数据被误判为恶意代码占样本总数的比例, 即:

$$FPR = \frac{FP}{FP+TN} \times 100\% \quad (9)$$

具体参数见表 4.

表 4 模型参数设置

TP	*FN*	*TN*	*FP*
良性数据被判对	良性数据被判错	恶意数据被判对	恶意数据被判错

3.3.2 检测方法分析

(1) 向量维度

在数据预处理时, 不同的向量维度对模型的训练和检测的效率具有不同的影响, 一个合适的向量维度才能使模型充分利用数据的信息. 向量维度过长会使得模型的收敛速度变慢, 向量维度过短会遗失大量的有效信息, 因此为了选择合适的向量维度, 本文对比了 30、50、100、150 这 4 个向量维度, 以此来观察向量维度与准确率和训练时间的变化关系. 实验结果如图 7 所示. 由图中的结果可以得出, 向量维度超过 50 后, 模型的检测效果相差不明显, 但是模型训练所用的时间却相差很大. 模型在向量维度为 50 和 100 时, 其检测效果是最好的, 然而选择 100 维向量时, 模型的训练时间几乎是 50 维向量的 2 倍, 因此, 本文在数据预处理时选择了向量维度为 50 维, 对不足 50 维的向量进行填充, 对超过 50 维的向量进行截断.

(2) 优化器

神经网络中的优化器是在利用损失函数计算出模型的损失值后, 再利用损失值对模型的各个参数进行优化, 使其达到最优的值. 最终使得模型的预测值和真实值的误差越来越小. 不同的优化器会导致模型性能的差异, 模型的收敛速度的不同. 本文对几种常见的优化器进行了对比试验, 损失函数的变化曲线如图 8 所示. 可以看出, 优化器 SDG 的效果最差, 并且收敛的速度不如其余几个优化器, Adam 优化器的收敛速度略微高于其余几种优化器而它的损失函数值更小. 故本文

的模型选择了 Adam 优化器, 使模型更加快速的收敛到一定的值.

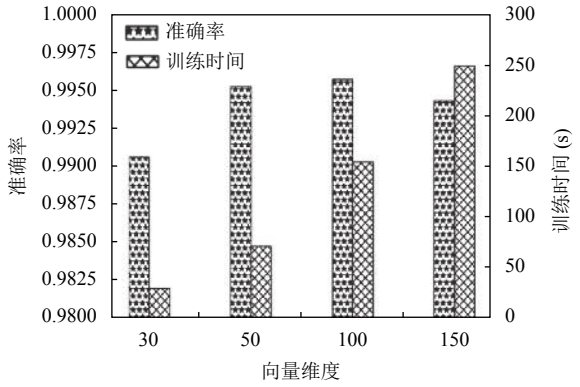

图 7 向量维度

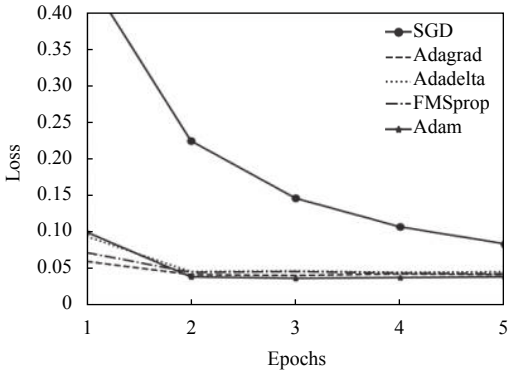

图 8 优化器

(3) 分类器

分类器是根据数据的真实标签来学习分类的规则, 随后在训练完毕后对未知的数据进行分类. 常用的两种分类器为 LR 分类器和 Softmax 分类器, LR 分类器主要用于二分类问题, Softmax 常用于互斥的多分类问题, 在分类数为 2 的时候, 会退化为逻辑回归分类.本文对两种分类器进行了对比实验. 从表 5 中的实验结果可以看出使用了 Softmax 分类器的模型的性能要优于使用 LR 分类器的检测模型, 因此这里选择 Softmax 作为分类器.

在实验环节, 本文对各种参数的设置进行了对比实验, 选择了能使模型达到最优效果的参数, 所得的检测模型的参数设置如表 6 所示.

表 5 分类器

分类器	准确率	召回率	*F*1值
LR	0.988	0.984	0.973
Softmax	0.995	0.986	0.988

表 6 模型参数设置

参数	设置
向量维度	50
网络结构	50×128–128–64–2
Dropout	0.5
优化器	Adam
分类器	Softmax

本文提出的实验方法结果如图 9~图 11所示, 随着训练次数的增加, 准确率和损失函数曲线会逐渐收敛. 通过预测值和真实值之间的误差反向传播调节各个参数的值, 直到模型的效果达到最优, 在 5 个 epoch 后, 此时准确率和损失函数的值会收敛到一定值. 模型训练完毕后, 使用测试集数据对模型进行测试, 实验发现, 本文提出的方法对于一条 script 标签的代码进行检测的时间约为 0.33 s, 并且数据随来随检测, 可以对网页中的 JavaScript 恶意代码实现实时的检测.

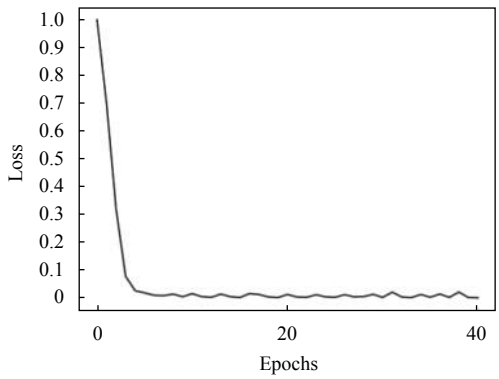

图 9 损失函数曲线

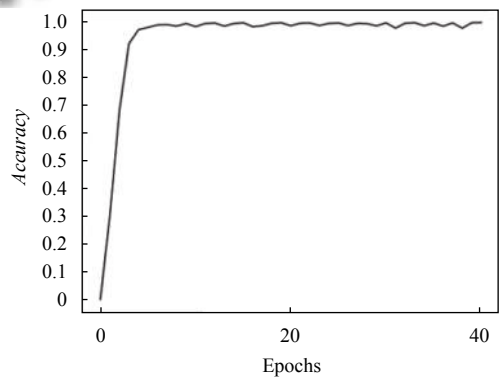

图 10 准确率变化曲线

3.3.3 对比分析

本文通过两组对比实验来评估模型的效果. 其中一组对比实验是将本文的方法和文献 [7] 的 SVM, AD

Tree 等检测方法进行对比. 另一组对比试验是将本文中的方法与其他深度学习算法如 TextCNN, RCNN 等进行对比.

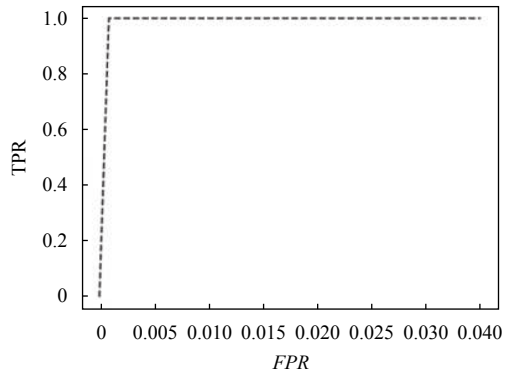

图 11 ROC 曲线

(1) 机器学习对比实验

从表 7 和图 12 可以看出, 传统的机器学习方法对 JavaScript 恶意代码实现了较好的检测, 但与本文的 BiLSTM 相比, 对恶意代码检测的准确率没有本文方法的高, 并且误报率相对而言较高, 而本文方法的准确率为 99.52% 而误报率仅为 1.4%, 并且不需要手动提取恶意代码特征, 节省了大量的人力.

表 7 BiLSTM 与机器学习算法对比结果

检测方法	准确率(%)	*RE*	*FPR*	*F*1
ADTree	93.86	0.937	0.121	0.937
AdaBoost	95.31	0.941	0.03	0.941
SVM	98.53	0.974	0.053	0.976
本文方法	99.52	0.986	0.014	0.988

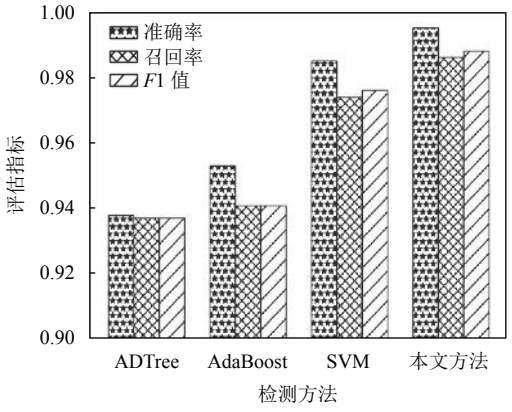

图 12 BiLSTM 与机器学习算法对比

(2) 深度学习对比实验

为了验证本文提出的方法效果更优, 将本文的方法与 TextCNN, RCNN 和 LSTM 进行对比实验, 结果如表 8 和图 13 所示, TextCNN 常用于文本分类, 其将一个句子中的每个词当作一个一维向量, 使用卷积的方式去获取句子的特征, 但是其网络结构忽略了句子之间的结构信息, 没有考虑到恶意代码中的函数、数组、变量的调用关系, 分类效果没有本文方法中的效果好. RCNN 则是在经过一个双向 RNN 后再经过一个最大池化层, 试图找到最重要的潜在语义因素, 然而在恶意代码检测领域, 无论是最重要的因素还是次重要的因素都可能对我们的安全造成威胁, 因此, 在本实验中, RCNN 的准确率不及本文方法. LSTM 没有充分的获取到上下文的相关信息, 因此它的误报率明显高于本文方法.

表 8 BiLSTM 与深度学习方法对比结果

检测方法	准确率(%)	*RE*	*FPR*	*F*1
TextCNN	96.6	0.88	0.032	0.927
RCNN	96.82	0.915	0.026	0.935
LSTM	97.34	0.934	0.021	0.947
本文方法	99.52	0.986	0.014	0.988

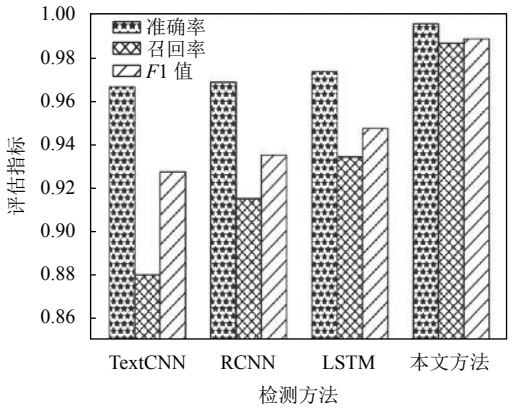

图 13 不同深度学习方法对比

综上所述, 与其他机器学习方法和深度学习方法相比, 使用本文提出的 BiLSTM 更适合对 JavaScript 恶意代码进行检测, BiLSTM 模型明显优于其他方法.

4 结论

本文在现有检测技术的基础上, 提出一种使用双向长短时记忆网络检测 JavaScript 恶意代码攻击的方法. 为了提高检测的精度, 首先使用解码技术对 JavaScript 进行反混淆的处理, 并利用深度学习工具 Word2Vec 将代码向量化作为神经网络的输入. 最后, 使用深度学

习算法 BiLSTM 对 JavaScript 恶意代码进行分类, 提高检测的性能. 和机器学习算法相比, BiLSTM 检测模型无需人工提取代码特征, 其会自动的提取出跟恶意代码检测相关的特征. 和深度学习其他算法相比, BiLSTM 检测模型更加充分的获取到代码的上下文相关信息, 模型的检测性能更加的优异, 通过测试检测模型并与机器学习和深度学习分类算法进行比较, 验证了本文提出的基于 BiLSTM 的 JavaScript恶意代码检测模型的可行性和有效性. 结合上述实验结果, 本文方法可以应用到实践中去, 将本文的提出方法以插件的形式嵌入到浏览器中, 如谷歌浏览器, 用户在不经意点开某些恶意站点如色情网站, 虚假的购物网站, 赌博网站时会发出警告来提醒用户有潜在的安全威胁.

参考文献

1 北京瑞星网安技术股份有限公司. 瑞星 2019 年中国网络安全报告与趋势展望. 信息安全研究, 2020, 6(2): 98–107. [doi: 10.3969/j.issn.2096-1057.2020.02.001]

2 Rad BB, Masrom M. Metamorphic virus variants classification using opcode frequency histogram. Proceedings of the 14th WSEAS International Conference on Computers: Part of the 14th WSEAS CSCC Multiconference. Stevens Point, WI, USA. 2011. 147–155.

3 Khan N, Abdullah J, Khan AS. Defending malicious script attacks using machine learning classifiers. Wireless Communications and Mobile Computing, 2017, 2017: 5360472.

4 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. 计算机应用与软件, 2015, 32(7): 293–296. [doi: 10.3969/j.issn.1000-386x.2015.07.070]

5 Pan JK, Mao XG. Obfuscated malicious JavaScript detection by machine learning. Proceedings of the 2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics (AMEII 2016). 2016. 805–815.

6 Likarish P, Jung E, Jo I. Obfuscated malicious javascript detection using classification techniques. Proceedings of the 4th International Conference on Malicious and Unwanted Software. Montreal, QC, Canada. 2009. 47–54.

7 Wang WH, Lv YJ, Chen HB, *et al*. A static malicious Javascript detection using SVM. Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering. 2013. 214–217.

8 Li YC, Ma R, Jiao RH. A hybrid malicious code detection method based on deep learning. International Journal of Software Engineering and Its Applications, 2015, 9(5): 205–216. [doi: 10.14257/ijseia.2015.9.5.20]

9 Cui ZH, Xue F, Cai XJ, *et al*. Detection of malicious code variants based on deep learning. IEEE Transactions on Industrial Informatics, 2018, 14(7): 3187–3196. [doi: 10.1109/TII.2018.2822680]

10 Wu F, Wang JG, Liu JQ, *et al*. Vulnerability detection with deep learning. Proceedings of the 3rd IEEE International Conference on Computer and Communications. Chengdu, China. 2018. 1298–1302.

11 Fang Y, Huang C, Liu L, *et al*. Research on malicious JavaScript detection technology based on LSTM. IEEE Access, 2018, 6: 59118–59125. [doi: 10.1109/ACCESS.2018.2874098]

12 Choi YH, Kim TG, Choi SJ. Automatic detection for JavaScript obfuscation attacks in web pages through string pattern analysis. International Journal of Security and Its Applications, 2009, 4(2): 13–26.

13 Visaggio CA, Pagin GA, Canfora G. An empirical study of metric-based methods to detect obfuscated code. International Journal of Security and Its Applications, 2013, 7(2): 59–74.

14 Lu G, Coogan K, Saumya Debray. Automatic simplification of obfuscated JavaScript code (Extended Abstract). Proceedings of the 6th International Conference on Information Systems, Technology and Management. Grenoble, France. 2012. 348–359.

15 马洪亮, 王伟, 韩臻. 混淆恶意 JavaScript 代码的检测与反混淆方法研究. 计算机学报, 2017, 40(7): 1699–1713. [doi: 10.11897/SP.J.1016.2017.01699]

16 Zeng Y, Yang HG, Feng YS, *et al*. A convolution BiLSTM neural network model for Chinese event extraction. Proceedings of the 5th CCF Conference on Natural Language Processing and Chinese Computing, NLPCC 2016, and 24th International Conference on Computer Processing of Oriental Languages. Kunming, China. 2016. 275–287.

17 马洪亮, 王伟, 韩臻. 面向 drive-by-download 攻击的检测方法. 华中科技大学学报 (自然科学版), 2016, 44(3): 6–11.