doi: 103969/j. issn. 0490-6756. 2019. 02. 015

基于语义分析的恶意 JavaScript 代码检测方法

邱瑶瑶[1], 方 勇[2], 黄 诚[2], 刘 亮[2], 张 星[3]
(1. 四川大学电子信息学院, 成都 610065; 2. 四川大学网络空间安全学院, 成都 610065;
3. 北京神州绿盟信息安全科技股份有限公司, 北京 100089)

摘 要: JavaScript 是一种动态脚本语言,被用于提高网页的交互能力. 然而攻击者利用它的动态性在网页中执行恶意代码,构成了巨大威胁. 传统的基于静态特征的检测方式难以检测经过混淆后的恶意代码,而基于动态分析检测的方式存在效率低等问题. 本文提出了一种基于语义分析的静态检测模型,通过提取抽象语法树的词法单元序列特征,使用 word2vec 训练词向量模型,将生成的序列向量特征输入到 LSTM 网络中检测恶意 JavaScript 脚本. 实验结果表明,该模型能够高效检测混淆的恶意 JavaScript 代码,模型的精确率达 99. 94%,召回率为 98. 33%.

关键词: 恶意 JavaScript 代码检测; 抽象语法树; 长短时记忆网络; 深度学习
中图分类号: TP391. 1 **文献标识码**: A **文章编号**: 0490-6756(2019)02-0273-06

Syntax-based malicious JavaScript code detection method

QIU Yao-Yao[1], *FANG Yong*[2], *HUANG Cheng*[2], *LIU Liang*[2], *ZHANG Xing*[3]
(1. College of Electronics and Information Engineering, SichuanUniversity, Chengdu 610065, China;
2. College of Cybersecurity, Sichuan University, Chengdu 610065, China;
3. Nsfocus Information Technology Company, Limited, Beijing 100089, China)

Abstract: JavaScript is a dynamic scripting language originally designed to improve the interactive capability of web pages. However, attackers use this peculiarity to execute malicious code on web pages, posing a huge threat. The obfuscated malicious code is difficult to detect using the traditional method based on static features, and the method based on dynamic analysis is inefficient. This paper proposes a static detection model based on semantic analysis. Specifically, the lexical unit sequence is extracted from abstract syntax trees; then the word vectors are generated by word2vec based on the lexical unit sequence; finally the generated vectors are input into the LSTM network to detect malicious JavaScript. Experiments show that the model can effectively detect obfuscated malicious JavaScript code and improve the detection speed, with a precision of 99. 94% and recall of 98. 33%.

Keywords: Malicious JavaScript code detection; Abstract syntax tree; Long short-term memory; Deep learning

收稿日期: 2018-12-13
基金项目: CCF-绿盟科技"鲲鹏"基金(2018008)
作者简介: 邱瑶瑶(1994-), 女, 硕士研究生, 研究方向为信息对抗技术. E-mail: fionsky911@gmail. com
通讯作者: 黄诚. E-mail: opcodesec@gmail. com

1 引 言

随着互联网的普及,人们的生活、娱乐以及工作已经离不开网络及其各类应用. JavaScript 是 Web 开发中最受欢迎的脚本语言之一,丰富了客户端的交互功能. 然而攻击者利用 JavaScript 的动态性将恶意代码嵌入到网页中以达到网页挂马和网页重定向等目的,对用户造成了巨大的威胁. 此外,攻击者还通过多样的混淆技术,例如随机混淆、编码混淆和逻辑结构混淆技术等[1]大幅提高检测的难度,传统的基于特征匹配的检测技术存在准确率低等问题. 如何有效地检测页面中的恶意 JavaScript 代码已成为亟需解决的问题.

目前,网络安全学者与专家提出了多种恶意 JavaScript 检测方法. 为了对抗混淆的恶意代码,Cova 等[2]提出了代码中的重定向次数、函数定义和调用频率、动态函数(例如 eval 和 setTimeout)执行次数等检测特征. Morishige 等[3]通过提取代码中分割的 URL 来检测混淆的恶意代码. 文献[4-6]则通过分析 JavaScript 代码的语义结构来构建检测模型. Aurore 等[6]提出 JaST 模型,将代码转化成抽象语法树,结合 N-gram 提取序列频率作为检测特征. 模型针对邮件中恶意 JavaScript 代码有不错的检测效果,但是对于 Web 页面中的脚本代码效果并没有那么突出. Yao 等[7]使用堆叠去噪自编码器从 JavaScript 代码中提取特征,再引入逻辑回归作为分类器判别恶性和良性的代码. 不过由于现在开发者为了减少 JavaScript 文件的体积会压缩代码,模型可能会将这类正常代码误判为恶意代码而导致高假阳率.

总体来说,基于静态分析的检测方法检测效率高,资源占用小,依赖特征选择和算法模型. 基于动态分析的检测方法需要消耗更多的系统资源且牺牲一定的时间. 对于混淆的恶意代码,在反混淆阶段无法完全做到自动化[8]. 纯代码文本层面的特征提取检测手段难以检测混淆代码,但是混淆前后的恶意代码在语义结构上具有一定的相似性. 因此文章模型的特征选择从代码的语义结构入手,将源代码转化成词法单元序列,通过 word2vec 训练词法单元的词向量模型,获取词法单元序列文本的词向量特征,结合深度学习构建恶意 JavaScript 代码检测模型. 实验表明,该模型检测准确率高且时间代价小,精确率达 99.94%,召回率为 98.33%.

2 恶意 JavaScript 代码检测模型

恶意 JavaScript 代码检测模型的训练流程和检测流程如图 1 所示. 首先,将训练样本解析成抽象语法树,对抽象语法树的节点进行深度优先遍历,得到词法单元序列保存成文本文件. 然后通过 word2vec 模型训练词法单元序列文本生成词向量. 最后,利用深度学习算法分析并检测恶意的 JavaScript 代码.

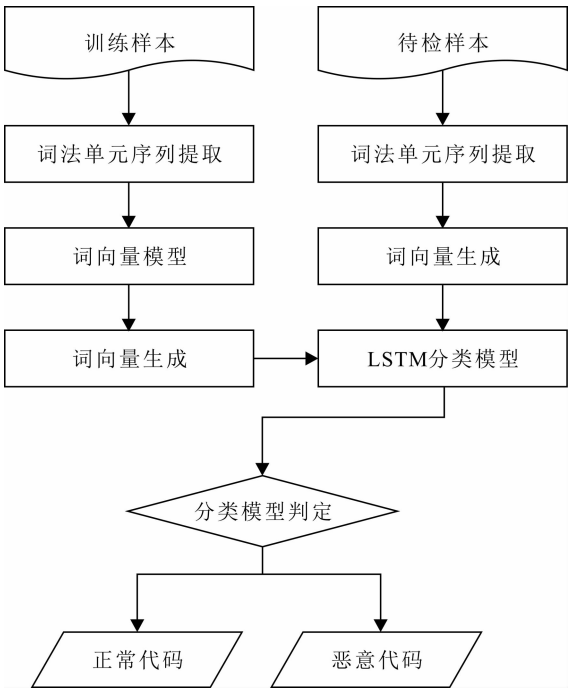

图 1 模型训练与检测流程图

Fig. 1 Model training and testing flow chart

2.1 JavaScript 代码的词法单元序列生成

抽象语法树(Abstract Syntax Tree, AST)是源代码抽象语法结构的树状表示[9]. JavaScript 源代码转化成抽象语法树的过程如图 2 所示. 首先,通过词法分析(Lexical analysis)将源代码转化成对应的 Token 流,然后,通过语法分析(Parse analysis)分析 Token 流中语法单元的关系生成抽象语法树.

文章使用 Esprima.js 将 JavaScript 代码解析成抽象语法树. Esprima.js 是一个高性能的 JavaScript 解析工具,会产生 69 个不同类型的词法单元. 首先通过 Esprima.js 对恶意代码进行解析,代码被映射成为一棵抽象语法树,不同的语句被映射成为不同类型的节点,之后通过深度后序优先遍历抽象语法树的节点,最终可以得到词法单元序列文

本，用于之后的训练. 下面展示了从代码中提取词法单元序列文本的实例. 如下是一段 JavaScript 代码.

```
var a="alert(1)". replace(/. +/,eval);
```

该语句经过 Esprima. js 解析后，生成如图 3 所示的抽象语法树.

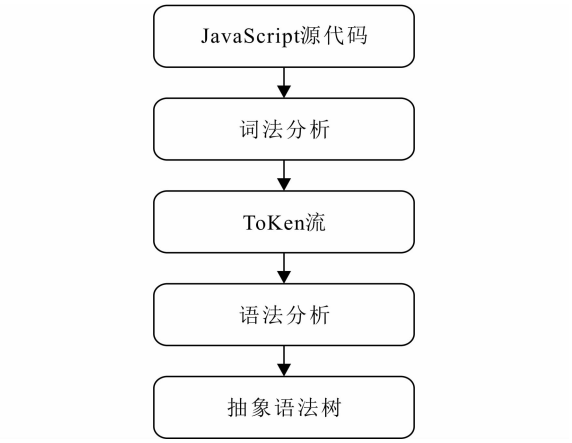

图 2　抽象语法树生成流程图
Fig. 2　Abstract syntax tree generation process

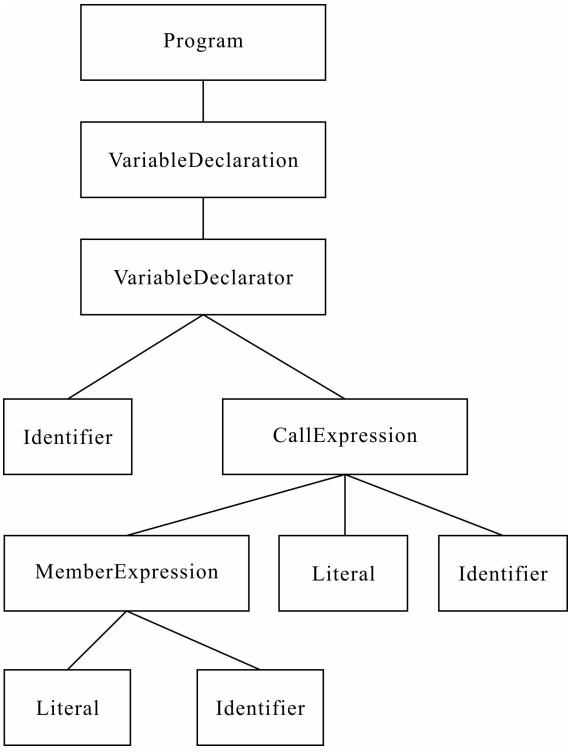

图 3　抽象语法树示例
Fig. 3　An example of abstract syntax tree

随后，我们深度后序优先遍历该语法树，得到以下序列，如表 1 所示.

表 1　词法单元序列示例
Tab. 1　An example of lexical units sequence

#	词法单元类型
1	Identifier
2	Literal
3	Identifier
4	MemberExpression
5	Literal
6	Identifier
7	CallExpression
8	VariableDeclarator
9	VariableDeclaration
10	Program

2.2　基于 word2vec 的词向量训练

2013 年，Google 团队开源了一款词向量特征训练工具 word2vec[10,11]. 它将文本内容的处理转化为 K 维向量空间中的向量运算，利用向量空间上的相似度表示文本语义上的相似度. Word2vec 工具主要包含两个训练模型，分别是 Skip-gram 模型和连续词袋模型（Continuous Bag Of Words，CBOW），此外，还提供了负采样（Negative Sampling）和 Hierarchical Softmax 两种优化方法来提升训练效率.

CBOW 基于上下文来预测当前词语的概率，而 Skip-gram 模型则是根据当前词语来预测上下文的概率. 虽然 Skip-gram 模型训练速度不如 CBOW 模型快，但是它能更好的处理出现频率较低的单词或元素，具有更高的准确率.

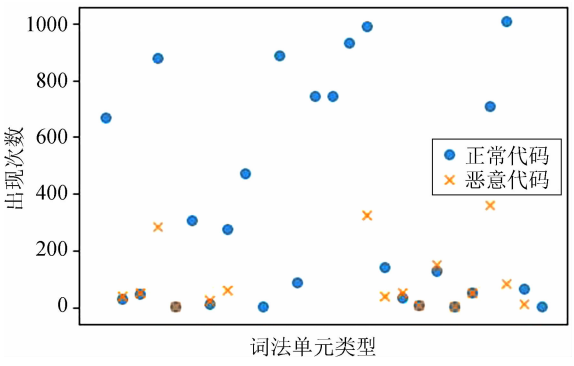

图 4　两类样本低频的词法单元分布情况
Fig. 4　The low-frequency lexical units distribution of two kinds of samples

在用于训练词向量的数据集中，提取的词法单元类型出现频率各有不同，从几十次到上万次不等. 正常代码和恶意代码在低频词的分布上也有一些差异，如图 4 展示了两类代码里低频的词法单元

分布情况,恶意代码中出现频率极低的一些词法单元在正常代码中出现频率较高,而在正常代码中的低频词法单元在恶意代码中出现频率略有上升. 为了更好的处理低频的词法单元类型,我们选择 Skip-gram 训练词向量模型.

2.3 基于 LSTM 的检测模型构建

利用 word2vec 算法得到序列的词向量后,输入到基于长短时记忆网络[12](Long Short-Term Memory,LSTM)的 JavaScript 恶意代码检测模型中. 循环神经网络模型(Recurrent Neural Networks,RNN)是一种节点定向连接成环的神经网络,处理单元之间除了内部的反馈连接还有前馈连接,对于处理一定长度的不分段的文本具有不错的效果. 然而,由于 RNN 只能记忆部分序列,在处理长序列时存在一些缺陷,会带来梯度消失或者梯度膨胀的问题.

LSTM 模型是 RNN 的一种特殊结构,用于解决长距离依赖问题. LSTM 通过增加单元状态(cell state)来存储长期状态,还引入了门(gate)结构来去除或者增加信息到单元状态的能力. LSTM 具有三个门结构,即输入门、输出门和遗忘门. 随着信息进入到模型中,该结构会将符合规则的信息留下,遗忘不符合的信息,以此来控制单元状态. 输入门、遗忘门和输出门的计算公式如下所示.

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

$$f_t=\sigma(W_f\cdot[h_{t-1},x_t]+b_f) \tag{2}$$

$$o_t=\sigma(W_o\cdot[h_{t-1},x_t]+b_o) \tag{3}$$

其中,W 代表权重向量;b 为偏移向量;σ 是 sigmoid 函数,决定通过的信息量. LSTM 的输出结果 h_t 由输出门和单元状态 C_t 决定.

$$\tilde{C}_t=\tan h(W_c\cdot[h_{t-1},x_t]+b_c) \tag{4}$$

$$C_t=f_t\cdot C_{t-1}+i_t\cdot\tilde{C}_t \tag{5}$$

$$h_t=o_t\cdot\tan h(C_t) \tag{6}$$

通过 word2vec 训练得到词向量字典后,利用该字典将词法单元序列文本生成对应的词向量矩阵,用其训练基于 LSTM 的恶意 JavaScript 代码检测模型. LSTM 模型包括一个 LSTM 层和全连接层,此外还通过 Dropout 层防止模型的过拟合,输出层使用 Sigmoid 激活函数对样本进行分类,得到预测标签.

3 实验与分析

3.1 实验数据及环境

本文从 Alexa 公布的排名网站收集正常样本,恶意的 JavaScript 代码样本则来自开放平台 VxHeavens、Github 和 Junjie[13] 公开的数据,最终获得样本共计 10674 份,其中正常样本为 8189 份,恶意样本为 2485 份. 实验采用五折交叉验证进行模型评估.

实验环境的软硬件配置信息如表 2 所示,首先在 Node. js 环境下,使用 Esprima. js 库来将代码转换成抽象语法树来提取词法单元序列. 之后的模型训练与测试则是基于 Python 语言,词向量训练采用了 Genism 库,LSTM 模型使用 Keras 库实现.

表 2 实验环境配置

Tab. 2 Experimental environment configuration

名称	信息
操作系统	Ubuntu 16. 04. 3 LTS
系统配置	CPU:Intel i7−7700,内存:16G GPU:GeForce GTX 1060 6G
Python 库	Genism、Karas、Scikit-learn、Matplotlib
Node. js	Node. js 9. 4. 0

3.2 实验评价指标

本文的恶意 JavaScript 代码检测是一个二分类问题,实验的结果使用以下 4 种类型表示.

(1) 真阳性(Ture Positive,TP),表示预测值为恶意代码并且样本标签为恶意代码;

(2) 假阴性(False Negative,FN),表示预测值为正常代码但是样本标签为恶意代码;

(3) 假阳性(False Positive,FP),表示预测值为恶意代码但是样本标签为正常代码;

(4) 真阴性(True Negative,TN),表示预测值为正常代码并且样本标签为正常代码.

本文模型的评估指标采用精确率(P),召回率(R),准确率(ACC)和 F_1 值,公式如下所示.

$$P=\frac{TP}{TP+FP} \tag{7}$$

$$\mathrm{R}=\frac{TP}{TP+FN} \tag{8}$$

$$ACC=\frac{TP+TN}{TP+TN+FP+FN} \tag{9}$$

$$F_1=\frac{2PR}{P+R} \tag{10}$$

3.3 实验设计与结果分析

在 LSTM 模型的训练过程中,我们将词法单元序列的输入长度设置为 512,隐藏神经元个数设置为 64,并采用 Adam 模型优化算法. 为了验证本文模型的有效性,设计了以下两组实验.

3.3.1　实验一　在输入相同训练特征的情况下，实验对比了朴素贝叶斯、随机森林和 LSTM 三种算法的准确率、精确率、召回率和 F_1 值，如表 3 所示.

表 3　不同分类算法检测结果

Tab. 3　Detection results of different classifiers

分类算法	准确率(%)	精确率(%)	召回率(%)	F_1 值(%)
朴素贝叶斯	95.46	97.69	82.47	89.43
随机森林	98.39	97.88	95.24	96.54
LSTM	99.60	99.94	98.33	99.13

由表 3 可看出，朴素贝叶斯的效果最差，召回率仅达到 82.47%，而 LSTM 算法的各项性能指标都优于其他两类算法，经过 30 轮训练后模型精确率达到 99.94%，召回率达到 98.33%，检测效果最优. 三类算法的 ROC 曲线如图 5 所示.

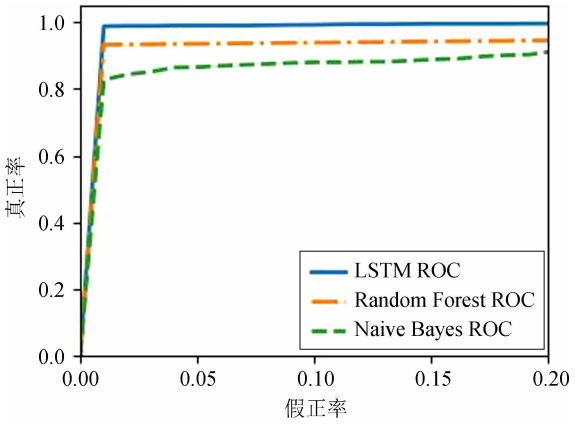

图 5　三类算法的 ROC 曲线

Fig. 5　ROC curve of different models

此外，将本文所提的恶意 JavaScript 代码检测模型(SJD)与文献[3]和文献[7]中的检测模型做了对比，结果如下所示. 本文模型在准确率、精确率、召回率等多个方面都有所提升，由此说明了模型的有效性.

表 4　与其他模型对比

Tab. 4　Compare with other models

分类算法	准确率(%)	精确率(%)	召回率(%)	F_1 值(%)
SdA-LR[7]	94.82	94.90	94.80	94.80
文献[3]	97.10	96.99	98.07	97.53
SJD	99.60	99.94	98.33	99.13

3.3.2　实验二　实验计算了 300 个数据样本从解析生成词法单元序列、加载训练好的模型、生成词向量特征到模型预测各个阶段执行所花费的总时间，具体情况如表 5 所示.

表 5　300 个样本在各阶段执行花费时间

Tab. 5　Processing time for 300 samples for different stage

阶段	花费时间(s)
解析生成词法单元序列	2.433
加载模型	1.085
生成词向量	1.338
LSTM 模型验证	0.659
总计	5.515

300 个样本检测共花费时间 5.515 s，平均每个样本检测花费时间为 18.3 ms，对比文献[6]提出 JaST 的静态分析检测模型，本文提出的模型在检测效率上大幅提升.

表 6　模型检测花费时间对比

Tab. 6　Models comparison of processing time

模型	平均每个样本检测时间(ms)
JaST[6]	226.86
SJD	18.3

4　结　论

为了更好的解决混淆恶意代码的检测问题，本文提出了一种基于语义分析和深度学习的恶意 JavaScript 代码检测模型. 通过 Esprima.js 将代码转换成抽象语法树，再从抽象语法树中提取词法单元序列训练词向量，最后输入到 LSTM 网络中训练模型. 实验通过与其他算法进行对比分析，验证了本文提出的检测模型的可行性和有效性，且检测花费时间短. 由于本文所使用的检测模型限制了模型输入长度，对于较大的样本数据会造成一定的损失，如何优化词法单元序列文本的向量表达将是我们进一步研究的方向. 本文仅采用代码的词法单元序列特征，未来还将引入其他维度的特征构建检测模型.

参考文献：

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

[2]　Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript code [C]//Proceedings of the 19th inter-

national conference on World wide web. [s. l.]: ACM, 2010.

[3] Morishige S, Haruta S, Asahina H, *et al*. Obfuscated malicious javascript detection scheme using the feature based on divided URL [C]//Proceedings of the 2017 23rd Asia-Pacific Conference on Communications (APCC). [s. l.]: IEEE, 2017.

[4] Curtsinger C, Livshits B, Zorn B G, *et al*. ZOZZLE: Fast and precise In-Browser JavaScript malware detection [C]//Proceedings of the 20th USENIX Conference on Security. San Francisco: ACM, 2011.

[5] Kapravelos A, Shoshitaishvili Y, Cova M, *et al*. Revolver: an automated approach to the detection of evasive Web-based malware [C]//Proceedings of the 22nd USENIX Conference on Security. Washington: ACM, 2013.

[6] 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. [s. l.]: Springer, 2018.

[7] Wang Y, Cai W D, Wei P C. A deep learning approach for detecting malicious JavaScript code [J]. Security Communi Networks, 2016, 9: 1520.

[8] 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 [J]. 计算机学报, 2017, 40: 1699.

[9] Yue C, Wang H. Characterizing insecure javascript practices on the web [C]//Proceedings of the 18th international conference on World wide web. [s. l.]: ACM, 2009.

[10] MikolovT, Sutskever I, Chen K, *et al*. Distributed representations of words and phrases and their Compositionality [J]. Adv Neural Inf Process Sys, 2013, 26: 3111.

[11] 周顺先, 蒋励, 林霜巧, *et al*. 基于 Word2vector 的文本特征化表示方法 [J]. 重庆邮电大学学报: 自然科学版, 2018, 30: 272.

[12] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural Comput, 1997, 9: 1735.

[13] Wang J, Xue Y, Liu Y, *et al*. JSDC: A hybrid approach for JavaScript malware detection and classification [C]//Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. [s. l.]: ACM, 2015.

[14] Morishige S, Haruta S, Asahina H, *et al*. Obfuscated malicious javascript detection scheme using the feature based on divided URL [C]//Proceedings of the 2017 23rd Asia-Pacific Conference on Communications (APCC). [s. l.]: IEEE, 2017.

[15] Takata Y, Akiyama M, Yagi T, *et al*. Understanding evasion techniques that abuse differences among javascript implementations [C]//Proceedings of theInternational Conference on Web Information Systems Engineering. Berlin: Springer, 2017.

[16] 郑荣锋, 方勇, 刘亮. 基于动态行为指纹的恶意代码同源性分析 [J]. 四川大学学报: 自然科学版, 2016, 53: 793.

[17] 李道丰, 黄凡玲, 刘水祥, 等. 基于行为语义分析的 Web 恶意代码检测机制研究 [J]. 计算机科学, 2016, 43: 110.

[18] Stokes J W, Agrawal R, McDonald G. Neural classification of malicious scripts: A study with JavaScript and VBScript [J/OL]. (2018-03-15)[2018-09-12]. http://www. doc88. com/p-2505055864204. html.

引用本文格式:

中 文: 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: 273.

英 文: Qiu Y Y, Fang Y, Huang C, *et al*. Syntax-based malicious javascript code detection method [J]. J Sichuan Univ: Nat Sci Ed, 2019, 56: 273.