软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(7):2124−2138 [doi: 10.13328/j.cnki.jos.005490]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 Jalangi 的广告代码调用路径追踪*

许 蕾[1], 刘蕊成[2], 陈贵美[2], 赵 晨[2], 张卫丰[2]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](南京邮电大学 计算机学院,江苏 南京 210003)

通讯作者: 张卫丰, E-mail: zhangwf@njupt.edu.cn

摘 要: 随着互联网的迅猛发展,网络广告成为互联网最重要的商业模式之一.网络广告在促进互联网发展的同时,也带来了用户信息泄露、影响用户网页浏览体验等负面问题.为了对网络广告进行系统的研究,需要获取广告生成过程中完整的调用路径.由于加载到页面中的 JavaScript 文件量大、函数调用路径链路长、路径中的 JavaScript 代码经过了一定的压缩和混淆,因此很难通过静态方法获取网络广告调用路径.分析了动态广告生成的过程,对相关代码进行动态插桩,利用函数参数实现广告调用信息的传递,并记录下每个 iframe 内部的调用信息,通过匹配与合并多个 iframe 的信息,生成了完整的广告调用路径并确定了广告插入的操作方式.针对 21 个真实网站进行了实验,结果表明:该方法能够在不太影响性能的前提下,获取到静态方法无法获取到的广告动态加载过程信息并生成广告代码调用路径.

关键词: 动态插桩;调用路径;广告代码分析

中图法分类号: TP311

中文引用格式: 许蕾,刘蕊成,陈贵美,赵晨,张卫丰.基于 Jalangi 的广告代码调用路径追踪.软件学报,2019,30(7):2124−2138. http://www.jos.org.cn/1000-9825/5490.htm

英文引用格式: Xu L, Liu RC, Chen GM, Zhao C, Zhang WF. Tracking call path of online advertisement based on Jalangi. Ruan Jian Xue Bao/Journal of Software, 2019,30(7):2124−2138 (in Chinese). http://www.jos.org.cn/1000-9825/5490.htm

Tracking Call Path of Online Advertisement Based on Jalangi

XU Lei[1], LIU Rui-Cheng[2], CHEN Gui-Mei[2], ZHAO Chen[2], ZHANG Wei-Feng[2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](School of Computer Science, Nanjing University of Posts & Telecommunication, Nanjing 210003, China)

Abstract: Online advertisement (short as ad) has become one of the most important business patterns, with the rapid development of Internet. Online advertisements are main economic sources of Web applications, but the negative affect is that ads may leak users' privacy, or increase loads of browsers' performance. In order to study online ads systematically, it is necessary to obtain a complete call path in the whole generating process. However, since the sizes of the loaded JavaScript files are usually large, the function call path is long, and even worse, the JavaScript code in the path is compressed and confused, it is difficult to get the call path of the online ads through static analysis method. This study tracks the call path of online ads dynamically, namely instruments the relevant codes at first, then uses the function parameters to transmit the call information and records the internal call information in each iframe, finally, by matching and merging the information in multiple iframes, a complete ad call path about the generating process of online ads is generated. The experiment focused on 21 real websites, and the results show that: the proposed method can obtain the dynamic loading information of ads and generate the whole call paths, which are impossible for static methods, and the overhead is acceptable.

Key words: dynamically instrument; call path; ad code analysis

* 基金项目: 国家重点基础研究发展计划(973)(2014CB340702); 国家自然科学基金(61272080, 91418202, 61403187)

Foundation item: National Basic Research Program of China (973) (2014CB340702); National Natural Science Foundation of China (61272080, 91418202, 61403187)

收稿时间: 2017-06-30; 修改时间: 2017-08-29; 采用时间: 2017-11-21

1 背景介绍

目前,互联网中最成熟的商业模式之一是以广告为基础的商业模式.据悉,谷歌公司 96%的收入都来自于网络广告.当用户使用搜索引擎时,除了返回与查询词相关的信息外,还会出现广告,一旦用户点击了这些广告,广告的发布者需要付费给搜索引擎,与此同时,用户有可能成为广告商品的买家.这样,在线广告把商家、用户、应用开发者有机地联系在一起,共同构成了互联网应用的生态系统.

为了便于发布、传播广告并提高用户点击率,广告提供商通常使用 JavaScript 代码动态生成广告页面,并收集用户交互、浏览器环境等信息.作为第三方代码,这些广告代码可能给网站以及用户带来极大的安全隐患.例如,这类广告代码可以获得用户在浏览器上的行为,从而获得用户的访问信息,增加了用户隐私信息泄露的可能性,同时也打破了网站本身的完整性.更可怕的是,一些恶意广告可能利用网页中的漏洞,偷偷将恶意软件安装到用户的电脑中.研究显示,网络上每天出现约 130 万个恶意广告,它们中的大部分都会下载恶意软件并且进行一定的伪装,以绕过安全软件的检测[1−4].

现今大部分网站都使用 JavaScript 代码动态显示信息.事实上,几乎所有的恶意广告都是通过 JavaScript 代码加载的.因此,为了提高网页的安全性,可以直接禁止所有 JavaScript 代码的加载.但是由于包含 JavaScript 代码的应用十分普遍,网页中几乎所有的动态效果和交互性强的显示效果都需要使用 JavaScript 代码,网站应用越来越趋向于使用 JavaScript 代码,以吸引用户和增强表现力.因此,直接屏蔽 JavaScript 代码会导致网页的交互性降低,不能因噎废食.

为了识别恶意广告,需要追踪广告代码调用路径.但追踪广告代码路径具有很大的挑战,这是由于:(1) 网页中混杂了大量 HTML 文件、JavaScript 脚本文件、CSS 文件,又由于有的脚本文件根本不执行,很难识别出广告相关的 JavaScript 文件;(2) JavaScript 脚本文件中包含大量函数,每个函数之间的调用关系错综复杂,形成了很长的函数调用链路,这些函数大都分布在不同的 JavaScript 文件中,也不容易识别函数的调用关系;(3) 为了保证用户的浏览体验以及代码本身的隐私和安全性,开发者往往对 JavaScript 代码进行了压缩,使得整体的代码可读性变差,导致 JavaScript 代码中函数调用链的获取工作更加困难.

为此,我们计划使用动态分析方法获取广告的调用路径.由于网页广告的调用过程实质上是网站主通过广告联盟获取广告相关的 JavaScript 代码,因此我们的工作可以认为是从广告联盟的代码中获取广告调用路径.本文的主要工作如下:(1) 对广告相关的 JavaScript 脚本进行分析,发现动态广告的生成特点;(2) 根据生成特点,使用动态插桩工具对网页进行插桩,以获得广告的调用路径;(3) 对 iframe 内部调用路径和跨 iframe 调用路径进行不同处理,获得完整的调用路径;(4) 通过实验分析以及与静态分析方法的对比,说明本文方法的有效性:能够在有限增加访问时间(2~10X)的前提下,获取到静态方法无法获取到的广告动态加载过程信息,并生成广告代码调用路径.

本文第 2 节通过实例说明广告调用路径获取的困难性.第 3 节设定规则使用动态插桩工具对广告相关的调用路径进行追踪.第 4 节分别介绍对调用路径中 iframe 内部和跨 iframe 调用路径的获取.第 5 节是实验部分,针对 21 个真实网站进行广告代码调用路径的追踪并给出具体的统计结果.另外,还与静态分析方法进行了对比,说明本文方法的有效性.第 6 节为相关工作,包括互联网广告研究和 JavaScript 程序分析.第 7 节总结全文并指出后续可开展的工作.

2 动态广告生成示例

网页中动态广告的加载和生成是一个复杂的过程,其传播路径可能涉及多个 JavaScript 脚本文件.本节我们用一个示例描述网页中动态广告的加载过程,并说明广告调用路径获取的困难所在.

以 www.gamefaqs.com 为例,图 1 虚线以上区域展示了该网站主页加载完毕后的代码及截图,红色椭圆区域是由 Google 提供的动态广告;虚线以下区域显示广告加载的动态过程,包括多个 JS 文件的调用.在该应用中,为了个性化推送广告和躲避检测,源代码经过了一些混淆处理.为了方便阅读,图中所示代码已经进行了一些简化处理.其广告加载的详细步骤描述如下.

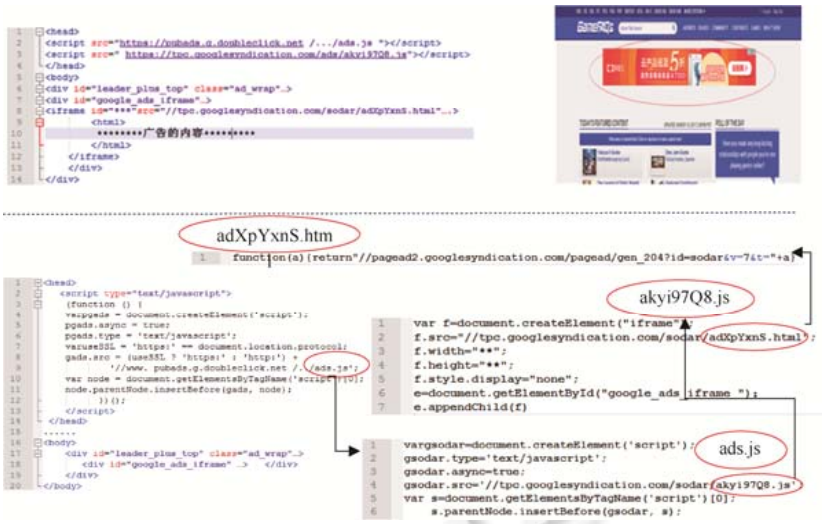

Fig.1 Example of ad’s dynamic loading

图 1 动态广告加载过程示例

(1) 浏览器加载主页,加载后由服务器返回的代码如图 1 中虚线以下区域的左半部分第 1 行~第 20 行所示.当第 3 行~第 14 行中的脚本语言执行后,一个新 script 标签及其包含的 JS 文件会加载到页面中(ads.js).

(2) 浏览器解析新增加的 script 标签并执行来自 pubads.g.doubleclick.net 中的 ads.js 文件.该文件第 4 行又向页面中插入新的 script 标签并执行 JS 文件 akyi97Q8.js.

(3) 浏览器加载来自 tpc.googlesyndication.com 中的 akyi97Q8.js 文件,该文件创建了一个 iframe 标签(文件中第 1 行),并在第 3 行中引用 tpc.googlesyndication.com 的 adXpYxnS.html 文件,并把该 iframe 嵌套到 id 为 google_ads_iframe 的 div 下(第 6 行、第 7 行).

(4) 浏览器从 tpc.googlesyndication.com 加载 adXpYxnS.html 文件.该 HTML 文件触发了广告内容相关函数的执行,该函数返回广告相关的内容(pagead2.googlesyndication.com).

(5) 最终,浏览器加载并显示实际的广告.页面截图和加载完毕后的源代码如图 1 中虚线上部所示.

从图 1 所示例子可以看出,网页中广告相关的 JavaScript 调用路径难以获得有以下几个原因.

(1) 由于网页中混杂着大量的 HTML 代码、JavaScript 脚本文件、CSS 文件,又由于有的脚本文件根本不执行,这给广告相关 JavaScript 文件的识别工作带来了相当大的困难.

例如,网站 www.gamefaqs.com 中的 measure.js 存在于网页中,却没有被执行,也难以判断该 JS 文件是否是广告相关代码.

(2) JavaScript 脚本文件中包含大量函数,包括匿名函数,各函数之间的调用关系错综复杂,形成了很长的函数调用路径.这些函数大都分布在不同的 JavaScript 文件中,这对观察和识别函数的调用关系造成了不便.

例如,图 1 中存在的文件调用路径为 ads.js→akyi97Q8.js→adXpYxnS.html,同时还有众多其他文件没有显示出来,要从中找出这些调用路径难度很大.

(3) 另外,为了保证用户的浏览体验以及代码本身的隐私和安全性,HTML 页面中的 JavaScript 代码通常会被压缩或混淆,使得整体的代码可读性变差,导致 JavaScript 文件中函数调用链的获取工作更加困难重重.

例如,图 1 所示 www.gamefaqs.com 网站的 ads.js 文件包含丰富的信息,实际找到的文件却只有 1 行代码,但却有上千列.这些代码压缩或混淆在一起,可读性极差,也进一步增加了调用路径获取的难度.

3 使用 Jalangi 获取广告调用路径

本节我们通过使用动态插桩工具 Jalangi 对包含 JavaScript 代码的网页进行插桩,并针对 JavaScript 函数进

行若干特殊处理,以获取广告的调用路径.

3.1 动态插桩工具Jalangi

Jalangi[5]是美国加州大学伯克利分校在 2013 年开发实现的 JavaScript 动态分析框架,能够对前端和后端 JavaScript 进行动态分析,允许监控每个 JavaScript 程序的操作以及编写自己的程序分析代码;通过记录-回放的机制来实现 JavaScript 代码的插桩,并通过影子执行的方法运行自定义的动态分析脚本.Jalangi 提供了丰富的分析 API,以实现各种动态程序分析.当使用 Jalangi 对网页进行分析时,浏览器通过使用一个代理服务器向网页所在服务器发送请求,并获得整个原始页面的 JavaScript 脚本文件;接着,Jalangi 对获得的脚本文件进行插桩.

Jalangi 在网页加载前对原始代码进行插桩,并生成插桩后代码.

```
var a=b                                  //插桩前的 JavaScript 代码
var a=J$.W(9, ‘a’, J$.R(5, ‘b’, b), a)   //插桩后代码
```

插桩后代码调用了两个 Jalangi 回调函数 J$.W 和 J$.R,分别是变量赋值和变量读取的分析回调函数,变量名和变量值等参数会被传递给这两个函数.J$.R(5,‘b’,b)表示读取名称为‘b’的变量,其中数字 5 表示回调函数的标识符;J$.W(9,‘a’,J$.R(5,‘b’,b),a)表示对变量名为‘a’的变量赋值为 J$.R(5,‘b’,b)的返回值,其中数字 9 也表示回调函数的标识符.此时在浏览器中加载网页,会执行插桩后的 JavaScript 代码,且不会破坏页面的原有设置.

另外,还可以通过在工具 Jalangi 源程序中设定一系列规则,对 JavaScript 代码进行动态插桩,进而得到执行轨迹、变量取值等定制化信息.因此,我们可以设定规则对网页进行插桩,以获取广告相关的 JavaScript 脚本.

Jalangi 提供了回调函数 invokeFunPre 和 putFieldPre,它们分别在函数执行前和在对象属性赋值前进行分析操作.其中,invokeFunPre 函数包含几个重要的参数:(1) iid,当前回调函数执行时的唯一标识符,以数字类型表示;(2) f,指向当前函数对象;(3) base,如果函数 f 是某个对象的方法,则 base 是该对象的引用,否则为 null;(4) args,函数 f 的参数列表,以数组形式表示;(5) isConstructor,判断函数 f 是否为构造函数,以布尔形式类型表示.putFieldPre 函数包含几个重要的参数:(1) iid,当前回调函数执行时的唯一标识符,以数字类型表示;(2) base,赋值的属性隶属的对象;(3) offset,属性名;(4) val,在 base[offset]中存储的值,即该对象属性存储的原始值.

3.2 广告调用路径相关定义

由于时间、精力有限,我们只追踪来自百度和谷歌的广告调用路径,且对于域名与当前网页所在服务器的二级域名不相同时才进行追踪,以下定义均建立在此前提之上.另外,根据同源策略,当 iframe 引用的 URL 与当前网页的域名不同时,DOM 元素是不可以被 iframe 所在页面内的 JavaScript 代码操作的,所以 document 内部的函数调用是指在一个 iframe 内的函数调用.

定义 1. document 内广告路径的开始节点 S 若函数调用时没有调用者或者调用者是 DOM 事件处理函数,则认为是广告路径的开始节点.另外,我们使用 unname 作为匿名函数调用的标识.

例如:图 1 中左侧代码片段的开始节点为第 3 行~第 15 行的匿名函数.

定义 2. document 内广告路径的结束节点 E 若使用函数 appendChild、insertBefore 以及 document.write 插入〈img〉、〈script〉、〈a〉、〈iframe〉标签,或使用 innerHTML 方式插入〈iframe〉标签,则将其作为广告路径的结束节点.

例如,图 1 右侧底部 ads.js 文件中的结束节点为第 6 行用 insertBefore 方式插入代码片段,图 1 右侧中部 akyi97Q8.js 文件中的结束节点为第 7 行用 appendChild 方式插入的〈iframe〉标签.

定义 3. document 内广告路径的中间节点 M 开始节点和结束节点之间的一系列函数调用均作为中间节点.

定义 4. document 内部的函数调用路径 P 记为 $S,M_1\ldots M_n,E,S',M'_1\ldots M'_n,E',S''\ldots$其中,$S$ 和 E 分别表示广告路径的开始节点和结束节点,$M_1\ldots M_n$ 表示中间节点,相邻节点表示调用关系,路径 P 上相邻节点的左边节点调用了右边节点.

$S,M_1\ldots M_n,E,S',M'_1\ldots M'_n,E'$表示若一条广告路径的结束节点为 script 脚本,且另一条广告路径的开始节点是此 script 脚本中的函数,则这两条广告路径是前后关联的.

例如,图 1 虚线下方左侧广告调用路径为%source%unnamed→insertBeforescriptads.js,虚线下部右侧底部广告调用路径为%source%unnamed→insertBeforescriptakyi97Q8.js,虚线下部右侧上部广告调用路径为%source%unnamed→appendChildiframe.

虚线下部代码片段组成一条 document 内部的函数调用路径,即%source%unnamed→insertBeforescriptads.js→%source%unnamed→insertBeforescriptakyi97Q8.js→%source%unnamed→appendChildiframe.

另外,我们注意到,JavaScript 函数中使用 setTimeout 和 setInterval 设置了定时执行函数,函数执行时调用者会变为浏览器.因此,为了广告链路的完整性,我们需要将调用 setTimeout 和 setInterval 的函数作为定时执行函数的父节点.

定义 5. 跨 document 的函数调用路径 AP 记为 $P_1 \ldots P_m$,其中,P_i 表示第 i 个 document 内部的广告路径,相邻节点 P_i,P_{i+1} 表示 P_i 结束节点产生了第 i+1 个 document.

例如:在图 1 虚线下右侧中部中使用 appendchild 插入一个 iframe,iframe 的 src 为 adXpYxnS.html,即这个 document 产生了另一个内容为 adXpYxnS.html 的 document.

3.3 改写Jalangi实现针对广告调用的动态插桩

我们考虑使用 Jalangi 对网页进行动态插桩,以获得广告的调用路径,具体方案如图 2 所示,其主要的工作流程是:在浏览器中设置代理,当我们访问一个网页时,代理服务器会获取从 Web 服务器返回的 HTML 网页文件及其引用的 JS 脚本文件,并调用 Jalangi 对 HTML 和 JS 文件中的 JavaScript 脚本代码进行插桩,然后记录执行轨迹并在浏览器上显示页面内容.

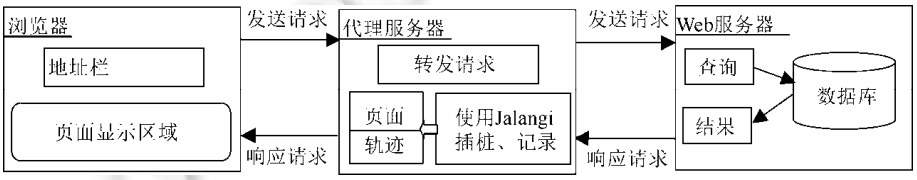

Fig.2 Flow chart of tracking ad’s call path

图 2 广告调用路径的追踪流程图

每执行一行 JavaScript 脚本代码,其执行轨迹均会被记录下来,通过分析将其关联起来,就可以追踪特定代码的调用路径.一个函数调用路径是一个函数调用的相关信息项的序列,每一项都包含:执行的函数;函数调用在代码中的位置,以行列数表示;函数调用所在的 js 文件的 url.函数调用路径中相邻的两项表示前一项中的函数调用了后一项中的函数.本文获取函数调用路径的方法是基于函数的参数对象 arguments 实现.一个函数的参数对象可以在该函数的作用域内被访问,也可以在这个函数直接调用的函数的作用域内被访问.参数对象是一个类数组对象,存放着包括本次函数调用传入的实参 arguments[0]~arguments[arguments.length-1]和对该函数的引用 arguments.callee.

我们通过改写 Jalangi 的 invokeFunPre 分析回调函数,在被分析代码中的函数调用之前,将函数调用路径作为一个额外的实参传递给它,则函数执行后该函数调用路径会出现在它的参数对象里;在其调用的任何函数作用域内,都可以通过 arguments.callee.caller.arguments 访问到前者的参数对象存放的函数调用路径.

一个函数对应的函数调用路径即以该次函数调用为终点的一系列函数调用.Jalangi 将原代码中的函数调用替换成一个包装函数的调用,该包装函数先调用 invokeFunPre 进行调用前分析,再调用原代码中的函数的插桩后版本,最后调用 invokeFun 进行调用后分析.原本传递给原函数的参数现在传递给包装函数,且该包装函数的调用者与原函数在插桩前原本的调用者是一致的;因为 Jalangi 将包装函数的参数对象传递给 invokeFunPre,所以我们可以获取到包装函数的调用函数的参数对象,提取后者对应的函数调用路径.与此同时,我们也可以通过 invokeFunPre 在一个函数的参数对象里初始化一个函数调用路径.

3.4 广告调用路径相关信息获取

为了捕捉广告调用路径的开始、过程和结束,我们设定一条广告路径由 0 或一串函数调用路径和一次插入某些特定 DOM 元素的操作组成.一个完整的网页广告生成流程包含不少于一个的广告路径.经过调研我们发现,广告调用路径涉及 JavaScript 中 3 类原生函数,对应于定义 2,即:使用 insertBefore 或 appendChild 插入 script、a、img、iframe 标签;使用 document.write 插入 script、a、img、iframe 标签;使用 innerHTML 插入 iframe 标签.另外,对于 setTimeout 和 setInterval 的处理,也需要特殊对待.

为了在函数调用过程中记录并传递执行轨迹(trace)信息,我们需要通过 invokeFunPre 中特定的函数参数记录并追踪 trace.为此,我们设置了特定的 trace 信息结构.

如图 3 所示,trace 中记录的 document 内广告调用路径 *P* 的信息结构为

%source%函数名称,位置,文件名,函数调用的位置||(函数名称,位置,文件名,函数调用的位置||)*插入元素的方式,插入的内容,插入内容的 src,位置,文件名,函数调用的位置

其中,“%source%”表示广告路径 *P* 的开始节点 *S*;“||”作为每次函数调用的分隔标识;“(函数名称,位置,文件名||)*”表示 0 个或多个中间节点 *M*;“插入元素的方式”表示结束节点的开始,包括了 insertBefore、appendChild、document.write 这 3 种方式.整个广告调用路径包括 1 个开始节点、0 个或多个中间节点、1 个结束节点.

由此构成 document 内的广告调用路径集合 CS 和所有 document 的广告调用路径集合 CSS:CS 的信息结构为 keyvalue 键值对,key 为文件名,每个 value 为一个二维数组,第 1 维表示 trace,第 2 维表示 trace 的每一项的具体信息;CSS 的信息结构为(url,P)*,其中,url 表示当前 document 所在的 url,*P* 表示当前 document 广告调用路径.

为了得到以上信息,我们需要在冗长、复杂、可能包含混淆代码的 JavaScript 文件中识别出广告相关的函数,并确定调用路径的起始点、更新以及终止点,具体处理过程如下所示.

Fig.3 Information structure of trace

图 3 执行轨迹 trace 的信息结构

3.4.1 广告相关函数调用路径初始化

发生函数调用时,如果满足以下任一条件,则构成广告相关的函数调用路径起始点.(1) 该函数来自于第三方且调用者为 null,即没有调用者;(2) 该函数来自于第三方,且它的调用者是 DOM 事件处理函数,如 window.onload.

需要说明的是,在引入的第三方脚本中,如果一个函数被网站服务商提供的脚本代码调用,则这次函数调用不被认为有关广告生成,而是第三方函数库的调用行为,因为现今主流的广告联盟不要求网站开发者调用它们的接口以生成广告.

图 4 是一个简单的广告调用路径代码示例:当用户访问网

```
1.window.onload=function(){
2.     A();
3.};
4.functionA(){
5.     setTimeout("B(x)",5000);
6.}
7.functionB(x){
8.     var s=document.createElement(script);
9.     document.body.appendChild(s);
10.}
```

Fig.4 Snippet code (C.js) of ad’s call path

图 4 广告调用路径代码示例 C.js

站 www.A.com 的主页时,页面主动加载了 C.js 文件,这段代码来自于 ads.B.com,即对于主页面来说,这段代码来自于第三方 JavaScript 库.因此,符合我们设定的广告调用路径起始点(满足条件 2).经过 Jalangi 插桩,可以记录下 trace:onload→A.

3.4.2 广告相关的函数调用路径更新

如果一个函数有调用者且调用者的参数对象中存在函数调用路径,则说明它已被标记为潜在广告路径的一部分,所以我们可以将它的函数调用路径和当前的函数调用相关信息拼接,形成更新后的函数调用路径.

对于函数调用路径更新,有一个特殊的情况是:当一个函数调用 setTimeout 延时执行或调用 setInterval 定时延时另一个函数时,从后者的参数对象中获取的调用者是 null,即从调用栈的角度来看,两者没有直接调用的关系.但是在调研中,我们发现:广告脚本中有很多地方使用了这种定时或延时执行函数.因此,需要捕获这样的调用关系.

如图 5 所示,在加载执行 A 函数时,使用 setInterval 每隔 5s 调用 B 函数,在 B 函数中调用了 C 函数,C 函数又做了一些操作.因此,函数的调用路径是 A→setInterval→B→C.但是由于 setInterval 设置的延时,使得调用路径在 B 之后发生中断,导致无法获取到 C 函数的调用信息.setTimeout 的情况与此类似.

因此,使用 setTimeout 或 setInterval 函数阻断了广告调用路径的传播,需要对这两个函数进行特殊处理.处理方法是:创建一个闭包,该闭包内存放着 setTimeout 函数或 setInterval 函数调用者参数对象的引用、要延时或定时执行的目标函数,其中,延时或定时执行的对象不再是原目标函数,而是这个闭包,这个闭包在运行时会先将 setTimeout 函数或 setInterval 函数调用者的参数对象传递给分析函数 invokeFunPre,然后再执行目标函数.图 6 展示了创建闭包的函数关键代码,其中,J$.invokeFun 函数会调用分析函数 invokeFunPre,并调用目标函数 fun;args 里只有 setTimeout 函数或 setInterval 函数调用者的参数对象,但它不作为目标函数真正的参数对象.

经过上述处理,在分析延时和定时的函数调用时,就可以获取到存放在设置延时和定时执行的函数的参数对象中的函数调用路径,所以能够捕获非直接的调用关系.仍然使用图 4 所示的例子,可以看到,A 函数在调用 B 函数时使用 setTimeout 等待了 5 000ms,这个操作会影响广告路径的延续,因此 Jalangi 在这里对 setTimeout 进行特殊处理,即调用 invokeFun 函数直接执行 B,以此对路径进行延续,这样获得的 trace 路径为 onload→A→B,没有丢失对 B 函数的调用.

```
1.window.onload=function(){
2.     A();
3.};
4.functionA(){
5.     setInterval("B()",5000);
6.}
7.functionB(){
8.     C();
9.}
10.functionC(){
11.     ……
12.}
```

Fig.5 Example of using setInterval

图 5 使用 setInterval 的代码示例

```
1.FunctionCall(…,fun,caller_args){
2.     var args={
3.          Callee:{
4.          Caller:{
5.          Arguments:caller_args
6.          }
7.     }
8.};
9.     ……
10.Returnfunction(){
11.     J$.invokeFun(…fun,args);
12.     }
13.}
```

Fig.6 Main code of creating closure

图 6 创建闭包的关键代码

3.4.3 广告相关的函数调用路径完成

广告调用路径可能会包含多次页面跳转(从一个 JS 文件链接到另一个 JS 文件),但最终会需要将包含广告内容的页面通过 DOM 操作插入到已有的 HTML 页面中,插入元素的类型可能是脚本元素〈script〉、内联框架元素〈iframe〉、超链接元素〈a〉以及图片元素〈img〉,这是因为网页广告的生成包括以下几种情况.

1) 广告有可能通过插入〈script〉脚本来加载外部源的 JS 脚本文件;

2) 插入〈a〉标签或〈img〉标签,由于很多网络广告通过插入一段链接指向某个真正广告内容的 url 地址,所以需要进行标记;插入〈img〉标签则为插入图片,这时往往也是广告生成的最后一步,即用图片展示广告内容,且可以起到与〈a〉标签一样的作用,因此也需要进行标记;

3) 插入〈iframe〉标签,由于网页上的广告位通常为一个既定的区域,且保证不受页面上其他脚本的 DOM 操作带来的影响,常用一个 iframe 引入一个广告页面.

另外,一个广告脚本可能会插入其他广告脚本,也可能会插入 iframe 显示广告页面,而且〈a〉和〈img〉元素都可以通过设置 src 属性实现广告链接.

使用 JavaScript 语言在页面 DOM 结构中插入 HTML 元素的方法有以下几种:调用 DOM 元素对象的 insertBefore 和 appendChild 方法、调用 document.write 函数和赋值 DOM 元素对象的 innerHTML 属性.

1) 对 insertBefore 和 appendChild 函数的分析

对于使用 insertBefore 和 appendChild 函数插入 iframe、script、a、img 标签的情况,我们认为该条路径已经到达了终止点,此时需要判断其 caller 的情况,如果有 caller 属性,即有调用者,而且其调用者不是事件处理函数,则对该条路径进行输出;如果 iframe、script、a 标签没有调用者,或者调用者就是事件处理函数,则对整条广告传播路径进行输出.

在图 4 中,B 函数创建了一个 script,然后用 appendChild 方法添加到网页中.由于 B 函数的调用者 A 函数的参数中已经添加过 trace 属性,即 A 和 B 都在广告调用路径上,因此为 B 函数也添加调用路径的属性:onload→A→B→insertscript.appendChild.这就表示了该文件内广告调用的结束,因此在控制台对 B 函数所绑定的参数进行打印输出.

2) 对 document.write 函数的分析

考虑到 document.write 函数接收的参数是字符串而非 DOM 元素对象,满足 HTML 表达式规范的字符串会被解析为 DOM 对象,所以我们对 document.write 的参数进行字符串匹配,确定它是否插入了之前提到的 4 类元素.同样,我们也要判断写入的 script 标签有没有 src 属性.图 7 所示例子说明了通过 document.write 插入元素的过程:第 7 行在 A.js 中使用 document.write 嵌入一个 script 标签,script 标签执行 B.js.

```
1.〈script type="text/javascript"〉
2.      var aa=document.createElement('script');
3.      aa.src="http://******/.../A.js";
4.      var node=document.getElementsByTagName("script")[0];
node.parentNode.insertBefore(aa,node);
5.〈/script〉
6.-----------------A.js 中---------------
7.document.write('〈scripttype="text/javascript"src="http://********/B.js"〉〈/script〉')
```

Fig.7 Calling process of using document.write

图 7 使用 document.write 的调用过程

3) 对 innerHTML 属性的分析

向一个 DOM 元素的 innerHTML 属性赋值一个合法的 HTML 表达式字符串,可以在其下面插入元素.使用 Jalangi 的 putFieldPre 分析回调函数,可以分析任意对象属性的写入行为.当写入一个对象的属性时,判断对象是否为 DOM 元素对象以及属性名是否为 innerHTML,且写入了与广告相关的标签.

最后需要将函数调用路径和 DOM 操作关联起来,组成单条广告路径.在使用 appendChild、insertBefore 和 document.write 函数来插入广告相关元素时,如果它们有调用者,则可以获取它们的调用者参数对象里的函数调用路径,并附上当前 DOM 操作的相关信息作为最后一项;而对于 DOM 元素对象的 innerHTML 属性的写入,我们扩展了 Jalangi 的代码,使 putFieldPre 函数接收一个额外的参数,即该条属性赋值语句所在的函数,从这个函数的参数对象里寻找函数调用路径并拼接上该 DOM 操作的相关信息,作为最后一项.

4 广告生成过程

4.1 document内的广告生成过程

动态生成的广告能够在网页广告位中显示来自于第三方广告联盟的广告资源,这些广告资源可能是图片、

视频,甚至是一个 URL 链接.而根据 JavaScript 的同源策略,iframe 引用的 URL 属于不同域名网页内的 DOM 是不可以被 iframe 所在页面内的 JavaScript 代码操作的,所以,为了不影响正常显示,这些来自于第三方的图片、视频资源等,一般都嵌套在 iframe 里,再放入网页中.一个页面和它的 iframe 引用的页面各自具有以 document 节点为根节点的 DOM 树.

在一个 document 内,我们规定在两条广告路径中,如果其中一条插入了一个〈script〉标签,引用了某个 JS 文件,而另一条的第1项对应的语句在该JS文件中,则两者是关联的,前者是后者的前继.我们将获取到的所有路径都放在一个路径集合中,方便之后获取到新的广告路径时通过算法回溯它的所有前继.

document 内的广告路径回溯算法如算法 1 所示.该算法接受 document 内所有广告路径集合 *CS* 和需要回溯的路径 *P*,遍历 *CS* 中的路径,若找到一条插入 script 脚本且 *P* 对应的语句处于该脚本中,则递归回溯该条路径的前继,直到回溯完毕.

算法 1. document 内广告路径回溯算法.

1.输入:document 内所有广告路径的集合 *CS*,需要回溯的路径 *P*;

2.输出:包含 *P* 和 *P* 的所有前继的路径序列 *PS*.

3.**Procedure** backtrackPathIntraDoc(*CS*,*P*)

4. **declare** *PO*

5. **for** *C* **in** *CS* **do**

6. **if** (*C* inserts a script and *P*[begin] is in the script) then

7. *PO*+=outputTrace(*CS*,*C*)+*P*;

8. **break**;

9. **return** *PO*;

4.2 跨documents的广告生成过程

在某些情况下,一个页面插入的 iframe 引用的页面里又会出现其他广告相关元素的动态插入,我们将这些行为关联在一起.在两条广告路径中,如果其中一条路径插入了一个 iframe,另一条广告路径起源于该 iframe 引用的网页中,则前者是后者的前继.

每条广告路径都对应一个 document,即它的每一项对应的文件都是被该 document 引用的.如算法 2 所示,为了关联跨 documents 的广告路径,遍历其他 document 的广告路径集合里的路径,并查看这些路径的尾部是否插入了一个 iframe 且 src 属性是否是被回溯路径对应的 document 的 url.如果存在这样的路径,则该路径是这个待回溯路径的前继.与此同时,一个 document 对应的所有广告路径都与插入该 document 的路径相关联.

算法 2. 跨 documents 广告路径回溯算法.

1.输入:所有 document 的广告路径的集合 *CSS*,需要回溯的路径 *P*;

2.输出:生成 *PN* 所经过的完整路径 *PO*.

3.**Procedure** backtrackPathInterDocs(*CSS*,*P*)

4. **declare** *PO*

5. **for** each *CS* **in** *CSS* **do**

6. **for** each *C* **in** *CS* **do**

7. **if** (*C* inserts an iframe and url of *P*[end] equals url of the iframe) **then**

8. *PO*←backtrackPathInterDocs(*CSS*-*CS*,*C*)+backtrackPathIntraDoc(*CS*,*C*)+*P*;

9. **return** *PO*

4.3 获取广告调用路径中涉及的JS文件url

在图 8 的示例中,A 函数创建了一个 iframe,用 appendChild 插入一个 iframe 标签,作为广告调用路径的终止点.此时得到的广告调用路径为 A→appendChild(ifr).在 iframe 内部又调用了 B 函数,B 函数又调用了 C 函数,C 函数插入一个图片,使用 appendChild 插入 img 标签,此时,这部分的广告调用路径结束,这部分的广告调用路径为 appendChild(ifr)→B→C→appendChild(img).

通过使用算法 3,可以将图 8 所示的跨 iframe 的两条广告调用路径进行拼接,从而获得该广告的完整调用路径: A→appendChild(ifr)→B→C→appendChild(img).

```
1.〈script〉
2.    functionA(){
3.    var ifr=document.createElement('iframe');
4.    document.body.appendChild(ifr)
5.    }
6.〈/script〉
-------------iframe 内部---------
7.functionB(){
8.    C();
9.}
10.functionC(){
11.    var img="〈img src="图片地址"〉";
12.    document.getElementById("**").appendChild(img);
13.    }
```

Fig.8 Example of cross-iframe

图 8 跨 iframe 示例

算法 3. 获取广告调用路径中涉及的 JS 文件 url.

1.输入:广告路径 *P*;

2.输出:路径 *P* 涉及的所有 JS 文件的 url 的集合 *UO*.

3.**Procedure** collectJsFiles(*P*)

4. **declare** *UO*={}

5. **for** each *N* **in** *P* **do**

6. *UO*←*UO*+url of *N*

7. **return** *UO*

5 实验分析

本节我们通过实验来展示使用动态插桩方法获得广告代码调用路径,通过对比实验来验证本文方法的优越性.

5.1 实验目标和环境设置

为了说明本文方法的有效性,我们对动态插桩工具 Jalangi 进行了扩展并开展了以下实验,以期动态追踪广告代码调用路径并获取到调用路径长度、广告插入方式等特征信息,然后具体分析广告相关的 JavaScript 脚本文件中原生函数 insertBefore、appendChild、document.wirte、innerHTML 所占的比重,从而更加明确广告代码的加载、传播方式.另外,还与静态分析方式进行了对比实验,以说明我们的方法在恶意广告的检测精度上更具优势.

JavaScript 的动态插桩工具 Jalangi 运行在 Linux 系统中,我们在代理服务器端使用 OSX 系统对网页进行插桩,在 Windows 7 系统上用 Firefox 上对网页进行浏览,并在控制台获取广告相关的调用路径,然后通过调用路径分析广告相关的 JavaScript 脚本文件.

5.2 实验过程

我们随机选取 Alexa 排名网站中的 21 个网站(包括 13 个国外网站和 8 个国内网站),作为实验对象,以追踪网站中广告相关的 JavaScript 文件调用、传递的详细过程.

本文实验获取的广告相关 JavaScript 函数调用路径中插入标签的方式包括:通过 JS 函数 insertBefore 和 appendChild 分别插入 script、img、a、iframe 标签;或通过 document.write 写入一些标签;或通过 innerHTML 插入 iframe 标签.之后,统计各种插入方式的操作次数.另外,我们还取到了广告路径中相关的 JavaScript 脚本文件,对每个文件解

析其抽象语法树,然后统计文件中 insertBefore、appendChild、document.wirte、innerHTML 出现的次数.这种静态方法只能获取到各种操作的次数,不能获取到插入标签的种类.实验过程中,比较这两种方式所获取信息的差异,并给出具体的结论.

5.3 实验结果

5.3.1 实验数据及其分析

通过对网页动态执行过程的记录和分析,我们不仅可以获取广告相关的函数调用路径,还可以获取网站 JS 文件数量、广告的插入方式.对于静态分析方法,可以通过遍历广告相关的 JS 文件的抽象语法树获取广告插入的操作方式,但是无法获取操作标签的种类.

表 1 中给出了网页动态执行中所涉及的 JS 文件、广告相关 JS 文件和广告相关 JS 片段数量,另外,还列出了插桩前后访问时间的对比情况以及实际记录的 Trace 长度情况.从平均数来看,这些网站平均要载入 72 个 JS 文件,其中广告相关的 JS 文件有 7 个,这说明,在运行过程中会动态载入或生成相当多数量的 JS 文件或片段.如果只通过静态方法分析网页中的 JS 文件,则不能获得动态执行过程中载入的 JS 文件或片段,因而也就无法精确识别网页中的广告.另外,通过访问时间的对比,我们发现本文方法的性能开销通常是 2~10X,处于用户可以接受的范围内.关于实际记录的 Trace 长度情况,我们发现一个页面上会包含多个广告(通常有 2~10 个),且广告代码调用路径的长度在 1~34 之间,平均数在 3~9 之间,这表明,广告的调用、传播路径还是比较长的,中间经过了多次跳转,这很大程度上增加了安全风险.

Table 1 Numbers of JS files, ads-related JS files and ads-related JS snippets

表 1 网页动态执行中所涉及的 JS 文件、广告相关 JS 文件和广告相关 JS 片段数量

网站名称	网站总的 JS 文件数	广告相关的 JS 文件数	广告相关的 JS 片段数	正常访问时间(s)	插桩后的访问时间(s)	网页中 Trace 个数	Trace 长度最小数	Trace 长度最大数	Trace 长度平均数
aintitcool.com	34	6	0	78	330	4	2	9	8
aol.com	58	8	2	114	300	4	2	5	4
bbb.org	25	4	0	60	240	3	4	4	4
Cbssports.com	226	26	2	138	342	9	2	7	9
deadspin.com	76	20	1	192	384	10	2	12	7
Newyorker.com	56	8	8	60	210	7	6	14	6
sbnation.com	46	7	1	90	186	3	2	10	4
walmart.com	32	4	0	34.36	84	2	2	3	3
Wonderhowto.com	38	8	3	126	258	2	2	3	3
Xfinity.com	81	9	0	66	216	5	3	24	6
tomsguide.com	63	10	13	108	258	8	2	20	9
sporcle.com	135	19	12	120	306	7	4	12	9
onet.pl	60	7	1	102	252	2	5	3	4
zhidao.baidu.com	16	7	2	15	120	4	1	9	4
news.ifeng.com	118	6	3	28	164	8	3	15	7
news.baidu.com/guoji	24	3	0	4.5	189	2	5	5	5
news.baidu.com/house	100	3	3	2.82	209	2	5	5	5
news.baidu.com/auto	16	3	0	4.6	201	2	5	5	5
music.baidu.com	53	7	0	17.25	166	4	1	9	5
finance.ifeng.com	130	8	4	19	230	7	1	34	7
csdn.net	61	5	3	17	187	4	2	7	4

图 9 中显示了网站 JS 脚本动态运行过程中 appendChild、insertBefore、document.write 和 innerHTML 的比例关系,可以看出,appendChild 使用得最多(超过 40%),其次为 insertBefore(37%),而 document.write 和 innerHTML 所占比例较少(分别为 14%和 6%).

我们同样使用静态分析方法统计了网站中 JS 文件中 appendChild、insertBefore、document.write 和 innerHTML 的比例关系(如图 10 所示),可以看出,appendChild 方法占比也是超过了 40%,而 insertBefore 的比例有所减少(25%),innerHtml 的数量大幅增加(25%).

通过比较图 9 和图 10,我们发现:通过 innerHtml 属性来动态加载 HTML 网页中广告的方式比较少见(6%),而在网页源代码中经常能看到 innerHtml 属性(25%),这表明,该属性虽然常见,但大部分是与广告加载无关的;与此相反,document.write 在网页源代码中不算常见(2%),但还较多地用来动态加载 HTML 网页中的广告(14%),

这表明,该属性虽然不大常见,但大部分是与广告加载相关的.

图 11 中给出了网页 JS 脚本运行过程中动态生成 iframe、script、image 和 a 标签所占的比例,可以看出,在这些含广告的网站动态生成的标签中 script(52%)和 iframe(28%)标签占了大部分(80%),这说明,广告脚本在运行过程中会动态生成很多脚本和网页,这正是广告分析的难点所在.作为广告展示的 image 标签占 17%,这说明,广告主要以图片的形式进行展示,而从广告网页的加载开始到最终显示出广告,会经历多次的动态生成脚本和网页的过程,有非常复杂的调用路径,这使得恶意广告的入侵成为可能.

我们的动态分析可以监控到广告加载的全过程,从而进行及时、有效的干预.而只进行静态分析,是无法检测到这些动态生成的恶意广告代码的.

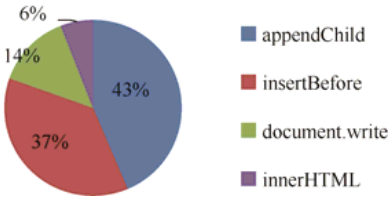

Fig.9 Percentages of appendChild, insertBefore, document.write and innerHTML by dynamic analysis

图 9 动态分析所得 appendChild、insertBefore、document.write 和 innerHTML 比例

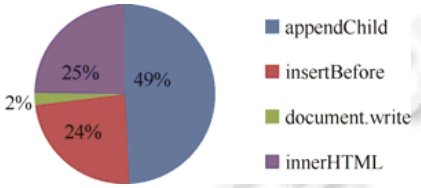

Fig.10 Percentages of appendChild, insertBefore, document.write and innerHTML by static analysis

图 10 静态分析所得 appendChild、insertBefore、document.write 和 innerHTML 比例

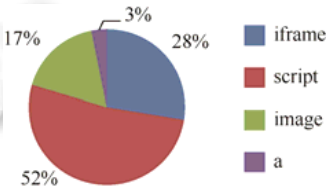

Fig.11 Percentages of iframe, script, image and a tags during executing JS files

图 11 网页 JS 脚本运行过程中动态生成 iframe、script、image 和 a 标签所占的比例

5.3.2 实例分析

如图 12 所示,匿名函数 1 中有一个三目运算,当条件(0===TRC.trkRequestStatus)不满足时,调用函数_(函数 2);函数_执行了一系列操作之后,调用函数 T(函数 3);函数 T 调用了函数 C(函数 4);函数 C 中(https:"==ea?"https://sb":"http://b")+".scorecardresearch.com/beacon.js)为一个三目运算操作和一个字符串拼接操作,并把运算后的字符串作为参数传递给 a,{async:!0}传递给参数 c,在函数 C 中,创建 script 标签 e,然后设置 e 的 src 为 a,即 https://sb.scorecardresearch.com/beacon.js 或者 http://b.scorecardresearch.com/beacon.js,并且判断 c.async 的值作为 e 的属性设置的依据,最后用 insertBefore 操作将 script 标签写入网页中.

```
1.function(box,data){
      TRC.hasTrk&& void 0 === TRC.trkRequestStatus ?a.setTimeout(_, ca) : _()
   }(window, document)
2.function _() {
        if (Y(), g(b.getElementsByTagName("script")), va = a[fa] = a[fa] || [], !va.registered) {
                  for (va.push = S, va.registered = !0; va.length;) S(va.shift());
                  aa.global["enable-cross-check"] && C(za, {
                  async: !0}),T(),aa.global["enable-visit-value"] && A()
   }
3.function T() {
      var b;
      aa.global["inject-comscore"] && (a._comscore = a._comscore ||
      [], C(("https:" == ea ? "https://sb": "http://b") + ".scorecardresearch.com/beacon.js", {
         async: !0}),
   }
4.function C(a, c) {
            var d = b.getElementsByTagName("script"),
            e = b.createElement("script");
            c&&c.async ? e.setAttribute("async", "") : e.setAttribute("defer", ""), e.src = a,
            d[0].parentNode.insertBefore(e, d[0]), TRC.pConsole("page", "debug", "loading : " +
e.src)
   }
```

Fig.12 Code snippet of Gamefaqs.com

图 12 实验对象代码(Gamefaqs.com)示例

由此,我们使用扩展后的 Jalangi 进行插桩并记录执行轨迹,形成的函数调用路径如下所示:

"unnamed(cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js:2:3492:114:162320)"

"_(cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js:114:162298:114:162301)"

"T(cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js:2:12101:2:12104)"

"C(cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js:2:10529:2:10616)"

"insertBeforescripthttps://sb.scorecardresearch.com/beacon.js(cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js:2:7152:2:7188)"

其中,“_”“T”“C”分别表示函数名,unnamed 是我们为匿名函数取的一个标识,cache/cdn.taboola.com/937d7f3e84aa424b9efca0a72b0ec608/loader.js 表示 loader.js 在 jalangi 中存放的文件路径,114:162298:114:162301 表示当前函数在 loader.js 脚本中的位置为从第 114 行的 162 298 列~第 114 行的 162 301 列,因为脚本文件的内容是经过压缩的,所以列数多,代码难理解.上述调用路径为“unnamed→_→T→C→insertBeforescriptbeacon.js”.在 beacon.js 中还有一系列的函数调用.仔细观察函数_,还调用了 A 函数,此处又有另一条函数调用路径.这表明,实际网页中存在很多代码压缩的情况,并且函数之间的调用关系非常复杂,如果采用人工审查代码的方式,是很难识别这些广告文件中的函数调用关系的.

6 相关工作

6.1 JavaScript程序分析

JavaScript 在 Web 应用中发挥着重要作用,具有语法灵活性和高度动态性,易于使用,但代码的可维护性不够好.比如,JavaScript 在运行时被广泛用来和 DocumentObjectModel(DOM[6])元素进行异步交互[7],其动态性和松散性使 JavaScript 代码易错,且定位困难[8–11].

基于 Jalangi 动态分析框架[5],可以检查 JavaScript 类型的一致性[12]或提高 just-in-time(JIT)性能[13].另外,工具 DLint[14]是在 Jalangi 的基础上实现的,使用动态分析方法检查 JavaScript 代码质量,由一个通用框架和一组可扩展的地址和特定规则的检查器组成,能够解决被静态方法遗漏的缺陷.

文献[15]在文献[16]的基础上提出了一个自动定位技术,通过追踪和后向切片,实现对 JavaScript 的动态分析,解决了包括 eval、匿名函数处理的困难.此外,HTML 元素和 JavaScript 代码之间通过浏览器相互作用,加剧了客户端 JavaScript 代码的维护问题.文献[17]提出了一种 JavaScript 动态切片技术 JS-Slicer,使得理解和调试客户端 JavaScript 代码变得容易.JS-Slicer 在动态分析框架 Jalangi 的基础上,结合动态和静态分析所得结果,精确地捕获运行时的依赖信息.

6.2 广告代码检测

随着互联网的发展,在线广告越来越多地被用于非法途径,如传播恶意软件、诈骗、点击诈骗行为等.为了理解这些恶意广告活动的严重性,文献[1–3]中研究了通过广告联盟所传播广告节点的拓扑结构,以此来获得恶意广告的传播行为和特点,文献[4]分析了 3 个月内爬取的广告相关的网页痕迹,揭示了恶意广告的猖獗,进而从恶意广告节点及其相关的内容传递路径来识别出恶意广告的特征,并构建了一个检测系统.

出于隐私、介入性和安全性等方面的考虑,现有一些技术和工具来拦截、屏蔽互联网广告,如:AdBlocke[18]、AdblockPlus[19]、Ghostery[20].这些工具通过维护一系列基于 URL 的正则表达式(EasyList[21]),将其与网页上获取的 URL 进行匹配而过滤广告.文献[22]使用针对 JavaScript 源代码的静态程序分析,识别用于加载并显示广告的 JavaScript 代码,通过特征训练,得到广告相关脚本的分类器,从而实现广告的拦截.与其相反,为了保障广告商的合法权益,WebRanz[23]利用随机化机制来使得广告拦截器失效,通过使用 WebRanz,内容发布者可以不断改变内部 HTML 元素 ID 以及元素属性,而不会影响它们的视觉效果和功能.

现有互联网广告的检测方法主要集中在静态模式匹配、静态特征匹配等,无法对混淆过后的域名和选择器进行有效检测,并且主要通过主观判定来识别、确定广告的特征,检测精度低.相应地,本文工作的主要目的是获

取广告代码运行时的调用路径,以得到广告相关的 JS 文件并确定广告插入的操作方式等信息,可以应用到广告代码(包括混淆代码甚至恶意广告代码等)的识别中,并有效提高检测精度,部分工作细节参加文献[24].

7 总结与展望

作为互联网最成熟的商业模式之一,在线广告一方面有助于促进互联网生态系统的健康发展,但也可能影响用户的网页浏览体验以及带来潜在的安全风险.现有的在线广告屏蔽插件通过设置黑名单的方式实现对网络广告的检测和屏蔽,但无法识别不在名单中的广告,也无法获取广告代码的调用路径.

本文的研究对象是来自于广告联盟的动态广告,这些广告是目前在线广告的主要存在形式.本文的工作是通过使用 JavaScript 的动态插桩工具 Jalangi 获取自动执行的广告代码第三方 JavaScript 函数调用路径.为此,我们为 JavaScript 函数动态绑定了一个存储调用路径信息的属性,分别识别广告代码调用路径的起始、中间以及终止状态,并对于 setTimeout、setInterval 等函数进行特殊处理以保证函数调用链的完整传递.另外,还对跨 iframe 的调用路径进行拼接处理.此外,我们统计了广告插入操作方式的类型(insertBefore、appendChild、document.wirte、innerHTML)和比例,并与遍历抽象语法树的静态方法操作数量进行了对比,以此说明本文方法的有效性.

在实验过程中,我们发现了一些用于分析用户行为和记录用户 cookie 的脚本文件,如 analytics.js,bk-coretag.js 等.这些文件的特征和广告代码文件的特征比较相似,尤其体现在动态生成、传播上.分析用户行为和记录用户 cookie 也会在一定程度上降低用户的浏览体验感受,实际上也是一种对用户隐私的侵害,可以考虑将其一并作为广告相关 JavaScript 文件进行屏蔽.因此,在后续研究中,我们会通过追踪代码调用路径的方式,进一步区分该类分析文件与广告文件的特征,并应用于恶意广告的检测和屏蔽方面.

References:

[1] Vogt P, Nentwich F, Jovanovic N, *et al*. Cross site scripting prevention with dynamic data tainting and static analysis. In: Proc. of the Network and Distributed System Security Symp. (NDSS 2007). San Diego: DBLP, 2007.

[2] Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proc. of the Int'l Conf. on World Wide Web (WWW 2010). Raleigh: DBLP, 2010. 281–290.

[3] Provos N, Mavrommatis P, Rajab MA, *et al*. All your iFRAMEs point to us. In: Proc. of the Conf. on Security Symp. USENIX Association, 2008. 1–15.

[4] Zhou L, Zhang KH, Xie YL, Yu F, Wang XF. Knowing your enemy: Understanding and detecting malicious Web advertising. In: Proc. of the 19th ACM Conf. on Computer and Communications Security (CCS 2012). 2012. 674–686.

[5] Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (FSE 2013). ACM, 2013. 488–498.

[6] Nicol G, Wood L, Champion M, Byrne S. Document objectmodel (DOM) level 3 core specification. W3C Working Draft, 2001,13: 1–146.

[7] Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side JavaScript bugs. In: Proc. of the Int'l Symp. on Empirical Software Engineering and Measurement (ESEM 2013). IEEEComputer Society, 2013. 55–64.

[8] Jones J, Harrold M. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2005). ACM, 2005. 273–282.

[9] Abreu R, Zoeteweij P, Gemund A. Spectrum-based multiple fault localization. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2009). IEEE, 2009. 88–99.

[10] Agrawal H, Horgan J, London S, Wong W. Fault localization using execution slices and dataflow tests. In: Proc. of the 6th Int'l Symp. on Software Reliability Engineering (ISSRE 1995). IEEE, 1995. 143–151.

[11] Cleve H, Zeller A. Locating causes of program failures. In: Proc. of the 27th Int'l Conf. on Software Engineering (ICSE 2005). ACM, 2005. 342–351.

[12] Pradel M, Schuh P, Sen K. Typedevil: Dynamic type inconsistency analysis for JavaScript. In: Proc. of the Int'l Conf. on Software Engineering (ICSE 2015). 2015. 314–324.

[13] Gong L, Pradel M, Sen K. Jitprof: Pinpointing jit-unfriendly JavaScript code. Technical Report, UCB/EECS-2014-144, University of California at Berkeley, 2014.

[14] Gong L, Pradel M, Sridharan M, Sen K. DLint: Dynamically checking bad coding practices in JavaScript. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis (ISSTA 2015). 2015. 94–105.

[15] Ocariza Jr. FS, Li GP, Pattabiraman K, Mesbah A. Automatic fault localization for client-side JavaScript. Software Testing, Verification and Reliability, 2016,26:69–88.

[16] Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side JavaScript bugs. In: Proc. of the Int'l Symp. on Empirical Software Engineering and Measurement (ESEM 2013). 2013. 55–64.

[17] Ye JB, Zhang C, Ma L, Yu HB, Zhao JJ. Efficient and precise dynamic slicing for client-side JavaScript programs. In: Proc. of the 23rd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER 2016). 2016. 449–459.

[18] The 2015 AdBlocking Report. 2015. http://blog.pagefair.com/2015/ad-blocking-report/

[19] Adblock Plus. https://adblockplus.org/

[20] Evidon, Inc., Ghostery. 2012. http://www.ghostery.com/

[21] Easy Blog-EasyList statistics: August 2011. 2011. https://easylist.adblockplus.org/blog/2011/09/01/easylist-statistics:-august-2011

[22] Orr CR, Chauhan A, Gupta M, Frisz CJ, Dunn CW. An approach for identifying JavaScript-loaded advertisements through static program analysis. In: Proc. of the 11th Annual ACM Workshop on Privacy in the Electronic Society (WPES 2012). 2012. 1–11.

[23] Wang WH, Zheng YH, Xing XY, Kwon YW, Zhang XY, Eugste P. WebRanz: Web page randomization for better advertisement delivery and Web-bot prevention. In: Proc. of the ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering (FSE 2016). 2016. 205–216.

[24] Chen GM. Dynamic advertisement analysis method and replay technology research [MS. Theisi]. Nanjing: Nanjing University of Posts and Telecommunications, 2018 (in Chinese with English abstract).

附中文参考文献:

[24] 陈贵美.动态广告分析方法与重现技术研究[硕士学位论文].南京:南京邮电大学,2018.

许蕾(1978－),女,江苏镇江人,博士,副教授,CCF 专业会员,主要研究领域为 Web 程序设计语言分析,Web 应用恶意代码识别分析.

刘蕊成(1991－),男,硕士,主要研究领域为程序分析.

陈贵美(1992－),女,学士,主要研究领域为程序分析.

赵晨(1992－),女,硕士,主要研究领域为程序分析.

张卫丰(1974－),男,博士,教授,CCF 专业会员,主要研究领域为代码仓库,持续集成,程序分析.