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.

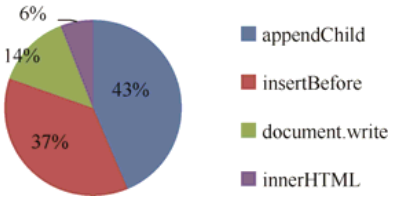


Fig.9 Percentages of appendChild, insertBefore, document.write and innerHTML by dynamic analysis

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

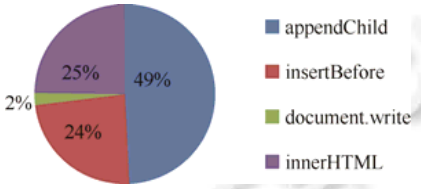


Fig.10 Percentages of appendChild, insertBefore, document.write and innerHTML by static analysis

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

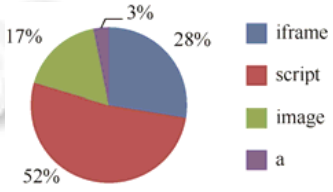


Fig.11 Percentages of iframe, script, image and a tags during executing JS files

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).

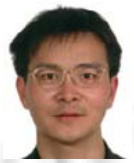
附中文参考文献:

[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.

赵晨(1992－),女,硕士,主要研究领域为程序分析.

刘蕊成(1991－),男,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.

陈贵美(1992－),女,学士,主要研究领域为程序分析.