Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations

Minh Tran[1], Xinshu Dong[2], Zhenkai Liang[2], and Xuxian Jiang[1]

[1] {mqtran, xuxian_jiang}@ncsu.edu
Department of Computer Science, North Carolina State University
[2] {xdong, liangzk}@comp.nus.edu.sg
School of Computing, National University of Singapore

Abstract. JavaScript-based applications are very popular on the web today. However, the lack of effective protection makes various kinds of privacy violation attack possible, including cookie stealing, history sniffing and behavior tracking. There have been studies of the prevalence of such attacks, but the dynamic nature of the JavaScript language makes reasoning about the information flows in a web application a challenging task. Previous small-scale studies do not present a complete picture of privacy violations of today's web, especially in the context of Internet advertisements and web analytics. In this paper we present a novel, fast and scalable architecture to address the shortcomings of previous work. Specifically, we have developed a novel technique called *principal-based tainting* that allows us to perform dynamic analysis of JavaScript execution with lowered performance overhead. We have crawled and measured more than one million websites. Our findings show that privacy attacks are more prevalent and serious than previously known.

Keywords: Privacy, Web security, Information flow, JavaScript, Dynamic analysis

1 Introduction

Privacy violation is common in web applications today, especially with the excessive power of the JavaScript language. The Same-Origin Policy (SOP) [1] governs the accesses by JavaScript to web page and network resources, which ensures objects in a web application are only accessible by JavaScript from the same origin (defined as the tuple $< protocol, host, port >$). Unfortunately, a web application often needs to include third-party scripts in the same origin of the web application itself. If the scripts have privacy violations, they are free of restrictions from SOP. Besides, although XMLHttpRequest is restricted by SOP, its successor Cross-Origin Resource Sharing (CORS) has more flexibility in sending requests to different origins. Moreover, SOP does not prevent information leakage through requests for external resources, such as images, and CSS background.

This concern has motivated researchers to search for an answer. Work by Krishnamurthy and Wills [2] examined 75 mobile Online Social Networks (OSNs) and showed that all of these OSNs exhibit some leakage of private information to third parties. In a

similar vein, Krishnamurthy et al. [3] surveyed 120 popular non-OSN websites. Using a proxy, they monitored the traffic and discovered 56% of the sites directly leak pieces of private information in clear text to third-party aggregators. From another perspective, Jang et al. [4] studied Alexa global top 50,000 websites and found multiple cases of privacy violation including history sniffing and behavior tracking. However, analysis on potential privacy-violation behaviors at the web page level is too coarse-grained, as it does not distinguish between different pieces of JavaScript on the page. On the other hand, although fine-grained taint tracking, such as [4], provides desirable accuracy in identifying information leakage behaviors, the performance overhead imposed by it has limited the scale of the study that can be performed on real-world web applications.

Our approach. In this work, we propose an effective mechanism to perform a large-scale study on privacy-violating behaviors in real-world web applications. We observe that web applications often include many third-party libraries. Although these libraries share the same namespace, they are loosely coupled. The reason is threefold. First of all, by virtue of the JavaScript language, local objects declared inside a function are only accessible inside that scope. Furthermore, functions are typically wrapped inside a closure (e.g. jQuery), restricting accesses to their objects. And even if a function does declare objects in the global scope, other code will probably not access these objects because the code is oblivious to the existence of the objects. As a result, the objects are only accessed and modified by their creator. Based on this fact, we propose to track web application behaviors at the granularity of the JavaScript libraries, which greatly boosts the performance in tracking potential privacy violation behaviors in web applications. We thereafter refer to this novel technique as *principal-based tracking*.

In this paper, we present the design, implementation, and evaluation of *LeakTracker*, a fast and scalable tool to study the privacy violations in web contents. We perform principal-based tracking on web applications to identify the libraries that have suspicious behaviors in accessing user privacy information, where the principal is defined as the URL of the source of the JavaScript. For each piece of JavaScript code going to be compiled by the JavaScript runtime, we introduce a tag specifying which principal this piece of code belongs to. The resulting script, when executed, can introduce more scripts into the system and these new scripts will in turn be tagged, and this process continues. Our system then monitors the behaviors of scripts with different principals, and identifies suspicious behaviors in accessing users' private information.

To verify that such suspicious scripts do leak private information to external parties, we perform an additional variable-level dynamic taint analysis only on them, and for the rest of the web application, our system runs at full speed without tainting. By applying the principal-based tainting technique, we manage to reduce the significant performance overhead associated with application-wide taint analysis [5, 6, 4], while directing the power of taint analysis directly towards those suspicious script principals.

We have implemented a prototype of LeakTracker by extending the JavaScript engine of Mozilla Firefox. Our prototype tracks not only JavaScript code originated from web pages but also JavaScript from browser plugins (e.g. Flash/Silverlight) and extensions. Based on this prototype, we have developed a HoneyMonkey-style [7] crawler and deployed it on production systems. Our computer cluster consists of ten monkeys and one monkey controller to drive them. Compared to previous work, we substan-

tially extend the scope of the study, by crawling the list of top one million global websites provided by Alexa. The cluster finished crawling these websites within one week, demonstrating that our design is fast and scalable. We have further evaluated the performance of LeakTracker with the SunSpider [8] benchmark suite, which indicates that our system incurs a reasonable performance impact.

In summary, this paper makes two primary contributions:

- **Principal-based tainting technique.** We design a less expensive dynamic tainting technique that tracks the taint flows in a web application. It tracks JavaScript originated from different sources, including that from NPAPI-based plugins and browser extensions. We show that our technique is robust in the presence of JavaScript obfuscation and redirection. Our implemented prototype confirms that the technique incurs reduced performance overhead.
- **Comprehensive evaluation through large-scale study.** We evaluate LeakTracker in terms of effectiveness and performance. With timeout parameter set at fifteen seconds, within a week of deployment, we finished crawling the global top one million websites. We show that even in such popular set of websites, privacy attacks are still prevalent. Specifically we found that 39.1% of websites exfiltrate certain private information about the user and her browser. Most alarmingly, 31,811 websites still spy on user behaviors and 7 websites still sniff on users' browsing history.

The rest of this paper is organized as follows. Section 2 provides an overview of the problem and existing works. Next, Section 3 and 4 detail the design and implementation of LeakTracker. After that, Section 5 presents our evaluation results. We cover the related work in Section 6 and finally Section 7 concludes this paper.

2 Background

In this section we review the economic and technical reasons why privacy violations come to exist, especially how the issue tends to relate to Internet advertisements (ads) and web analytics.

Internet advertising is an important business model to support free Internet content. To sustain the publishing effort and to earn a profit, web publishers display ads on their sites and get paid by the advertisers, or more often, the ad networks (who are in turn paid by the advertisers). The arrangement is somewhat similar to the situation in traditional media like television and newspapers. In other word, the user unwittingly gives up some of his time viewing ads in exchange for the content provided by the publishers. This win-win arrangement between the users, the publishers, and the advertisers is one of the reasons that contribute to the booming of web media since the late 1990s.

With the rising popularity of the web, advertisers get more and more sophisticated. Instead of serving the same ad to everyone visiting a particular website, advertisers or ad networks adopt a practice called *targeted advertising*. In this practice, they dynamically decide what kind of ads to display to the site's visitors. For example, the ads can be chosen based on the type of content users are viewing, thus making Internet advertising

more relevant and presumably more helpful to visitors. The net result is generally more ad clicks, more sales, and ultimately more profit for the advertisers and the publishers.

Unfortunately, advertisers do not stop at this point. To further enhance the relevancy of their ads, they start to adopt more aggressively practices, namely, tracking and profiling visitors. According to a study by the Wall Street Journal (WSJ) [9] in August 2010, "the nation's 50 top websites on average installed 64 pieces of tracking technology onto the computers of visitors, usually with no warning." They also identified that between the web users and the advertisers, there are "more than 100 middlemen: tracking companies, data brokers and advertising networks". Data about the users' interests are collected, aggregated, packaged, and sold to the advertisers, who use analytics software to determine the most relevant ads to serve. For example, BlueKai, one of the major data brokers, sells 50 million pieces of personal information on a daily basis.

Besides the economic reasons for tracking, some technological advances make tracking more effective. For example, a study [10] by the Stanford Center for Internet and Society (CIS) found that Microsoft Advertising network places a syncing cookie on its **live.com** domain, effectively respawning the tracking cookie even though the user has cleared cookies in an attempt to preserve her privacy. This mechanism is referred to as "supercookies". Flash Local Shared Objects (LSO) can also be used to achieve the same purpose. The second tracking method, reported by the Panopticlick project at the Electronic Frontier Foundation [11], is called "fingerprinting". In this method, public configuration information provided by the web browser can be used to create a signature that uniquely identifies the browser and its user. Their study [12] shows that this method is highly effective: a randomly picked browser will have a unique signature among 286,777 other browsers, resulting in 18.1 bit of entropy.

In this paper, we focus on privacy attacks that happen when embedded JavaScript code abuses its privileges to track and leak confidential user information such as history, behaviors, interests and so forth. Out-of-band methods like traffic fingerprinting are out of the scope of this work. Our goal in this work is to conduct an extensive study and quantify the state of privacy violations in web contents.

3 System Design

We have developed a comprehensive and efficient system called LeakTracker to track privacy violations in web applications, whose overall architecture is shown in Figure 1. The parts in dark colors are major components of LeakTracker. In essence, we first identify JavaScript principals in a web application that have suspicious behaviors in accessing privacy information, and then apply the principles of dynamic taint analysis to track their executions. If the taint flows from critical sources to critical sinks our reference monitor logs an alert. In other words, by observing the flow of taint we can detect possible privacy information leakage in a web application.

Next, we will explain the principal-based tracking technique, followed by the elaboration on how we assign principals to JavaScript in different cases with principal tagging. Finally, we elaborate on the taint sources and sinks tracked by our system.

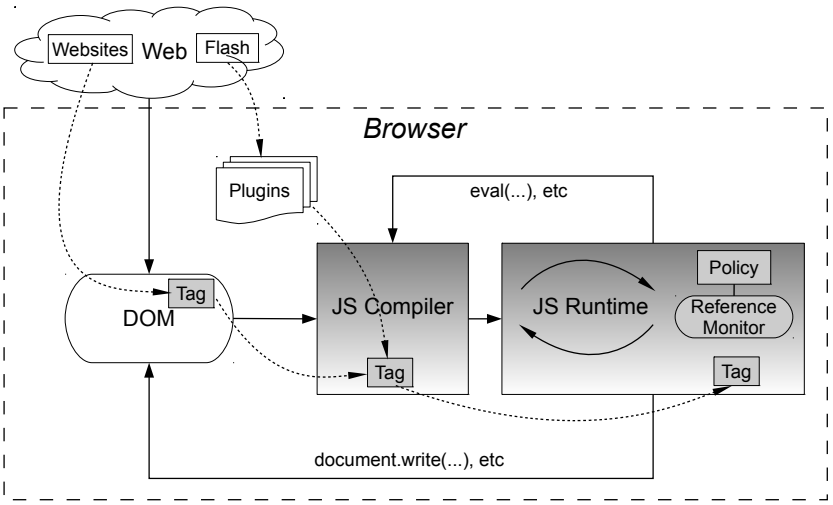

Fig. 1. LeakTracker architecture

3.1 Principal-based tracking

Despite being a powerful technique, taint analysis incurs a significant performance overhead. This overhead is mainly associated with the introduction, propagation and checking of taint. One needs to perform these for every JavaScript bytecode instructions executed. As a result, the performance impact from taint analysis is prohibitive. To reduce this overhead, we introduce a novel technique called principal-based tainting. Our key insight is: even though an embedded JavaScript widget shares the global namespace with other JavaScript code, its objects are naturally isolated from other code. JavaScript libraries, such as jQuery, tend to wrap functions and variables inside closures, and even for global variables or functions, they are generally not accessed by other JavaScript since other scripts are oblivious to their existence. The net result is that the objects are only accessed and modified by the creator. As a result we only need to perform taint analysis on the relevant functions, instead of the entire web application as implemented in previous approaches [5, 6, 4]. This lessens the performance overhead introduced by taint analysis.

To differentiate relevant functions from irrelevant ones, we introduce the concept of principal. A principal is defined as the URL of the JavaScript code introduced to the page. We observe that modern web applications typically employ third-party tracking libraries to track user behaviors, and such JavaScript libraries put users' privacy information at risk. During our analysis on privacy violations, it is necessary to distinguish first-party and third-party logic mixed in a web application to further lower the need for full-blown taint analysis. For example, it is quite natural and even expected that a web page will set and later read the cookie object to restore view settings, user preferences, among other things. On the other hand, it would be suspicious for a third party code to read the cookie and then transmit its content to a third party server. In fact, this con-

stitutes the cookie stealing attack. However, due to the complex interactions between principals, it is challenging to differentiate the owner of a piece of code and accurately assign a principal to each piece of JavaScript. We detail our principal assignment strategy next.

3.2 Principal tagging

Table 1. Script Sources

Source	Type
DOM	Direct script
DOM	Unnamed script
Source of the plugin object	Plugins
Source of the extension	Extensions
JavaScript	JS features
JavaScript	DOM APIs

Although JavaScript libraries tend to be independent of each other during execution, they may modify or overwrite other pieces of scripts. For instance, a script principal can overwrite the handler on a DOM element created by another principal. In another case, a principal can introduce more code into the page by features like **document.write**, by modifying the DOM tree, and so forth. To address this challenge, we introduce a second technique termed principal tagging.

As can be seen in Figure 1, our system completely tracks the provenance of all JavaScript code and assigns appropriate principal tags. The flow of tags is as follows. We start with the most common source of JavaScript code: the DOM. A <**script**> tag requests the browser rendering engine to send the corresponding JavaScript code to the JavaScript compiler. Our instrumented JavaScript compiler, in addition to parsing and compiling JavaScript code into bytecode, determines the principal of the script from the DOM and attaches a tag to the resulting **JSScript** object. We note that a <**script**> tag is not the only way to introduce JavaScript code into a web application. Another way of doing so is to use unnamed JavaScript code blocks, e.g. event handler code. Yet JavaScript code can also be introduced into the system by browser plugins (e.g. Flash and Silverlight) or browser extensions. A list of all sources is shown in Table 1.

As Table 1 indicates, JavaScript code can also be generated dynamically by the running JavaScript code through the use of JavaScript language features like **eval()** or DOM APIs like **document.write()**. We interpose on these vectors and propagate the principal tag from the creator script to the created script accordingly. In this way, we always keep track of the true identity of every piece of code. We also note that by keeping the full URL of the script in the tag, our system offers much better precision and accuracy than the Same Origin Policy(SOP), which assigns the same origin to all scripts embedded in the same page.

3.3 Taint sources and sinks

The Reference Monitor (RM) component guards the taint sources and sinks according to our policy. In our system, we track all major taint sources and sinks. For example, to detect possible cookie stealing attacks, the RM keeps track of all principals that has accessed the **cookie** object. Similarly, we treat the **getComputedStyle** function as a taint source to detect history hijacking attacks. We also track the registration of event handlers and other sensitive sources, as further discussed in Section 4. After our principal-based tracking technique identifies the script principals that exhibit suspicious behaviors in accessing user privacy information and sending data through the sinks (e.g. the network), we apply taint analysis on these principals to confirm whether they actually leak privacy information to the network. Upon detection of a policy violation (e.g. sending private information to a third party server) the RM will record the identity of the offender to the audit log, and let the operation succeed. We note again that the focus in this work is to detect policy violation and not to implement a protection system, so currently when any tainted data is flowed to the taint sink, we log such event with the principal of the JavaScript that generates such behaviors.

4 Implementation

We have implemented a prototype of LeakTracker by instrumenting Mozilla Firefox 3.6.13. The reference monitor is implemented with 700 source lines of code (SLOCs) in C++. About 1200 SLOCs is used to implement the principal tracker. We add about 2400 SLOCs into the SpiderMonkey JavaScript engine to track and propagate taint. Overall, we add 4300 SLOCs to Mozilla Firefox code base.

The crawler is implemented using AutoIT, a specialized scripting language for application automation. The monkey controller is implemented in 500 SLOCs. 300 SLOCs is needed to implement a monkey. The total number of SLOCs used to implement the crawling infrastructure is 800 SLOCs. In the following subsections, we examine each component of LeakTracker and present its implementation details.

4.1 Instrumented Browser

Table 2. DOM Functions and Properties

Family	Functions
Write	write, writeln
Timer	setTimeout, setInterval
DOM Tree Manipulation	createElement, insertBefore, appendChild, ...
DOM Node Property	innerHTML, text, textContent

To implement the principal-based tainting technique, we first need to be able to track the principal of any piece of JavaScript code. We therefore instrument the browser

to introduce and propagate the principal tag. The tag is stored as a property of the **JSScript** object. As shown in Table 1, there are six sources and each of them need to be handled a little differently. We provide details on each of them as follows. The first one is JavaScript code directly included on the page using the **script** tag. If the **src** property is specified, the tag will be set to the value of the **src** property. This case is relatively straight-forward. On the other hand, if script code is specified directly in the body of the **script** tag, then the principal should be carefully determined based on who created that **script** tag. If the creator was another JavaScript, LeakTracker will assign the principal tag of the created script to that of the creator. If, however, the **script** tag is part of the original HTML document then the newly created script's tag will have the principal of the hosting page. The second case is handled in a similar way. In both cases, we enhanced the Firefox's **JSTokenStream** class to achieve the goal.

In the third case, the principal of the script should be that of the embedded object and not of the hosting page. For example, for JavaScript code introduced by Flash plugin, the principal should be set to the URL of the Flash file. We modified the internal NPAPI implementation inside *nsNPAPIPlugin.cpp* for this purpose. The fourth case is handled similar to the first case, in that the newly created script will inherit the principal of the extension, which starts with "`chrome://`". The fifth case is the least complicated of all. In this one, we copy the tag directly from caller of **eval()** to the newly created script. The final case requires the most engineering effort, as we need to instrument every relevant DOM API functions. This careful implementation is needed to thwart obfuscation attempts. Table 2 lists the APIs that we instrument. With the above mechanisms in place, we can now realize the principal-based dynamic taint analysis technique.

Recall that we apply additional variable-level taint analysis to principals detected as suspicious by the principal-based tracking. Variable-level taint analysis helps confirm the information flow between the sources and the sinks. For example, if we introduce taint at any reads from the **cookie** object, and later find a tainted object being send to a remote server, we can conclude that the content of the cookie is being leaked to the server. Tracking taint propagation for the entire web application is very expensive. Therefore our instrumented JavaScript engine only performs taint propagation for the script originated from suspicious principals. For every piece of JavaScript code, upon determining that the principal is trusted, the engine switches to an alternate execution path that is free of taint propagation operations. From that moment on, the piece of code is executed at full speed to completion. This helps reduce the overall overhead incurred by taint analysis. On the other hand, if the principal is determined to be suspicious by the principal-based taint analysis, we propagate taint as the code executes. To achieve this goal, we instrumented the 235 opcodes of the JavaScript bytecode interpreter. These opcodes operate on the unified **jsval** type. We introduced a taint bit into this **jsval** structure. The instrumented opcodes then propagate this taint bit. In this way, taint analysis is achieved with minimum space overhead.

The second component of our instrumented browser is the reference monitor (RM). To implement the RM, we instrument the Firefox's JavaScript interpreter and introduce guards at the critical sinks. When our system tracks suspicious principals, the RM will inject taint into any objects derived from taint sources. There are four critical sinks

that JavaScript code can use to send data to a remote server: XMLHttpRequest, form submission, CSS property misuse, and **src** property misuse. As further discussed in Section 5, our findings indicate that the last method is the most frequently used for exfiltrating data. Every time there is an access to a critical sink, the RM will check if the taint bit is present. If it is, the RM will raise an alert and record the principal of the executing code into the audit log.

4.2 Crawler

To explore and scan the web, we implement a HoneyMonkey-style[7] crawler. In this approach, a machine, which we refer to as the monkey controller, maps and dispatches tasks to individual monkeys. Note that a monkey can actually be a controller itself, and further map the tasks to its children. In this way, a tree is formed with monkeys as the leaves. Therefore the LeakTracker architecture supports scalability by allowing the flexible addition of computing power should the need arise.

We now discuss the monkey itself. Inside each monkey there is a software agent that acts as a liaison between the controller and the instrumented browser. The agent, upon receiving tasks from the controller, will drive the browser to visit websites. This approach is more advantageous than the use of an indexer in traditional crawler like Heritrix[13] in that a full-blown browser allows us to examine dynamically generated page while an indexer does not.

The monkey controller takes as input a list of websites, and divides it into tasks, each consists of one hundred websites. It then maps the tasks to idle monkeys, and instructs them to start crawling. The monkeys crawl the websites and when they have finished with the tasks, they compress the logs and send them to a log server. They then signal the controller for more tasks. The system continues to operate as described until there is no more tasks, at which point it remains idle. For each task, the software agent iterates over the list and drives our instrumented Firefox browser to visit each of the websites. It also sends keystrokes and mouse events to the browser to simulate keyboard and mouse activities. This is necessary to observe possible behavior tracking of websites. After letting the website run for a timeout period of fifteen seconds, the agent will close the browser, collect the logs and continue with the next website. Our empirical experience shows that the threshold of fifteen seconds is enough to let most websites finish loading.

5 Evaluation and Findings

We have deployed LeakTracker on production systems to survey the web. In our prototype, ten monkeys were used to run the crawling, each is a virtual machine running Microsoft Windows XP Service Pack 3 . The virtual machine was configured with one virtual CPU running at 3.0GHz and 320MB of memory. The entire computing cluster ran on an IBM eServer BladeCenter HS21 with VMWare ESX Server 3i as the hypervisor. Within one week our computing cluster finished crawling the Alexa global top one million websites, demonstrating that our design is fast and scalable. Overall, we found 817,831 instances of leakage in 391,837 out of one million websites (roughly

39.1%), resulting in an average of 2.08 instances per website. In the following sections, we provide detailed discussion of our findings.

5.1 General Findings

Examining the leakage cases identified in our experiment, we found that most of them are caused by web analytics software embedded in the websites. Manual examinations of a random sample of two hundred leakages reveal that most of them leak screen resolutions, color depth, and JavaScript version. More than a dozen of them transmit the full list of installed browser plugins to the tracking server. One typical case is shown below:

```
http://charter.122.2o7.net/b/ss/charternetprod/1/H.22.1/s436952264
8743?AQB=1&ndh=1&t=4%2F7%2F2011%209%3A49%3A29%204%20240&ce=UTF-8&
ns=charter&pageName=homepage&g=http%3A%2F%2Fcharter.net%2F&cc=USD&
ch=home&server=web12.charter.synacor.com&events=event1&c1=home&v1=
D%3Dc1&v7=D%3Dc7&v9=D%3Dc9&v17=D%3Dc17&c23=Repeat&v23=D%3Dc23&c
27=4&v27=D%3Dc27&c28=Less%20than%201%20day&v28=D%3Dc28&c29=10%
3A30am&v29=D%3Dc29&c30=thursday&v30=D%3Dc30&c51=non-customized&c52
=not%20a%20premium%20owner&c53=logged-out&s=1680x1050&c=24&j=1.7&v
=N&k=Y&bw=1680&bh=900&p=Test%20Plug-in%3BMozilla%20Default%20Plug-
in%3BGoogle%20Update%3BShockwave%20Flash%3BSilverlight%20Plug-In%3
BAdobe%20Acrobat%3BQuickTime%20Plug-in%207.6.8%3B&AQE=1
```

This is the actual query string captured by LeakTracker when it was being set to the **src** property of an image. The taint sources in this case are the **screen** object (`height`, `width` and `colorDepth` properties) and the **navigator** object (the `plugins` property). As pointed out in [12], the tracking server could then use the captured information and other public information such as IP addresses, User-Agent strings and so forth to compute a fingerprint of the user. We therefore consider this a potentially dangerous privacy attacks. To ease the examination effort, we implemented a classifier based on data provided by Ghostery (www.ghostery.com). This classifier allows us to identify the provider of each suspicious JavaScript principal found by our system. Running the classifier over our log database, we found 480 known providers and they are responsible for the majority of the leakages. Our system detects six new trackers that were not previously identified by Ghostery, one of which is ClickHeat, a behavior tracker (further discussed in section 5.2). We observe a long-tail effect in the distribution of leakages over trackers. In other words, some providers are more popular than others (for example Google Analytics is used in 339,147 sites, Quantcast - 34,351 sites) but the majority of leaks are in the long-tail.

Regarding the distribution of trackers per website (we count one distinct script file as one tracker, regardless of how many leaks resulted by that tracker), we found that the majority of them employ less than ten trackers. Specifically, 384,535 sites (98.1%) have less than ten trackers on their page. A minority of the sites, however, have a large amount of trackers, four of which even have more than 50. Overall, we found 7,302 sites that have more than ten trackers. The list of 20 websites with the most trackers is shown in Table 3. We also observe that less popular websites tend to have many more

Table 3. 20 Sites Embedded the Most Trackers

Site	Rank	Trackers
pmcarpenter.blogs.com	430602	62
freestore.spb.ru	534501	58
minordetails.typepad.com	800405	55
thefraserdomain.typepad.com	307401	51
exopolitics.com	570650	46
travel.ru	10106	44
tuschistesmachistas.com	990528	44
meine-preis.de	686600	43
mylol.net	268601	42
sysopt.com	251904	40
thediscountdivaguide.com	733649	40
markwebberforum.com	857301	40
thelifeofluxury.com	299507	40
blackberrydownload.net	305801	39
bigwigbiz.com	505509	39
smallbizlabs.com	428284	38
democracyforums.com	762002	38
usaonlinemall.net	559693	38
community.web.id	286006	37
bestpriceproduct.com	838801	37

trackers than popular websites. As can be seen in Table 3, 19 out of 20 websites have ranking more than 200,000.

On the prevalence of the four popular kinds of attacks, we did not observe any case of cookie stealing or location hijacking. This finding is consistent with those reported by Jang et al. [4]. We, however, did find notable cases of history sniffing and behavior tracking, which we discuss in the following subsections.

5.2 Behavior Tracking Cases

We found multiple cases of behavior tracking in Alexa global top one million websites. In this kind of privacy attack, a website will stealthily record the behaviors of visiting users. It achieves this goal by installing event handlers on various relevant events like mouse movements, mouse hovering and so forth. The collected log is then silently transferred back to the server using XMLHttpRequest, form submission, CSS property misuse, or **src** property misuse. The aggregated behavior data can then be used to generate a profile about the specific interests of that user. This proves to be very effective in tailoring highly relevant ads to that user. This is the main reason why user interest data command a premium in data exchange, as pointed out in [9] . As this level of surveillance is more than most people are comfortable with [14], we consider it a dangerous kind of privacy attack.

Table 4 summarizes our findings. The first column shows the name of the tracking provider we have identified. The number of sites that embeds a particular tracker

Table 4. Behavior Tracking Providers

Tracking Provider	Sites	Events Tracked
Tynt Insight	1602	copy, mouseover
ClickDensity	259	mousedown
ClickHeat	475	mousedown, focus
Omniture	6884	mousedown, keydown
Woopra	2263	mousedown, mousemove, keydown
Pagealizer	5	mousedown
Statcounter	15149	mousedown, load, unload
Visual Website Optimizer	651	blur, focus, focusin, focusout, load, resize, scroll, unload, click, dblclick, mousedown, mouseup, mousemove, mouseover, mouseout, mouseenter, mouseleave, change, select, submit, keydown, keypress, keyup
ClickTale	2187	load, unload, scroll, mousemove, mouseover, mouseout, mousedown, mouseup, click, contextmenu, resize, keydown, keyup, keypress, focus, blur, select, change
Etracker	1334	mousedown
Reinvigorate	722	mouseup
SeeVolution	203	onscroll, onpaste, keydown, on blur, change, focus, mousedown
Mouseflow	77	mousemove, mouseover, mousedown, mouseup

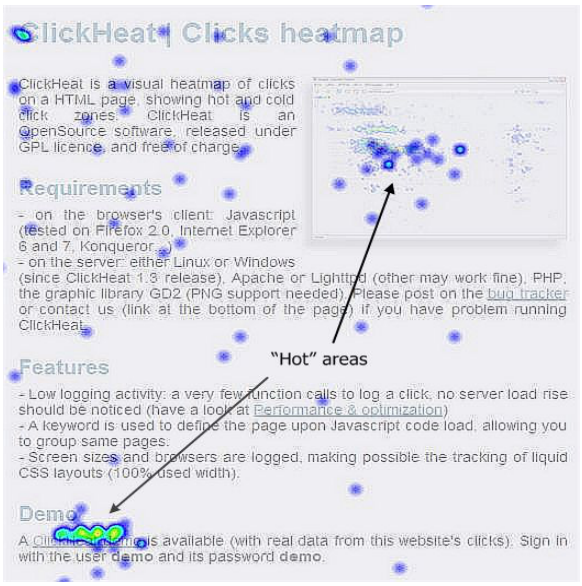

Fig. 2. ClickHeat

is listed in the second column. The last column shows the event handlers identified by LeakTracker. Overall, we found that 31,811 websites spy on user behaviors, using trackers provided by 13 companies. All of them are interested in tracking mouse events, with most of them register event handlers for mousedown and mousemove. This preference is expected, as mouse cursor movement is shown to have strong correlation with eyes movement and thus user's attention [15]. Some of the trackers seem very aggressive in collecting data, as indicated by the number of event they track. For instance, Visual Website Optimizer registers handlers for 23 events, while ClickTale registers 18.

We identified one new behavior tracker not previously recognized by Ghostery, who refers to itself as ClickHeat. Figure 2 shows a demo by ClickHeat itself, where data collected from users visiting its websites are used to compute a heatmap. As can be seen in the heatmap, hot and bright colors like yellow and green represent most clicked areas while cold colors like blue represent less frequently clicked areas.

5.3 History Sniffing Cases

History sniffing attacks have received considerable attention recently. In this kind of attack, a website gains unauthorized accesses to its visitors' browsing history and exfiltrates the data. Due to its highly invasive nature, history sniffing is generally considered an unsavory practice and several high profile lawsuits have been made against the perpetrators [16, 17]. These lawsuits have drawn considerable scrutiny over the behavioral advertising industry, leading to the establishment of AdChoices and opt-out programs. One of our goals in this paper is to assess how the current situation is.

Table 5. Sites Performing History Sniffing

Site	Description	Inspected URLs
golfdigestschool.com	Sports	casinogolfschools.com, executivegolfschool.org, +18
webleads-tracker.fr	Business	quivisite.com, +2
gearyi.com	Business	businessol.com, +2
netfactor.com	Business	leadlander.com, +3
pagealizer.com	Business	yahoo.com, +13
beencounter.com	Business	sitebrand.com, +5
bitbang.it	Business	facebook.com, +2

As shown in Table 5, history sniffing attacks still exist in the wild and we observed several cases of history sniffing occurring in the long-tail of the web. Specifically, we found seven websites still sniff on users' browsing history, querying for a total of 53 websites. Portions of the attack code captured from `http://www.netfactor.com/` are shown below:

```
var b_c_urls = [ 'iuuq://xxx.fyb68768hkih878nqmf.dpn', 'iuuq://xxx
.mfbemboefs.dpn', 'iuuq://xxx.wjtjtubu.dpn', 'iuuq://xxx.efnboecbtf
.dpn'];
var b_c_urls_id = [ '-1', '7850', '12973', '12972'];
```

```
b_c["3"] = "8301265327807-1";
...
if (ciphertext.length>2)
{
ciphertext = ciphertext.replace("][", String.fromCharCode(92));
ciphertext = ciphertext.replace(")(", String.fromCharCode(39));
Decrypt();
}
...
style_to_add_to_page += ".beencounter-id-" + b_c_urls_id_ +
":visited {background:url("+String.fromCharCode(39)+"http://www.
beencounter.com/b.php?one=" + b_c_urls_id_ + b_c_url_ending
 +String.fromCharCode(39)+")}";

links_to_add_to_page += "<a href="+ String.fromCharCode(39) +
plaintext + String.fromCharCode(39) +" class="+String.fromCharCode(
39)+"beencounter-id-" + b_c_urls_id_ +String.fromCharCode(39) +
"> </a>";
...
document.write( "<style type=" + String.fromCharCode(39) + "text/css"
+String.fromCharCode(39)+ ">" );
document.write( style_to_add_to_page );
document.write( "</style>" );
document.write( "<div  style=" + String.fromCharCode(39) + "position:
absolute;left: -4000px ;text-indent:-999px"+
String.fromCharCode(39)+">" );
document.write("<img src="+String.fromCharCode(39)+"http://www.
beencounter.com/b.php?one=" + b_c_url_ending+
String.fromCharCode(39)+">");
document.write(links_to_add_to_page );
document.write( "</div>" );
```

We have reverse-engineered the attacks and found that all of them are caused by a tracker called Beencounter. In the following, we provide a quick description of the attack. The list of URLs to sniff for is pre-encrypted and stored in the **b_c_urls** array. This encryption apparently is to avoid detection. Each of the URL is associated with a predetermined unique ID stored in **b_c_urls_id**. At runtime, the URLs are decrypted and used to build a list of links, stored in **links_to_add_to_page**. The associated IDs are then used to build respective CSS styles and then stored in **style_to_add_to_page**. As can be seen in the attack code, the CSS **background** property is misused to trigger the browser into sending requests to the web server. Therefore when the links are inserted into the page, the web server can learn about which sites the user has visited, based on what is requested and what is not requested by the browser.

5.4 Performance

In order to assess the performance overhead of LeakTracker, we conducted experiments with SunSprider [8], an industry standard in browser benchmarking. The version of

the benchmarking suite used was 0.9.1 and our experiments were done with a Dell Optiplex 760 workstation running Windows 7 x64 Service Pack 1. This workstation was equipped with an Intel Core 2 Quad Q9550 CPU (2.83Ghz 12 MB L2 Cache) and 4 GB of RAM. The baseline for evaluation was the original Mozilla Firefox 3.6.13. Each experiment was repeated ten times, and the average result was recorded. Overall, our instrumented browser completes the test in 1979ms versus 899.5ms of the original browser, resulting in a 2.2 times slowdown. We consider this satisfactory given the amount of additional computation needed to propagate and check taints, as a typical dynamic taint analysis system can incur up to 7.9x slowdown [4–6].

6 Related Work

Web-based attacks have received significant attention over the last few years. The threats from malicious web pages have motivated researcher into developing mechanisms to protect web surfers from being exploited. We first review related work on detection of malicious websites followed by a brief discussion on specific protection approaches.

6.1 Detection

Recent work by Curtsinger et al. [18] presents a mostly static approach to detect JavaScript malware. Their tool uses Bayesian classification of hierarchical features of the JavaScript AST to identify syntax elements that are highly predictive of malware. The results show that the system can achieve a very low false positive rate at negligible performance overhead. As a result, it can be deployed inside an end-user browser or as a first-level filter to reduce the workload of a dynamic, high-overhead but more effective approach like NOZZLE [19]. In a similar fashion to ZOZZLE, Prophiler by Canali et al. [20] combines JavaScript, HTML and URL features into one classifier that is able to quickly discard benign pages. According to their evaluation, the tool is able to reduce the load on a more costly dynamic analysis tools Wepawet [21] by 85%. In addition to using classifiers, it is also possible to detect attacks through the static analysis of other content retrieved from the web server, as shown in [22], [23], [24].

Work by Jang et al. [4] focuses on detecting privacy violating information flows in JavaScript web applications. Using dynamic taint analysis, they are able to detect the leaking of sensitive information through taint sinks and show four kinds of attack: cookie stealing, location hijacking, history sniffing, and behavior tracking. Their empirical study of Alexa global top 50,000 websites shows that many websites, including several in the top 100 sites, leak private information about users' browsing behavior. Their implementation parses JavaScript code and inject additional taint processing logic before sending them to the V8's JavaScript engine for execution. In contrast, LeakTracker implements taint tracking at one level below, inside the JavaScript engine itself. As a result, it incurs lower performance overhead and is more resilient to subversion. In the same vein as [4], using the Fiddler framework, Krishnamurthy et al. [3] surveyed 120 popular non-social-network websites and show that 56% of the sites directly leak pieces of private information in clear text to third-party aggregators.

6.2 Protection

Cross-site scripting (XSS) attacks can be prevented in several ways. BrowserShield [25], for instance, rewrites dynamic scripts into safe equivalents before sending them to the clients. BLUEPRINT [26], on the other hand, integrates with web applications to encode user-generated HTML content into a syntactically inert format and decodes it at the client. This allows them to bypass the anomalous parsing behaviors from client web browsers. ConScript [27] leverages aspect-oriented programming techniques to insert hooks into various interfaces, thus allowing the tool to restrict how JavaScript can interact with its environment. Document Structure Integrity (DSI) [28] takes a different approach and cast the XSS problem as a document structure problem where client and server have inconsistent views of the document structure. Their evaluation shows that the tool is effective and incurs low performance overhead.

The possibility of history sniffing attack was first considered nine years ago by Clover in a BUGTRAQ mailing list post in February of 2002 [29]. This, however, has not been considered seriously by the browser vendors until recently when Jang et al. [4] reported 46 popular websites did sniff on users' history. After the disclosure, a prevention mechanism was proposed by L. David Baron of Mozilla [30] and has been adopted in the latest version of major browsers. Right after that, in May 2011, Weinberg et al. [31] demonstrated a new kind of history sniffing attack that circumvents even Baron's defense and to date, no newer protection measure has been proposed.

Dynamic taint analysis has widespread application in the research community. Vogt et al. [5] proposed to instrument the browser's JavaScript engine to track taint flow in web appplications. Their system reported an XSS attack if sensitive data is sent to a third party domain. In the same vein, Dhawan et al. [6] applied a similar method to study confidentiality properties of Firefox's extensions.

From another perspective, the JavaScript language can be constrained to allow sandboxing of untrusted widgets. AdSafety [32] uses a novel type system to analyze the robustness of web sandboxes. They found several new bugs in the implementation of AdSafe, a web sandbox by Yahoo. Concurrent with AdSafety, Taly et al. [33] also studies JavaScript reference monitors and devises a restricted version of the JavaScript language. They then develop a tool that can soundly prove that an API cannot be circumvented or subverted, hence ensuring the robustness of sandbox protection.

7 Conclusions

We have presented the design, implementation and evaluation of LeakTracker, a fast and scalable tool to study the privacy violations in web content. Particularly, we developed a novel technique called principal-based tainting that allowed us to perform dynamic analysis of JavaScript execution at reduced performance overhead. We have implemented a Firefox-based prototype of LeakTracker and deployed it on production systems. We have crawled and surveyed the Alexa global top one million websites. Our findings demonstrated that privacy attacks are more prevalent and serious than previously known.

Acknowledgments

We thank the anonymous reviewers for their insightful comments that helped improve this paper. This work is supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640, and the Singapore Ministry of Education under the grant R-252-000-460-112. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Mozilla: Same origin policy for javascript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript
2. Krishnamurthy, B., Wills, C.: Privacy Leakage in Mobile Online Social Networks. In: Workshop on Online Social Networks. (2010)
3. Krishnamurthy, B., Naryshkin, K., Wills, C.: Privacy Leakage Vs. Protection Measures: The Growing Disconnect. In: Web 2.0 Security and Privacy. (2011)
4. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In: 17th ACM Conference on Computer and Communications Security. (2010)
5. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Network and Distributed System Security Symposium. (2007)
6. Dhawan, M., Ganapathy, V.: Analyzing Information Flow in JavaScript-based Browser Extensions. In: ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference. (2009)
7. Wang, Y.M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In: Network and Distributed System Security Symposium. (2006)
8. Webkit: SunSpider JavaScript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html
9. The Wall Street Journal: The Web's New Gold Mine: Your Secrets. http://online.wsj.com/article/SB10001424052748703940 904575395073512989404.html
10. The Center for Internet and Society: Tracking the Trackers: Microsoft Advertising. http://cyberlaw.stanford.edu/node/6715
11. Electronic Frontier Foundation: Panopticlick. http://panopticlick.eff.org/
12. Eckersley, P.: How Unique Is Your Browser? In: Proceedings of the Privacy Enhancing Technologies Symposium (PETS), Springer Lecture Notes in Computer Science. (2010)
13. Heritrix: Heritrix. https://webarchive.jira.com/wiki/display/Heritrix/Heritrix
14. The Wall Street Journal: What They Know About You. http://online.wsj.com/article/SB10001424052748703999 304575399041849931612.html
15. Cooke, L.: Is the Mouse a 'Poor Man's Eye Tracker?'. In: Society for Technical Communication Conference. (2006)
16. Forbes: Class Action Lawsuit Filed Over YouPorn History Sniffing. http://www.forbes.com/sites/kashmirhill/2010/12/06/class-action-lawsuit-filed-over-youporn-history-sniffing/
17. Forbes: McDonald's, CBS, Mazda, and Microsoft Sued for 'History Sniffing'. http://www.forbes.com/sites/kashmirhill/2011/01/03/mcdonalds-cbs-mazda-and-microsoft-sued-for-history-sniffing/

18. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. In: 20th USENIX Security. (2011)
19. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A Defense Against Heap-Spraying Code Injection Attacks. In: 18th USENIX Security. (2009)
20. Canali, D., Cova, M., Kruegel, C., Vigna, G.: Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In: Proceedings of the World Wide Web Conference (WWW). (2011)
21. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In: Proceedings of the World Wide Web Conference (WWW). (2010)
22. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All Your iFRAMEs Point to Us. In: 17th USENIX Security. (2008)
23. Seifert, C., Komisarczuk, P., Welch, I.: Identification Of Malicious Web Pages With Static Heuristics. In: Australasian Telecommunication Networks and Applications Conference. (2008)
24. Spoor, R.J., Kijewski, P., Overes, C.: The HoneySpider Network: Fighting Client-side Threats. In: FIRST. (2008)
25. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: Vulnerability-driven Filtering of Dynamic HTML. In: Proceedings of the Symposium on Operating Systems Design and Implementation. (2006)
26. Louw, M., Venkatakrishnan, V.: Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In: Proceedings of the IEEE Symposium on Security and Privacy. (2009)
27. Meyerovich, L., Livshits, B.: ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In: Proceedings of the IEEE Symposium on Security and Privacy. (2010)
28. Nadji, Y., Saxena, P., Song, D.: Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: Network and Distributed System Security Symposium. (2009)
29. Clover, A.: CSS Visited Pages Disclosure. BUGTRAQ Mailing List Posting. http://seclists.org/bugtraq/2002/Feb/271
30. Baron, L.D.: Preventing Attacks on a User's History Through CSS :visited Selectors. http://dbaron.org/mozilla/visited-privacy.
31. Weinberg, Z., Chen, E., Jayaraman, P.R., Jackson, C.: I Still Know What You Visited Last Summer: Leaking Browsing History Via User Interaction and Side Channel Attacks. In: 31st IEEE Symposium on Security and Privacy. (May 2011)
32. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADsafety: Type-Based Verification of JavaScript Sandboxing. In: 20th USENIX Security. (2011)
33. Taly, A., Erlingsson, U., Mitchell, J.C., Miller, M.S., Nagra, J.: Automated Analysis of Security-Critical JavaScript APIs. In: Proceedings of the IEEE Symposium on Security and Privacy. (2011)