Toward Exposing Timing-Based Probing Attacks in Web Applications

Jian Mao[1,3](✉), Yue Chen[1], Futian Shi[1], Yaoqi Jia[2], and Zhenkai Liang[2,3]

[1] School of Electronic and Information Engineering, BeiHang University, Beijing, China
maojian@buaa.edu.cn

[2] School of Computing, National University of Singapore, Singapore, Singapore

[3] Department of Computer Science, George Washington University, Washington, DC, USA

Abstract. Timing attacks in web applications have been known for over a decade. Recently, new attacks have been reported to exploit timing techniques to probe sensitive information from web applications. In this paper, we present a tool to detect timing-based probing attacks in web applications. The main idea of our approach is to monitor the browser behaviors and identify anomalous timing behaviors. We prototyped our approach in the Google Chrome browser, and demonstrated its effectiveness.

1 Introduction

In web applications, the same origin policy (SOP) [1] prevents a web application from directly accessing information belonging to other web applications. Since the web application share the same browser environment with web applications under other origins, it may *indirectly* figure out information from other web applications through analyzing the shared states in the browser environment [5,7,9,11,14–16]. For example, attackers can infer whether a website has been visited by the user via checking the color of a link to the site, as demonstrated in the browser-history sniffing attack [10]. Using this technique, via including a list of links pointing to a list of websites, a malicious website can obtain a user's browsing history by checking the color of the links[1]. This example shows that an attacker can steal users' information indirectly, i.e., inferring the information about another site by checking the browser states that are affected by visiting the website.

Observation. Compared to direct access to users' private information in the browser environment, this type of inference attack can only obtain limited information in each attempt. For example, the browser history inference is obtained

[1] Traditional ways to do this is by calling "getComputedStyle" method, but this method has already been modified to prevent this kind of misuse. However, there are still other ways to check the links colors [13,18].

Q. Yang et al. (Eds.): WASA 2016, LNCS 9798, pp. 499–510, 2016.
DOI: 10.1007/978-3-319-42836-9_44

site-by-site. In other words, the “data rate” of information leakage in such attacks is low. In order to obtain a significant amount of information, attackers have to *repeatedly* check and infer information from the other origin. We call them *browser probing attacks*, and the repetitive nature of browser probing attacks forms the basis to detect them.

We focus on timing-based probing attacks [8,12,13,17] in this paper, which are a popular browser probing attacks. They indirectly access sensitive information, such as cryptographic keys and states from other virtual machines [2,4]. They rely on the variations in the time taken by the systems to process different inputs [8,13]. This type of attacks has been adopted to exploit web applications. In a timing-based probing attack, a malicious web application check the time required to perform various tasks that can be affected by other websites.

Our approach. In this paper, we present a tool to detect timing-based probing attacks in web applications. The main idea of our approach is to monitor a web application's runtime behaviors, and identify anomalous timing operations to detect timing probing attacks. We summarize behavior patterns for timing-based probing attacks, and our approach detects timing probing attacks by matching monitored behaviors with such behavior patterns. Our approach alarms users with the potential risk of the privacy leakage to the website, and shows users the suspicious behaviors embedded in a malicious web page.

We prototype our approach in the Google Chrome browser. We evaluate the effectiveness of our approach using malicious probing applications built from known attacks.

Contributions. Our contribution is as follows.

- *New understanding of the timing-based probing attack.* We studied common timing probing attacks and define a general behavior model to describe the timing probing attacks. Based on the proposed model, we generate behavior patterns corresponding to different timing probing attacks respectively.
- *Light-weight approach to expose and limit the timing-based probing attack.* We propose an extention-based approach that monitors web application's behaviors and detect timing probing attacks, based on the repeat rate of sturctured-probing-behavior patterns. Our approach makes the timing-base probing attacks more difficult to succeed.
- *System prototype and evaluation.* We prototyped our approach as an extension of Google Chrome. Our evaluation demonstrates the effectiveness of our approach.

2 Overview

In this section, we discuss the threat from timing-based probing attacks and analyze their core features that can be used as the basis for detecting them.

2.1 Background

In this paper, we assume the adversary to be a web attacker. That is, the attacker controls a website, and is able to run JavaScript in the victim's browser. But the attacker cannot run native code in the victim's system, nor can the attacker exploit vulnerabilities in the victim's system or browser. The attacker aims to infer the victim user's private information in the browser environment through timing probing attacks.

Timing-Based Probing Attacks. Felten and Schneider [8] introduces a timing attack to web applications. This attack measures the time required to load a web resource. As the time needed to load a web resource is affected by the resource's cache status, the attacker can learn the resource's cache status, and then infer the user's browsing history.

From this example, we generalize *timing-based probing attacks* as follows. The attacker first retrieves time from the system, which is either to record system time or to start a timer. We refer to this activity as T_1 and the time obtained as the starting time t_1. It then starts a workload W, such as loading resources or perform a computation. Once W is finished, the attacker immediately takes another time measurement. We refer to this activity as T_2 and the time obtained as the ending time t_2. The time difference $t = t_2 - t_1$ is the time spent on W. If W depends on browser states that cannot be directly accessed by attackers, t reveals information about such states.

Properties of Timing Probing Attacks. Timing-based probing attacks are usually invisible to users. Since they are launched to indirectly infer other users' sensitive data with the presence of strong security mechanisms in browsers, each probing attempt typically infer only limited information. For example, in the above attack, every time the attacker can only infer whether a web resource is cached, among tens of thousands of resources that may reveal users' privacy. In other words, the "data rate" of leaked information in such probing attacks is very low. Due to the limited information accessible through probing, attackers/malicious websites need a large amount of repetitive operations to extract enough information for inferring a small amount of users' privacy. For example, to probe users' browsing history, the attacker must prepare a list of URLs, and check each of them repeatedly (using the behavior sequence (T_1, W, T_2)). In addition, the result of a practical timing based probing will differ depending on the speed of the hardware on which the browser is running. To achieve accurate time-based measurement results, attackers have to repeat time measurement operations to achieve the desired calibration.

Challenge. Though timing-based probing attacks do require repetitive behaviors of accessing time and carrying out the workload, benign web applications also have legitimate reasons to frequently access time and perform regular activities. Simply repeating such behaviors cannot be considered as the distinguished feature to identify the timing based probing attacks. We need to distinguish benign repeated timing behaviors from malicious ones.

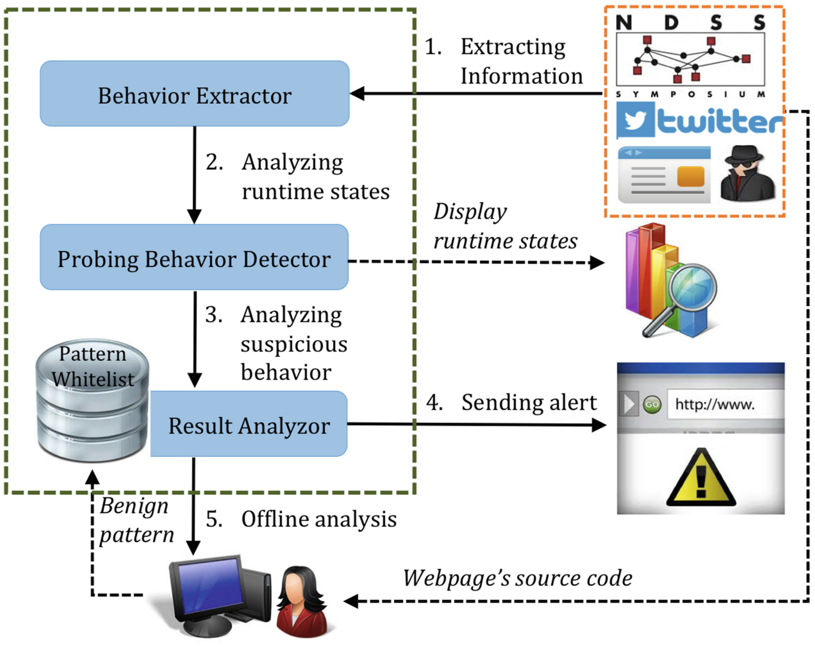

Fig. 1. Architecture of our approach(Color figure online)

2.2 Approach

The overall architecture of our approach is illustrated in Fig. 1. It monitors a web application's behaviors for abnormal patterns of timing-related events to alert the user of potential timing-based probing attacks. Our approach consists of three kernel components: *Behavior Extractor*, *Probing Behavior Detector*, and *Result Analyzer*.

In particular, the *Behavior Extractor* component monitors the web application and extracts its runtime execution information; the *Probing Behavior Detector* analyzes the runtime states gathered by the *Behavior Extractor* and detects suspicious timing-based probing behaviors; According to the knowledge in *Pattern Whitelist*, *Result Analyzer* analyzes the suspicious behaviors detected by *Probing Behavior Detector*; once a malicious behavior is confirmed, it displays *Alert* to the browser user. The intercepted behaviors and detection results are made available to an *Offline Analysis* component, where analysts identifies benign repetitive timing behaviors and update the *Pattern Whitelist*, so that such benign behaviors will not be flagged as attacks in the future.

Extracting Runtime Behaviors. Behavior extraction is the basis of our approach. Since most of a web application's dynamic behaviors are carried by JavaScript, our tool intercepts JavaScript API calls to represent the behavior of a web application. The behavior extractor module records the API together with its parameters. To help understand the behavior and distinguish APIs used under different scenarios, the behavior extractor also extracts the runtime JavaScript stack of the API.

In order to extract these runtime behaviors without modifying the browser, we build our tool as a browser extension. Though our focus is mainly on the timing-related behaviors, the interception mechanism is flexible to allow users to specify general JavaScript APIs to monitor. It intercepts JavaScript behaviors of web applications through rewriting JavaScript functions. Our extension preloads the rewriting JavaScript code before any code in the web application is loaded. It then takes a list of APIs to be hooked, which is specified by the users, and interpose the APIs to output them in a log of JavaScript behaviors, which records the functions' names, the arguments, and the functions' call stack information.

While obtaining API call information is straightforward, getting JavaScript call stack information needs more effort. We take the advantage of an error-handling feature of JavaScript, which reports the call stack information.

Detecting Probing Behaviors. Timing-related APIs are commonly used in benign web applications. To distinguish benign usage of timing APIs from probing attacks, the probing behavior detector adopts a two-level detection mechanism to expose probing behaviors. First, it flags suspicious timing behaviors based on the *frequency pattern*, i.e., the frequency of timing behaviors and their distribution over time. It serves as the first-level detector. To further prevent mistakenly reporting benign timing behaviors as probing attacks, the detector analyzes the *structure* of timing behaviors, and performs detection based on structured timing behavior pattern.

Distribution-based probing behavior pattern. The first-level detection is based on the distribution of timing-behaviors. We gather statistics on how many times timing functions are called over a period of time.

At this level, our approach provides light-weight real-time attack detection. Taking the cache timing attack explained above as example, our approach displays statistics for monitored JavaScript APIs. The user can pay particular attention to the *getTime* API. Once the frequency exceeds a certain threshold, the user can be alarmed about possible timing attack on the website he is visiting. To further increase the detection accuracy, we use another level detection via structure information of web application's timing behavior.

Structure-based probing behavior pattern. Timing probing attacks always contain two operations of obtaining system time value T_1 and T_2, as well as a workload in the middle W, which is the operation to be timed. So it forms the following pattern:

1. A *get-time* activity T_1 (such as a *Date.getTime* API call or a *set-timer* event);
2. One or several workload operations, such as loading an image or drawing a frame. We refer to the collection of such operations[2] as W;
3. Another *get-time* operation T_2.

[2] Note that the behaviors in W might not always be shown as the form of a function call (e.g. when loading an image, it's just an assignment to the src attribute of the image element).

Due to the asynchronous nature of the JavaScript language, T_1 and T_2 often appear in completely different program locations and contexts. However, in many timing probing attacks, W is carried out by an asynchronous operation, where T_2 can only be obtained in the callback functions of the asynchronous operation. When T_2 is executed, it is difficult to decide the corresponding T_1 and W. Moreover, in actual timing attacks, the (T_1, W, T_2) pattern may repeat multiple times consecutively, making T_2 of one iteration the T_1 of later iterations, making the following pattern $(T_1, W, T_2, W', T_2', ...)$. In this case, we can use the connection between the last two operations as our pattern for the timing attack, and treat abnormal amount of repeated pattern as indicator of a timing attack.

Some attacks that have distinguishing features that can be identified inside the behavior log can be detected automatically. Take the cache timing attack as an example. This attack always contains two system time acquire operations (normally by calling *Date.getTime* function), one before an resource (often an image) begins to load and another after the resource is loaded (i.e., being called inside the *onload* function of the resource). For this attack, the connection between T_2 and W is the *Date.getTime* function call for T_2 is inside an *onload* function of an *HTMLImageELement* for W. This connection, between the second system time acquire operation T_2 and the image loading operation W, plays an important role in distinguishing benign behaviors from probing behaviors in websites in our experiment.

3 Evaluation

We have prototyped our tool as an extension to the Google Chrome browser. Our prototype is based on the 64-bit Chrome browser. We have evaluated our approach from the following aspects. We used recent timing probing attacks and showed that our approach can successfully detect them. We then analyzed the performance overhead of our approach.

3.1 Timing-Based Probing Attack Detection

To evaluate the effectiveness of our approach, we used recent timing probing attacks in web applications. We implemented the attacks according to their technical descriptions. We first introduce the attacks and then describe how they are exposed by our approach.

Web Caching Based Probing Attack
Attack Mechanism. Web browsers use caching to speed up the access to the recently visited files/resources. A web-cache-based probing attack may make use of this functionality by measuring the time required to access a particular file belongs to another origin. If that file is in the user's cache, the access must be faster. According to the time cost for accessing, the attacker may infer whether the file is in the browser's cache, and whether the user has accessed the target origin as well (deducing the users' browsing history indirectly). We illustrate

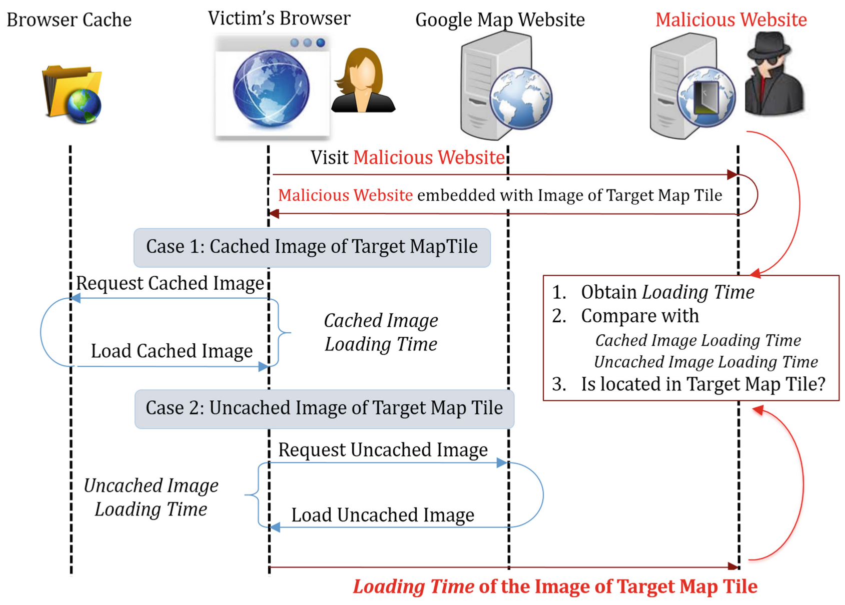
Browser Cache
Victim's Browser
Google Map Website
Malicious Website
Visit Malicious Website
Malicious Website embedded with Image of Target Map Tile
Case 1: Cached Image of Target MapTile
Request Cached Image
Load Cached Image
Cached Image Loading Time
1. Obtain Loading Time
2. Compare with
Cached Image Loading Time
Uncached Image Loading Time
3. Is located in Target Map Tile?
Case 2: Uncached Image of Target Map Tile
Request Uncached Image
Load Uncached Image
Uncached Image Loading Time
Loading Time of the Image of Target Map Tile

Fig. 2. Web caching based timing probing Attack(Color figure online)

the basic principle of the cache-based probing attack in Fig. 2. To evaluate the effectiveness on web caching based attack, we use the attack proposed by Jia et al. [12] as the test case.

Effectiveness. In Jia's attack, malicious webpage needs to measure the loading time of image files, so it is required to call *Date.getTime* function after image is loaded. The corresponding behavior pattern is *calling* Date.getTime *function inside* onload *function of* HTMLImageElement *element*. The pattern matching can be done by searching for lines containing both string *Date.getTime* and regular expression `HTMLImageElement \.[^]*\.onload` inside function call stack data.

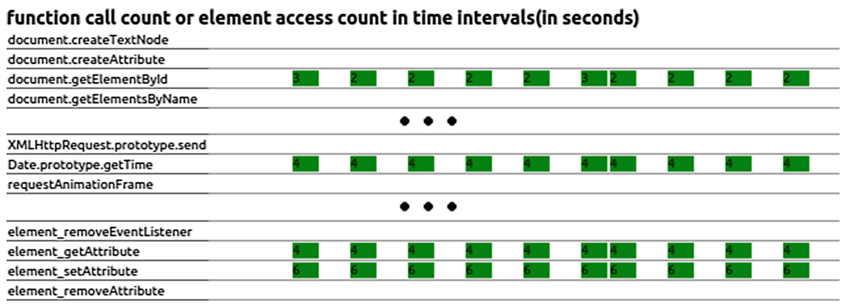

Fig. 3. Real-time function call and element access chart(Color figure online)

Figure 3 illustrates the real-time function call and element access chart (first-level detection), as well as key behavior log of the web caching based timing probing attacks (second-level detection).

Pixel-Based Attacks. Kotcher et al. [13] proposed another timing probing attacks, which allows the attacker to "see" the target website.

The shader inside browsers often takes different time to draw pixels of different color. So theoretically by measuring the time taken to draw a pixel, attackers can know the color of this pixel. And by traversing all pixels inside a target region of the target website, the attacker can get an image of this region, i.e. "seeing" this region of the target website.

Then attackers will measure the webpage's drawing time to infer the color of the pixels, which can be done by measuring the refreshing rate of the web page. By consecutively calling *requestAnimationFrame* function until certain amount of frames have been drawn, and measuring the time required to draw these frames, attackers can know the web page's refreshing rate, which is mostly influenced by the drawing speed of pixels of different color. By checking the web page's refreshing rate, the attackers can know the color of the current targeting pixel.

The attack needs to measure the frame rate of websites, which need to call *Date.getTime* function after a frame is drawn. And the function used as *requestAnimationFrame* function's argument will be called after a frame is drawn. so the pattern used to indicate this attack is *calling* Date.getTime *function inside functions that are used as* requestAnimationFrame *function's argument.* The pattern matching can be done by first searching for regular expression ˆ `requestAnimationFrame:function` [ˆ (]*(inside function call data and get the argument function's names, and then searching for lines containing both string *Date.getTime* and the argument function's names, inside function call stack data.

Repainting-Based Attacks. Stone [17] proposed a way to check the visit status of web links and sniff the browsing history by measuring the web page's repainting time.

According to the synchronizing property of the Chrome browser, when the browser found a link's target URL has changed, it will check whether this link's visit status has change too, i.e., from unvisited to visited or the other way around. If the visit status has changed, the Chrome browser will repaint this link element, while doing nothing if the visit status remains the same. The repainting operation may be inferred by measuring the drawing time of the web page.

This attack needs to measure the frame drawing time of a certain frame, which needs to call the *Date.getTime* function after a frame is drawn, the same as the last attack. Also note that the function used as *requestAnimationFrame* function's argument will be called after a frame is drawn. so the pattern used to indicate this attack is still *calling* Date.getTime *function inside functions that are used as* requestAnimationFrame *function's argument.* The pattern matching can be done the same way as in the last attack. The pattern distribution is the same as in that of the last attack, too.

3.2 Performance

We evaluate the performance overhead of our tool. We test the average time cost of *baidu.com* under different configurations. The result is shown in Table 1.

Table 1. Time cost on different configurations

Performance Test Environment	
CPU: Intel®Core$^{\text{TM}}$i5-4570	
Memory: 4 Gigabyte	
Operating system: Linux 12.04.2 (64-bit)	
Browser: Chrome, Version 42.0.2311.90 (64-bit)	
Enabled Functionalities	**Time Cost**
None	241 ms
Write function name to console	454 ms
Write function call stack to console	1420 ms
Write function name to console	1583 ms
Write function call stack to console	
Draw real-time behavior chart	310 ms
Warn attack(white list enabled)	1180 ms
Draw real-time behavior chart	1193 ms
Warn attack(white list enabled)	

The first three configurations are for offline behavior analysis. The additional overhead only affects the speed of evaluating a large amount of websites, which can be optimized by our automatic website parallel testing method. The last three configurations are for real-time behavior analysis. Drawing the real-time behavior chart causes neglectable overhead. And with warning functionality enabled, the time cost only increases to approximately 1 second. This is only the starting phase of visiting a website, after the website has been loaded and stabilized, the difference can be neglected too, users will not feel any difference.

4 Related Work

Previous work has discovered several classic browser probing attacks, along with some detection and prevention methods.

Kotcher et al. [13] discovered two timing probing attacks using CSS default filters. The first attack can check whether a user has an account with a website by exploiting the DOM rendering time difference. And the second attack can sniff user browsing history or read text token by stealing pixels on the user's screen. They conducted evaluations of their attacks, and proved the attacks' feasibility.

Felten et al. [8] described a class of timing attacks used to sniff browsing history too, but their attacks focus on operations whose time consumption depends on the cache status. They also proposed *web cookies*, which are a series of traditional cached files used as one traditional cookie. Using Web Cookies in attacks can make them harder to be detected. To prevent these attacks, one can turn off caching or alter the hit or miss performance of the cache, but both will make caching lack of usefulness.

Weinberg et al. [18] proposed a way to sniff browsing history. The traditional way to do this is secretly checking the status of the browser environment. But since browsing history will affect the appearance of the display to the user, the attacker can trick the user to tell him what the user has seen on the screen, which can be used to infer the user's browsing history. The links are disguised as CAPTCHAs, so that while the user is finishing the CAPTCHAs, he is actually telling the attacker his browsing history. They also come up with a way to "see" the user's screen by monitoring the reflection of the screen using the user's web camera. They proved that this is also a practical way to sniff the user's browsing history.

Bansal et al. [3] exploited the Web Worker APIs to make the cache probing operation parallel, which speeds up the traditional cache probing attacks. They also proposed the idea of canceling resource requests once the attacker can confirm that the resource being probed is from browser cache. In this way, the attacker can avoid polluting the cache. They applied their improved cache timing attack on four scenarios, including attacks on web environment and operating systems. At the end of their paper, they discussed potential countermeasures, such as separating cache among different operating system components, and setting no-cache headers to private data. However, their improved attacks doesn't reduce the number of probes that the attacker requires, so our approach can still detect these attacks.

Chen [6] found a vulnerability in four web applications that can leak users sensitive information. The base of this vulnerability is that, since the application needs to provide different contents according to the user's choices, different user inputs will result in different network traffic sizes. Furthermore, usually the possible user input at one application state is very few, making it easy to guess the user's input. The authors also use history traffic data to aid the process of guessing the user's input. In their opinion, in order to effectively and efficiently mitigate the impact of this vulnerability, the method has to be application-specific.

5 Conclusion

In this paper, we studied common browser probing attacks and develop a solution to detect such attacks based on generalized behavior patterns. We present a browser-extension-based tool to detect browser probing attacks. Our approach enables users to be aware of the potential risk of the privacy leakage during their surfing, and exposes the suspicious behaviors embedded in a malicious web page.

Acknowledgment. This work was supported in part by the National Natural Science Foundation of China (No. 61402029), the National Key Basic Research Program (NKBRP) (973 Program) (No. 2012CB315905), the National Natural Science Foundation of China (No. 61370190), Beijing Natural Science Foundation (No4162020), Singapore Ministry of Education under NUS grant R-252-000-539-112.

References

1. Same-origin policy. http://en.wikipedia.org/wiki/Same-origin_policy
2. Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P.: The EM side—channel(s). In: Kaliski, B.S., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002. LNCS, vol. 2523, pp. 29–45. Springer, Heidelberg (2003)
3. Bansal, C., Preibusch, S., Milic-Frayling, N.: Cache timing attacks revisited: efficient and repeatable browser history, OS and network sniffing. In: Federrath, H., Gollmann, D., Chakravarthy, S.R. (eds.) SEC 2015. IFIP AICT, vol. 455, pp. 97–111. Springer, Heidelberg (2015). doi:10.1007/978-3-319-18467-8_7
4. Brier, E., Joye, M.: Weierstraß elliptic curves and side-channel attacks. Public Key Cryptography. Springer, Heidelberg (2002)
5. Cabuk, S., Brodley, C.E., Shields, C.: IP covert timing channels: design and detection. In: Proceedings of the 11th ACM Conference on Computer and Communications Security (2004)
6. Chen, S., Wang, R., Wang, X., Zhang, K.: Side-channel leaks in web applications: A reality today, a challenge tomorrow. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE (2010)
7. Chevallier-Mames, B., Ciet, M., Joye, M.: Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. IEEE Trans. Comput. **53**(6), 760–768 (2004)
8. Felten, E.W. Schneider, M.A.: Timing attacks on web privacy. In: Proceedings of the 7th ACM Conference on Computer and Communications Security (2000)
9. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$a: A shared cache attack that works across cores and defies VM sandboxingand its application to AES. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (2015)
10. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Protecting browser state from web privacy attacks. In: Proceedings of the 15th International Conference on World Wide Web (2006)
11. Janc, A., Olejnik, L.: Feasibility and real-world implications of web browser history detection. In: Proceedings of Web 2.0 Security and Privacy Workshopp (2010)
12. Jia, Y., Dong, X., Liang, Z., Saxena, P.: I know where you've been: Geo-inference attacks via the browser cache. Internet Comput. IEEE **19**(1), 44–53 (2015)
13. Kotcher, R., Pei, Y., Jumde, P., Jackson, C.: Cross-origin pixel stealing: timing attacks using CSS filters. In: Proceedings of the ACM Conference on Computer and Communications Security (2013)
14. Lee, S., Kim, H., Kim, J.: Identifying cross-origin resource status using application cache (2015)
15. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (2015)
16. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: Practical cache attacks in Javascript. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (2015)

17. Stone, P.: Pixel perfect timing attacks with html5. Context Information Security(White Paper) (August 2013)
18. Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C.: I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In: Proceedings of the IEEE Symposium on Security and Privacy (2011)