An Adaptive Method for Cross-Platform Browser History Sniffing

Anxin Huang
Xiamen University
zjkuabjt@stu.xmu.edu.cn

Chen Zhu
Xiamen University
zhuchen@stu.xmu.edu.cn

Dewen Wu
Xiamen University
windwood@xmu.edu.cn

Yi Xie*
Fujian Key Laboratory of Sensing and Computing for Smart City
csyxie@xmu.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University
luoxiapu@gmail.com

***Abstract*—To accelerate browsing, most browsers keep track of visited URLs and create a browser history. Unfortunately, history sniffing attacks exploit the saved state to learn users' private information and infer about their visits to other sites. Although browser developers have taken measures to defend against these attacks, Smith's method, a kind of side-channel sniffing attack, could still work effectively. However, since different devices perform differently, this method requires that a list of parameters should be manually set up according to the device platform. In this paper, we propose an adaptive method that introduces multiple auxiliary links and adopts a dynamic parameter search algorithm to improve Smith's method. Our adaptive method attains nearly 100% accuracy on most popular browsers in different operating systems and platforms.**

***Keywords*—Browser history sniffing; Cross-platform attack; Adaptive method**

I. Introduction

Web applications with rich functions are very popular nowadays. People can easily and conveniently access websites. However, along with the rapid development of browsers, many problems, such as privacy disclosure, have ensued. Browsers store a user's browsing history, which can reveal the user's gender, behavior, location, even who they are in the real world [15]. Thus, a great threat to a user's privacy is that, using various methods, attackers can sniff the browsing history. The existing methods for sniffing browsing history can be divided into the following three categories: vulnerability attacks, interactive attacks, and side-channel attacks.

Vulnerability attacks. To sniff the history of a browser, vulnerability attacks, which usually focus on one or a few design weaknesses in the browser, use methods like harvesting the visited status of a link by reading its color [2]. By doing so, a malicious site can effectively check thousands of URLs per second to see if a user has visited it. However, browsers have already been enhanced to defend against such attacks by adopting the solutions proposed by David Baron [1].

Interactive attacks. Interactive attacks, by styling links in clever ways, such as CAPTCHAs and game pieces, trick users into revealing what they see on the screen and then rely on the users to visually identify visited links by clicking on them [15]. However, these methods are time-consuming.

Side-channel attacks. Side-channel attacks are based on information gained from the use of computer systems rather than the weaknesses of browsers. Side-channel attacks are of great concern because of the advantages of stealthiness, flexibility, and convenience. For example, the method of timing attack [13], a kind of side-channel attack, leaks information by measuring the computational time required for system operation when a browser renders one frame. Although this attack became invalid, Smith et al. [11] proposed an improved method that still works in some devices.

However, our experimental results show that Smith's method lacks robustness and practicability, because it works well only under carefully adjusted parameters, which heavily depend on the platform and operating system of the device. In this paper, to address the limitation in Smith's method, we propose an adaptive method for browser history sniffing, which uses multiple auxiliary links to amplify the frame-number difference instead of one and adopts a dynamic parameter search algorithm to determine the optimal number of auxiliary links for the current browser. Our major contributions include:

- We analyze the limitations of Smith's method and design a new approach[1] using multiple auxiliary links to improve its accuracy and robustness.
- We propose an adaptive method that adopts a dynamic parameter search algorithm to rapidly determine the optimal number of auxiliary links for the most popular browsers in different operating systems and platforms.
- We conduct extensive experiments to evaluate our adaptive method and study the influence of important parameters . The experimental results show that our

*Corresponding author: Yi Xie, Fujian Key Laboratory of Sensing and Computing for Smart City, School of Informatics, Xiamen University, Xiamen, China, csyxie@xmu.edu.cn.

[1] The code is available at https://github.com/onlyvae/Browser-History-Sniffing

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-63-0
https://dx.doi.org/10.14722/madweb.2020.23006
www.ndss-symposium.org

method can obtain nearly 100% accuracy when detecting visited URLs and outperform Smith's method.

The rest of the paper is organized as follows. In Section II, we introduce Smith's method in detail and analyze its limitations. In Section III, we propose an adaptive method for sniffing browser history as a cross-platform improvement, and present a detailed procedure for the dynamic parameter search algorithm. In Section IV we evaluate the performance of our proposed adaptive method. In Section V, we discuss the limitations of our method and propose the possible methods for defending against browser history sniffing. In Section VI, we introduce related works on browser history sniffing. Finally, in Section VII, we conclude the paper.

II. Background

A. Smith's Method

Smith's method is a kind of side-channel attacks. First, it applies a complicated style, such as adding text shadows or making 3D transforms, to a hyperlink element whose address is an unvisited URL [7]. For this complicated style, a browser will require a lengthy computation. But, when it initially draws this element, a browser will perform this computation only once and then reuse the rendered result unless the computed style of the element is changed. Then, for hyperlinks that the user has already visited, Smith's method defines a different style, which makes a browser experience a tough re-paint when the visit status of a link changes from unvisited to visited. Therefore, an attacker can switch the address of a hyperlink element from an unvisited URL to a target URL and measure the time of rending frames. If an obvious delay occurs due to the re-paint, the attacker will learn that the target URL has been visited. In the old version of browsers, attacker can use the API **requestAnimationFrame()** [8] to measure the accurate time of rending frames. But the latest version browsers reduce the precision of the timestamp that can be obtained through **requestAnimationFrame()**. To solve this problem, Smith et al. [11] improved the timing attack by introducing many re-paints and measuring the frame rate rather than the re-paint time of a single frame. Here, an attacker repeatedly switches the address of a hyperlink element between a target URL and an unvisited URL over a fixed time window and records how many times the browser invokes the callback function in **requestAnimationFrame()**. This number equals approximately the number of frames rendered by browsers over this time. If this number is obviously lower than the value measured by toggling the address of the hyperlink element between two unvisited URLs, then the target URL is identified as a visited URL.

B. Analysis

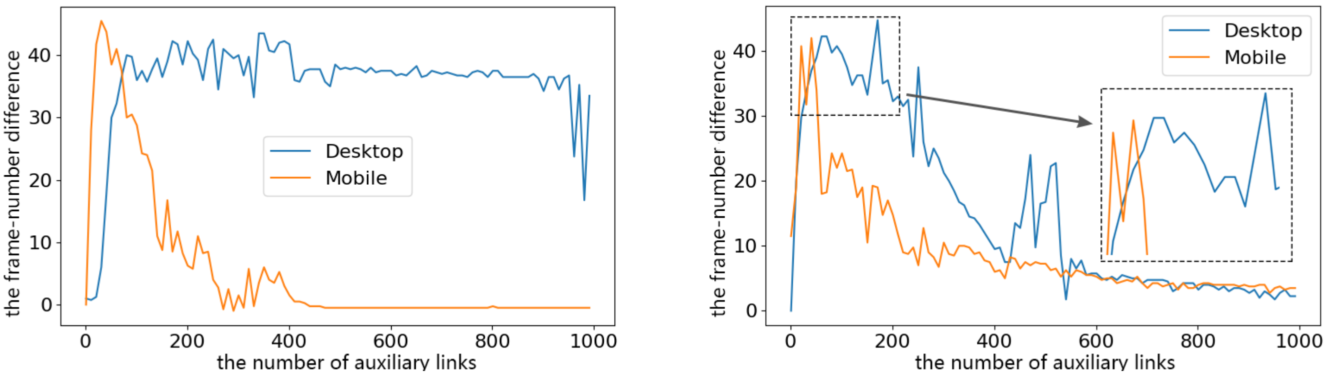
One key step in Smith's method is to require a browser take much time to re-paint a hyperlink element by setting a very complex style for the hyperlink element whose address points to an unvisited URL. Then, the attacker repeatedly switches the address of the hyperlink element between a target URL and an unvisited URL and records the number of rendered frames. If the target URL has not been visited before, the browser will render more frames over a fixed time window. Therefore, because the number of frames for an unvisited target URL is larger than that for a visited target URL, Smith's method recognizes target URLs by observing rendered frames. However, recognition accuracy highly depends on the difference between these two numbers, hereafter called the frame-number difference. The larger the frame-number difference, the higher the accuracy of the browser history sniffer. To amplify the frame-number difference, Smith's method requires different complex styles oriented towards the browsers in different OSs and hardware configurations. This obviously limits the reliability and practicability of the method.

We improve on Smith's method by using many auxiliary links, whose addresses simultaneously point to the target URL, to amplify the frame-number difference. Our new approach introduces a new problem: how many auxiliary links are needed for a given browser?

To study the relationship between the frame-number difference and the number of auxiliary links, we record the frame-number differences on the latest desktop and mobile versions of Chrome and Firefox[2] when the fixed time window is 1 second and the number of auxiliary links ranges from 1 to 1000. Fig.1 shows the results. In general, the frame-number difference increases steeply when the number of auxiliary links is small, and the peak or the sub-peak appears quickly. When the number of auxiliary links further increases, the browsers in the two devices have different tendencies. The frame-number difference for Chrome fluctuates near the peak in the desktop device, but shows an obvious downtrend for the mobile phone. The frame-number difference for Firefox fluctuates strongly with a decreasing trend in both devices. Both Chrome and Firefox have a similar tendency on mobile device, while have a different behavior on desktop device. One possible reason is that Chrome for desktop has a better performance optimization than Firefox. We cannot simply chose a big number of auxiliary links because the frame-number difference is not keeping increase with the number of auxiliary links increases. Therefore, we need to find the optimal number of auxiliary links that initially leads to the largest frame-number difference. It is worth noting that this optimal number should not be large; otherwise, it will cost the browsers much computational time to deal with many auxiliary links, thus weakening the concealment of our sniffing method.

We first consider a simple search algorithm (i.e., Hill climbing [10]), which searches the optimal number of auxiliary links by increasing the number from a small value until the frame-number difference decreases. But, because the frame-number difference does not increase monotonically when the number of auxiliary links is small, this algorithm may not always work. For example in Fig.1(b), the algorithm ceases at the second largest frame-number difference and mistakenly selects a local optimal number of auxiliary links. Therefore, to make our method works for different browsers and different platforms, we propose a new dynamic parameter search algorithm (see III-B) to determine the optimal number of auxiliary links.

[2] The desktop configuration: Windows 10, Intel Core i7, 16GB memory, Chrome 79.0.3945.79, Firefox 70.0.1; the mobile phone configuration: Android 7, Qualcomm 636, 6GB memory, Chrome 77.0.3865.116, Firefox 68.2.2



(a) Chrome

(b) Firefox

Fig. 1. The frame-number difference when the fixed time window is 1 second and the number of links ranges from 1 to 1000.

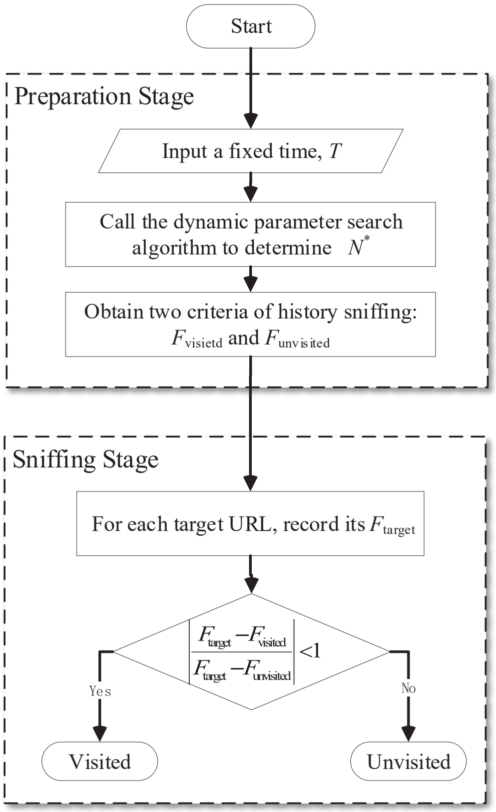

Start
Preparation Stage
Input a fixed time, T
Call the dynamic parameter search algorithm to determine N^*
Obtain two criteria of history sniffing: F_{visietd} and $F_{\text{unvisited}}$
Sniffing Stage
For each target URL, record its F_{target}
$\left|\frac{F_{\text{target}}-F_{\text{visited}}}{F_{\text{target}}-F_{\text{unvisited}}}\right|<1$
Yes
No
Visited
Unvisited

Fig. 2. Flow chart of the adaptive method of browser history sniffing

III. Methodology

A. Overview of the adaptive method

To overcome the deficiencies in Smith's method, we propose an adaptive method of browser history sniffing (see Fig.2), which consists of two stages: preparation and sniffing.

Preparation stage. In the preparation stage, a dynamic parameter search algorithm is used to find an optimal number of auxiliary links, N^*, which introduces an adequate frame-number difference within a fixed time, T, given a browser on some platform. Then N^* auxiliary links are generated to obtain two criteria for history sniffing: $F_{unvisited}$, the number of frames when repeatedly switching the addresses of the auxiliary links between two random unvisited URLs within T , and $F_{visited}$, the number of frames repeatedly switching the addresses of the auxiliary links between a given visited URL (denoted as URL$_{visited}$) and a random unvisited URL within T.

Sniffing stage. In the sniffing stage, for each target URL, we measure the number of frames, F_{target}, by switching the addresses of the auxiliary links between the target URL and a random unvisited URL within T. If F_{target} is closer to $F_{visited}$, i.e. $\left|\frac{F_{target}-F_{visited}}{F_{target}-F_{unvisited}}\right| < 1$, then the target URL has been visited and vice versa.

B. Dynamic parameter search algorithm

Algorithm 1: Dynamic parameter search

```
Output: The optimal number of auxiliary links $N^*$
$i \longleftarrow 0, increment \longleftarrow 1, stop \longleftarrow$ False
$\boldsymbol{N}[i] \longleftarrow$ N$_0$
$\boldsymbol{diffList} \longleftarrow$ getFrameDifference($\boldsymbol{N}[i]$)
$\boldsymbol{D}[i] \longleftarrow$ average($\boldsymbol{diffList}$), $\boldsymbol{V}[i] \longleftarrow$ variance($\boldsymbol{diffList}$)
while True do
    $\boldsymbol{N}[i{+}1] \longleftarrow \boldsymbol{N}[i] + increment\ /\ \frac{\boldsymbol{D}[i]}{\boldsymbol{N}[i]}$
    $\boldsymbol{diffList} \longleftarrow$ getFrameDifference($\boldsymbol{N}[i+1]$)
    $\boldsymbol{D}[i+1] \longleftarrow$ average($\boldsymbol{diffList}$)
    $\boldsymbol{V}[i+1] \longleftarrow$ variance($\boldsymbol{diffList}$)
    if $stop$ = True then
        break
    endif
    if $\boldsymbol{D}[i+1] > \boldsymbol{D}[i]$ then
        $increment \longleftarrow increment{*}2$
    else
        $increment \longleftarrow 1$
        $stop \longleftarrow$ True
    endif
    $i++$
end
search the largest and the second largest elements in $\boldsymbol{D}$,
  whose indexes are denoted as $I_1$ and $I_2$
if $\boldsymbol{V}[I_1] < \boldsymbol{V}[I_2]$ then
    $N^* \longleftarrow \boldsymbol{N}[I_1]$
else
    $N^* \longleftarrow \boldsymbol{N}[I_2]$
endif
return $N^*$
```

```
Function getFrameDifference(x)
Generate $x$ auxiliary links
for $j \longleftarrow 1$ to $4$ do
    Repeatly switch these auxiliary links' address in $T$
    diffList[$j$] = frame-number difference
end
return diffList
```

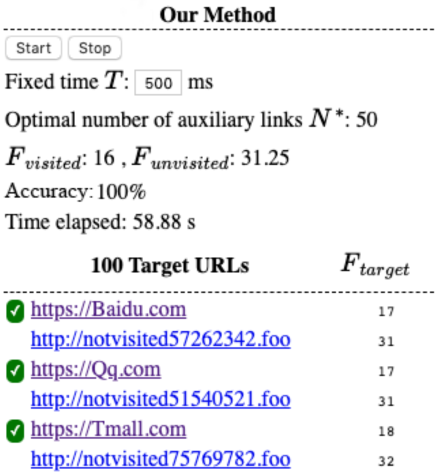
The dynamic parameter search algorithm (Algorithm1) is designed to quickly find the optimal number of auxiliary links, N^*, which makes the frame-number difference as large as possible, while avoiding excessive fluctuations. This algorithm uses an iterative technique, where $\boldsymbol{N}[i]$ records a candidate value of N^* in the i^{th} iteration, $i = 0, 1, 2, \ldots$. The algorithm begins with initialization: $\boldsymbol{N}[i]$ $(i = 0)$ equals a small number of auxiliary links, N_0[3]; an increment size is set as 1; a *stop* flag is set as False. Then we use the function, *getFrameDifference(*$\boldsymbol{N}[i]$*)* to obtain a vector ***diffList***, which records four experimental results of the frame-difference[4]. In the j^{th} experiment, we generate $\boldsymbol{N}[i]$ auxiliary links and, to calculate the frame-difference ***diffList***$[j]$, repeatedly switch their addresses within T. The average and variance of ***diffList*** are calculated and recorded in the i^{th} element of two vectors, $\boldsymbol{D}$ and $\boldsymbol{V}$, respectively.

In the $(i+1)^{th}$ iteration, $\boldsymbol{N}[i]$ increases by *increment*/$\frac{\boldsymbol{D}[i]}{\boldsymbol{N}[i]}$, where $\frac{\boldsymbol{D}[i]}{\boldsymbol{N}[i]}$ is used to estimate the current frame-difference caused by one auxiliary link. Then we caculate *getFrameDifference(*$\boldsymbol{N}[i+1]$*)*, $\boldsymbol{D}[i+1]$ and $\boldsymbol{V}[i+1]$. If the average frame-number difference continues to increase (i.e. $\boldsymbol{D}[i+1] > \boldsymbol{D}[i]$), the *increment* size is doubled to increase the frame-number difference rapidly. Otherwise, the *increment* size returns to 1 and the same process is repeated once before the iterations are stopped.

Next, we search the largest and the second largest elements in $\boldsymbol{D}$, whose indexes are denoted as I_1 and I_2, respectively. Then, the optimal number of auxiliary links, N^* is selected between $\boldsymbol{N}[I_1]$ and $\boldsymbol{N}[I_2]$ by comparing their variances. To obtain a stable frame-number difference, we prefer to choose N^* with a smaller variance. If $\boldsymbol{V}[I_1] < \boldsymbol{V}[I_2]$, then $N^* = \boldsymbol{N}[I_1]$; otherwise, $N^* = \boldsymbol{N}[I_2]$.

C. Implementation

A sniffer tool, based on a simple HTML page with JavaScript, has been implemented to evaluate the adaptive method for sniffing the browser history of a victim browser. As shown in Fig.3, our adaptive method can automatically determine the optimal number of auxiliary links N^* and requires only one input, T. For comparison, Smith's method also has been implemented. In this paper, the same CSS complex style recommended by Smith's paper [11] (shown below), and the same T, are applied in two methods. In general, the sniffer tool has higher accuracy using a longer T. How to choose a suitable T to gain a satisfied sniffing result will be discussed further in Sub-section IV-B.



Our Method

Fixed time T: 500 ms

Optimal number of auxiliary links N^*: 50

$F_{visited}$: 16 , $F_{unvisited}$: 31.25

Accuracy: 100%

Time elapsed: 58.88 s

100 Target URLs	F_{target}
✓ https://Baidu.com	17
http://notvisited57262342.foo	31
✓ https://Qq.com	17
http://notvisited51540521.foo	31
✓ https://Tmall.com	18
http://notvisited75769782.foo	32

Fig. 3. Adaptive browser history sniffer tool

```
a {
   transform: perspective(100px) rotateY(37deg);
   filter: contrast(200%) drop-shadow(16px 16px 10px
        #fefefe) saturate(200%);
   text-shadow: 16px 16px 10px #fefffe;
   outline-width: 24px;
   font-size: 2px; text-align: center;
   display: inline-block;
   color: white;
   background-color: white;
   outline-color: white;
}
a:visited {
    color: #feffff;
    background-color: #fffeff;
    outline-color: #fffffe;
}
```

For testing, the sniffer tool focuses on a set of target URLs, whose visiting signs have been labeled with colors: half are visited (violet) and half are unvisited (blue). In each experiment, the sniffer tool lists these URLs at random and operates according to the input T. Two criteria for history sniffing ($F_{visited}$ and $F_{unvisited}$), the total elapsed time, and the sniffing result of each target URL (with its F_{target}) are shown. For example, in Fig.3, **https://Qq.com** is judged to have been visited, because its $F_{target} = 17$ is closer to $F_{visited} = 16$. This judgement is consistent with its violet label, denoted to be correct with green check. To evaluate the performance of browser history sniffing, this sniffer tool calculates the accuracy of browser history sniffing, which is the ratio of the number of green checks to the number of visited target URLs (violet).

IV. EXPERIMENTAL RESULT

In this section, we evaluate our method by answering two Research Questions (RQs).

A. RQ1: Can our new method work effectively on different devices?

Motivation. Through this RQ, we examine whether our new method can effectively work on the most popular browsers [12] and different operating systems.

[3] We set $N_0 = 50$ for the desktop platform and $N_0 = 10$ for the mobile platform by default, which are determined according to extensive experiments.

[4] The experiment is repeated multiple times to get a precise value of the frame-number difference. We empirically choose 4 times.

TABLE I. The sniffing results of our and Smith's methods

Platform	OS	Browser	Average accuracy			
			Ours	Ours,$N_d = 200$	Ours,$N_m = 10$	Smith's
Desktop	Win10	1. Chrome 79.0.3945.79	**99.84%**	99.71%	74.90%	6.99%
		2. Firefox 70.0.1	**98.36%**	95.23%	69.03%	38.81%
	macOS	3. Chrome 78.0.3904.108	**99.15%**	99.80%	39.25%	15.3%
		4. Firefox 71.0	**99.86%**	100%	27.39%	36.46%
		5. Safari 12.1.1	**99.75%**	100%	30.26%	87.4%
	Ubuntu	6. Chrome 79.0.3945.79-1	**99.45%**	99.96%	43%	51.79%
		7. Firefox 71.0	**99.62%**	100%	30.89%	50.04%
Mobile	Android	8. Chrome 77.0.3865.116	**99.8%**	89.21%	99.07%	0.23%
		9. UC Browser 12.7.9.1059	**99.41%**	64.79%	100%	10.75%
		10. Samsung Internet 10.2.00.53	**99.78%**	68.62%	99.60%	1.48%
	iOS	11. Chrome 75.0.3770.103	**100%**	100%	100%	0.26%
		12. Firefox 18.1	**100%**	100%	100%	0.12%
		13. Safari 12.1.1	**100%**	100%	100%	8.06%

Approach. We evaluated and compared our adaptive method and Smith's method using the most popular browsers [12]: Chrome, Firefox, Safari, UC Browse, and Samsung Internet, those operate on two different platforms. The desktop platform includes three devices with different operating systems[5]: Windows 10, macOS Mojave, and Ubuntu 18. The mobile platform includes two devices with different operating systems[6]: iOS 12 and Android 7. The sniffer tool selects 100 target URLs, and a given browser has visited fifty target URLs in advance. To obtain reliable performance metrics, the sniffer tool repeats each experiment 100 times and then calculates the average accuracy.

We operated the sniffer tool in thirteen situations where two platforms, five operating systems and five browsers were involved. For a fair comparison between our adaptive method and Smith's method, the same value of T was adopted. For further studying the importance of multiple auxiliary links, two variants of our adaptive method are also compared, whose numbers of auxiliary links are fixed as $N_d = 200$ and $N_m = 10$ respectively. These selected numbers of auxiliary links can obtain nearly 100% accuracy for Chrome in Windows (Situation 1) and Chrome in Android (Situation 8). Then, sufficient experiments were launched to compare the performance of our method, our method fixing $N_d = 200$, our method fixing $N_m = 10$ and Smith's method.

Result. From the sniffing results in Table 1, we have the following findings.

1. Smith's method fails in the mobile platform because its average accuracy is extremely low, such as 0.12% in Situation 12. On the Desktop platform, Smith's method is unstable. It is valid in Situation 5 with the average accuracy of 87.4%, but fails in other six situations where the average accuracy is near or less than 50%. The sniffing results of Smith's method confirm the analysis in Subsection II-B, the frame-number difference due to one single link, may not be large enough to correctly recognize visited URLs, and a fixed complex style can not adapt to different browsers, OSs and platforms.

2. Our adaptive method achieves much higher accuracy than Smith's method in all situations when using 200 auxiliary links. By employing 10 auxiliary links, our adaptive method also achieves higher accuracy than Smith's method in nine situations. These experimental results suggest that using multiple auxiliary links is helpful to increase the accuracy of browser history sniffing. However, the number of auxiliary links may have a great influence on our method, which shall be changed according to different situations. For example, when fixing $N_m = 10$, our method is valid in Situation 8, but basically fails in Situation 3 through Situation 7, because of the average accuracy less than 50%.

3. Our adaptive method is superior to its variant methods and Smith's method, because it detects visited target URLs in all situations, where the average accuracy is nearly 100% or even exactly 100%. These excellent sniffing results were obtained, because an optimal number of auxiliary links, N^*, were automatically found by our dynamic parameter search algorithm in each situation, thereby leading to high-quality criteria of history sniffing.

Conclusion. Our adaptive method, which attains nearly 100% accuracy on the most popular browsers in different operating systems and platforms, works well in real scenarios.

B. RQ2: How does the fixed time window T influence the performance of browser history sniffing?

Motivation. The fixed time T, the input of our adaptive method, plays an important role to the performance of browser history sniffing. In general, the longer T, the higher precision it can get, also more slowly to sniff. However, if T is too large, the sniffing process is revealed easily, because it costs too much time and resources. We explore how the fixed time T influences the sniffing results and decide a good value for T in this RQ.

Approach. We set the fixed time T from 200 to 1000 ms with a step size of 200 ms and obtain the sniffing results on different situations, and then select an appropriate value of T.

Result. Taking the Chrome browser in Windows OS as an example, as shown in Fig.4, when T is as small as 200ms, the average accuracy is less than 80%. The reason might is because the difference between the two criteria using a small T may be insufficiently large to correctly differentiate the visited and unvisited target URLs. It is obvious that the average accuracy increases with T and researches nearly 100% when T is equal to or greater than 400ms. Experimental results for other browsers show a similar tendency.

Conclusion. Therefore, we recommend an appropriate T of 400ms or slightly larger value (e.g.500ms in Fig.3).

[5] Windows 10 (Intel Core i7, 16GB memory, NVIDIA GeForce GT 330), macOS Mojave (Intel Core i7, 16GB memory, Intel HD Graphics 4000) and Ubuntu 18(Intel Core i7, 8GB memory, Intel HD Graphics)

[6] iOS 12 (Apple A10 Fusion, 2GB memory) and Android 7 (Qualcomm 636, 6GB memory, Adreno 509 GPU).

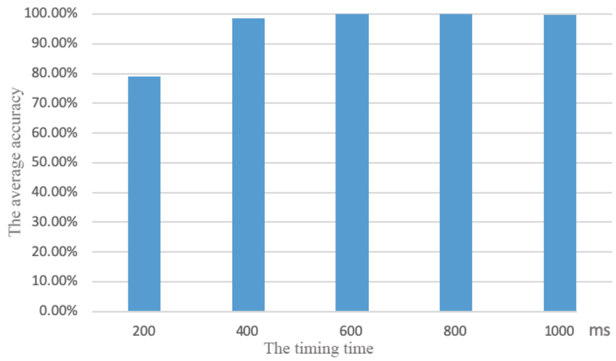


Fig. 4. The average accuracy vs. T, Chrome on Windows

V. Discussion

Our method achieves very high accuracy for sniffing browser history in the browser family of Chrome and Firefox under different operating systems and devices. However, it also has some limitations. First, because the time cost in sniffing each target URL is controlled by the fixed time, T, our sniffing method is not fast enough. For example, as seen in Fig.3, it requires approximately 50 seconds to detect 100 target URLs, where the average time consumed for each target is near $T = 500ms$. That is, our method is effective for sniffing a set of target URLs but the time cost increases with the number of targets. Second, a few immune browsers, such as Edge and IE, still exist, according to the security error code of Microsoft Edge (SEC7115) [9] — **:link** and **:visited** styles on Edge and IE can differ only by color, and other styles were not applied to **:visited**.

To defend against this attack, one possible solution is that the browser does not reuse previous calculation results and recalculates every element every time. But this will increase the burden of browsers and decrease performance, leading to a bad user experience. Alternatively, the browsers may use a temporary variable to store and search the browser history during the current session, and append the temporary variable to place for saving browser history when the user closes the browser. In this way, the attacker can only sniffer a small number of URLs. It can significantly reduce the probability of being attacked.

VI. Related work

This section describes the existing methods of browser history sniffing. We divided them into the three categories.

A. Vulnerability attacks

There have been found several browser vulnerabilities due to design flaws, which may leak a user's access status. The most classic vulnerability is based on the different colors of the links for different visiting status. Cascading Style Sheets (CSS) has a selector **:link** that matches unvisited links and a selector **:visited** that matches visited links [7]. These selectors are used to set different colors for the links in different visiting statuses. Thus, a user can conveniently recognize which links he has visited. But, because browsers provide the function, **getComputedStyle()**, to obtain the CSS style property of any element [6], it is also convenient for attackers who want to steal the browser history. Therefore, an attacker can obtain the visit status of a link by simply calling this JavaScript function to read its color [2]. In fact, not only the color of the link, but also the position, background image, and some other CSS attributes of the link can be revealed [1]. Another typical vulnerability exists in the browsers, which are required to use HTTP Strict Transport Security (HSTS) [4] when visiting websites [5] [14]. HSTS can sniff a user's browser history because it utilizes the different ports between HTTP and HTTPS. That is, an attacker can detect some HSTS-enable websites those do not exist in the HSTS preloading list [3].

In general, because vulnerability attacks exploit some browser flaws directly, they are easy to defend. For example, Baron et al. [1] proposed a defense scheme with some simple changes to the browsers. First, the scheme modifies the function, **getComputedStyle()**, to return a definitive result for each attribute query of one link, no matter whether or not this link has been visited. Second, it makes some limitations, such as forbidding the **:visited** selector to set background images.

B. Interactive attacks

Interactive attacks use special ways to trick users into revealing what they see on the screen. For example, an attacker places several links on the screen, with the color of the unvisited link as background and the color of the visited link highlighted and then prompts that these highlighted links are CAPTCHAs [15]. When entering highlighted texts in an input box, a user reveals to the attacker which links have been visited before. Because it is quite difficult to defend, we believe that this kind of interactive attack is effective to sniff browser history for quite a long time. But, the obvious shortcoming of an interactive attack is that it aims only at a small number of links at once. If testing many links simultaneously, attackers must design more complex interactions, which easily reduces users' experiences or even arouses users' suspicions.

C. Side-channel sniffing attacks

Side channel attacks leak information through a mechanism not intended to provide that information. For example, a side-channel attack [15] found visited links by using a webcam to detect the color of the computer screen from the reflected light, because the dominant color of the screen depends on whether the link was visited before. However, as users are careful about granting access to webcams, this attack may not be practical.

VII. Conclusion

The protection of user privacy, such as browser history, has become a rising concern. So far, most existing methods for browser history sniffing have been prevented. Although Smith's method is still effective, due to parameter sensitivity, it lacks robustness. We proposed an adaptive method for sniffing browser history, which is applicable to different browsers, operating systems, and hardware devices. This cross-platform method improves Smith's method by automatically searching the optimal number of auxiliary links using the dynamic parameter search algorithm. The experimental results show that on the five major browsers in the most popular operating systems and platforms, our adaptive method attains nearly 100% accuracy.

References

[1] D. Baron, "Preventing attacks on a user's history through css :visited selectors," https://dbaron.org/mozilla/visited-privacy, 2010.

[2] A. Clover, "Css visited pages disclosure," https://lists.w3.org/Archives/Public/www-style/2002Feb/0039.html, 2002.

[3] Google, "Hsts preload list submission," https://hstspreload.org.

[4] J. Hodges, C. Jackson, and A. Barth, *RFC 6797 - HTTP Strict Transport Security (HSTS)*, IETF, https://tools.ietf.org/html/rfc6797, November 2012.

[5] Imfaster, "Remote websites can know which hsts enabled websites the users have visited," https://bugs.chromium.org/p/chromium/issues/detail?id=436451, 2014.

[6] MDN-Web-Doc, "window.getcomputedstyle()," https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle, 2019.

[7] MDN-Web-Doc, *css3*, https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3, 2019.

[8] MDN-Web-Doc, *window.requestAnimationFrame()*, https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame, 2019.

[9] Microsoft, "Devtools - console error and status codes - microsoft edge development — microsoft docs," https://docs.microsoft.com/zh-cn/microsoft-edge/devtools-guide/console/error-and-status-codes.

[10] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[11] M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan, "Browser history re: visited," in *12th USENIX Workshop on Offensive Technologies*, 2018.

[12] StatCounter-Global-Stats, "Browser market share worldwide," http://gs.statcounter.com/browser-market-share, 2019.

[13] P. Stone, "Pixel perfect timing attacks with html5," Context Information Security, Tech. Rep., July 2013.

[14] V. Tsyrklevich, "Possible to track users visits to servers with particular hsts configurations," https://bugzilla.mozilla.org/show_bug.cgi?id=1090433, 2014.

[15] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.